Understanding Bulk-Bitwise Processing In-Memory Through Database Analytics

Ben Perach, Ronny Ronen, Fellow, IEEE, Benny Kimelfeld, and Shahar Kvatinsky, Senior Member, IEEE

Abstract—Bulk-bitwise processing-in-memory (PIM), where large bitwise operations are performed in parallel by the memory array itself, is an emerging form of computation with the potential to mitigate the memory wall problem. This paper examines the capabilities of bulk-bitwise PIM by constructing PIMDB, a fully-digital system based on memristive stateful logic, utilizing and focusing on in-memory bulk-bitwise operations, designed to accelerate a real-life workload: analytical processing of relational databases. We introduce a host processor programming model to support bulk-bitwise PIM in virtual memory, develop techniques to efficiently perform in-memory filtering and aggregation operations, and adapt the application data set into the memory. To understand bulk-bitwise PIM, we compare it to an equivalent in-memory database on the same host system. We show that bulk-bitwise PIM substantially lowers the number of required memory read operations, thus accelerating TPC-H filter operations by $1.6 \times -18 \times$ and full queries by $56 \times -608 \times$, while reducing the energy consumption by $1.7 \times -18.6 \times$ and $0.81 \times -12 \times$ for these benchmarks, respectively. Our extensive evaluation uses the gem5 full-system simulation environment. The simulations also evaluate cell endurance, showing that the required endurance is within the range of existing endurance of RRAM devices.

Index Terms—Memory technologies, Emerging technologies, Database processing.

1 INTRODUCTION

promising technique for accelerating memory- ${
m A}$ bounded computation moves some computation from the processor to the memory unit (processing-in-memory, PIM), thus reducing the data movement latency and energy. Due to the ever-increasing performance and energy gaps between processing units and memory, numerous PIM architecture types and techniques, such as near-memory [20], analog [34], and CAM [7,17], have gained attention in recent years. One particular PIM technique relies on using the memory cell arrays to both store the data and perform logical operations. This PIM technique can be implemented with DRAM [14,33] or nonvolatile memory technologies [13, 16, 23, 24, 30, 37]. Because of the regular structure of the memory cell array, logical operations can be performed concurrently between many sets of cells, resulting in bulkbitwise operations. Hence, in addition to the reduced data movement compared to other PIM techniques [31], bulkbitwise PIM can achieve significant operation throughput, potentially being performed on all memory cell arrays in parallel. Nevertheless, building such a system and executing complete applications using bulk-bitwise PIM remains a challenge.

Previous works focused on certain aspects of bulkbitwise PIM. These aspects include exploration of tech-

Manuscript received September 30, 2022. This work was supported by the European Research Council through the European Union's Horizon 2020 Research and Innovation Programme under Grant 757259.

nology [12, 23, 30, 33, 40], programming language abstraction [14, 45], memory interface [37], logic design and abstraction [3, 24, 36], and combining bulk-bitwise with other PIM technologies [16, 17]. A full system perspective of bulkbitwise PIM is, however, still missing. Insights about the benefits and limitations of such an approach need to be educed. Aspects such as what applications are suitable for this type of processing and what the system requirements have yet to be resolved. In this paper, our goal is to investigate bulk-bitwise PIM and determine how it can be utilized. We ask the above questions and seek to understand the performance gains, drawbacks, and bottlenecks of bulkbitwise PIM.

To achieve our goal of bulk-bitwise PIM investigation, we propose PIMDB, an example of a full PIM system based on bulk-bitwise logic. PIMDB accelerates operations from a real-world application: analytical processing of relational databases. We focus on a single application case study to concisely present its adaptation for bulk-bitwise PIM (for software and hardware) and its performance analysis. By utilizing only bulk-bitwise PIM techniques and addressing the requirements of a real-world application, the benefits and limitations of bulk-bitwise PIM can be identified. PIMDB is also the first to introduce a host programming model that works in virtual memory for PIM bulk-bitwise operations. The programming model is an essential part of our evaluation, as it reflects the interaction of the software, host hardware, and PIM hardware.

We target analytical processing of relational databases as our example application for several reasons. First, it is an important real-world application, found in business decision support processes [28, 39]. Second, relational databases are stored in tabular structures called *relations*, each a collection of small and independent items called *records*. Records can be stored in a small physical memory region and operated

B. Perach, R. Ronen, and S. Kvatinsky are with the Andrew and Erna Viterbi Faculty of Electrical and Computer Engineering, Technion – Israel Institute of Technology, Haifa.
 E-mail: benperach@campus.technion.ac.il, ronny.ronen@technion.ac.il,

shahar@ee.technion.ac.il.

B. Kimelfeld is with the Taub Faculty of Computer Science, Technion – Israel Institute of Technology, Haifa. E-mail: bennyk@cs.technion.as.il.

on concurrently. The ability to fit each independent data unit into a small physical memory region is crucial, as bulk-bitwise operations require the operands from the same data unit to be in physical proximity in the memory cell array [14, 36]. Third, a database can reside in memory in a dedicated structure and be repeatedly used, amortizing the one-time cost of loading the database into the PIM module. When executing queries, no data transfer to the PIM module is required before execution starts. Fourth, database analytical processing is a memory-bound application [28], requiring the performance of simple operations (e.g., comparison, addition, multiplication) on large data sets (from a few to thousands of GBs [39]). Fifth, analytical processing queries perform aggregation (e.g., sum, average) on many database records [28]. Performing the aggregation, or part of the aggregation, in memory can potentially reduce data movement by several orders of magnitude.

PIMDB uses a byte-addressable, nonvolatile memristive memory utilizing stateful logic [5,13,16,24,30,31,37], which stores a copy of the database. We chose memristive stateful logic as the underlying bulk-bitwise technology as it can operate on both bitlines and wordlines [36], enabling a richer functionality compared to other (e.g., DRAM-based) bulk-bitwise PIM. Nevertheless, our techniques are not limited only to memristive stateful logic; they can be applied to other bulk-bitwise PIM technologies as well. PIMDB's speedup is primarily driven by the reduction of memory accesses, where the PIM operations reduce the amount of data read out of the memory array. In this paper, we show that over 99% of the reads can be eliminated for some queries when using bulk-bitwise operations. The proposed design supports in-memory record filtering and aggregation operations over database relations (similarly to commercial in-storage analytical processing [43]). For queries involving a single relation, most query execution can be performed solely in PIMDB. For queries involving multiple relations, PIMDB performs the filtering while the rest of the query is executed at the host in a standard non-PIM fashion.

PIMDB is evaluated in a gem5 full-system simulation [4], using queries from the TPC-H benchmark [39] where the SQL queries are compiled using our in-house compiler. The proposed design uses stateful logic with memristive devices [31], specifically, RRAM-based MAGIC NOR gates [21]. As PIMDB focuses on bulk-bitwise operations, seeking to reveal their strengths and weaknesses, our evaluation is of use for other PIM architectures beyond the context of databases.

In summary, this paper makes the following contributions:

- We evaluate the performance of and derive insights regarding bulk-bitwise PIM operations in general and in stateful logic specifically.
- We define a programming model for bulk-bitwise logicbased PIM architectures, enabling the use of bulkbitwise logic as part of host applications in virtual memory space.
- We present PIMDB, a PIM-based accelerator for analytical processing of relational databases.
- We establish database querying as an empirical proof of application for bulk-bitwise PIM where the latter accelerates the performance of the underlying operations.



Fig. 1: (a) A 3×4 memristive crossbar array with a single device per cell (1R [41]) and a single sense amplifier. (b) Logic operations between crossbar columns. The inputs are marked in red, the output in green. Devices in black do not participate in the logic operation. Specifically, row 2 is isolated and does not participate in the operation. Row-wise operations can be performed similarly [36].

- We demonstrate speedup improvement of $1.6 \times -18 \times$ on filter operations and $56 \times -608 \times$ for full queries from TPC-H, and an energy reduction of $1.7 \times -18.6 \times$ and $0.81 \times -12 \times$, respectively.
- We provide a gem5 full-system simulation for bulkbitwise PIM¹.

2 BACKGROUND

2.1 Bulk-Bitwise PIM and Stateful Logic

Bulk-bitwise PIM refers to a class of PIM techniques categorized by the location and capabilities of the PIM processing elements. In bulk-bitwise PIM, the computation elements are the memory cells and their periphery circuits (e.g., sense amplifiers, voltage drivers, decoders), thus the memory storage medium itself can compute (i.e., PIM, referred to also as *processing using memory*). Several technologies were suggested for implementing such processing capabilities, including DRAM [14,33] and emerging nonvolatile memory technologies [13, 16, 30, 37], all of which execute simple logic operations (e.g., AND, OR, NOT). These simple logical operations use the memory cells as inputs and write the result directly to a memory cell. Furthermore, due to the regular structure of memory cells in a memory array, these operations can be concurrently performed on many sets of cells within an array, *i.e.*, executing bitwise operations. As different memory arrays are independent circuits, many memory arrays can concurrently perform these bitwise operations, resulting in a wide bitwise operation, *i.e.*, a bulkbitwise operation. The concurrent operation on many memory arrays, each performed on many sets of cells, produces substantial computational throughput.

Implementing bulk-bitwise PIM using emerging memory technologies is often referred to as stateful logic [5, 21] and specifically refers to memristive technology (*e.g.*, ReRAM, PCM, MTJ) [44]. Memristive devices are devices that can change their resistance when electrically stimulated.

The resistive state of memristive devices is nonvolatile, and can, therefore, be used to construct nonvolatile memories [41], where each memory cell is a single memristive device. A memory array of memristive devices is constructed

1. https://github.com/benperach/gem5_PIM_extension

as a crossbar [41] (Fig. 1a shows an example of a 3×4 crossbar array), where the horizontal and vertical wires are referred to as wordlines and bitlines, respectively. These memories are random-access memories, as they can read or write specifically selected cells.

Bulk-bitwise operations are performed by applying a voltage across the bitlines and/or wordlines in a certain manner, independently of the stored cell values. Furthermore, this logic technique can be applied in parallel on a crossbar row or column [36], as illustrated for columnwise operations in Fig. 1b. It is possible to exclude certain rows or columns from a parallel operation. For example, Fig. 1b shows how a row can be excluded in a crossbar with a single memristive device per cell (1R crossbar [41]). The row exclusion from a parallel operation is done by setting the corresponding wordline to an isolation voltage, leaving the output device undisturbed [36]. Note that the input and output operands for stateful logic, as for other bulk-bitwise technologies, must reside on the same wordline/bitline. This physical proximity of operands poses a challenge for bulk-bitwise PIM, as the data has to be arranged in an appropriate way to enable PIM use.

An example of a stateful logic technique is MAGIC NOR [16, 21]. Since NOR is functionally complete, any logical function can be performed using a series of NOR operations (requiring additional devices for the intermediate values). In PIMDB, we use MAGIC NOR as the basic stateful logic gate. Nevertheless, most of the concepts presented for MAGIC NOR apply to any functionally complete bulkbitwise PIM technology [14, 23, 33] (all concepts apply as long as row-wise and column-wise bulk-bitwise operations are available).

Memristive random-access memory can be used as the system's main memory. Memristive memory has several potential advantages over DRAM; the small device size and the dense layout of memristive arrays result in high-capacity memory [36]. The nonvolatility of memristive devices eliminates the need for memory refreshes, reducing idle power consumption. The disadvantages of memristive memory are its longer access time relative to DRAM and its limited cell endurance [44]. Due to these limitations, to date, memristive memories have not been replacing DRAM as main memory; rather, memristive memories have been augmenting DRAM as a new memory hierarchy tier [18].

2.2 Memory Organization

DRAM and memristive memory are constructed similarly. The memory is divided into *channels, ranks, chips,* and *banks* [19,41]. Each channel has a dedicated bus, and though the ranks in a channel share the bus, they can operate independently. Each rank comprises one or more banks and one or more chips. Each bank is logically distributed across several chips where all bank parts operate in lockstep. The banks of a rank share the rank's memory bus interface and can operate independently.

For practical reasons, banks are not built as a single memory array in each chip; they are split into units called *subarrays* [41]. In memristive memory, a subarray comprises several crossbars (including periphery circuits, *e.g.*, sense amplifiers, voltage drivers, decoders), which operate in lockstep. When accessing a bank as part of a traditional memory read/write operation, only the necessary subarray is activated.

Traditionally, the host-memory interface is designed especially for DRAM (e.g., the DDRx standard), which results in lower performance for memristive technologies [37]. Furthermore, traditional memory interfaces specify a fixed set of memory operations (e.g., read, write, refresh, etc.), limiting the use of novel memory designs (such as PIMenabled memory). To address these issues, new technologyagnostic interfaces have been proposed, e.g., OpenCAPI, GenZ , and CXL . These new interfaces issue technologyindependent operations, with possible vendor extensions, and delegate the technology-dependent management to the memory module. The general structure of these new interfaces is shown in Fig. 2, where the host-side memory controller communicates with a *media controller* using a technology-agnostic interface, while the media controller communicates with the memory chips using a technologyspecific interface (e.g. DDRx for DRAM, R-DDR [37] for RRAM). The media controller is part of the memory module, enabling optimization of the media controller for the specific memory technology and design.

2.3 Relational Databases and Analytical Processing

A relational database is a data model that organizes data into relations (tables) [28]. A relation can be thought of as a set of records and a set of attributes. Attributes may be of different types, and each record is assigned a value for each attribute. In the relation, each record is represented by a different row, and each attribute is represented by a different column. The value of a record for a specific attribute is the relation entry located at the intersection of the record row and the attribute column. Fig. 5a visualizes a relation.

A query over the database is a question about the content of the database's relations, *i.e.*, about attributes of a specific record or a group of records fulfilling a certain condition. Analytical processing refers to a class of queries requiring the selection of records according to some criteria and their aggregation (*e.g.*, sum, average) [28]. These kinds of queries appear in applications such as business decision support processes and have dedicated benchmarks [39].

Databases can be represented and stored in various data structures. Two main categories of such data structures are *row-store* and *column-store*. In row-store, a relation is stored as an array of records, each record having a field for each attribute. In column-store, each relation attribute is stored as a single array, and a relation is a collection of such arrays. Due to data locality, row-store is better suited for accessing single records, while column-store is better suited for accessing several attributes over many records [28]. In PIMDB, reads and writes access relations as row-stored, while PIM operations, due to their access pattern, access the same relations as column-stored (see Section 4).

3 PIMDB ARCHITECTURE OVERVIEW

We now present the proposed PIMDB design. This section also presents the PIMDB programming model, which specifies the PIM interface for the software (*e.g.*, for compiler



Fig. 2: The general structure for emerging interfaces (*e.g.*, OpenCAPI, GenZ, CXL). The host-memory interface is agnostic to the memory technology; the technology-specific operations are performed by the memory module.

or library writers; for databases, these are the database management system writers). The programming model can be used by a compiler from a high-level language such as SQL, as used for our evaluation (Section 5.4).

The PIMDB system, shown in Fig. 4, comprises a host processor, DRAM main memory, and additional byteaddressable memory ranks based on RRAM to support bulk-bitwise MAGIC NOR stateful logic operations [21]. These bulk-bitwise PIM ranks are termed *PIM modules*, and their memory is mapped to the host memory address space and accessed using standard reads and writes. The PIM modules have a predefined instruction set, performing operations, such as add, multiply, less-than, *etc*.

3.1 Programming Model

The programming model defines how the host software interacts with the PIM modules, *i.e.*, how it utilizes the PIM module's instructions on the relevant data. We show here how this interaction can support virtual memory and how the software can control the physical proximity of PIM operands.

To operate on data in a PIM module, the host software sends a sequence of requests, named *PIM requests*, to the PIM module. PIM requests contain address and data fields, similar to a memory write operation. In PIMDB, for example, PIM requests are sent by a dedicated host instruction, similar to loads and stores. Regardless of how PIM requests are sent, each PIM request contains information for a single PIM instruction, *e.g.*, the instruction's opcode and locations of operands. The possible instructions are determined by the PIM module's ISA, possibly high-level operations such as addition and multiplication (the translation to basic operations supported by the PIM module's crossbars, *e.g.*, a sequence of MAGIC NOR operations as in PIMDB, is done at the PIM module).

Virtual memory: Since bulk-bitwise PIM is done in the main memory of the host processor, and since user-level processes use virtual memory, it is essential for bulk-bitwise PIM to support virtual memory to be integrated into such a system. Furthermore, supporting virtual memory is essential not only from a compatibility point of view but because of the advantages virtual memory provides: holding multiple and independently operated-on data structures (such as database relations) in memory and enabling dynamic data allocation. Virtual memory support can be achieved by allowing PIM instructions to operate within the boundaries of a specific virtual memory page, *i.e.*, a PIM request is sent to an individual virtual page and can only access and change data within that page. Such a mechanism utilizes the existing virtual memory implementation, issuing the PIM requests with virtual addresses that are translated in the usual way to physical addresses, and thereafter sent to the required memory module. When operating on a data structure spanning multiple pages, a PIM request should be sent to each page belonging to that data structure.

Nevertheless, there are challenges in supporting virtual memory as suggested: (1) Bulk-bitwise PIM operation's operands must be located on the same memory crossbar, possibly on the same crossbar row or column [33, 37]. The physical memory abstraction, where the memory channels, ranks, banks, and crossbars appear as a monolithic address range, hides the knowledge required for such data allocation. Moreover, the virtual memory abstraction hides the specific physical memory address from the user-level software. To allocate data for bulk-bitwise PIM, these two abstraction levels, virtual and physical memory, must be circumvented in some manner. (2) An essential property for bulk-bitwise PIM performance is its high parallelism. Hence, a PIM request should initiate its specified PIM instruction on as many crossbars as possible. Restricting a single PIM request to a single page, however, will restrict the number of crossbars the PIM request targets, limiting the benefit of bulk-bitwise PIM. This situation is especially acute when dealing with standard 4KB pages, which are smaller than a single crossbar [19, 41]. In the following paragraphs, we address these challenges, explaining how our programming model entails the address mapping of physical addresses to memory cells and utilizes huge-pages.

Using huge-pages: To maximize the number of crossbars within a page and increase the parallelism of a single bulkbitwise operation, we use huge-pages. Huge pages can range in size from a few MB to a few GB (PIMDB uses 1GB huge-pages), sufficiently large to contain multiple memory crossbars. These huge pages are only required for the PIM memory, which is in addition to the standard DRAM main memory (see the start of Section 3 and Figure 4). Hence, using huge pages for PIM does not affect the usability and performance of the DRAM main memory.

PIM operand allocation: To address the challenge of how to allocate PIM operands, we give the software finegrained control over the location of values in the memory array. Using this fine-grained control, the software will decide how each data structure will be located across crossbars, crossbar rows, and crossbar columns. As the software is the entity that knows how the data is going to be operated on, allowing it the flexibility to set data structures according to its specific needs will allow our programming model to suit different applications. To enable such fine-grained control, the virtual-to-physical address translation and the physicaladdress to memory cell translation have to be controlled in some manner by the software.

To control the virtual-to-physical address translation we note that the address translation does not change the page offset. This allows the software to directly control the page offset bits of the physical address. Because our programming model allows bulk-bitwise PIM operations to be performed only within a single page, controlling the page offset, which encodes the relative location within the page, is sufficient for PIM data allocation.

To control the physical-address to memory cell translation, our programming model reveals the necessary details



Fig. 3: Example of address mapping, revealed to the programmer as part of the programming model. PIMDB uses this configuration for 1GB pages with 1024×512 cell crossbars

(see Section 5.2). The fields of this mapping are not consecutive due to the internal structure of the memory, *e.g.*, the number of accessed crossbars in a read/write and the number of bits read/written together from a single crossbar.

of that translation to the software. The necessary details are which bits in the physical page offset decode the crossbar index within the page, and the row and column index within the crossbar. An example of such mapping is shown in Fig. 3. Hence, by controlling the crossbar, row, and column bits in a virtual address, user-level software can target and manage each cell in each crossbar of a single page. By using loads, stores, or PIM requests, the value of specific cells can be read, written, or operated on. Note that a continuous data unit on a crossbar row (*e.g.*, the relation record in Section 4.1) is not necessarily continuous in the virtual address space. Non-continuous data structures can be handled by software abstraction (as our compiler does in Section 5.4).

PIM requests: PIM requests encode the details of the PIM operation they transfer with their address and data. The page number in the *address* specifies the page for which the instruction is meant. PIM requests target all crossbars within their targeted page, thus ignoring the crossbar index field within the address. The column and/or row index fields within the page offset indicate the location of the instruction *result* within each crossbar. The PIM request's *data* specify all additional fields: opcode, location of input operands within the crossbar, immediate operands, *etc.* As this work focuses only on bulk-bitwise operations, no PIM instructions are allowed to move data between crossbars or operate on operands from different crossbars, whether within the same page or among pages. To move data among crossbars, standard reads and writes should be used.

For memory consistency purposes, the same rules that apply to write requests apply to PIM requests, as PIM requests are similar to writes in the sense that they change the memory data. To enforce the desired memory ordering of reads, writes, and PIM requests, programmers should use the memory order rules of the host processor, treating PIM requests as write requests, as well as using the available ordering primitives (*e.g.*, memory fences). Furthermore, the processor caches ignore PIM requests and forward them to the memory. It is the responsibility of the programmer to ensure coherency between the host caches and the PIM memory, using cache flushes.

The use of PIM requests abstract the PIM module details for the host hardware. The host hardware is simply required to transfer the PIM requests to the PIM modules – the host is not required to know their exact meaning. Therefore, the host is able to work with different PIM modules, utilizing different bulk-bitwise PIM technologies and instruction sets. The software determines the address and data of the PIM requests, cognizant of their full meaning, and thus controls the PIM modules. Hence, our proposed programming model is general and can apply to applications other than database analytical processing. The extent of the host hardware knowledge is presented in Section 3.4.

Additional computation area: To execute PIM instructions, the PIM modules require some crossbar area for their internal computations (for storing intermediate results). The software is responsible for allocating this crossbar area by configuring the PIM module. To provide the software with the necessary knowledge, the computation area requirements for each PIM instruction are given as part of the PIM module instruction set architecture. Each page has its own computation area configuration, which applies to all crossbars within the page. This configuration, conveyed via a special PIM instruction to the targeted page, configures the page but does not access the crossbars themselves. The software can change the page configuration at any time to suit the needs of the upcoming PIM instructions.

3.2 The PIM Module

The proposed PIM module is a single memory rank and is constructed similarly to an RRAM memory module: several RRAM memory chips and a single media controller chip (as shown in Fig. 4). The media controller communicates with the host memory controller using the OpenCAPI interface [15], and with the memory chips through an interface similar to the R-DDR interface [37]. The media controller schedules the incoming requests using a First-Ready-First-Come-First-Serve (FR-FCFS) policy, which considers the dependencies between reads, writes, and PIM requests.

The R-DDR interface is similar to DDRx [19]. It has an address bus, data bus, and several command signals. The address bus and command signals are connected in parallel to all memory chips, making the chips work in lockstep, while the data bus is divided among the chips. The difference between R-DDR and DDRx is in the operations they specify; R-DDR targets RRAM technology, while DDRx targets DRAM (see [37] for details). For the interface used in this work, reads and writes are kept the same as in R-DDR, while PIM operations are handled differently. A PIM operation is specified, similar to a read or write operation, by a certain combination of the command signals. On issuing a PIM request to the RRAM chips, the media controller sets the command signals accordingly, passes the address of the PIM request to the address bus, and duplicates the PIM request's data to all the chip data buses (according to the timing required by the interface), issuing the same PIM request to all chips in parallel.

Each memory chip is constructed similarly to an RRAM memory chip [41]. Reads and writes are performed as in a standard RRAM chip. To support PIM operations, *PIM controllers* are incorporated in the memory banks, as illustrated in Fig. 4. Each PIM controller controls all crossbars across multiple subarrays, all of which belong to the same memory space of a single huge-page. A huge-page is assigned to a single bank of a single memory rank, requiring several PIM controllers in each chip to cover all the page's subarrays. When a PIM request is issued to the chip, its information is routed to the destination bank as a read or a write, and then intercepted by the targeted PIM controllers. Each targeted PIM controller then issues the required MAGIC NOR sequence of operations to all crossbars connected to it,



Fig. 4: PIMDB system structure and PIM-enabled memory bank schematic.

according to the specific instruction. While a PIM controller is executing its operation, since it does not require the use of the bank's global resources, the bank can perform reads, writes, or PIM operations on subarrays that are not being controlled by that PIM controller.

The media controller is responsible for orchestrating the requests sent to the memory chips and for identifying the bank or subarray that can operate (similar to the operation of the memory controller in a DDRx interface).

PIM architectures occasionally need inter-crossbar data movements. These data movements can be accomplished indirectly via simple, yet slower, CPU load/store instructions or using faster, but more complex, direct inter-crossbar communication [16, 40]. Direct inter-crossbar communication can be added to PIMDB at the cost of added hardware and complexity. Here, we decided not to include direct inter-crossbar communication, as adding such a PIM technique obscures the explicit contribution of bulk-bitwise PIM and our evaluation (Section 6) shows that PIMDB achieves significant speedup without it. Hence, inter-crossbar data movements are performed via standard load/stores.

3.3 Instruction Design

PIM controllers perform each PIM instruction as a sequence of bulk-bitwise NOR [21] operations. This sequence is implemented as a finite-state-machine (FSM) included in each PIM controller. Some instructions require operations on in-memory values of various lengths (Section 4.2). Implementing a predefined operation sequence for every possible length, such as suggested in [3, 38], might require numerous FSM states. Instead, we implement an *n*-bit operation by iterating a single-bit operation *n*-times. The operand's length determines the number of iterations. For instance, addition and multiplication with variable-length operands are implemented by iterating over a full-adder sequence. Such implementation increases the length of the bulk-bitwise logic sequence for each instruction, since there is no optimization across iterations, but it supports many operand lengths with the same states in the FSM. Moreover, such implementation reduces the number of cells required for the internal computation, given that the iterated short logic sequence can naturally reuse the same cells.

Additionally, we added an optimization that uses immediate values (instruction constants) in the control paths. When performing PIM operations with an immediate value, **Algorithm 1:** PIM controller algorithm for equality operation between an in-memory value and an immediate value. **Input :** $v_{n-1}...v_0$ - in-memory value bits

	input : $v_{n-1}v_0$ - in-memory value bits
	$c_{n-1}c_0$ - immediate bits at the PIM controller
	Output: in-memory bit m_{eq}
	Result: $m_{eq} = 1$ if for all $i v_i = c_i$, else $m_{eq} = 0$
1	$InMemory(m_{eg} \leftarrow 1)$
2	foreach $i \in [0,, n-1]$ do
3	if $c_i = 1$ then
4	$InMemory(m_{eq} \leftarrow v_i \ AND \ m_{eq})$
5	else
6	$InMemory(m_{eq} \leftarrow NOT(v_i) \ AND \ m_{eq})$

the control sequence of the FSM depends on this value. Thus, instead of writing the immediate value to the crossbar (possibly duplicating it on every row) and performing the operation between two in-memory values, the bulk-bitwise logic sequence is determined according to the immediate value. An example of an equality comparison between an in-memory value and an immediate value is shown in Algorithm 1. Note that an instruction uses the same FSM states for all immediate values. The use of the immediate value for control reduces the number of logic operations and the latency, eliminates the need to store the immediate value in each row, and improves the lifetime of the PIM module due to fewer cell writes. To the best of our knowledge, this work is the first to implement bulk-bitwise logic operations in this manner.

3.4 Modifications to the Host Processor

To support PIM requests, some adjustments to the host are needed. First, the host must be capable of issuing a softwareinitiated PIM request, similar to the way the host sends a read/write request. For memory operation ordering, a PIM request is considered as if it were a write request. Caches at the host, however, should not use the content of the PIM requests' address or data since their meaning is dependent on the PIM specifications. In this work, caches forward PIM requests as is and do not change their state or data when encountering PIM requests. As explained in Section 3.1, it is the programmer's responsibility, *e.g.*, using flushes, to ensure coherence between the memory and the caches when using PIM requests. We leave further investigation of PIM request consistency and coherence support for future work.

Further, the host processor should have memory controllers that support OpenCAPI. These controllers are assumed to preserve the order of requests passed to them since they cannot know the meaning of the PIM requests' data and addresses.

4 MAPPING DATABASES TO PIMDB

In this section, we show how a database is mapped to PIMDB. First, the memory layout of a database relation is presented. Then, the PIM instructions necessary to support database applications are described. These PIM instructions need to be implemented as a sequence of bulk-bitwise logic operations by the PIM controller (described in Section 3.2).

PIMDB holds a copy of subset of the database in the PIM modules to accelerate certain operations. Other operations



Fig. 5: (a) Visualization of a database relation. (b) The layout of the relation in a crossbar (partly shown). The attributes of a record are within a single crossbar row. Attributes are aligned across rows.



Fig. 6: Example of a column-transform operation: Transferring a single crossbar column to several rows. In this example, a column in an 8×7 crossbar is transformed into two rows, each with four cells. (a) The initial state of the crossbar with the input column. (b) Negating each value to its targeted result column. (c) Negating each value to its targeted result row. (d) The final state of the crossbar with input column and result.

are performed in a non-PIM manner using the host and DRAM main memory [28]. The database copy is constructed offline once and then used for query execution. The query execution does not modify the database copy, so further data copying is not necessary prior to subsequent query execution. Note that maintaining database copies is common in database systems [29].

4.1 Database Relation Memory Layout

Database relations are stored in PIMDB memory such that each record is placed in a crossbar row. For each relation, enough huge-pages are assigned to it to hold all of its records. Records can be assigned to the rows of a crossbar in any order, and to any crossbar in the huge-pages assigned to the relation (no two relations share a page). If a relation requires more records, additional pages can be assigned to it dynamically. Each attribute is aligned in all the crossbar rows and is contained in consecutive crossbar cells² (Fig. 5 shows this mapping). Since reads and writes are performed on crossbar rows, a relation is held in a row-store manner in the PIM memory, making accessing and updating records straightforward. Nevertheless, a single record is not entirely consecutive in the host's virtual memory due to the mapping of the crossbar cells to memory addresses (see Section 3.1).

If relation records are too large to fit into a single crossbar row, the attributes can be distributed across several crossbars, with each crossbar on a different huge-page. To



Fig. 7: Example of a reduce operation: Reducing (SUM/MIN/MAX) values stored in multiple rows to a single value in a single row. In this example, a column of two-cell values (green) in an 8×8 crossbar is reduced to a single value.

Unlike the example, for some operations (*e.g.*, SUM) the number of bits after each reduction may change. (a) Copying half of the values to the rows of the other half of the values. (b) Reducing values on the same row to a single value using a column-wise operation (SUM/MIN/MAX). (c) Copying half of the reduced values to the rows of the other half. (d) Reducing values again using a column-wise operation. (e) Copying half

of the reduced values to the rows of the other half. (f)Reducing values for the last time, resulting in a single value.

operate on such a relation, the host might need to transfer partial operation results between the pages through reads and writes, slowing the execution. Note that for TPC-H, the physical size of the crossbar rows is sufficient for all relations and there is no need to split attributes of a relation among different pages.

4.2 PIM Instructions

To support database queries, PIMDB performs *filter* and *aggregate* operations within the crossbars. This section explains which PIM instructions PIMDB implements to support the filter and aggregate operations. These instructions consist of traditional arithmetic instructions and three additional complex instructions, Table 4 lists all implemented instructions.

Filter operations operate on every record in a database relation and specify which records pass the filter condition. A filter condition is defined using the attributes of the relation and can be evaluated as True or False for each record, according to the attribute values of the record. A condition can be a single comparison or a logical combination (*e.g.*, OR, AND, NOT) of several comparisons. A comparison is one of the following operations: $=, \neq, <, >, \leq, \geq$, between attributes, functions of attributes, and constants.

To support filter operations, the PIM controller supports comparison between two in-memory values or an inmemory value and an immediate (constant), bitwise logical operations, and simple arithmetic operations (*e.g.*, addition, multiplication), all of which are performed by bulk-bitwise logic on all crossbar rows. These operations should be

^{2.} PIM instructions are supported on consecutive cells; see Section 4.2.

Relation	In	# of	# of Crossbar	# of PIM	Memory	
	PIMRecordsPART \checkmark 2×10^8		Row Bits	Pages	Utilization	
PART			124	12	24.1%	
SUPPLIER	\checkmark	1×10^7	99	1	12%	
PARTSUPP	\checkmark	8×10^8	80	48	15.5%	
CUSTOMER	\checkmark	1.5×10^8	106	9	20.6%	
ORDERS	ORDERS \checkmark 1.5×10^9		133	90	25.8%	
LINEITEM \checkmark 6×10^9		6×10^{9}	191	358	37.3%	
NATION	Х	25	-	-	-	
REGION	X	5	-	-	-	
Total		-	518	32.6%		

TABLE 1: PIM layout summary for TPC-H relations with $SF=1000\,$

supported for an arbitrary length of consecutive crossbar columns since a database attribute can have any length; *e.g.*, we may need to compare 8-bit numbers as well as 32-bit numbers. When filtering using these operations, the Boolean result will appear as the value of a single cell in each crossbar row (the same cell in all rows, indicated by the PIM request address and constituting a crossbar column). The Boolean result indicates whether or not the record associated with that row passed the filter condition. Thus, by performing filter operations in memory, instead of reading all the attributes for filtering per record, only a single bit per record is read, substantially reducing the data transfer.

Since the reads from the crossbar are row-oriented, the PIM controller also supports a *column-transform* instruction to enable efficient retrieval of the filter result. The column-transform operation takes a single cell column at each crossbar, and transfers the values of the column's cells into several crossbar rows. The number of values in each target row is fixed to the size of a read from the crossbar. After transforming the filter results into a row orientation, the results can be read with higher efficiency than with a column orientation. The column-transform operation is shown in Fig. 6.

Aggregate operations are a reduction of many values to a single value, e.g., taking the sum/average/minimum of an attribute from selected records in a relation. To support aggregation, the PIM controller implements reduce instructions on consecutive crossbar columns across all rows, as shown in Fig. 7. The cells in each crossbar row for the given consecutive columns are assumed to constitute a single value, and the values from all rows are reduced to a single value result according to the specific arithmetic operation. The result location is specified by the PIM request. If a value of a specific row needs to be ignored, it should be adjusted beforehand by PIM operations according to the specific reduce operation. For instance, to ignore rows in a SUM reduce, a filter should be computed and AND with the column value to be reduced. This will zero undesirable values and the SUM will operate on the desired values only. After the reduce operation, the reduced values from all crossbars are read and combined by the host processor. Thus, when performing filtering and aggregation, instead of reading the attributes for each record, only a single value is read from each crossbar per aggregation (e.g., in our evaluation, 1024 records per crossbar are reduced to a single value, different number of records in a crossbar changes the read reduction amount).

Filter-Only Queries								
02	DADT CUDDITED	03	CUSTOMER,					
Q2	FART, SUFFLIER	25	ORDERS, LINEITEM					
04	ODDEDS ITNETTEM	Q5	SUPPLIER,					
Q4	ORDERS, LINEIIEM		CUSTOMER, ORDERS					
07	SUPPLIER,	00	PART, ORDER,					
Q7	CUSTOMER, LINEITEM	Qο	CUSTOMER					
Q10	ORDERS, LINEITEM	Q11	SUPPLIER					
Q12	LINEITEM	Q14	LINEITEM					
Q15	LINEITEM	Q16	PART					
Q17	PART	Q19	PART, LINEITEM					
020	SUPPLIER,	021	SUPPLIER,					
Q20	LINEITEM	Q2⊥	ORDERS, LINEITEM					
	Full Queries							
Q1	LINEITEM	Q6	LINEITEM					
Q22_sub	CUSTOMER	-	-					

TABLE 2: PIM operated relations for each query in our evaluation. Filter-only queries may operate on more relations in a non-PIM manner.

Since reduce operates first on values in the same crossbar, and then on values across crossbars, only commutative and associative operations can be supported, such as SUM, MIN, and MAX. Non-commutative or non-associative aggregate operations (*e.g.*, average) can be implemented using several supported instructions and additional host computation. For example, to perform an average, the PIM module performs a SUM on the requested attribute and then another SUM on the filter result (in a column orientation, providing the record count). Thereafter, the host performs the division between the two SUM results.

As stated before, PIM instructions operate on crossbar columns, and relation attributes are set along crossbar columns. Thus from the PIM perspective, the relation is held in PIMDB as a column-store. Hence, reads and writes see relations in a row-store (Section 4.1), while PIM operations see a column-store.

5 EVALUATION METHODOLOGY

To evaluate PIMDB, the gem5 full-system simulator [4] is used with the necessary additions and modifications. As benchmarks, TPC-H [39] queries are used. In this section, we elaborate on PIMDB and workload implementation details, including the database layout in the memory, query execution, and micro-architecture details. Following this, the evaluation results are presented and discussed.

5.1 TPC-H Benchmark

TPC-H [39] is a database benchmark for business decision support, which specifies a database and a suite of queries. TPC-H controls the size of the database through a *scale factor (SF)*. The size of some relations is linearly dependent on the *SF*, and the database total size is approximately the *SF* in units of GB. We evaluate the proposed design with SF = 1000, *i.e.*, a database approximately 1TB in size (before compression).

Table 1 summarizes the layout of the TPC-H relations in the PIM modules. Relations with few records are not assigned to the PIM modules. Instead, they are columnstored in DRAM, as directly accessing a few records in DRAM is more efficient than PIM operations. For PIM module relations, attributes are compressed using simple schemes, without limiting the relevant PIM operations,

when applicable: dictionary encoding [28] (allowing equality comparisons), or leading-zero suppression [1] (allowing all operations). Large text attributes that are almost unused by the TPC-H queries are not included in the PIM module database copy; this is done to preserve space for computation. These attributes are the NAME, ADDRESS, and COMMENT of the various relations. If a query requires the use of these attributes, they are accessed from the DRAM. Additionally, a valid attribute is added to each relation in the PIM module so that unused crossbar rows can be ignored in query execution. Decisions on which relations should reside in the PIM module, what attributes to include, and how to encode them, were made manually for the evaluation presented here. Such decisions can be automated in the same manner as other implementation decisions in database management [8].

The memory utilization listed in Table 1 considers the data size of the relations in the PIM module and the allocated memory space in huge-pages, capturing the memory overhead for enabling PIM. The relatively low memory utilization is primarily due to the low utilization of a crossbar row, rather than unused rows, *i.e.*, most of the unoccupied space can be used for intermediate results during the computation and is not wasted. Furthermore, this space can be used for extending the life of memory cells with wear leveling (see Section 6.4).

Using the TPC-H benchmark, we evaluate the query operations that can be performed with the PIM modules, as the focus is on bulk-bitwise operation performance. Three of the queries, Q1, Q6, and a sub-query of Q22 (Q22_sub), operate on single relations only, so both filtering and aggregation can be performed in the PIM modules. The rest of the queries operate on multiple relations and can perform only the filter part in the PIM module. These two groups are referred to as *full queries* and *filter-only queries*, respectively. Three of the filter-only queries, Q9, Q13, and Q18, only filter attributes that are not included in the PIM memory (see the start of this section), so they do not involve any bulk-bitwise operations and are not evaluated. Table 2 presents a byquery summary of the relations that are operated on using the PIM modules (Table 2 only specifies relations involving PIM operations).

5.2 PIM Module Micro-Architecture

A single PIM module is a single memory rank, comprising a media controller chip and eight PIM-enabled memory chips. The capacity of a single memory rank is 128GB (equivalent to the capacity of the small Intel Optane module [18]). The details of the PIM module are listed in Table 3 and explained here.

5.2.1 Interfaces

The operation timing for the R-DDR interface, between the media controller and the memory chips, is taken from [37]. The media controller communicates with the host memory controller using the OpenCAPI protocol [15]. Our simulation implementation of the OpenCAPI protocol considers the added protocol header sizes and computes the timing according to the OpenCAPI bandwidth of 25GB/s [15].

Single PIM Module (8 PIM Modules)							
Capacity	128GB (1TB)	Banks	64 (512)				
Subarrays per PIM controller	64	Crossbars per subarray	4				
Crossbar rows	1024	Crossbar columns	512				
Crossbar read	16 bit	Stateful logic cycle	30 ns [37]				
Crossbar write energy	6.9 pJ/bit [37]	Single stateful logic energy	81.6 fJ/bit [36]				
Crossbar read energy	0.84 pJ/bit [37]	Single PIM controller power	126 uW				
	Evaluatio	n System					
Processor Cores	6 cores, X86, OoO, 3.6GHz	Main memory	64GB DRAM, DDR4-2400, 2 channels				
L1 cache	Private, 64KB, 64B block, 4-way	L2 cache	Shared, 8MB, 64B block, 16-way				
Coherence protocol	MESI	PIM modules	8				

TABLE 5. ATCHIECTURE and System Computation	TABLE	3: Architec	ture and	system	configuration
---	-------	-------------	----------	--------	---------------

5.2.2 Instruction Implementation

The PIM controller supports the operations listed in Table 4, performed on all crossbars connected to the controller in parallel. The *column-transform*, shown in Fig. 6, instruction is implemented by negating the source column into each destination column (Fig. 6b), and negating each bit a second time within its designated column to its designated row (Fig. 6c). The *reduce* instructions, shown in Fig. 7, are implemented as iterations of move and reduce steps, resulting in a binary-tree reduction scheme. In a move step, half of the values are moved to the rows of the other half, while in the reduce step, the two values in each row are reduced according to the relevant operation. Hence, each iteration reduces the number of values by half. The process stops when a single value is left. All other instructions are implemented as iterations on a single-bit operation (Section 3.3).

5.2.3 Crossbar Operations

We take a conservative and practical approach regarding the capabilities of bulk-bitwise logic operations in a crossbar, *i.e.*, the operations sent by a PIM controller to a crossbar. We impose restrictions on bulk-bitwise logic operations to minimize the area overhead of the crossbar peripherals and communication from the PIM controllers to the crossbars. This overhead is due to the additional voltage levels added for each enabled stateful logic operation [21], which must be multiplexed on bitlines and wordlines, and for adding crossbar peripherals. In addition, allowing full flexibility to the combination of operated crossbars rows and columns in a bulk-bitwise logic operation, as in [3], requires controlling each row or column independently. This requires passing a wire for each wordline and bitline (1536 wires in our design) from a PIM controller to its crossbars. As there are many crossbars per PIM controller, the required wiring can reduce the memory density and increase area.

To address this challenge, we enable only a minimal number of operation types and a minimum allowed combination of crossbar rows and columns, while still maintaining complete functionality. First, column-wise operations can be NOR2, NOT, single-column-SET, or single-column-RESET and can only be performed on all the rows in parallel. Exclusion of rows from computing should be done through

Instruction	Cycle Count	Inter. Cells
Equal imm	$imm_0 + 3 \cdot imm_1 + 1$	1
Not Equal imm	$imm_0 + 3 \cdot imm_1 + 3$	2
Less Than imm	$11 \cdot imm_0 + 3 \cdot imm_1 + 4$	5
Greater Than imm	$11 \cdot imm_0 + 3 \cdot imm_1 + 2$	6
Add imm	18n + 3	8
Equal	11n + 3	5
Less Than	16n + 2	6
Set/Reset	n	0
Bitwise NOT	2n	0
Bitwise AND	6n	2
Bitwise OR	4n	1
Addition	18n + 1	6
Multiply	24nm - 19n + 2m - 1	6
Reduce Sum	2254n + 3006	n + 15
Reduce Min/Max	2306n + 200	n+7
Column-Transform	2050	1

TABLE 4: Instruction characteristics. n, m: operands and immediate lengths. imm_0/imm_1 : immediates' number of 0/1 bits. Intermediate cells: cells required for intermediate results in addition to input and output cells per crossbar row.

Crossbar size affects bold-marked operations' coefficients; values in the table reflect a crossbar size of 1024×512 .

software, *e.g.*, using a bit-mask and including it in the computation. Second, row-wise operations can be performed only on a single column at a time and can either be a NOT or single-row-SET [36]. These minimum capabilities are essential for a functionally complete column-wise computation, maintaining high parallelism, and enabling data movement between rows (for column-transform and reduce instructions). These restrictions make certain optimizations impossible, posing an interesting topic for future research. For example, in reduce instructions, the reduce steps are performed on all rows, including rows not participating in a step. The resulting instruction cycle count and intermediate cell requirements (computed as in [36]) are listed in Table 4.

5.3 System Configuration

As the reduction of memory reads is done by the PIM modules and the evaluated workload is memory constrained, the choice of a host (*e.g.*, CPU, GPU, FPGA, near-memory processing) will primarily affect the memory accessing speed. Thus, although the choice of the host will affect performance, it will not change the number of memory reads that are eliminated due to the PIM execution. Hence, for simplicity of evaluation, we chose a multicore with six outof-order X86 cores as the host, having private L1 caches and a shared L2 cache. The host has a 64GB DDR4-2400 DRAM main memory, along with eight OpenCAPI channels³, each with a single PIM module, totaling 1TB⁴ of memory in the PIM modules. Table 3 summarizes the system configuration.

The system was implemented in the gem5 simulation [4], running in full-system mode (running a Linux kernel), and includes the gem5's ruby cache system. A PIM request instruction was added to the host instruction set. Similarly to a write, a PIM request instruction requires two registers as input: address and data.

5.4 Query Compilation and Execution

We built an SQL compiler to abstract PIMDB and its programming model, allowing it to be programmed using a high-level language. The compiler receives the configuration of the database and the SQL statements as input and generates a C++ code for the query execution, where PIM requests are issued using an inline assembly. The C++ code is then compiled using gcc. This compilation process is done offline and is not measured as part of the query execution.

The compiler extracts the query operations required for the PIM modules' execution, *i.e.*, filtering the relations, aggregating if possible, and reading the results. The query execution starts by operating on the small relations residing in the DRAM memory, if required. For relations in the PIM module, the operation execution is divided among four threads, one per core. Each receives part of the huge-pages for each relation; thus, each thread operates on separate pages. For each relation participating in the query, each thread performs several computations and read phases (possibly one of each). A computation phase sends the necessary PIM requests to the thread's pages and the read phase reads the results from those pages. Each computation phase fits into the available crossbar area unoccupied by data, and following it, a read phase clears the area for reuse. To avoid unnecessary cache flushes, the compiler assigns the results of different computation phases to different cache blocks by controlling the addresses of the PIM requests. Memory ordering among PIM requests and reads is enforced using memory fences. Phases of two relations are not interleaved in each thread.

Before query execution is simulated, the required hugepages are first allocated using the operating system and then populated according to [39]. To maintain reasonable memory requirements for the simulation resources, the 1GB huge-pages are emulated using 2MB pages. For the emulation, the number of allocated 2MB pages is the same as the required number of 1GB pages. To emulate the execution time, the required number of reads from each 2MB page in the query execution is matched to the required number of reads on a 1GB page (we made sure each read is retrieved from the PIM modules and not from the caches). Since the latency of PIM operations is independent of the page size, it does not need adjustments. For energy and power considerations, however, the relevant operation energies were counted as operating on 1GB pages, (e.g., number of bulk-bitwise logic operations, number of PIM controllers, etc.).

5.5 Baseline

To show the bulk-bitwise PIM benefits, we take as a baseline the execution of the same query operations evaluated on PIMDB, performed on a database contained in the host main memory (*i.e.*, *in-memory database* [28]). The baseline is executed on the same PIMDB host system configuration, using the gem5 full-system simulation. Comparing our system to the same host system without PIMDB demonstrates the memory access reduction achieved with bulk-bitwise PIM operations. For the baseline, all database relations are column-stored and use the same encoding and compression techniques as in PIMDB. For each query, a C++ code was

^{3.} IBM's Power9 has 16 OpenCAPI channels [35].

^{4.} A single PIM module has a capacity of 128GB, the same capacity as the smallest Intel Optane module [18].



Fig. 8: PIMDB execution time speedup and reduction in LLC misses compared to the baseline (left y-axis of sub-figures). An estimated total query speedup for the filter-only queries is shown using the right y-axis of sub-figure (a).

written to perform the same query operations as produced by our PIMDB compiler. The relevant attributes for each query are brought into the main memory prior to execution (no disk access during execution). The same huge-page emulation is used for memory allocation as in the PIMDB evaluation. Hence, the baseline represents operations on a column-store in-memory database [28] equivalent to the operations executed in PIMDB. The execution of these operations, however, cannot be easily broken to its different components (host compute, PIM compute, memory reads) as the PIMDB execution since the host executes out-of-order and the operation of these components overlap in time.

The baseline execution structure is also similar to that of PIMDB. The small tables (*i.e.*, REGION and NATION) are operated on if necessary, and then the large relations operation is split into four threads where each thread is responsible for a quarter of each relation's records. The threads serially traverse the records of the required relations, filtering according to the required attributes (using nested if-statements) and performing aggregation. The attribute filtering order is chosen offline to minimize memory access to non-required records.

6 RESULTS

6.1 Query Execution Time

To evaluate the benefits of PIMBD, we measure the query execution time, capturing the operations that PIMDB accelerates, and compare it to the query execution time of the baseline. PIMDB execution time includes the operations performed by the PIM modules, the operations to support the PIM operations (such as operations on small DRAM relations, spawning threads, etc.), and the results read from the PIM module. Hence, for the full queries, the entire query execution is captured. For the filter-only queries, however, only the filter operation is measured (including reading the results), as the rest of the query execution does not involve the PIM modules or their database copy and is considered out-of-scope for this work. Our evaluation does not compare PIMDB to other bulk-bitwise PIM technologies and architectures [12,14,17,23,33,37,40] as we are interested in evaluating bulk-bitwise PIM, not comparing the various technologies. We discuss this issue further in Section 7.

Fig. 8 shows the execution time and the LLC miss ratios between PIMDB and the baseline on the accelerated operations. The filter-only and full queries achieve a speedup



Fig. 9: PIMDB execution time breakdown. For the filter-only queries, the PIM ops portion is too small to be visible.

	Filtor	Arith	Col.	Inter.		Filtor	Arith	Col.	Inter.
	rinter	Anu.	trans.	cells		rinter	Anu.	trans.	cells
Q2	619	0	2050	80	Q3	97	0	2050	32
Q4	216	0	2050	49	Q5	220	0	2050	33
Q7	200	0	2050	30	Q8	200	0	2050	31
Q10	220	0	2050	33	Q11	22	0	2050	30
Q12	678	0	2050	39	Q14	252	0	2050	39
Q15	228	0	2050	39	Q16	271	0	2050	48
Q17	37	0	2050	32	Q19	606	0	2050	64
Q20	220	0	2050	39	Q21	216	0	2050	30
	Filter	Arith.	Agg. col/row	Inter. cells		Filter	Arith.	Agg. col/row	Inter. cells
Q1	190	20498	$\begin{array}{c} 2.2 \times \\ 10^{5} / \\ 2 \times 10^{6} \end{array}$	313	Q6	346	3390	$\begin{array}{c} 9.9 \times \\ 10^{3} / \\ 9.4 \times 10^{4} \end{array}$	189
Q22 _sub	453	106	${6.2 \times 10^3 / \over 4.9 \times 10^4}$	122					

TABLE 5: The number of PIM bulk-bitwise logic cycles by type and intermediate results' cells used on a single crossbar. Filter-only queries do not use aggregation; full queries do not use column-transform. The Agg. col/row column shows the cycles of the aggregation column and row operations,

respectively.

of $0.82 \times -14.7 \times$ and $62 \times -787 \times$, respectively. The speedup difference between full and filter-only queries is due to the data reduction advantage of reduce versus filter operations (Section 4.2). Q11 is the only query where using PIMDB results in a slowdown. Q11 has only a small filter operation on a small relation, achieving little read reduction and making the read latency of the results (from DRAM in the baseline and from PIM memory in PIMDB) more prominent.

The reduction in memory accesses is perceived through the LLC misses. The LLC miss and execution time ratios, however, do not correlate entirely. The LLC misses are for all the memory accesses (not just the database) and code execution and memory access patterns for PIMDB and baseline differ substantially. Given that PIMDB accelerates only the filter operations on the filter-only queries, to provide a full performance perspective, Fig. 8(a) shows the total query speedup for the filter-only queries using data from [20]. As our work aims to enable and investigate the capabilities of bulk-bitwise PIM, we focus on the parts accelerated by PIMDB and leave advanced algorithm implementation (*e.g.*, JOIN, UPDATE, GROUPBY [28]) and mapping of filterheavy databases (*e.g.*, key-value store [27]) for future work.

For calibration, we estimate how bulk-bitwise PIM compares with real systems. We compared PIMDB with the two top-ranking state-of-the-art systems for TPC-H with





SF = 1000 [9, 10, 39]. These systems have a higher core count than our base system. The published results, however, are for full queries only, allowing us to compare only Q1 and Q6. On Q1, we attain a speedup of $9.3 \times$ and $8.2 \times$, while on Q6, we achieve a speedup of $19.6 \times$ and $11.6 \times$, over [9] and [10], respectively. Note that this comparison is between simulation and real hardware, giving only a rough estimation of the PIMDB system benefits. Also we point out that taking either [9] or [10] as the PIMDB host system may further improve the performance of PIMDB, as it will likely benefit from the higher core count of these systems.

Fig. 9 shows the breakdown of the PIM-executed queries into PIM operations, reading data from the PIM modules, and other operations (e.g., spawning threads, operating on DRAM relations). The breakdown shows that for the filteronly queries, the read time from the PIM modules dominates, and comprises over 99% of execution time for all queries, with the exception of Q2, Q11, Q16, and Q17 due to operation on smaller relations (*i.e.*, not operating on the LINEITEM or ORDERS relations). For the full queries Q1 and Q6, the read time is also the bottleneck, but more moderately, 70% and 55% of execution time, respectively. For the full query Q22_sub, due to the small read size resulting from the aggregation read reduction and the operation on the smaller CUSTOMER relation, the read time is no longer the bottleneck. These results imply that although the data transfer bottleneck was successfully relaxed, for most cases, it still remains the main component of the query execution time.

To further investigate the PIM operations, Table 5 shows the breakdown of bulk-bitwise PIM logic cycles by type and the number of cells used to hold intermediate results. The dominant operations are the column-transform operations in the filter-only queries and the aggregate operations in the full queries. Although these operations may consume many cycles, they are amortized over all the records in many huge pages operating concurrently, where each such page (1GB) contains 16M records. Both column-transform and aggregate operations mostly comprise row-wise operations performing serial bit-by-bit data movement between crossbar rows, showing that even the bulk-bitwise logic latency is chiefly dedicated to data movements. This result is due to the crossbar row-wise operation restriction, requiring that only one column be active at a time. We analyze the case where row-wise operations are allowed to operate on multiple columns in any combination (only increasing the rowwise data movement bandwidth). Our analysis shows that the full queries' bulk-bitwise logic latency can be reduced by 80%-86%, and the execution time improved by 25% for Q1



Fig. 11: PIMDB energy saving over baseline.



Fig. 12: PIMDB energy breakdown.

and Q6 and by 39% for Q22_sub, turning the bulk-bitwise logic latency from 29%–48% into 5.4%–15% of execution time.

Our results show that despite using a non-optimum loop-based n-bit operation design and restricted crossbar operations, the bulk-bitwise PIM latency used for computation, as opposed to data movement between crossbar rows, does not have a major impact on query execution time (less than 1% for filter-only queries and less than 6% for full queries on large relations). These results, although dependent on data set size, signal that bulk-bitwise PIM computation latency can be traded-off with other system aspects, *e.g.*, complexity, area, and power.

6.2 Area

To estimate the area of the PIM controller, we implemented it in Verilog, synthesized it, and evaluated its area using Cadence Innovus and Synopsys Design Compiler with TSMC 28nm technology. For the PIM module chip area, NVSim [11] was modified to include a single PIM controller per 64 subarrays. Fig. 10 shows a PIM module chip area breakdown, with the PIM controller consuming only 0.17% of chip area.

6.3 Power and Energy

To evaluate the system's energy and power consumption, we evaluate and sum the energies of the DRAM main memory, the host, and the PIM modules. The DRAM energy is evaluated by the integrated gem5 DRAM power model [6]. The McPAT [22] tool is used to assess the host power consumption.

The PIM modules' energy is taken as the sum of the PIM controller, bulk-bitwise (stateful) logic, read and write



Fig. 14: Peak and average power demand for PIM module chip.

operations, and chip IO energies according to the simulation behavior. The PIM controller energy is evaluated using the Synopsys Design Compiler as explained in Section 6.2. The energy per stateful logic operation was taken from [36], and the energies per read and write operations were taken from [37]. Table 3 lists the different energy values. For chip IO energy costs, the gem5 DRAM energy model was used. The power of a PIM module is sampled as the average power in a 100ns time window.

The energy ratio between the baseline (host and DRAM) and PIMDB (host, DRAM, and PIM modules) is shown in Fig. 11. Overall, an energy reduction of $0.88 \times -15.3 \times$ is achieved for filter-only queries, and a reduction of $1.14 \times$ and $15.8\times$ is achieved for the full queries. Q1 has a relatively small energy reduction as it performs many reduction operations, completely offsetting the memory traffic energy saving. The energy breakdowns of the PIMDB system and the PIM module are shown in Figs. 12 and 13, respectively. For the filter-only queries, most of the energy is consumed by the DRAM module (standby energy). While in the PIM modules, most of the energy is dedicated to bulk-bitwise (stateful) logic operations. For the full queries, the PIM modules dominate the energy consumption. For Q22_sub, the PIM energy component is more moderate than in Q1 and Q6 since Q22_sub operates on a smaller relation (CUSTOMER vs. LINEITEM). At the PIM module, more than 99% of the energy is spent on bulk-bitwise logic for the full queries. This is mainly due to the substantial reduction in read operations and longer PIM operations of full queries. Our results indicate that the main energy-consuming component is now the bulk-bitwise logic. In all queries, the host uses relatively little energy since, using PIM, the host is mainly involved in producing the memory operations (read, write, and PIM request), which require light arithmetic. As our



Fig. 15: Required endurance for a ten-year operation with a 100% duty cycle.

	Filter	Arith.	Col. trans.	Agg. col/row		Filter	Arith.	Col. trans.	Agg. row/col
Q2	91%	0	9%	0/0	Q3	60%	0	40%	0/0
Q4	77%	0	23%	0/0	Q5	77%	0	23%	0/0
Q7	76%	0	24%	0/0	Q8	76%	0	24%	0/0
Q10	77%	0	23%	0/0	Q11	26%	0	74%	0/0
Q12	91%	0	9%	0/0	Q14	80%	0	20%	0/0
Q15	78%	0	22%	0/0	Q16	81%	0	19%	0/0
Q17	37%	0	63%	0/0	Q19	90%	0	10%	0/0
Q20	77%	0	23%	0/0	Q21	77%	0	23%	0/0
Q1	1%>	8%	0	85%/ 7%	Q6	2%	23%	0	68%/ 6%
Q22_sub	6%	1%	0	87%/ 6%					



workload was chosen to be memory-bound, this behavior is expected.

Fig. 14 shows the peak and average chip power demand measured by simulations, as well as theoretical peak chip power demand. The theoretical peak power demand is computed as the power required to perform a stateful logic operation across all accessed pages, for the PIM module with the maximum number of pages. The theoretical peak power demand shows that when all pages accessed by a query are operating in parallel, the power demand can reach up to 330W per chip, which is high but reasonable for such an accelerator [26]. The peak and average measured power, however, is substantially lower, up to 125W and 10W, respectively, since pages are not synchronized in their operations and stateful logic operates only in part of the query execution. If we look at a bulk-bitwise stateful logic operation across all crossbars in a PIM module chip, which no query in the evaluation performs, we see that the chip power demand can reach 730W. These results indicate that power-aware scheduling for the PIM operations is required.

6.4 Endurance

As device endurance is a main challenge for memristive technologies [44], the required endurance for the proposed design and workloads are evaluated. For this evaluation, we assume that through the life of the PIM module, the computation at a crossbar row is uniformly distributed across all cells of that row. This is a reasonable assumption since the locations of all values in a crossbar row are controlled by software and can be shifted periodically. Furthermore, previously proposed techniques [32] can be applied to achieve a uniform distribution. Under this assumption, the maximum number of operations on a cell is calculated as follows: the maximum number of operations experienced by a single crossbar row is extracted per query and divided by the number of crossbar row cells. Note that not all crossbar rows experience the same number of operations. Different relations experience different operations and the column-transform and reduce operations do not operate on all crossbar rows uniformly. Fig. 15 shows the maximum number of operations applied per cell, for each query, where our measured execution (Section 6.1) is performed backto-back, i.e., with a 100% duty cycle, for ten years. Based on previously reported RRAM endurance numbers of 10^{12} cycles [44], the results show that the PIMDB lifetime can exceed ten years, with the exception of Q22_sub due to its operation on a relatively small relation, resulting in frequent writes to the same memory cells.

Table 6 shows the breakdown of operations contributing to the maximum number of operations per cell. For the filteronly queries, the filter PIM operations are the dominant contributors (as opposed to the PIM operation latency). This is because the column-transform operation mainly performs bit-by-bit movements between rows, resulting in few operations per row. For the full queries, the reduce operations are still the dominant factor but with the columnwise operations rather than the row-wise operations. These results imply that endurance, rather than latency, may be more important for arithmetic design with stateful logic.

Overall, we see that there are two reasons for the low number of operations applied per cell. First, most of the PIMDB execution time is spent on read operations and not PIM operations, as shown in Fig. 9. Second, the most timeconsuming PIM operations have a low number of operations per cell. These operations are the column-transform and row-wise operations for aggregation, for the filter-only and full queries, respectively.

7 RELATED WORK

Several bulk-bitwise PIM technologies and architectures were previously proposed [12, 14, 23, 33, 37, 40]. These works showed how to perform the bulk-bitwise PIM operations, but did not delve into full system performance. Furthermore, these works only demonstrated bulk-bitwise PIM on micro-benchmarks (including database micro-benchmarks), missing the implementation details and operation mix of a standard benchmark. All these works only show bulkbitwise operations on either crossbar bitlines or wordlines, enabling only the query's filter, not aggregation, to be performed with solely bulk-bitwise PIM. As such, our work has a more comprehensive and more fundamental evaluation of bulk-bitwise PIM than previous works in general, and on database applications specifically. As the evaluation of the filter-only queries shows, the read latency, and not the PIM latency, dominates the query latency. Hence, other technologies will have similar query execution behavior as PIMDB, with execution time differences depending on the difference in read latency from the PIM memory. The work in [40] also supports direct inter-crossbar communication in addition to bulk-bitwise PIM. This communication enables aggregation

without the host, potentially increasing performance further with additional hardware and control costs.

An RRAM-based CAM architecture for database acceleration was presented in [17]. The design included analog and stateful logic techniques on the same CAM crossbars, making it hard to distinguish the contribution of each technique separately and involving further hardware and control overheads. While PIMDB and [17] support similar basic database primitives, they significantly differ in hardware, operation implementations, relation layout, and host–PIM interaction. Moreover, [17] did not include a programming model and their evaluation consists of only a few custom micro-benchmarks, rather than a standard benchmark.

To the best of our knowledge, none of the previous works for bulk-bitwise operations, where the PIM module is part of the main memory, presents a full programming model [14,16,17,37,40]. [45] presented a high-level programming language interface for bulk-bitwise PIM. [14] and [23] presented a programming model that avoids the PIM memory management and addressing issues, delegating them to the operating system (OS) to handle. These solutions are complementary to our programming model as they are on different abstraction levels. [14] also suggested a host instruction set extension to issue PIM instructions similar to our PIM requests. [2] and [25] proposed instruction offloading schemes for near-memory architectures. Near-memory uses distinct components for processing elements and memory, having different characteristics than bulk-bitwise PIM. [2] proposed a host instruction set extension to indicate PIMenabled instructions, similar to the PIM request proposed in our work, while [25] suggested intercepting the host processor atomic instructions for PIM execution.

In-storage [42, 43] and near-memory [20] are other processing techniques that are similar to bulk-bitwise PIM. Both in-storage and near-memory processing try to reduce data movement by bringing the computation closer to where the data reside. Both techniques, however, use standard CMOS technology (e.g., cores, dedicated accelerators), rather than the data storage medium, as the processing elements. Hence, although in-storage and near-memory processing goals are similar to those of bulk-bitwise PIM, the control and operation are different. Additionally, commercial instorage processing [43] offers to accelerate filtering and aggregation for databases, which are the same primitives inherently available by bulk-bitwise PIM. As mentioned, the approach to accelerating these primitives differs from bulkbitwise PIM, producing different characteristics and tradeoffs. Nevertheless, in-storage and near-memory processing are not competing with bulk-bitwise PIM, but complementing it. As in-storage and near-memory use logic outside of the memory array, they can be used as hosts for bulk-bitwise PIM, benefiting from the data movement reduction achieved by bulk-bitwise PIM.

8 CONCLUSION

In this paper, we investigated bulk-bitwise PIM by proposing PIMDB, a bulk-bitwise PIM architecture based on memristive stateful logic for accelerating analytical processing operations on relational databases, including a host programming model. By focusing on bulk-bitwise PIM for a real-world application and introducing a programming model, the properties of this PIM method were demonstrated. Insights about application mapping, execution and energy breakdowns, PIM logic latency, endurance, and PIM module power were offered. The design reduced the data transfer in the system by supporting filtering and aggregation operations within the memory and included mapping and logic techniques to support these operations. The programming model, as well as other aspects of our work, can be applied to other bulk-bitwise PIM technologies and architectures. PIMDB accelerates TPC-H filter operations by $1.6 \times -18 \times$ and full queries by $56 \times -608 \times$, while achieving a $1.7 \times -18.6 \times$ and $0.81 \times -12 \times$ energy reduction for these benchmarks, respectively.

REFERENCES

- [1] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 671–682. [Online]. Available: https://doi.org/10.1145/1142473.1142548
- [2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), 2015, pp. 336–348.
- [3] 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.
- [4] 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
- [5] 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.
- [6] K. Chandrasekar, B. Akesson, and K. Goossens, "Improved power modeling of ddr sdrams," in 2011 14th Euromicro Conference on Digital System Design, 2011, pp. 99–108.
 [7] D. Chen, Z. Li, T. Xiong, Z. Liu, J. Yang, S. Yin, S. Wei, and L. Liu,
- [7] D. Chen, Z. Li, T. Xiong, Z. Liu, J. Yang, S. Yin, S. Wei, and L. Liu, "Catcam: Constant-time alteration ternary cam with scalable inmemory architecture," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 342–355.
- [8] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin, "Automatic sql tuning in oracle 10g," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, p. 1098–1109.
- [9] "TPC Benchmark H Full Disclosure Report For Dell Technologies PowerEdge MX740c Modular Server While Using Microsoft SQL Server 2019 Enterprise Edition and Red Hat® Enterprise Linux® 8.0," http://tpc.org/3369, Dell Technologies, March 2021.
- [10] "TPC Benchmark H Full Disclosure Report For Dell Technologies PowerEdgee R7515 Server While Using Microsoft SQL Server 2019 Enterprise Edition 64 bit and Red Hat® Enterprise Linux® 8.0," https://tpc.org/3374, Dell Technologies, April 2021.
- https://tpc.org/3374, Dell Technologies, April 2021.
 [11] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "NVSim: A Circuit-Level Performance, Energy, and Area Model for Emerging Nonvolatile Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 7, pp. 994–1007, 2012.
 [12] I. Giannopoulos, A. Singh, M. Le Gallo, V. P. Jonnalagadda,
- [12] I. Giannopoulos, A. Singh, M. Le Gallo, V. P. Jonnalagadda, S. Hamdioui, and A. Sebastian, "In-memory database query," *Advanced Intelligent Systems*, vol. 2, no. 12, p. 2000141, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10. 1002/aisy.202000141
- [13] S. Gupta, M. Imani, B. Khaleghi, V. Kumar, and T. Rosing, "Rapid: A reram processing in-memory architecture for dna sequence alignment," in 2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED), 2019, pp. 1–6.

- [14] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. a. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "SIMDRAM: A Framework for Bit-Serial SIMD Processing Using DRAM," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 329–345. [Online]. Available: https://doi.org/10.1145/3445814.3446749
- [15] OpenCAPI 4.0 Transaction Layer Specification, online: https://opencapi.org/, IBM, October 2017, version 1.2.
- [16] 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.
- [17] M. Imani, S. Gupta, S. Sharma, and T. S. Rosing, "Nvquery: Efficient query processing in nonvolatile memory," *IEEE Transactions* on Computer-Aided Design of Integrated Circuits and Systems, vol. 38, no. 4, pp. 628–639, 2019.
- [18] "Intel® Optane[™] Persistent Memory," https://www.intel. com/content/www/us/en/products/memory-storage/ optane-dc-persistent-memory.html, Intel, accessed: 2021.
- [19] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
- [20] T. R. Kepe, E. C. de Almeida, and M. A. Z. Alves, "Database processing-in-memory: An experimental study," *Proc. VLDB Endow.*, vol. 13, no. 3, p. 334–347, November 2019. [Online]. Available: https://doi.org/10.14778/3368289.3368298
- [21] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristoraided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.
- [22] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009, pp. 469–480.
- [23] 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.
- [24] Z. Lu, M. T. Arafin, and G. Qu, RIME: A Scalable and Energy-Efficient Processing-In-Memory Architecture for Floating-Point Operations. New York, NY, USA: Association for Computing Machinery, 2021, p. 120–125. [Online]. Available: https://doi.org/10.1145/3394885.3431524
- [25] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," in 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 457–468.
- [26] "Whitepaper NVIDIA Tesla P100," https://images.nvidia.com/ content/pdf/tesla/whitepaper/pascal-architecture-whitepaper. pdf, NVIDIA, 2016.
- [27] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann, "Fast scans on key-value stores," *Proc. VLDB Endow.*, vol. 10, no. 11, p. 1526–1537, aug 2017. [Online]. Available: https://doi.org/10.14778/3137628.3137659
- [28] H. Plattner, A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases. Springer Publishing Company, Incorporated, 2013.
- [29] T. Pohanka and V. Pechanec, "Evaluation of replication mechanisms on selected database systems," *ISPRS International Journal of Geo-Information*, vol. 9, no. 4, 2020. [Online]. Available: https://www.mdpi.com/2220-9964/9/4/249
- [30] 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.
- [31] J. Reuben, R. Ben-Hur, N. Wald, N. Talati, A. H. Ali, P. 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.
- [32] M. M. Sabry Aly, T. F. Wu, A. Bartolo, Y. H. Malviya, W. Hwang, G. Hills, I. Markov, M. Wootters, M. M. Shulaker, H. S. Philip Wong, and S. Mitra, "The n3xt approach to energy-efficient

abundant-data computing," *Proceedings of the IEEE*, vol. 107, no. 1, pp. 19–48, 2019.
[33] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim,

- [33] 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.
- [34] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 14–26.
- [35] J. Stuecheli, S. Willenborg, and W. Starke, "IBM's Next Generation POWER Processor," in 2019 IEEE Hot Chips 31 Symposium (HCS), 2019, pp. 1–19.
- [36] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic design within memristive memories using memristor-aided logic (magic)," *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 635–650, 2016.
- [37] 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.
- [38] V. Tenace, R. G. Rizzo, D. Bhattacharjee, A. Chattopadhyay, and A. Calimera, "Said: A supergate-aided logic synthesis flow for memristive crossbars," in 2019 Design, Automation Test in Europe Conference Exhibition (DATE), 2019, pp. 372–377.
- [39] "TPC benchmark H standard specification revision 3.0.0," http: //tpc.org/tpch/, Transaction Processing Performance Council, February 2021.
- [40] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "Racer: Bitpipelined 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
- [41] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015, pp. 476– 488.
- [42] S. Xu, T. Bourgeat, T. Huang, H. Kim, S. Lee, and A. Arvind, "Aquoman: An analytic-query offloading machine," in 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 386–399.
- [43] Y. Yang, M. Youill, M. Woicik, Y. Liu, X. Yu, M. Serafini, A. Aboulnaga, and M. Stonebraker, "Flexpushdowndb: Hybrid pushdown and caching in a cloud dbms," *Proc. VLDB Endow.*, vol. 14, no. 11, p. 2101–2113, jul 2021. [Online]. Available: https://doi.org/10.14778/3476249.3476265
- [44] F. Zahoor, Z. Azni, Z. Tun, and F. A. Khanday, "Resistive Random Access Memory (RRAM): an Overview of Materials, Switching Mechanism, Performance, Multilevel Cell (mlc) Storage, Modeling, and Applications," *Nanoscale Research Letters*, vol. 15, April 2020. [Online]. Available: https://doi.org/10.1186/ s11671-020-03299-9
- [45] M. Zhou, M. Imani, Y. Kim, S. Gupta, and T. Rosing, "Dp-sim: A full-stack simulation infrastructure for digital processing inmemory architectures," in 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC), 2021, pp. 639–644.



Ronny Ronen (Fellow, IEEE) received his B.Sc. and M.Sc. degrees in computer science from the Technion, Haifa, Israel, in 1978 and 1979, respectively. He is a Senior Researcher at the Andrew and Erna Viterbi Faculty of Electrical & Computer Engineering at the Technion. He was with Intel Corporation from 1980 to 2017 in various technical and managerial positions. In his last role there, he led the Intel Collaborative Research Institute for Computational Intelligence. He was the Director of Microarchitecture

Research and a Senior Staff Computer Architect at the Intel Haifa Development Center until 2011. He led the development of several system software products and tools, including the Intel Pentium processor performance simulator and several compiler efforts. In these roles, he led/was involved in the initial definition and pathfinding of major leading-edge Intel processors. He holds over 80 issued patents and has published over 35 papers.



Benny Kimelfeld is a Professor at the Computer Science Faculty at Technion, Israel, and the head and founder of the Technion Data and Knowledge (TD&K) Laboratory. After receiving his Ph.D. from The Hebrew University of Jerusalem, and before joining Technion, he has been a Research Staff Member at IBM Research Almaden, then a Computer Scientist at LogicBlox. Benny's research spans a spectrum of both foundational and systems aspects of data management, such as probabilistic and incon-

sistent databases, information retrieval over structured data, and infrastructure for text analytics. Benny was the program-committee chair of the 2018 International Conference on Database Theory (ICDT), a co-chair of the 2016 Web and Databases Workshop (WebDB), and a co-chair of the 2014 SIGMOD/PODS workshop on Big Uncertain Data (BUDA). He currently serves as an associate editor in the Journal of Computer and System Sciences (JCSS).



Shahar Kvatinsky (Senior Member, IEEE) is an Associate Professor at the Viterbi Faculty of Electrical and Computer Engineering, Technion. Shahar received the B.Sc. degree in Computer Engineering and Applied Physics and an MBA degree in 2009 and 2010, respectively, both from the Hebrew University of Jerusalem, and the Ph.D. degree in Electrical Engineering from the Technion in 2014. From 2006 to 2009, he worked as a circuit designer at Intel. From 2014 to 2015, he was a post-doctoral research fellow at Stan-

ford University. Kvatinsky is a member of the Israel Young Academy. He is the head of the Architecture and Circuits Research Center at the Technion and chair of the IEEE Circuits and Systems in Israel. Kvatinsky has been the recipient of numerous awards including: 2020 MDPI Electronics Young Investigator Award, 2019 Wolf Foundation's Krill Prize, 2015 IEEE Guillemin-Cauer Best Paper Award, ERC starting grant, and the 2017 Pazy Memorial Award.



Ben Perach is a Ph.D. candidate at the Andrew and Erna Viterbi Faculty of Electrical and Computer Engineering, Technion – Israel Institute of Technology. Ben received his B.Sc. degree in mathematics from The Hebrew University of Jerusalem in 2010 and his M.Sc. degree in electrical engineering from Tel Aviv University in 2017. Ben's current research interests include computer architecture with a focus on processor design, and also field-programmable gate arrays, security, and data networks.