PIM-tree: A Skew-resistant Index for Processing-in-Memory

Hongbo Kang
khb20@mails.tsinghua.edu.cn
Tsinghua University

Yiwei Zhao
yiweiz3@andrew.cmu.edu
Carnegie Mellon University

Guy E. Blelloch
guyb@cs.cmu.edu
Carnegie Mellon University

Laxman Dhulipala
laxman@umd.edu
University of Maryland

Yan Gu
ygu@cs.ucr.edu
UC Riverside

Charles McGuffey
cmcguffey@reed.edu
Reed College

Phillip B. Gibbons
gibbons@cs.cmu.edu
Carnegie Mellon University

ABSTRACT

The performance of today's in-memory indexes is bottlenecked by the memory latency/bandwidth wall. Processing-in-memory (PIM) is an emerging approach that potentially mitigates this bottleneck, by enabling low-latency memory access whose aggregate memory bandwidth scales with the number of PIM nodes. There is an inherent tension, however, between minimizing inter-node communication and achieving load balance in PIM systems, in the presence of workload skew. This paper presents *PIM-tree*, an ordered index for PIM systems that achieves both low communication and high load balance, regardless of the degree of skew in the data and the queries. Our skew-resistant index is based on a novel division of labor between the multi-core host CPU and the PIM nodes, which leverages the strengths of each. We introduce *push-pull search*, which dynamically decides whether to push queries to a PIM-tree node (CPU → PIM-node) or pull the node's keys back to the CPU (PIM-node → CPU) based on workload skew. Combined with other PIM-friendly optimizations (*shadow subtrees* and *chunked skip lists*), our PIM-tree provides high-throughput, (guaranteed) low communication, and (guaranteed) high load balance, for batches of point queries, updates, and range scans.

We implement the PIM-tree structure, in addition to prior proposed PIM indexes, on the latest PIM system from UPMEM, with 32 CPU cores and 2048 PIM nodes. On workloads with 500 million keys and batches of 1 million queries, the throughput using PIM-trees is up to 69.7× and 59.1× higher than the two best prior PIM-based methods. As far as we know these are the first implementations of an ordered index on a real PIM system.

1 INTRODUCTION

The mismatch between CPU speed and memory speed (a.k.a. the "memory wall") makes memory accesses the dominant cost in today's data-intensive applications. Traditional architectures use multi-level caches to reduce data movement between CPUs and memory, but if the application exhibits limited locality, most data accesses are still serviced by the memory. This excessive data movement incurs significant energy cost, and performance is bottlenecked by high memory latency and/or limited memory bandwidth.

Processing-in-Memory (PIM) [30, 39], a.k.a. near-data-processing, is emerging as a key technique for reducing costly data movement. By integrating computing resources in memory modules, PIM enables data-intensive computation to be executed in the PIM-enabled memory modules, rather than moving all data to the CPU to process. Recent studies have shown that, for programs with high data-intensity and low cache-locality, PIM provides significant advantages in increasing performance and reducing power consumption by reducing data movement [17, 18]. Although proposals for processing-in-memory/processing-in-storage date back to at least 1970 [37], including forays by the database community in active disks [33], PIM is emerging today as a key technology thanks to advances in 3D-stacked memories [23, 27] and the recent availability of commercial PIM system prototypes [39]. Typical applications exploiting state-of-the-art PIM architectures include neural networks [3, 26, 28, 41], graph processing [1, 22, 31, 32, 36, 44, 45], databases [7, 8], sparse matrix multiplication [16, 42], genome analysis [2, 43] and security [4, 20, 21].

PIM systems are typically organized as a host (multicore) CPU that pushes compute tasks to a set of P *PIM modules* (compute-enhanced memory modules), and collects the results. Thus, cost is incurred for moving both task descriptors and data—the sum of these costs is the *communication cost* between the CPU and the PIM modules. The host CPU can be any commodity multicore processor, and is typically more powerful than the wimpy CPUs within the PIM modules. Thus, an interesting feature of a PIM system is the opportunity to use both sets of resources (CPU side and PIM side) in service of applications.

In this paper, we focus on designing a PIM-friendly ordered index for in-memory data. Ordered indexes (e.g., B-trees [13]) are one of the backbone components of databases/data stores, supporting efficient search queries, range scans, insertions, and deletions. Prior works targeting PIM [12, 29] proposed ordered indexes based on *range partitioning*: the key space is partitioned into P subranges of equal numbers of keys, and each of the P PIM modules stores one subrange. Each PIM module maintains a local index over the keys in its subrange, and the host CPU maintains the top portion of the index down to the P roots of the local indexes. This approach works well for data and queries with uniformly random keys—the setting studied by these works—but it suffers from load imbalance under data or query skew. In more realistic workloads, batches of queries/updates may concentrate on the data in a small subset of the partitions, overwhelming those PIM modules, while the rest are idle. In the extreme, only one PIM module is active processing queries and the rest are idle, fully serializing an entire batch of queries on a single (wimpy) processor. The approach also suffers the cost of all data movements required to keep partitions (roughly) balanced in size. In a recent paper [24], we designed a PIM-friendly skip list that asymptotically overcomes this load-imbalance problem (details in Section 2.4), but the solution is not practical (up to 69.7× slower than the ordered index we present in this paper).

To address the above challenges with query and data skew, we present the *PIM-tree*, a practical ordered index for PIM that achieves both low communication cost and high load balance, regardless of the degree of skew in data and queries. Our skew-resistant index is based on a novel division of labor between the host CPU and PIM nodes, which leverages the strengths of each. Moreover, it combines aspects of both a B+-tree and a skip list to achieve its goals. We focus on achieving *high-throughput*, processing *batches* of queries at a time in a bulk-synchronous fashion. The PIM-tree supports a wide range of batch-parallel operations, including point queries (Get, Predecessor), updates (Insert, Delete), and range Scan.

We introduce *push-pull search*, which dynamically decides, based on workload skew, whether (i) to *push* queries from the CPU to a PIM-tree node residing on a PIM module or (ii) to *pull* the tree-node's keys back to the CPU. Combined with other PIM-friendly optimizations—*shadow subtrees* and *chunked skip lists*—our PIM-tree provides high-throughput, (guaranteed) low communication costs, and (guaranteed) high load balance, for batches of point queries, updates, and range scans. For example, each point query and update is answered using only $O(\log_B \log_B P)$ expected communication cost, where B is the expected fanout of a PIM-tree node and P is the number of PIM modules, independent of the number of keys n in the data structure, or the data skew. Note that it would take over 10^{19} PIM modules for $\log_B \log_B P$ to exceed 1, under our selection of $B = 16$; hence, the communication cost is constant in practice.

We implement the PIM-tree on the latest PIM system from UPMEM [39], with 32 CPU cores and 2048 PIM modules. Our codes can be found at https://github.com/cmuparlay/PIM-tree. We choose four state-of-the-art ordered indexes as competitors, including two PIM-friendly approaches [24, 29] implemented by ourselves, and two traditional approaches[9, 11] implemented in the SetBench benchmark suite [5]. On workloads with 500 million keys and batches of 1 million queries, the PIM-tree achieves (i) up to 59.1× higher throughput than the range-partitioned solution [29], (ii) up to 69.7× higher throughput than the prior skew-resistant solution [24], and (iii) comparable throughput in all cases regardless of skew and as low as 0.3× less communication than two state-of-the-art non-PIM indexes [9, 11].

The main contributions of the paper are:

- We design the PIM-tree, a high-throughput skew-resistant PIM data structure that efficiently supports a wide range of batch-parallel point queries, updates, and scans, even under highly-skewed workloads. It causes nearly constant data movement (communication cost) for a point query or update, and linear data movement for scans. Key ideas include push-pull search and shadow subtrees.
- We implement and evaluate the PIM-tree on a commercial PIM system prototype, demonstrating significant performance improvements at modest skew, and performance gains that increase linearly with larger skew. As far as we know these are the first implementations of an ordered index on a real PIM system.

2 BACKGROUND

2.1 PIM System Architecture and Model

The Processing-in-Memory Model. We use the Processing-in-Memory Model (PIM Model) (first described in [24]) as an abstraction of generic PIM systems. It is comprised of a host CPU front end (*CPU side*) and a collection of *P PIM modules* (*PIM side*). The CPU side is a standard multicore processor, with an on-chip cache of M words. Each PIM module is comprised of a DRAM memory bank (*local PIM memory*) with an on-bank processor (*PIM processor*) and a local memory of $\Theta(n/P)$ words (where n denotes the problem size). The PIM processor is simple but general-purpose (e.g., a single in-order core capable of running C code). The CPU host can send code to the PIM modules, launch the code, and detect when the code completes. It can also send data to and receive data from PIM memory. The model assumes there is no direct PIM-to-PIM communication, although we could take advantage of such communication on PIM systems supporting it.

As the PIM model combines a shared-memory side (CPU and its cache) and a distributed side (PIM modules), algorithms are analyzed using both shared-memory metrics (work, depth) and distributed metrics (local work, communication time). On the CPU side, the model accounts for ***CPU work*** (total work summed over all cores) and ***CPU depth*** (all work on the critical path). On the PIM side, the model accounts for ***PIM time***, which is the maximum local work on any one PIM core, and ***IO time***, which is the maximum number of messages to/from any one PIM module.[1] Programs execute in bulk-synchronous rounds [40], and the overall complexity metrics of an algorithm is the sum of the complexity metrics of each round. We focus on IO time and IO rounds in this paper.

Programming Interface. For concreteness, we assume the following programming interface for our generic PIM system, although our techniques would also work with other interfaces. Programs consist of two parts: a *host program* executed on the host CPU, and a *PIM program* executed on PIM modules. The host program has additional functions (discussed below) to communicate with the PIM side, including functions to invoke PIM programs on PIM modules and to transfer data to/from PIM modules. The PIM program is a traditional program (no additional functions) that is invoked in all PIM processors when launched by the host program. It executes using the module's local memory, with no visibility into the CPU side or other PIM modules. The specific functions are (named MPI-style [19]):

- **PIM_Load(**PIM_Program_Binary**)**: loads a binary file to the PIM modules.
- **PIM_Launch()**: launches the loaded PIM program on all PIMs.
- **PIM_Status()**: checks whether the PIM program has finished on all PIMs.
- **PIM_Broadcast(**src, length, PIM_Local_Address**)**: copies a fixed length buffer to the same local memory address in each PIM module.
- **PIM_Scatter(**srcs[], length[], PIM_Local_Address**)**: similar to PIM_Broadcast, but with a distinct buffer and length for each PIM module.
- **PIM_Gather(**dsts[], length[], PIM_Local_Address**)**: the reverse of PIM_Scatter, reading into the buffer array dsts[].

Algorithm 1. Batch-parallel Execution(O: batch of operations)

```
Repeat the following steps until done processing $O$:
```

[1]There is no separate accounting needed for messages to/from the CPU side because any well-balanced system should provide bandwidth out of the host CPU that matches bandwidth into the PIM modules (and vice-versa).

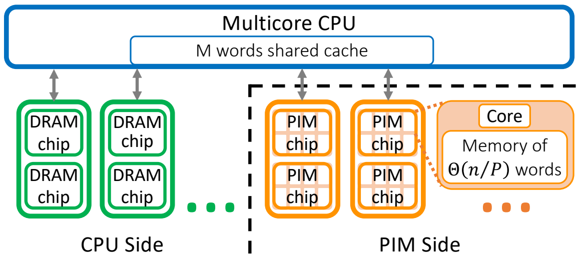


Figure 1: The architecture for the UPMEM PIM system, a specific example of our generic PIM system architecture. PIM modules are packed into memory DIMMs connected to the host CPU via normal memory channels. The CPU side also includes traditional DRAM modules, which are not part of the PIM model.

(1) Prepare a buffer of tasks for each PIM module.
(2) Scatter the task buffers to the local memory of each PIM module using either PIM_Scatter or PIM_Broadcast.
(3) Launch PIM programs using PIM_Launch, to run their tasks and fill their reply buffers. Wait until all the tasks finish (PIM_Status).
(4) Gather reply buffers from the PIM local memories using PIM_Gather.

Based on this interface, our PIM-friendly ordered index processes batches of operations in *bulk-synchronous rounds*, like in [35], using the steps in Algorithm 1.

As discussed in Section 5, when implementing our PIM-friendly programs, we use pipelining to overlap the above steps, e.g., overlapping step 1 at the CPU and step 3 at the PIM modules.

A Concrete Example: UPMEM. We evaluate our techniques on the latest PIM system from UPMEM [39]. UPMEM's architecture (Figure 1) is one way to instantiate the PIM model. Its PIM modules are plug-and-play DRAM DIMM replacements, and therefore can be configured with various ratios of traditional DRAM memory to PIM-equipped ones (current maximum available configuration has 2560 PIM modules). The CPU has access to both the main memory (traditional DRAM) and all the PIM memory, but each PIM processor only has access to its local memory, because PIM modules are physically separated in different memory chips. Each PIM module has up to 628 MB/s local DRAM bandwidth, so a machine with 2560 PIM modules can provide up to 1.6 TB/s aggregate bandwidth [18]. To move data *between* PIM modules, the CPU reads from the origin and writes to the target. UPMEM's SDK supports the programming interface functions listed above, but with the restriction that the scatter/gather functions must transmit same length buffers to/from all PIM modules (i.e., the buffers are *padded* out to equal lengths).

UPMEM's main memory (a component not in the PIM model) enables running programs with CPU-side memory footprints over M words, but these additional memory accesses bring another type of communication not existing in the PIM model: *CPU-DRAM communication*. Thus it is important to write programs with good cache efficiency. Our solution in the PIM-tree is to use only a small amount of CPU-side memory: $\Theta(S) < M$ words for a batch of S operations.

2.2 Load Balance Preliminaries

A key challenge for PIM systems is to keep load balance among the PIM modules, which we define as follows:

Definition 2.1. A program achieves ***load balance*** if the *work* (unit-time instructions) performed by each PIM program is $O(W/P)$ and the *communication* (data sent/received) by each PIM module is $O(C/P)$, where W and C are the sums of the work and communication, respectively, across all P PIM modules. For programs with multiple bulk-synchronous rounds, the program achieves load balance if each round achieves load balance.

The challenge in achieving load balance is that the PIM module with the maximum work or communication must be bounded. Note that randomization does not directly lead to load balance, e.g., randomly scattering P tasks of equal work and communication to P PIM modules fails to achieve load balance. This is because one of the PIM modules receives $\Theta(\log P/\log\log P)$ tasks with high probability (*whp*)[2] in P [6], causing the work and communication at that module to be a factor of $\Theta(\log P/\log\log P)$ higher than balanced.

We use balls-into-bins lemmas to prove load balance, where a bin is a PIM module and a ball with weight w corresponds to a task with w work or w communication. We will use the following:

Lemma 2.2 ([34]). *Placing $m = \Omega(P \log P)$ balls into P bins uniformly randomly yields $O(m/P)$ balls in each bin whp.*

Lemma 2.3 ([24, 34]). *Placing weighted balls with total weight $W = \sum w_i$ and each $w_i < W/(P\log P)$ into P bins uniformly randomly yields $O(W/P)$ weight in each bin whp.*

Lemma 2.4. *Placing $m = \Omega(P \log P)$ weighted balls with weights following a geometric distribution of expectation μ into P bins uniformly randomly yields $O(\mu m/P)$ weight in each bin whp.*

We prove Lemma 2.4 using the following lemma:

Lemma 2.5 ([10]). *Let $Y(n,p)$ be a negative binomially distributed random variable that arises as the sum of n independent identically geometrically distributed random variables with expectation p.*

Then $E[Y(n,p)] = np$, and for $k > 1$, $Pr[Y(n,p) > knp] \leq \exp(\frac{-kn(1-1/k)^2}{2})$

Proof of Lemma 2.4. Each bin gets $O(m/P)$ balls *whp*, according to Lemma 2.2. Then the sum of the weights of the balls in each bin is $O(\mu m/P)$ *whp* according to Lemma 2.5. □

2.3 Prior Work on Indexes for PIM

There are several prior works for indexes on PIM systems. Two prior works [12, 29] proposed PIM-friendly skip lists. Their skip lists are based on *range partitioning*: they partition the skip list by disjoint key ranges and maintain each part locally on one PIM module. As discussed in Section 1, such range partitioning can suffer from severe load imbalance under data and query skew.

The load imbalance problem of range-partitioned ordered indexes is also studied in traditional distributed settings. Ziegler et al. [46] discussed other choices for tree-based ordered indexes in order to avoid load imbalance, including: (i) partitioning by the

[2] We use $O(f(n))$ with high probability (*whp*) (in n) to mean $O(cf(n))$ with probability at least $1 - n^{-c}$ for $c \geq 1$.

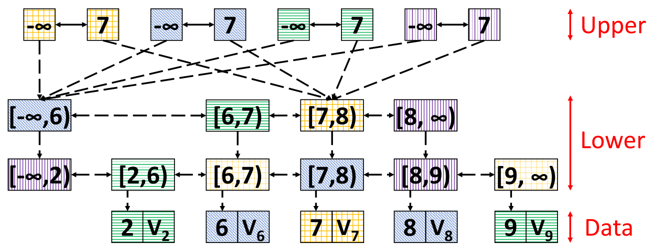


Figure 2: PIM-balanced skip list [24] with the upper part replicated on a 4-PIM system. Nodes on different PIM modules are different colors. PIM pointers are dashed lines. The lower part is $\log P$ levels.

hash value of keys, (ii) fine-grained partitioning that randomly distributes all index nodes, and (iii) a hybrid method that does fine-grained partitioning in leaves, and range partitioning for internal nodes. They also experimentally evaluated their approaches on an 8 machine cluster. However, each of these choices has its own problem in the case of a PIM system with thousands of PIM modules: (i) partitioning by hash makes range operations costly, because they must be processed by all PIM modules, (ii) fine-grained partitioning causes too much communication because all accesses will be non-local, and (iii) the hybrid method suffers from the load balance problem in its range partitioned part.

2.4 Prior Work: PIM-balanced Skip List

In a recent paper [24], we presented the first provably load-balanced batch-parallel skip list index, the ***PIM-balanced skip list***, under adversary-controlled workloads on the PIM model. A key insight was to leverage the CPU side to solve the load balance problem.

The **PIM-balanced skip list** horizontally splits the skip list into two parts, an *upper part* and a *lower part*, replicating the upper part in all PIM modules and distributing lower part nodes randomly to PIMs. This is shown in Figure 2, where nodes in different PIM modules have different colors, and the replicated upper part is explicitly drawn as four copies. For a system with P PIM-modules, the lower part is $\log P$ levels. We can afford to replicate (only) the top part because (i) it is small relative to the rest of the skip list and (ii) it is updated relatively infrequently (recall that an inserted key reaches a height h in a skip list with probability $1/2^h$).

Queries are executed by pointer chasing in the "tree" of skip list nodes. The batched queries are first evenly divided and sent to all PIM modules, each progressing through the upper part locally. Then the skip list goes through the lower part by sending the query to the host PIM module of each lower part node on the search path one-by-one, until reaching a leaf. We call this the ***Push*** method, because queries are sent ("pushed") to PIM modules to execute.

Executing a batch of parallel queries using only *Push* can cause severe imbalance, despite the lower part nodes being randomly distributed. For skewed workloads, many queries may share a common node on their search path, meaning that they are all sent to the host PIM module of that node, causing a load imbalance. These nodes are called *contention points*. An example is when multiple non-duplicate PREDECESSOR queries return the same key, with *all* nodes on the search path to that key being contention points.

The **PIM-balanced skip list** [24] solves this problem by avoiding contention points, based on a key observation: once the search paths of keys l and r share a lower part node v, searching any key $u \in [l, r]$ will also reach node v. Thus the search for u can start directly from the LCA (lowest common ancestor) of these two paths. We call this the ***Jump-Push*** method. *Jump-Push* search has a preprocessing stage to record search paths. It is a multi-round sample search starting with one sample: In each round, it doubles the sample size and uses the search paths recorded in previous rounds to decide start nodes of sample queries in this round. This approach limits the contention on each node, avoiding load imbalance.

However, the preprocessing cost is high. For P PIM modules and a batch of $P \log^2 P$ operations, it takes $O(\log P)$ sampling rounds, each of which takes $O(\log P)$ steps of inter-module pointer chasing to search the lower part. The main stage, in contrast, takes only $O(\log P)$ steps. Moreover, the preprocessing stage requires recording entire search paths—another overhead for the CPU side.

Our new ordered index (PIM-tree) uses some of the same ideas as this work, but includes key new ideas to make it simpler, and more efficient both theoretically and practically.

3 PIM-TREE DESIGN

Overview. The PIM-tree is a batch-parallel skew-resistant ordered index designed for PIM systems. It supports fundamental key-value operations, including GET(key), UPDATE(key, value), PREDECESSOR(key), INSERT(key, value), DELETE(key), and SCAN(Lkey, Rkey). It executes operations in same-type atomic batches in parallel, similar to [35]. As such, the data structure avoids conflicts caused by operations of different types. We design it starting from the structure discussed in §2.4.

In this section, we describe PIM-tree's design by studying the impact of our techniques/optimizations on the PREDECESSOR operation. We review the basic data structure design discussed in §2.4 in detail, then introduce our three key techniques/optimizations, and finally analyze the resulting communication cost and load balance. Later in §4, we describe the design of PIM-tree's other operations.

Notations. We refer to the number of PIM modules as P, the total number of elements in the index as n, the batch size as S, and the expected fanout of PIM-tree nodes as B.

Key Ideas. We observe that the two components of the PIM architecture, the CPU side and the PIM side, prefer different workloads. The distributed PIM side prefers uniformly random workloads and suffers from the load imbalance caused by highly skewed ones. Meanwhile, the shared CPU side prefers skewed workloads since we can explore spatial and temporal locality in these workloads that leads to better cache efficiency. This difference shows the complementary nature of shared-memory and distributed computing, and their coexistence in the PIM architecture motivates our hybrid method: we design a dynamic labor division strategy that automatically switches between the CPU side and the PIM side to use the more ideal platform. This strategy, called *Push-Pull search*, is the core technique of the PIM-tree.

With load balance achieved by Push-Pull search, we further propose two other optimizations, called *shadow subtrees* and *chunking*, to reduce communication. These optimizations are motivated by two basic ideas respectively: caching remote accesses at PIM modules to build local shortcuts (thereby eliminating communication), and blocking nodes into chunks (for better locality). We will show how these "traditional" techniques combine with the Push-Pull search optimization to bring an asymptotic reduction of communication from $O(\log P)$ to $O(\log_B \log_B P)$ (B is the expected fanout of chunked nodes), and a throughput increase of up to 69.7×, compared with the PIM-friendly skip list of §2.4.

3.1 Basic Structure

The *PIM-balanced skip list* is a distributed skip list horizontally divided into three parts, the *upper part*, the *lower part*, and the *data nodes* (Figure 2). Data nodes are key-value pairs randomly distributed to PIM modules to support hash-based lookup in one round and $O(1)$ communication. Every upper part node is replicated across all PIM modules, and every lower part node is stored in a single random PIM module. The ID of the PIM module hosting a lower part node is called the node's ***PIM ID***. Remote pointers, called ***PIM pointers***, are comprised of (PIM ID, address) pairs. In Figure 2, PIM pointers are represented by dashed arrows, while traditional (i.e., intra-PIM) pointers are represented by solid arrows. To save communication during search, each lower part node stores the key of the next skip list node at the same level (called a ***right-key***). There is a node at the lowest level for each key, and the probability of a node joining the next higher level in the skip list is set to $1/2$.

The upper part is replicated to enable local executions for queries on PIM modules, but the replication brings an overhead of P to both space complexity and update costs. To mitigate this overhead, the lower part height is set to be $H_{\text{low}} = \log P$, so that only a $1/2^{\log P} = 1/P$ fraction of the keys reach the upper part. By replicating the upper part, the number of remote accesses needed for a Predecessor query is reduced from $O(\log n)$ to $O(\log P)$.

In the following sections, we call the upper part ***L3*** and the lower part ***L2***. After applying the Shadow Subtree optimization (§3.3), we will further divide the lower part horizontally into two parts, called ***L2*** and ***L1***.

3.2 Push-Pull Search

Push-Pull search is our proposed search method that guarantees load balance even under skewed workloads. In the ***Push*** method, the CPU sends the query to the host PIM module of the next node along the search path, the PIM module runs the query, then the CPU fetches the result; in the ***Pull*** method, the CPU retrieves the next node along the path back to the CPU side, running the query itself. ***Push-Pull*** search chooses between Push and Pull by counting the number of queries to each node: when the number of queries to a node exceeds a specific threshold, denoted as K in the following sections, we Pull that node, otherwise we Push the query.

In further detail, Push-Pull search performs multi-round pointer chasing over the basic structure mentioned in §3.1 in three stages, where the CPU records the next pointer for each query as an array of PIM pointers throughout the process.

(1) Traverse L3 using the replicated upper parts. The CPU evenly distributes queries to PIM modules. Each PIM module runs its queries using its local copy of L3, until reaching a pointer to an L2 node. The CPU retrieves these pointers (using PIM_Gather).
(2) Traverse L2 using contention-aware Push-Pull. The CPU performs multiple Push-Pull rounds. In each round, the CPU counts the number of queries to each L2 node. If there are more than K queries to a node, choose Pull by sending a task to the PIM-side to retrieve that node to the CPU-side, then partition the queries (in parallel) based on the PIM IDs in the retrieved node's pointers on the CPU-side. Otherwise, choose Push to send a Query task to the PIM and retrieve the next pointer for the query.
(3) When the search reaches a data node, return the data.

We can record the addresses of all nodes on the pointer-chasing path for a query on the CPU side to get the ***search trace*** for each query. Note that these traces are used when performing updates (in §4). For the basic structure mentioned in §3.1, we choose $K = 1$, as it minimizes communication for constant size nodes.

Discussion. The most interesting part of Push-Pull search is that it is based on integrating two fundamental methods from distributed and shared-memory computing to achieve provable load balance with low cost (see §3.5 for analysis). We observe that the Push method is a distributed computing technique, as it uses the CPU as a router and always runs queries on PIM modules. Meanwhile, the Pull method is a shared memory technique, treating the PIM modules as standard memory modules and running the queries on the CPU. As discussed in §3, combining such fundamental methods works because of the complementary nature of the CPU side and the PIM side in the load balance issue: contention-causing (thus PIM-unfriendly) workloads are meanwhile CPU-friendly workloads.

As a solution only to the load balance issue, Push-Pull search provides no asymptotic improvements in worst-case bounds compared with Push-Only or Pull-Only methods. Such improvements are provided by our optimizations, *shadow subtrees* and *chunking*, which we describe next.

3.3 Shadow Subtrees

Shadow subtrees are auxiliary data structures in L2 that act as shortcuts to reduce communication from $O(\log P)$ to $O(\log\log P)$ for each query, while ensuring that the space complexity is still $O(n)$. The shadow subtree optimization is based on the idea of the search tree defined by a skip list, which is an imaginary tree generated by merging all possible search paths of a skip list. It contains all nodes and all edges of the skip list, except some horizontal edges. The ***shadow subtree*** of each node is a shadow copy of its search subtree stored together with this node. By using shadow subtrees, a PIM module can run queries locally through L2. Although shadow subtrees and replicating the top of the tree both involve copying nodes across different PIM modules with the purpose of reducing communication, they are actually quite different. When replicating the top, a single tree is copied P times across the modules. In the shadow subtree, every ancestor of a node has a copy of that node as part of its shadow subtree (in our case just the ancestors in L2).

Building shadow subtrees on all ($O(n)$) L2 nodes would require $O(n \log P)$ space. Instead, to maintain $O(n)$ space, we build them only on a small proportion of L2 nodes. In particular, we divide L2 into two layers, denoting the upper levels to be the new L2 and the lower levels to be L1. We build shadow subtrees only on the new L2. We set the height of L1 to be $H_{\text{L1}} = \log\log P$, so only $(1/\log P)$-fraction of nodes ($O(n/\log P)$ nodes) are in the new L2, and the

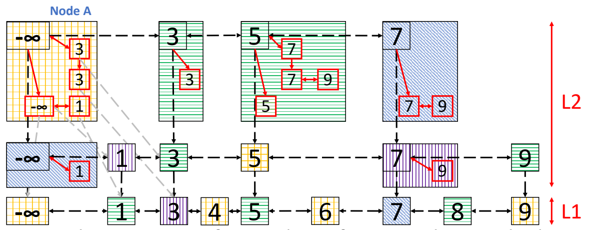


Figure 3: The structure of L2 and L1 after introducing shadow subtrees. Shadow nodes and shadow pointers are marked in red. Note that blue 1 does not have a shadow tree node for 3 because node 3 is not in its search subtree. Right-keys are omitted. We also omit pointers from shadow nodes to physical nodes except for node A. The L1 part (L2 part) is $\log\log P$ levels ($\log P - \log\log P$ levels, respectively).

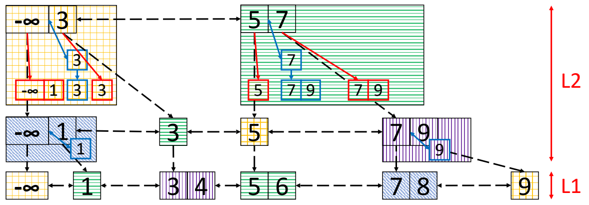


Figure 4: The intermediate state of the chunking transformation. We merge non-pivot nodes (nodes whose keys do not go to upper levels) to their left-side neighbors. Redundant shadow subtrees after merging are marked in blue, and will be removed in Figure 5. All physical pointers from shadow nodes are omitted.

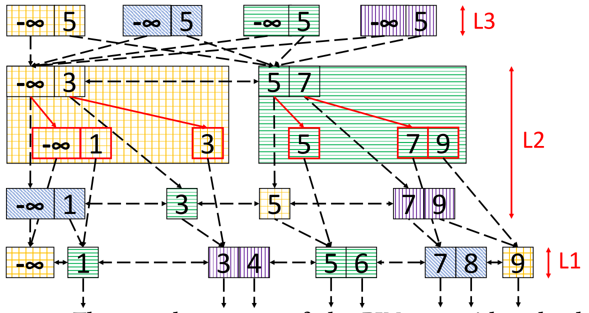


Figure 5: The actual structure of the PIM-tree with redundant shadow subtrees removed from Figure 4. We no longer need right-keys after chunking. Data nodes are omitted.

space complexity summing over all shadow subtrees is $O(n)$. Thus, the PIM-tree now has three layers: L3 under full replication, L2 with shadow subtrees, and L1 under random distribution without any replication. Each layer requires $O(n)$ space, so the total space complexity is $O(n)$. This is shown in Figure 3. We refer to original tree nodes and pointers to them as ***physical*** nodes (pointers), and mark them in black. Shadow-tree nodes and pointers to them are referred to as ***shadow*** nodes and pointers, and are marked in red.

Accelerating Predecessor using Shadow Subtrees. Shadow subtrees strengthen the Push side of Push-Pull search: a single Push round can send a query through the whole L2, rather than going forward by just one level, by running the query on the shadow subtree. Therefore the search process takes only $O(\log\log P)$ rounds for uniform-random workloads: one Push through L3, one Push through L2, and $O(H_{L1}) = O(\log\log P)$ Push-Pull rounds for L1. However, for skewed workloads, we cannot simply perform a single Push round through L2, because multiple queries may still be pushed to the contention points in L2 and cause load imbalance.

Scheme	Overhead factor	Maximum query number
Full Replication	P	Perfect Balance
Range Partitioned	1	P
Shadow Subtrees	$O(\log P)$	$\log P$

Table 1: Comparison between three types of replication schemes that run queries with $O(1)$ communication. The larger the overhead factor, the more space it takes and the slower updates will be. The larger the maximum query number is, the more imbalanced the execution will be under skewed workloads.

We solve this problem again by Pull, by introducing a multi-round Pull process to eliminate contention points.

In further detail, Push-Pull search in L2 has two stages: we first perform up to $O(H_{L2})$ Pull rounds for nodes with $\geq K$ queries until no such node exists, where $H_{L2} = \log P - \log\log P$ denotes the new L2's height, then execute one "Push" round to send all queries through L2. We set the threshold $K = H_{L2}$ instead of 1 since "Push" is now more powerful and we tend to use it more. Both stages take $O(1)$ balanced communication per query, as we will prove in Lemmas 3.4 and 3.5.

In practice, we use another optimization to reduce the number of Pull rounds. Note that although contention points are the only source of load imbalance, we may reach a reasonable level of load balance before eliminating all contention points. Therefore, to avoid unnecessary Pull rounds, before starting a Pull round, we measure the load balance across PIMs by counting the number of queries that will be sent to each PIM module; if the one with the most is below 3× the average load, we stop the Pull round and start to Push.

Replication and Space/Imbalance Tradeoffs. Compared with full replication used for L3 and range partitioning (which performs no replication) used in related works, shadow subtree is a novel scheme that supports queries with $O(1)$ communication by improving the locality of a distributed ordered index. Specifically, shadow subtree is a selective replication approach that lies between these two prior schemes. If we replicate nodes not only to their L2 ancestors, but to all PIM modules, we obtain full replication. On the other hand, if we only keep the shadow subtrees of the L2 roots, we obtain the range partitioned scheme.

The cost and the skew-resistance of shadow subtrees also lies between those of the other two schemes. Table 1 shows the bounds when applying different schemes to a skip list with height $\log P$ and size P in expectation. In the full replication scheme, we can run queries with perfect load balance, but it brings an overhead factor P to both space complexity and update costs. On the other hand, for the range partitioned scheme, each query can only be executed by a single PIM module. We can still do Push-Pull to avoid contention with a threshold of $K = P$: we will choose to simply Pull the whole tree when the number of queries exceeds the size of the whole part. There can be load imbalance, as some part gets up to P queries and others get none. For this approach there is no overhead on space complexity or for updates. Lastly, using shadow subtrees, the overhead factor is $O(\log P)$ as each node is replicated in all its L2 search tree ancestors, and the maximum query number is $\log P$ according to our choice of Push-Pull threshold $K = H_{L2}$.

Shadow subtrees therefore yield a balanced compromise between the two schemes, providing a sweet spot for both overhead and skew-resistance.

3.4 Chunked Skip List

Chunking or "blocking" is a classic idea widely used in locality-aware data structures, e.g., B-trees and B+-trees. To improve locality, we apply a similar chunking approach to improve the access granularity of the PIM computation, while decreasing the tree height. As chunking increases the access granulariy, each PIM processor obtains larger local memory bandwidth, therefore better performance. The effect of access granularity in PIM is discussed in detail in [18].

We apply chunking to all layers of the PIM-tree. In L3, we replace the multi-thread skip list with a batch-parallel multi-threaded B+-tree [35]. In L2 and L1, we chunk the nodes in our skip list to obtain a ***chunked skip list***. We first merge horizontal non-pivot nodes (whose keys do not go to upper levels) into a single chunk, then remove redundant shadow subtrees. Applying this two step process on Figure 3 first gives Figure 4 as an intermediate state, and finally the PIM-tree in Figure 5.

The result with shadow subtrees looks similar to a B+-tree. The difference is that while the B+-tree sends nodes to upper levels on overflow of lower level nodes, the chunked skip list uses random heights generated during Insert, so the fanout holds in expectation. We decrease the probability of reaching the next level in the skip list from $1/2$ to $1/B$, so that the expected fanout is B. We choose the same chunking factor B in L3, L2 and L1 for simplicity, but different factors could be used in each part. As discussed in §4.2, we use a chunked skip list instead of a classical B+-tree in L2 to make batch-parallel distributed Insert and Delete simpler and more efficient. We use a B+-tree in L3 because the structure is not distributed, making batch-parallel Insert and Delete easier.

Chunking reduces tree height at all levels, which improves multiple aspects of our design. We denote the new L2 (L1) height as H'_{L2} (H'_{L1}, respectively). The L2 part of the search path to each node is reduced from $O(H_{\text{L2}}) = O(\log P)$ to $H'_{\text{L2}} = \log_B P - \log_B \log_B P$, as there are no longer horizontal pointer-chasing processes. Therefore, the space and replication overhead of shadow subtree from $O(H_{\text{L2}})$ to H'_{L2}, as each node is only replicated in its L2 ancestors. Furthermore, lower overhead enables us to reduce H'_{L1} to $\log_B \log_B P$. H'_{L1} is effectively 1 in practice, because with our choice of $B = 16$ it will take over 10^{19} PIM modules for H'_{L1} to exceed 1.

Therefore, in practice, the height of L2 is reduced to 2 levels, and L1 reduced to 1 level. The probability for a key to reach L3 is $1/4096 < 1/P$, and the probability of reaching L2 is $1/16 < \log_{16} P$.

Implementing Predecessor after Chunking. Chunking brings only one modification to the search process: changing the Push-Pull threshold K from H_{L2} to $B \cdot H'_{\text{L2}}$, because we now "Pull" chunks with expected size $O(B)$ instead of $O(1)$. The detailed algorithm is explained in §3.5.

Chunking also improves the communication costs of Predecessor. First, as the height of L1 is reduced from $\log \log P$ to $\log_B \log_B P$, each query now causes only $O(\log_B \log_B P)$ communication in L1. Second, chunking reduces the maximum possible number of Pull-only rounds in L2 from $O(H_{\text{L2}}) = O(\log P)$ to exactly H'_{L2}, which is $\log_B P - \log_B \log_B P$. This helps reduce the number of communication rounds under skewed workloads.

3.5 Predecessor Algorithm and Bounds

Next, we describe the complete algorithm for Predecessor, and discuss its cost complexity. We provide proofs for the communication cost and load balance of Predecessor queries. For simplicity throughout the paper, our cost analyses assume that hash functions provide uniform random maps to PIM modules, so that the lemmas in §2.2 can be applied. Algorithm 2 summarizes the search process.

Algorithm 2. Predecessor (Q: batch of query keys)

```
(1) Push queries from $Q$ evenly to PIM modules, and traverse L3.
(2) While the number of queries that will be sent to each PIM
    module for L2 is not balanced (i.e., the busiest PIM module
    gets more than 3× the average load), do the following:
    (a) Pull all nodes with more than $K = B \cdot H'_{\text{L2}}$ queries back
        to the CPU.
    (b) Use these nodes to progress the pointer-chasing process
        of these queries by one step.
(3) Push each query to the PIM module holding its search node,
    and traverse L2 using the shadow subtrees.
(4) Perform $H'_{\text{L1}}$ Push-Pull rounds with $K = B$ to traverse L1, and
    retrieve the data nodes.
```

We demonstrate here a mini step-by-step example of a Predecessor batch with four queries on the PIM-tree in Figure 5 (note that real batches should have more queries on this tree to achieve load-balance). The queries request the Predecessors of keys 1, 3, 4 and 7. PIM-tree first evenly distributes one query for each of the four PIM modules to search through L3, returning three queries falling onto the L2 node $[-\infty, 3]$ and one falling onto node $[5, 7]$. The context of node $[-\infty, 3]$ will be pulled to the CPU from the yellow-masked PIM module due to its large contention, and the pointer-chasing searching of keys 1, 3 and 4 over L2 will be executed on the CPU side. After that, query 1, query (3, 4), and query 7 will be pushed to the PIM module containing the blue-masked node $[-\infty, 1]$, green-masked node $[3]$ and green-masked node $[5, 7]$ respectively on the local shadow subtrees to search through L2. Finally, all queries will be carried out in a similar Push-Pull way to return the results from L1 and data nodes.

Theorem 3.1. *A batch of Predecessor queries can be executed in $O(\log_B P)$ communication rounds, with a cost of $O(\log_B \log_B P)$ communication for each operation in total whp. The execution is load balanced if the batch size $S = \Omega(P \log P \cdot B \cdot H'_{L2}) = \Omega(P \log P \cdot B \cdot \log_B P)$. The CPU-side memory footprint is $O(S)$.*

The following lemma is useful for bounding the communication of queries that do not incur contention. We use Lemmas 3.2 and 3.3 to prove Lemmas 3.4 to 3.6, then combine them to prove Theorem 3.1.

Lemma 3.2 (Push). *For batch size $S = \Omega(K \cdot P \log P)$, using Push for queries whose target nodes have fewer than K queries incurs $O(S/P)$ communication per PIM module whp.*

Proof. We analyze the communication using a weighted balls-into-bins game, where we take the target nodes as balls, the numbers of queries on the target nodes as weights, and PIMs as bins. The weight limit is K by assumption, as each node gets at most K queries, and the weight sum is at most S. Applying Lemma 2.3, each PIM module incurs $O(S/P)$ communication. □

Lemma 3.3 (Pull). *For batch size $S = \Omega(K \cdot P \log P)$, and nodes with geometrically-distributed chunk sizes, with an expected chunk size of B, using Pull to fetch nodes with more than K queries incurs $O(B \cdot \frac{S}{KP})$ communication for each PIM module whp.*

Proof. We Pull no more than S/K nodes in each round. The amount of communication caused by using Pull to fetch each node is equal to its node size, which is geometrically distributed, and is B in expectation. Using weighted balls-into-bins again, we treat the balls as target nodes, communication on each target node as weights, and PIMs as bins. As $S = \Omega(K \cdot P \log P)$, we have that the number of balls is $\Omega(P \log P)$. Thus, applying Lemma 2.4 with $\mu = B$, each PIM module incurs $O(B \cdot \frac{S}{KP})$ communication *whp*. □

Lemma 3.4 (Pull-Only rounds for L2). *The multi-round Pull stages for L2 (the loop in Algorithm 2) are load balanced. Overall, the loop incurs $O(1)$ communication and PIM work whp per operation, and finishes in $H'_{L2} = \log_B P - \log_B \log_B P$ rounds.*

Proof. Because the number of rounds is upper bounded by the L2 height, the loop must finish in H'_{L2} rounds. To prove load balance and bound communication and work, we show that the CPU-PIM communication per PIM is $O(\frac{S}{H'_{\text{L2}} \cdot P})$ in each of the rounds. This result follows by applying Lemma 3.3 with $K = B \cdot H'_{\text{L2}}$. □

Lemma 3.5 (Push round for L2). *Push using the shadow subtrees (stage 3) takes 1 round, $O(1)$ communication whp for each query, and is load balanced.*

Proof. In this stage we send each query as a task to the corresponding PIM module, incurring $O(1)$ communication per query and 1 round overall. The load balance follows by applying Lemma 3.3 with $K = B \cdot H'_{\text{L2}}$. □

Lemma 3.6 (Push-Pull rounds). *For chunk nodes with geometric-distributed sizes of expectation B, the Push-Pull search rounds (stage 4) with batch size $S = B \cdot P \log P$ and threshold $K = B$ incur $O(H'_{L1})$ communication and PIM work whp for each query, and achieve load balance. The data nodes can be treated as chunk nodes with $B = 1$.*

Proof. As Push-Pull is a combination of both Push and Pull, this can be proved by apply $K = B$ to both Lemma 3.2 and Lemma 3.3, for each of the H'_{L1} rounds. □

Proof for Theorem 3.1. The CPU-side memory footprint is $O(S)$ because the only datum needed for all steps is an array of S (key, address) pairs, taking $O(S)$ space, and auxiliary memory used in each step (e.g. reading/writing the task buffer mentioned in Algorithm 1) is also $O(S)$.

We prove the communication bounds and load balance separately for each stage of Algorithm 2:

It's obvious that stage 1 incurs $O(1)$ communication, one communication round. It's load balanced because L3 is replicated across the PIMs, and we evenly distribute the queries to the PIMs.

Bounds for the Pull-only rounds and Push-only rounds in L2, and lastly the Push-Pull rounds for L1 and data nodes are proven seperately in Lemmas 3.4, 3.5 and 3.6, respectively. □

4 PIM-TREE: OTHER OPERATIONS

Having described the design of the PIM-tree data structure in §3, using the Predecessor operation as the running example, we now describe how other PIM-tree operations are implemented.

4.1 Get and Update using Hashing

Get and Update are operations with a given key. These operations also do not modify the structure of the data structure. Therefore, we solve these in one round and $O(1)$ communication per operation through a hash-based approach by first (i) using a fixed hash function to map keys to PIM modules, and (ii) building a local hash table on each PIM module to map keys to the local memory addresses of their data nodes.

Because the data nodes are distributed by a hash function, we achieve good load balance even for skewed workloads, assuming that there are not duplicate operations to the same key. If such redundant operations exist, we can solve this by preprocessing to combine operations on the CPU-side, using a user-defined combinining mechanism. In practice, we use a linear-probing hash table on the PIM-side, but other hash-table variants could also be used.

Algorithm 3 summarizes the steps for Get, and Theorem 4.1 shows the bounds; the steps for Update are identical, except that each data value is updated in place.

Algorithm 3. Get (Q: batch of query keys)

(1) Host program finds the PIM module for each operation's key using the hash function.
(2) Each PIM module uses its local hash table to find the data node, and return the data value.

Theorem 4.1. *A batch of Get (or Update) operations is executed in one communication round, incurring $O(1)$ communication and $O(1)$ expected PIM-work per operation. The execution is load balanced if the batch size $S = \Omega(P \log P)$. It requires $O(S)$ shared-side memory.*

Proof. It takes one communication round, $O(1)$ communication and $O(1)$ expected PIM work, because Get (Update) operations are packed into *constant-size* Get (Update) tasks, then sent to PIM modules and executed with *constant expected PIM-work* by the local hash table of each PIM module.

Since each Get (Update) incurs constant communication and PIM-work, Lemma 2.2 proves load balance for $S = \Omega(P \log P)$. □

4.2 Insert

An Insert(key,value) operation inserts the key into nodes on its search path, and the primary challenge is to avoid contention and conflicts when multiple Inserts in a batch go to the same node(s). We solve this by preprocessing: we perform searches in parallel and record the trace of each search, and use the traces to detect and handle contention points. Our algorithm has three stages: (1) perform

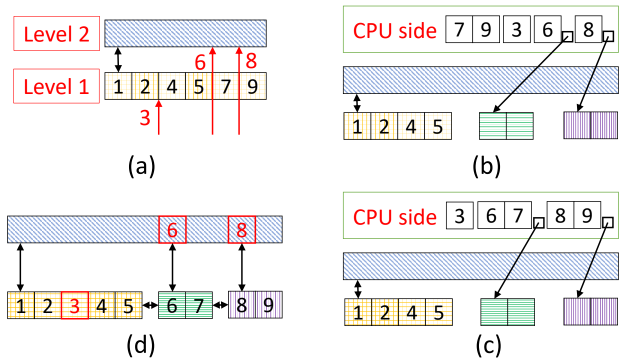


Figure 6: The process to insert keys 3, 6, and 8 into L2 of the PIM-tree. Insertion 3 has height 1, and insertion 6 and 8 both have height 2. These heights are generated beforehand by coin tossing with probability $1/B$. The height of the yellow node is 1, and that of the blue node is 2. As a result, key 3 is inserted into the yellow node, and keys 6 and 8 split the yellow node.

searches to record the search trace, (2) modify the physical skip list based on the search trace, and (3) update the shadow subtrees.

Record Search Trace. Recording the search trace is simple when there are no shadow subtrees, as the addresses of all of the searched nodes correspond to their actual (non-replicated) locations. Specifically, we can obtain a search trace per-query by recording the PIM pointers used in the search process in each Push-Pull step. However, the recording becomes challenging when using shadow subtrees, because they enable the search to run in the shadow copy (using local pointers), but we still need the true physical node addresses to modify the skip list. To solve this problem, we also store the physical pointers in shadow nodes. Note that for each red shadow pointer in Figure 5, there is a black physical pointer referencing the physical origin of the shadow node. When the search uses a shadow subtree, the PIM module records the corresponding physical pointer, and therefore records the correct search trace.

Update Physical Skip List. After obtaining the search traces, we Insert into these nodes according to random heights we generate prior to updating the PIM-tree. According to the height of each insertion and each layer, all insertions go to L1, $1/\log_B P$ goes to L2, and $1/P$ goes to L3. Insertions to different layers are applied differently. Insertions to L3 are broadcasted to all PIM modules and applied to their L3 copy, which is a local B+-tree. For L1 and L2, insertions are applied to nodes on the search trace according to their heights. Figure 6 is an example of insertions with contention, where we insert key 3 into the node as well as splitting the node with key 6 and 8, according to the pre-generated height. The insertion takes three steps: in (b) we fetch the right-side part of the node to the CPU side and generate empty new nodes in random PIM modules; in (c) we derive the correct element to be inserted to each node in CPU; finally in (d) we insert them.

The splitting policy of PIM-tree enables it to insert into different levels and different layers at once, rather than level-by-level bottom up from the leaves like the B+-tree. Complicated update processes (broadcast in L3, fetch and calculation in L2) over a small proportion of data can also be executed in parallel with the simple update processes (direct insert in L1) for all data, because they don't need results from lower layers. In practice, we insert to all levels in 2 rounds by (1) initializing new nodes and fetching the right-side parts in the first round, then (2) inserting into existing nodes and new nodes in the second round.

Each node receives at most $O(B)$ insertions in these stages, therefore we avoid load imbalance. Although the number of concurrent insertions to a single shared node isn't limited, these insertions split this node with probability $1/B$, and only insertions with keys less than the minimum-keyed split will actually be applied to this node, which are $O(B)$ insertions.

Update Shadow Subtrees. To maintain the invariant that shadow subtrees are copies of the search subtrees, we need to update shadow subtrees after performing insertions to the physical skip list. There are three types of updates: (1) **build** the new shadow subtree for a new node, (2) **insert** a new node into the shadow subtrees of its ancestors, and (3) **trim** a shadow subtree after a node split.

Our shadow subtree updating technique is straightforward. For build, we pull the L2 search tree and send it to the new node. For insert and trim, we observe that only shadow subtrees of nodes on the search trace need updating, so we send the newly-inserted node to all these nodes.

Algorithm. Algorithm 4 summarizes the steps for Insert.

```
Algorithm 4. Insert (K: batch of keys, V: batch of values)
(1) Generate a height for each Insert according to the geometric
    distribution with probability $1/B$.
(2) Run Predecessor on the keys to obtain/record search traces.
(3) Distribute new empty nodes to PIM modules, and record the
    address of these nodes in the CPU side.
(4) For each node that is split, send the minimum splitting key
    to fetch the right-side part.
(5) On the CPU side, compute the contents of each new node,
    and the new horizontal pointers.
(6) Apply all insertions to L1 and L2 nodes for both old and new
    nodes, and build horizontal pointers.
(7) Broadcast insertions that reach L3 to all PIM modules.
(8) Update shadow subtrees in existing L2 nodes.
(9) Build shadow subtrees for newly inserted L2 nodes.
```

Note that steps 3–4 and 6–8 are both done using a single communication round.

Optimizations. We don't need the whole search path for most insertions. First, we only record the search trace in L1 and L2, as insertions to L3 are executed locally by the PIM modules. Also, L1 insertions with height $h < H_{L1}$ only need and affect h nodes at the bottom of its search trace, therefore we only record these nodes, which is $O(1)$ nodes in expectation. This optimization ensures that each insertion records $O(1)$ nodes in their search path on average, because L1 insertions need only $O(1)$ expected nodes, and only a $1/\log_B P$-fraction of the insertions reaching L2 require a search trace with $\log_B P$ nodes.

Discussion: Load Balance in Insert. There is a load balance issue in our shadow subtree update algorithm: To keep shadow subtrees up to date, an L2 node may need updates of size $O(P/\log_B P)$. First, new subtrees can have up to $O(P/\log_B P)$ (given $H'_{L2} = \log_B P - \log_B \log_B P$); second, a single shadow subtree may receive up to $O(P/\log_B P)$ updates. (We derive this bound in a similar way to the concurrent insertions to a single node. An insertion splits the shadow subtree if it reaches L3. Each insertions reaching L2 splits the shadow subtree with probability $\log_B P/P$, so the number of actual insertions to a shadow subtree is $O(P/\log_B P)$). This contention factor $O(P/\log_B P)$ will grow faster than the factor $K = B \cdot \log_B P$ of Predecessor as P grows. This contention has minor effect at present, but it may cause issues when P gets larger in the future. Here we propose an algorithm to address this problem (not implemented at present).

A solution to avoiding load imbalance caused by contention, even as the number of PIM modules scales, is to delay the update process. To identify contented shadow subtrees, we keep track of the number of shadow-subtree updates of any existing node, as well as the unbuilt shadow-subtree sizes for new nodes. When the unfinished work on any node exceeds a threshold, applying the pending updates will cause contention, which leads to load imbalance. Therefore, instead of fully updating this node in a single round, we mark the node as *unfinished*, and prevent future queries from calling Push on it to avoid using its unfinished shadow subtree. Whenever the overall number of unfinished nodes in the entire tree reaches a threshold K_i, we start an update phase of multiple update rounds, where we send constant information to each contention point, until the number of unfinished nodes drops below K_i. According to Lemma 2.2, $K_i = P \log P$ suffices to ensure the communication in each update round is balanced.

Unfinished nodes bring only one modification to the Predecessor algorithm: In the multi-round Pull-Only phase, we have to Pull any unfinished nodes on the search path of any query.

Theorem 4.2. *Theorem 3.1 still holds with unfinished nodes.*

Proof. We only need to prove for Lemma 3.4 because we only change the Pull-Only stage. The lemma still holds because it takes only $O(B \log P)$ additional communication to Pull at most $K_i = P \log P$ unfinished nodes in each round, and $O(B \log P) \le O(\frac{S}{H'_{L2} \cdot P})$. □

Lemma 4.3. *An Insert only modify $O(1)$ expected nodes.*

Proof. An Insert affects the PIM-tree differently according to which (L1, L2, L3) layer it reaches. This is decided by the pregenerated height.

All Inserts affects L1. An L1-only Insert affects $O(1)$ expected nodes because its height is $O(1)$ expected. $O(1/\log_B P)$ Inserts are L2 Inserts. An L2 Insert affects $H_{L1}+H_{L2} = \log_B P$ nodes, because it needs to update all shadow subtrees of its L2 ancestors. $O(1/P)$ Inserts are L3 Inserts. An L3 Insert affects $O(P)$ nodes because we need to update all replicas of the modified L3 nodes. □

Theorem 4.4. *A batch of Insert operations can be executed in $O(\log_B P)$ IO rounds, incurring $O(\log_B \log_B P)$ communication for each operation. The execution is load balanced if the batch size $S = \Omega(P \log P \cdot B \cdot H'_{L2}) = \Omega(P \log P \cdot B \cdot \log_B P)$. The CPU-side memory footprint is $O(S)$.*

Proof. We treat an Insert execution as a Predecessor and additional steps. As the bounds for Insert is the same as that of Predecessor in IO rounds, communication and batch size requirements, here we prove that the additional steps also follows this bound.

We first prove for the $O(S)$ CPU-side memory footprint. As we've already proved $O(S)$ memory footprint for Predecessor in Section 3.5, we only prove that the additional memory space required by Insert is $O(S)$. These memory is mainly comprised of search paths and new nodes. (1) A batch of S Inserts needs $O(S)$ *whp* addresses for search paths, because we only need the addresses of the affected nodes, which is $O(1)$ for each Insert according to Lemma 4.3. (2) For new nodes, $O(S/B)$ *whp* new nodes are generated, and the size of each follows a geometric distribution with expectation B, given a $O(S)$ *whp* total size.

We then prove for the $O(\log_B \log_B P)$ communication and $O(\log_B P)$ IO rounds. The additional steps take $O(1)$ IO rounds. For communication, each operation causes $O(1)$ communication because the costs of both inserting into existing nodes and generating new nodes are $O(1)$ expected for each node. Amortizing over a batch gives $O(\log_B \log_B P)$ *whp* communication.

We then prove for the $O(\log_B \log_B P)$ communication, $O(\log_B P)$ IO rounds, and load balance for the additinoal steps. step 1 and 5 causes no communication; step 7 is a broadcast of $O(S/P)$ data, hence causes load-balanced $O(S)$ communication; step 3 distributes $O(S/B) = O(P \log P \log_B P)$ unit-size empty nodes each to a random PIM module.

In step 4, we pull the right-side part of $O(S/B) = O(P \log P \log_B P)$ nodes *whp*, each node's size following geometric distribution with expected value B. The proof for $O(S/P)$ IO time is similar to that in Lemma 3.3.

In step 6, we prove $O(S/P)$ IO time seperately for two type of insertions: building new nodes and insert into existing nodes. The proof of $O(S/P)$ IO time for new nodes is similar to that of step 4, as we build $O(S/B)$ new nodes with the same size distribution. Bounds for insertion into non-leaf existing nodes can be proved similarly, as there's $O(S/B)$ such nodes. For insertion to leaf existing nodes, there is (1) up to S such insert; (2) up to S target leaf existing nodes for uniform random workload; (3) up to B *whp* insert to a single node in case of data skew (geometric distribution with expectation B). We divide target leaf nodes into two types by whether this node has more than B insertions. For nodes with less than B insertion, we apply Lemma 2.3, where the weight limit is B and the total weight is the total number of insert (no more than S). For nodes with more than B insertion, there is no more than (S/B) such node, and the number of additional inserts (other than the first B inserts) for each node follows geometric distribution with possibility $1/B$. so we apply Lemma 2.2 and Lemma 2.4 respectively for the first B inserts and additional inserts, to prove $O(S/P)$ IO time.

For shadow subtrees, we proof the sum of updates to all shadow subtrees is $O(S)$. We don't prove for for load balance, becaused they're executed by the update rounds that ensure load balance. We prove seperately for the two types of updates: updates to existing shadow subtrees, and building new shadow subtrees. There's $O(S)$

whp updates to existing shadow subtrees, because $O(S/log_BP)$ *whp* insertions reach L2, and each insertion does $O(1)$ update to less than $\log_B P$ shadow subtrees (of its L2 ancestors). Building new shadow subtrees also take $O(S)$ updates: we denote L2 leaves as level 0, the L2 includes $\log_B P - \log_B \log_B P < \log_B P$ levels. Since the probability of PIM-tree is $1/B$, there's $O(S/\log P/B^{i+1})$ *whp* new nodes generated on level i, and we need to build a shadow subtree with size $O(B^{i+1})$ *whp* for each node. Multiplying the number of new nodes on each level and the size of these nodes gives $O(S/\log P)$ updates on each level, and $O(S)$ updates for all levels. □

4.3 Delete

We perform deletions in a similar way to insertions: first get the search trace, then delete keys from nodes on the search trace, and finally apply updates to shadow subtrees. While insertion causes node split, deletion causes node merge when the pivot key is deleted from a node.

We need the height of each key to modify the tree. While these heights are pre-generated when doing inserts, we need to get them before doing deletions. We store the height of each key in the data node. Before deletions, we do a batch Get to get these heights and filter out invalid deletions. With these heights, we do deletions just like insertions: modify the physical tree by removing keys and merging nodes, then update shadow subtrees for nodes on the search path.

A batch of Delete operations can be executed in $O(\log_B P)$ communication rounds, incurring $O(\log_B \log_B P)$ communication for each operation. Algorithm 5 summarizes the steps.

Algorithm 5. Delete (Q: batch of query keys)

```
(1) Preprocess: get the height for each valid DELETE, and remove
    any invalid DELETEs.
(2) Batch search to obtain search traces for each key.
(3) Remove the deleted nodes from the search traces.
(4) Fetch the remaining part if a node's pivot key is deleted.
(5) Merge horizontal consecutive remaining parts in the CPU.
(6) Insert the remaining parts to the left-side node of the removed
    nodes.
(7) Build horizontal pointers.
(8) Broadcast deletions that reach L3 to all PIM modules.
(9) Update shadow subtrees in existing L2 nodes.
```

Note that steps 3–4 and 6–8 are both done using a single communication round.

4.4 Scan

The Scan(LKey,Rkey) operation (a.k.a. range query) returns all the (key, value) pairs whose keys fall into the range of [Lkey, Rkey]. Algorithm 6 summarizes the Scan process.

When running a batch of Scan queries, we first on the CPU side merge all overlapping ranges in the batch into groups of disjoint larger ranges, by sorting the Lkey of the ranges and then carrying out a prefix sum on the Rkey, with the binary associative operator set to be $max()$. The leftmost bound of these merged disjoint ranges are the Lkey$[i]$ where Lkey$[i]$ > Rkey$[i-1]$. PIM-tree can use user-defined thresholds to split too large merged ranges into several small disjoint ranges to avoid overflows on hardware as well as achieve load-balance. Then the PIM-tree carries out batched scans on these disjoint ranges and eventually rearranges corresponding results from the fetched (key, value) pairs. PIM-tree then carries out Scan throughout the L3 traversal, by evenly distributing the batched range quries to all PIM modules and maintaining two boundary nodes (the predecessors of Lkey and Rkey on the current level) for each range query through the level-by-level L3 search. Within a single range, only the two boundary nodes and their intermediates will be involved later. The two boundary nodes are marked with SearchRequired labels, while the intermediates are marked with FetchAll labels.

FetchAll nodes are required to return all their leaf data nodes. Note that the range queries sent to the PIM modules are disjoint, so FetchAll nodes do not generate contention points throughout the L2 searching. Thus, FetchAll queries can be simply pushed to PIM modules and take advantage of shadow subtrees.

The SearchRequired nodes are processed similar to Predecessor. Since SearchRequired nodes might overlap on some highest levels in L2 and thus cause load-imbalance, similar Push-Pull search is taken. SearchRequired nodes with large contentions are pulled to the CPU side, while the others are pushed to the PIM modules. PIM-tree maintains the two boundary nodes in each range query by a Predecessor-like searching throughout L2, while the newly-generated intermediate nodes in lower levels are labeled with FetchAll.

Algorithm 6. Scan (R: batch of range queries)

```
(1) Preprocessing: On the CPU side, merge the overlapping range
    queries in $R$; then split too large ranges in the merged results.
    (a) Sort $R$ with the $LKeys$ in ascending order. Store the sorted
        ranges in $R'$.
    (b) Construct a new array $RRkeys$ with $RRkeys[i]$ represent-
        ing the maximum value in $R'.RKeys[0 : i]$, by using a
        parallel prefix sum on $R'.RKeys$ with the binary associa-
        tive operator set to be $max()$.
    (c) Each $i$ with $R'.LKeys[i] > RRkeys[i-1]$ represents a
        starting of a merged range required. Construct merged
        range queries $R''$ based on a parallel comparison and
        packing.
    (d) Split too large merged ranges in $R''$ based on user-defined
        thresholds.
(2) Push range queries in $R''$ evenly to the PIM modules and
    traverse L3. For L3 nodes $N_1, N_2, ..., N_m$ that falls into a single
    range query, mark $N_1$ and $N_m$ with SEARCHREQUIRED labels,
    and the others with FETCHALL labels.
(3) Return all leaf nodes for FETCHALL queries.
```

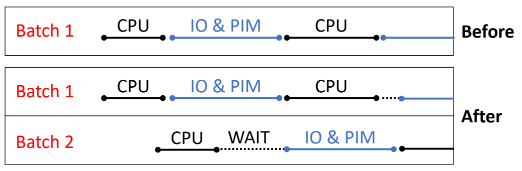


Figure 7: Program traces before/after CPU-PIM pipelining

   (a) Push each FetchAll query to the PIM module holding its search node, and traverse L2 using the shadow subtrees.
   (b) Traverse L1 and retrieve the data nodes.

(4) Use push-pull methods to process SearchRequired nodes with different contentions.
   (a) Pull all nodes with more than $K = B \cdot H'_{L2}$ queries back to the CPU. Use these nodes to process the pointer-chasing process of these queries by one step.
   (b) Push other nodes to the PIM module. Traverse L2 using the shadow subtrees. Maintain two boundary nodes in a range as SearchRequired through all levels of the PIM-tree and perform Predecessor-like searches until retrieving the data nodes. Mark the intermediate nodes as FetchAll and process them using Step 3.

(5) Rearrange the result with R and the returned key-value pairs on the CPU side.

5 IMPLEMENTATION

CPU-PIM Pipelining. Thus far, we have introduced algorithms where tasks on the CPU and PIM run in a synchronized, tick-tock manner in each round as depicted in Algorithm 1. The total execution time of this approach consists of three non-overlapping components: CPU-only time, PIM-only time, and communication time. Communication requires both CPU and PIM, but the other two components only utilize one part of the system, which presents an opportunity to reduce execution time by pipelining the CPU-only and PIM-only components.

For pipelining, we consider executions that run multiple batches in parallel in the PIM-tree. This is shown in Figure 7, where "CPU" represents time spent in CPU-only execution, while "IO & PIM" represents time spent in CPU-PIM communication and the PIM program. On our UPMEM system, CPU-PIM communication requires exclusive control of the PIM side, and any concurrent use of the PIM side will cause a hardware fault. Hence, one batch needs to wait for the PIM side to finish the current execution tasks. We only pipeline queries in our experiments, since update batches cannot be carried out concurrently. For mixed operations, we protect the PIM-tree by a read-write lock to prevent update batches from running concurrently with other batches.

PIM Program. PIM-tree's PIM program is a parallel executor of the tasks in the buffer sent from the CPU. It is designed to address two features of UPMEM's current PIM processors. First, the PIM processor is a fine-grained multi-threaded computing unit [18], and requires at least 11 threads to fill the pipeline, so we write PIM programs in the form of 12 threads. Second, UPMEM's system only supports PIM programs with fewer than 4K instructions, but the implementation of PIM-tree exceeds this bound. To bypass this restriction, we write the PIM program as multiple separate modules, and load each module when needed. Only Insert and Delete operations require swapping modules; program loading currently takes around 25% of the execution time. The remaining operations on PIM-tree fit within the 4K instruction limit.

6 EVALUATION

In this section, we evaluate our new PIM-optimized indexes on a PIM-equipped machine provided by UPMEM, and two traditional state-of-the-art indexes on a machine with similar performance. We summarize our expeirmental results from this section as follows:

(1) The PIM-tree performs better than the range-partitioned skip list under skewed workloads in terms of throughput, memory-bus communication, and energy consumption.
(2) The PIM-tree causes lower communication on the memory bus compared with traditional indexes without PIM.
(3) All optimizations mentioned, including Push-Pull search, shadow subtrees, chunked skip list and CPU-PIM pipelining, yield performance increases to (some) PIM-tree operations.

6.1 Experiment Setup

UPMEM's PIM Platform. We evaluate PIM-tree on a PIM-equipped server provided by UPMEM(R). The server has two Intel(R) Xeon(R) Silver 4126 CPUs, each CPU with 16 cores at 2.10 GHz and 22 MB cache. Each socket has 6 memory channels: 4 DIMMs of conventional DRAM are implemented on 2 channels, while 8 UPMEM DIMMs are on the other 4 channels. Each of the 16 UPMEM DIMMs has 2 ranks, each rank has 8 chips, and each chip has 8 PIM modules. There are 2048 PIM modules in total.

Traditional Machine w/o PIM. We evaluate traditional indexes on a machine with two Intel(R) Xeon(R) CPU E5-2630 v4 CPUs, each CPU with 10 cores at 2.20 GHz and 25 MB cache. Each socket has 4 memory channels. There are no PIM-equipped DIMMs. We cannot evaluate traditional indexes on the server of UPMEM because 2/3 of its memory channels are used by PIM-equipped DIMMs, which cannot be used as the main memory. Directly running traditional indexes on the server would cause unfairness in main memory bandwidth for the traditional indexes. In our experiments we choose the state-of-the-art binary search tree [9] and (a,b)-tree [11] as competitors. Both implementations are obtained from the SetBench benchmarking suite [5].

Range-Partitioned and Jump-Push Baselines. We implement a range-partitioned-based ordered PIM index as our primary baseline, where both data nodes and index nodes are distributed to PIM modules based on the ranges of the key [12, 29]. We record the range splits in the CPU side, and use these splits to find the targeted PIM module of each operation. Point operations are sent to and executed on the corresponding PIM module. Running a batch of Scan operations is similar, except that it runs an additional splitting in queries according to the range splits before tasks are sent to the PIMs. We also build a local hash table on all PIM modules for Get.

We also implement the PIM-balanced skip list [24] described in §2.4 as another baseline. We experimentally evaluate this approach when discussing the impact of the optimizations proposed in this paper, in Figure 11, where the algorithm is called "Jump-Push based".

Test Framework. We run multiple types of operations on the PIM-tree, range-partitioning skip list we implemented, and the state-of-the-art traditional indexes. In all experiments, we first warm up the index by running the **initialize set** that INSERT key-value pairs, then evaluate the index by the **evaluation set** of multiple operations. All operations are loaded from pre-generated test files. PIM algorithms (the PIM-tree and range-partitioning) run operations in batches, and traditional indexes run them directly with multi-threaded parallelism. In all experiments, the sizes of both keys and values are set to 8 bytes.

To study the algorithms, we measure both the time spent, and the memory bus traffic. Memory bus traffic is measured by adding CPU-PIM and CPU-DRAM communication, the prior one measured by a counter increased whenever a PIM function(e.g., PIM_Broadcast) is called, and the later one measured as cache misses by PAPI. We bind the program to a single NUMA node and disable the CPU-PIM pipeline when measuring cache misses for an accurate traffic measurement. As each CPU of the PIM-equipped machine has two NUMA nodes, the effective cache of the PIM algorithms is reduced to 11 MB, half of the full cache, under this setup. Time is measured with full interleave over all NUMA nodes.

Instability in performance exists in the current generation of PIM hardwares. We oberserved an approximately ±15% fluctuation in the measured metrics mentioned above in our experiments.

6.2 Microbenchmarks

Workload Setup. Each test first warms up the index by inserting 500 million uniform random key-value pairs;[3] then for testing it executes (i) 100 million point operations or (ii) 1 million SCAN operations that each retrieve 100 elements in expectation. Point operations use batch size $S = 1$ million, and SCAN operations use batch size $S = 10$ thousand.

We generate skewed workloads with Zipfian distribution [47]. However, workloads generated by Zipf-skew over elements is not ideal for evaluating batch-parallel ordered indexes, because this skew can be easily handled by a deduplication in preprocessing on the CPU side, by merging operations of the same key into one. To better represent the spatial bias, where keys in some ranges are more likely to be accessed in the same batch, we slightly modify the way to generate our Zipfian workload, as follows: (i) we divide the key space evenly into $P = 2048$ parts; (ii) for each operation, we first choose a part according to the Zipfian distribution, then choose a uniformly random element in that part. For operations for existing keys (GET, DELETE), we divide and choose among the keys currently in the index; for operations on arbitrary keys (PREDECESSOR, INSERT, SCAN), the key space consists of all 64-bit integers. We periodically shuffle the probability of each part in Zipfian distribution. This helps alleviate, but not eliminate the PIM memory overflow problem of the range-partitioned baseline caused by INSERTs accumulating in high-probability parts. PIM-tree gains no benefit from this shuffle.

[3]This is a favorable setting for the range-partitioned baseline, because the range boundaries are stable. The performance of the PIM-tree is not impacted by the distribution of key-value pairs over time.

To show results on different amounts of skew, we evaluate the algorithms on different α values in the Zipfian distribution, ranging from 0 (uniformly random) to 1.2. With this skewness generation approach, less than 10% operations are eliminated because of duplicated keys, under the most skewed case ($\alpha = 1.2$).

Performance. Figure 8 illustrates the throughput of the range-partitioned skip list and the PIM-tree on microbenchmarks. The performance of the range-partitioned baseline drops drastically as the query skew increases, while the PIM-tree shows robust resistance to query skew. In fact, across all operations, it is observed that PIM-tree is essentially unaffected by data skew, obtaining similar running times for $\alpha = 0$ and $\alpha = 1.2$. For $\alpha = 1.2$, PIM-tree outperforms the range-partitioned baseline by 3.87–59.1×.

It is observed that GET operations are significantly simpler and achieve higher throughput since a hash table is used as a shortcut (the same holds for UPDATE operations), whereas PREDECESSOR and SCAN operations must go through the entire ordered index. In Figure 8(c), INSERT on the range-partitioned baseline crashes when $\alpha = 1.2$, because skewed INSERT causes imbalanced data placement over PIM modules, then causes overflow of local memory on some PIM modules. Although this problem could be solved by a rebalancing scheme, the rebalancing process itself will cause load imbalance as it requires sending data from the overflowing PIM modules to other less-loaded PIM modules. It is observed that even if this improvement to the baseline (with which the existing range-partitioning solutions in the literature are not equipped) were to be made, the throughput of PIM-tree would still be significantly larger than range-partitioning, as this would be extra work that the baseline must perform during the execution.

Figure 9 shows the performance of PIM-tree compared with state-of-the-art binary search tree [9] and (a,b)-tree [11] under our workload. PIM-tree outperforms traditional indexes in all test cases, except throughput of PREDECESSOR compared with (a,b)-tree.

Execution breakdown. Figure 10 shows the percentage of time spent in each component mentioned in Section 5. These results are derived with our pipelining optimization turned off, because pipelining would cause an overlapping of different components. We select as typical examples the throughput of PREDECESSOR and INSERT, on range-partitioned skip list and the PIM-tree, for uniform random workload and Zipfian-skewed workload with $\alpha = 0.6$. Similar results also exist in the cases of other α values.

For range-partitioned skip list, *PIM Execution* and *CPU-PIM Communication* dominates the time cost of skewed workloads, mainly because the bottleneck of the entire execution—the busiest PIM modules— are receiving a growing number of tasks.

For uniform random workloads, *PIM Execution* only takes a small proportion of the total time cost, though almost all comparisons are executed in PIM modules. It is inferred that parallelism is fully exploited when a large number of PIM modules are involved in this case. We believe that this implies that PIM-based systems are an ideal platform for parallel index structures.

PIM-tree INSERT spends time loading PIM program modules during execution, as the full program size exceeds the current size of instruction memory on PIMs intended to store the PIM program. This limit is discussed in Section 5.

Effect of Optimizations. Figure 11 shows the impact of different

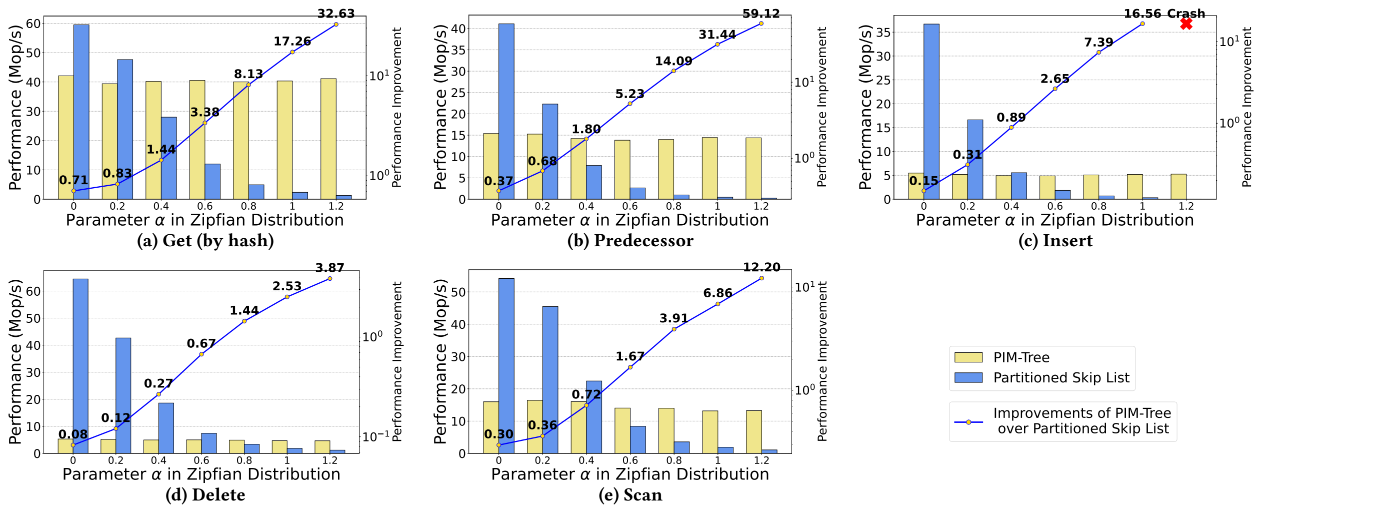


Figure 8: Throughput performance of ordered index operations. All operations other than Scan are run using a batch size of 10^6; Scan uses a batch size of 10^4, with 100 elements retrieved by each Scan operation in expectation.

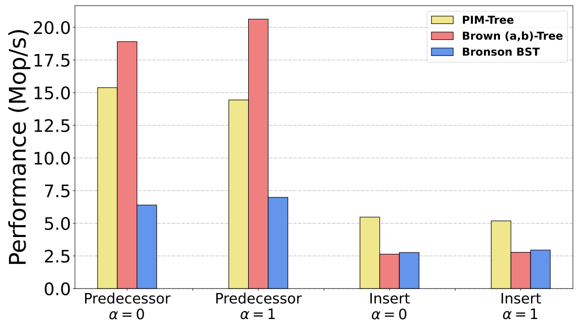


Figure 9: Throughput of PIM-tree versus SOTA traditional indexes.

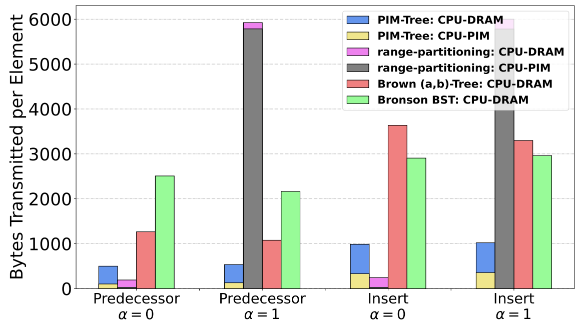


Figure 12: Average communication on the memory bus per operation in bytes.

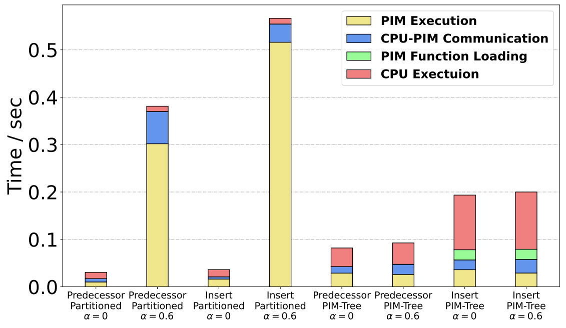


Figure 10: Time spent on each component of the program (without pipelining).

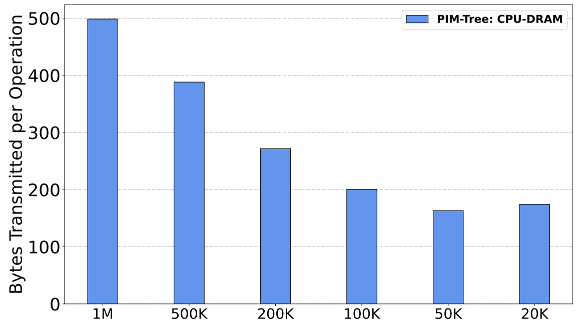


Figure 13: Average CPU-DRAM communication per predecessor operation for PIM-Tree with different batch size in bytes.

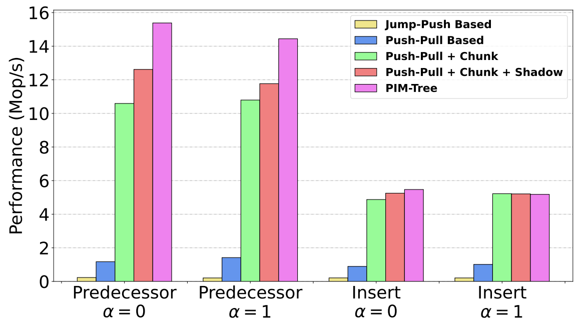


Figure 11: Impact of three optimizations on final performance.

optimizations on our ordered index. Here, we start with our Jump-Push baseline (the PIM-balance skip list in [24]). Replacing Jump-Push with Push-Pull provides up to 6.8× throughput improvements across all test cases. Adding Chunking provides the biggest improvement jump, up to 9.0×, across all test cases, while adding shadow subtrees mostly benefits Predecessor under no skew. (Insert get minor benefits because it needs to maintain this supplementary data structure.) Finally, adding pipelining—thereby implementing the complete PIM-tree algorithm—provides additional benefit for Predecessor. (Pipelining is not implemented for Insert because it would require interleaved Insert batches, which is not supported in our implementation.) Compared to the Jump-Push baseline, PIM-trees are up to 69.7× higher throughput for the settings studied.

Memory bus communication. Figure 12 shows the average amount of communication for PIM-tree, range-partitioned skip list, and traditional non-PIM indexes. PIM-tree needs less communication than all traditional indexes. Range partitioned skip lists outperform all

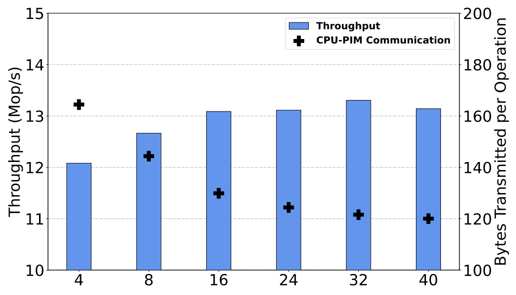


Figure 14: PIM-tree PREDECESSOR performance under different Push-Pull threshold factor k with $\alpha = 1$.

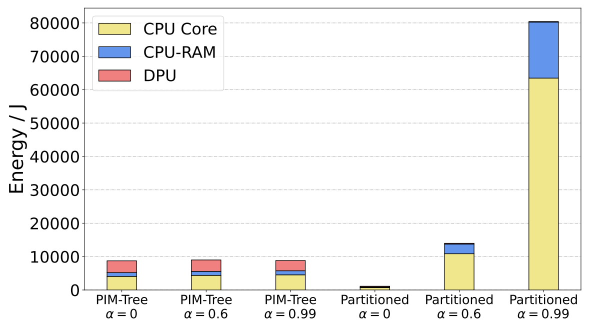
competitors by much under uniform random workload, but perform much worse in skewed workloads.

Another observation is that, while the PIM-tree stores all the data and does most comparisons in PIM modules, most memory bus traffic is between CPU and the DRAM. This is because though PIM-tree algorithms requires $O(S)$ CPU-side memory for a batch of S operations, the available setup with 11 MB cache is too small for batches of one million operations. As the result, CPU side data overflow to DRAM and cause significant CPU-DRAM communication. To show the effect of this overflow, in Figure 13 we study the CPU side communication as we run the 100 million uniform random predecessor operations with different batch sizes. Results show that the CPU-DRAM communication is reduced by 67% as we reduce batch size from 1M to 50K. We cannot directly use smaller batch size because of the load balance requirements, but this result hints that we can get much less CPU-DRAM communication when running the PIM-tree on a machine with larger cache size.

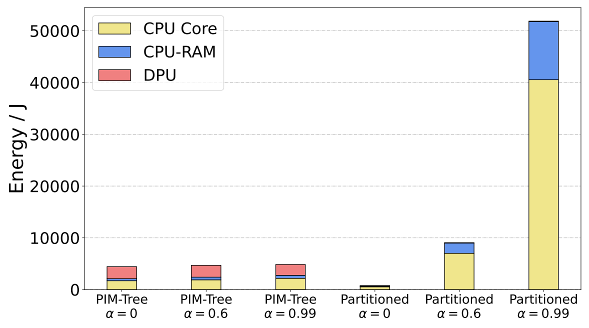
Push-Pull threshold choice Figure 14 study the PIM-tree PREDECESSOR performance under different Push-Pull threshold factor k. Recall that we use different threshold in L1 and L2: the threshold of the Push-Pull rounds in L1 is $K = k$, and the threshold of the Pull-Only round in L2 is $K = H'_{L2} * k$. We run this experiment on the PREDECESSOR workload of our microbenchmark with $\alpha = 1$. Results show that choosing a lower threshold leads to about 10% throughput drop and up to 28% more CPU-PIM communication. A threshold factor higher than B brings minor performance increase.

We believe that this improvement comes from the chunked skip list structure. If the size of each node is strictly $B = 16$, choosing threshold factor $k = 16$ will be optimal. However, the chunked skip list nodes could have larger sizes. According to the geometric distribution, there will be large nodes with more than 200 keys when the PIM-tree has millions of nodes. Larger nodes have both higher Pull cost and probability: they covers a larger key range, which collects more queries. Meanwhile, smaller nodes have both lower Pull cost and probability. Increasing the threshold factor can help alleviate this effect.

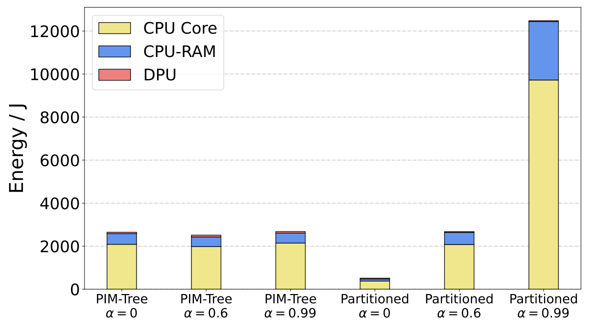
Energy Evaluation. We also evaluate the energy consumption of PIM-tree versus the range-partitioned baseline. The energy evaluation is carried out separately for the CPU host and the PIM modules. Intel RAPL [25] is used to collect CPU energy consumption statistics. Meanwhile, the number of instructions, cycles, WRAM accesses, and MRAM accesses are collected on the UPMEM PIM modules (called *DPUs*). WRAM and MRAM are the two types of memories on



(a) 100M INSERT operations



(b) 100M PREDECESSOR operations



(c) 1M SCAN operations with size 100

Figure 15: Energy Cost of operations on PIM-tree and range-partitioned skip list.

each DPU. We estimate the DPU energy consumption by multiplying these values by the hardware-related energy weights provided by UPMEM [39], and summing the result.

Figure 15 illustrates the energy consumption of PIM-tree and range-partitioned skip list, on the critical operations of INSERT, PREDECESSOR and SCAN with the same element size per batch on a same dataset. The figure shows an energy break down into three components: *CPU Core*, *CPU-RAM* (i.e., CPU-PIM communication and CPU-DRAM communication), and *DPU* (i.e., executing PIM programs). Overall, PIM-tree sacrifices roughly 1.5×–2× energy consumption on unskewed cases, in return for roughly 5×–10× energy reduction on skewed cases.

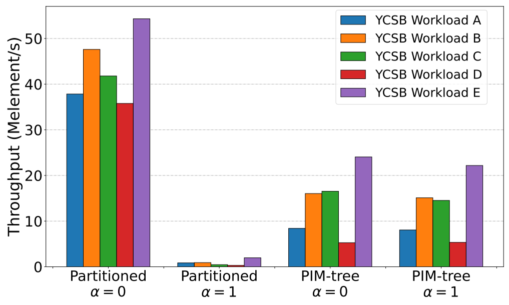


Figure 16: YCSB workload throughput.

Two additional findings can be drawn from this evaluation. First, DPU energy consumption is relatively stable against skew in all designs. This is because, under the assumption of DPU energy evaluation provided by UPMEM, an energy-efficient DPU can be turned on only when it is called for a task and turned off as soon as it returns the results to the CPU side. Therefore, the DPU energy consumption is positively correlated only to the number of executed operations. Meanwhile, even in the skewed cases, the total number of operations required to execute is approximately the work of these parallel operation batches and remains relatively constant.

Second, the energy consumption of the CPU and CPU-PIM communication is highly sensitive to skew in the range-partitioned baseline, while robust to skew in PIM-tree. We found on microbenchmarks that CPU energy consumption is strictly linearly proportional to the operating time, regardless of PIM data structure designs and operation batch types. One possible explanation is that all these designs assure that the CPU runs on its maximum capacity of processing and data communication, so both required time and energy consumption are proportional to the number of required CPU operations. Under such explanation, we suggest that the energy inefficiency on CPU of the range-partitioned baseline under skewed cases is equivalent to the time inefficiency.

We argue from the above analysis that the PIM-tree is a highly energy-efficient design in skewed cases, with the cost of only a little more energy consumption in unskewed cases (used for more complicated data structure construction).

6.3 YCSB Workload

Finally, we test our indexes using five YCSB-like [14] workloads:

A write-intensive (50% Predecessor and 50% Insert)
B read-intensive (95% Predecessor and 5% Insert)
C Predecessor-only
D Insert-only
E short-ranges (95% Scan and 5% Insert)

We use the method in §6.2 to generate zipfian-skewed operations with $\alpha = 0, 1$. All workloads are warmed up with 500 million elements. Workloads A–D are tested with 100 million operations. Workload E is tested with 20 million operations, because each Scan range is expected to return 100 elements, so the total number of tested elements is enough for the analysis. For workloads with mixed-type operations (A, B, E), we load operations into separate same-type batches, and run a batch atomically when its expected returned size exceeds 1 million [38].

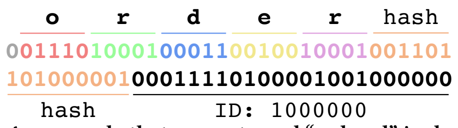


Figure 17: An example that convert word "ordered" in document id "1000000" to a 64-bit integer.

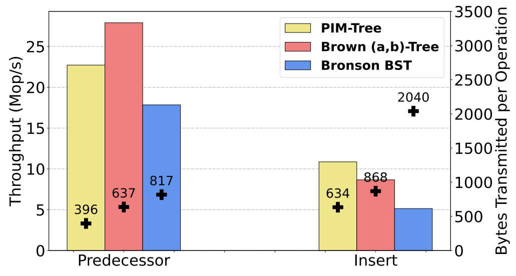


Figure 18: Throughput on the wikipedia workload.

The results are shown in Figure 16, and again show the fragility of the range-partitioned skip list and the robustness of PIM-tree under skewed workloads.

6.4 Workload of Real-world Skewness

In this section, we test the PIM-tree over a workload with real-world skewness using the publicly available wikipedia dataset [15], which is a collection of documents from wikipedia. To use this dataset in our test framework, we need to transform it into a collection of 8 byte key-value pairs, then run operations over them. To be specific, we first extract words from each document, lowercase them, then use (word, document id) pairs as keys, and a random 8 byte integer as values. Because our indexes only support 8 byte integer keys, we need to transform the (word, document id) pairs into 8 byte integers. The transformation is shown in Figure 17. We use 40 bits to represent the word, and 23 bits to store the document id (as there's less than 2^{23} documents). In the word part, we use 5 bits for each of the first 5 letters, then store the hash value of the whole word in the following 15 bits to avoid collision. After this transformation, the generated integers preserve the two skewness of english words: (i) word frequency skewness (some words are used more than others) and (ii) word distribution skewness in the dictionary order space (words with some prefix are used more).

In this test, we pick the first 1.2 billion keys: the first 1 billion words used for initialization, and the following 200 million used for evaluation. These keys covers the first 5.1 million documents, which is 63% of the whole dataset. There are 3.9 million unique words, and pairing the word and document id generates 529 million unique keys. We get duplicated keys only for the same word in the same document. Because the duplication rate of keys is about $2X$, we also double the batch size of the PIM-tree to two million.

The result is shown in Figure 18, where the throughputs are shown as the bars, and communication (bytes transmitted per-operation) as labeled points. All indexes experience higher throughput and lower communication in this workload than in microbenchmarks because of the replicated keys. Comparing different indexes gives results similar to that of microbenchmarks: PIM-tree has lower predecessor throughput than the (a,b)-tree, but outperforms traditional indexes in all other metrics.

7 DISCUSSION

PIM-tree outperforms conventional indexes in throughput in most cases, but very occasionally cannot win, e.g. only in PREDECESSOR compared with (a,b)-tree in our paper. We address here three hardware limits of the current PIM system by way of explanation, and to describe future changes to the hardware that would result in even better performance for PIM-optimized data structures.

The first factor is the limited CPU-PIM bandwidth on UPMEM's newly developed hardware. When carrying out a 50% read - 50% write task, the bandwidth obtained on UPMEM machine is 16GB/s, 1.9× slower than the shared-memory machine we use with a bandwidth of 31GB/s on the same workload. Even under such significant bandwidth limitations, PIM-tree still achieves better or comparable performance to DRAM-only indexes, primarily because it greatly reduces inter-module communication. Designing hardware to improve CPU-PIM bandwidth is thus an important direction, and one that we expect improvements for in the future. Therefore, we believe that PIM-tree will outperform conventional indexes in all cases in terms of throughput in the future.

Another issue is that the limited size of PIM program prevent us from more complicated designs. Current workaround, the dynamic program loading is too costly. We believe this problem will be solved in future hardware by a larger instruction memory, in other ways.

The last limit is that of inadequate CPU cache, as mentioned in Section 6.2. CPU-DRAM communication caused by cache overflow makes most of memory bus communication, and this can be alleviated by a larger cache. We believe an adequate cache will be important in future PIM systems.

8 CONCLUSION

This paper presented *PIM-tree*, the first ordered index for PIM systems that achieves both low communication and high load balance in the presence of data and query skew. We presented the first experimental evaluation of ordered indexes on a real PIM system, demonstrating up to 69.7× and 59.1× higher throughput than the two best prior PIM-based methods and down to 0.4× less communication than two state-of-the-art conventional indexes. Key ideas include *push-pull search* and *shadow subtrees*—techniques likely to be useful for other applications on PIM systems due to their effectiveness in reducing communication costs and managing skew. Our future work will explore such applications (e.g., radix-based indexes, graph analytics).

ACKNOWLEDGMENTS

We thank Rémy Cimadomo, Julien Legriel, Damien Lagneux, and all the other folks at UPMEM for providing extensive access to the UPMEM system and help whenever needed. This work would not have been possible without this level of support. This research was supported by NSF grants CCF-1910030, CCF-1919223, CCF-2028949, and CCF-2103483, VMware University Research Fund Award, Parallel Data Lab (PDL) Consortium (Alibaba, Amazon, Datrium, Facebook, Google, Hewlett-Packard Enterprise, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Salesforce, Samsung, Seagate, and TwoSigma) and National Key Research & Development Program of China (2020YFC1522702).

REFERENCES

[1] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA). 105–117. https://doi.org/10.1145/2749469.2750386

[2] Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. 2020. PIM-Assembler: A Processing-in-Memory Platform for Genome Assembly. In 2020 57th ACM/IEEE Design Automation Conference (DAC). 1–6. https://doi.org/10.1109/DAC18072.2020.9218653

[3] Shaahin Angizi, Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. 2018. CMP-PIM: An Energy-Efficient Comparator-Based Processing-in-Memory Neural Network Accelerator. In Proceedings of the 55th Annual Design Automation Conference (San Francisco, California) (DAC '18). Association for Computing Machinery, New York, NY, USA, Article 105, 6 pages. https://doi.org/10.1145/3195970.3196009

[4] Md Tanvir Arafin and Zhaojun Lu. 2020. Security Challenges of Processing-In-Memory Systems. In Proceedings of the 2020 on Great Lakes Symposium on VLSI (Virtual Event, China) (GLSVLSI '20). Association for Computing Machinery, New York, NY, USA, 229–234. https://doi.org/10.1145/3386263.3411365

[5] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. 2018. Getting to the Root of Concurrent Binary Search Tree Performance. In 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 295–306. https://www.usenix.org/conference/atc18/presentation/arbel-raviv

[6] Petra Berenbrink, Tom Friedetzky, Zengjian Hu, and Russell Martin. 2008. On Weighted Balls-into-bins Games. Theoretical Computer Science 409, 3 (2008), 511–520.

[7] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Rachata Ausavarungnirun, Kevin Hsieh, Nastaran Hajinazar, Krishna T. Malladi, Hongzhong Zheng, and Onur Mutlu. 2019. CoNDA: Efficient Cache Coherence Support for near-Data Accelerators. In Proceedings of the 46th International Symposium on Computer Architecture (Phoenix, Arizona) (ISCA '19). Association for Computing Machinery, New York, NY, USA, 629–642. https://doi.org/10.1145/3307650.3322266

[8] Amirali Boroumand, Saugata Ghose, Minesh Patel, Hasan Hassan, Brandon Lucia, Kevin Hsieh, Krishna T Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. IEEE Computer Architecture Letters 16, 1 (2016), 46–50.

[9] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010, R. Govindarajan, David A. Padua, and Mary W. Hall (Eds.). ACM, 257–268. https://doi.org/10.1145/1693453.1693488

[10] Daniel G Brown. 2011. How I wasted too long finding a concentration inequality for sums of geometric variables. (2011).

[11] Trevor Brown. 2017. Techniques for constructing efficient lock-free data structures. Ph.D. Dissertation. University of Toronto (Canada).

[12] Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 297–308.

[13] Douglas Comer. 1979. Ubiquitous B-tree. ACM Computing Surveys (CSUR) 11, 2 (1979), 121–137.

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In Proceedings of the 1st ACM symposium on Cloud computing. 143–154.

[15] Wikimedia Foundation. 2016. Wikipedia:Database download. https://en.wikipedia.org/wiki/Wikipedia:Database_download. Accessed March 15, 2022.

[16] Christina Giannoula, Ivan Fernandez, Juan Gómez Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. Proc. ACM Meas. Anal. Comput. Syst. 6, 1, Article 21 (feb 2022), 49 pages. https://doi.org/10.1145/3508041

[17] Christina Giannoula, Nandita Vijaykumar, Nikela Papadopoulou, Vasileios Karakostas, Ivan Fernandez, Juan Gómez-Luna, Lois Orosa, Nectarios Koziris, Georgios I. Goumas, and Onur Mutlu. 2021. SynCron: Efficient Synchronization Support for Near-Data-Processing Architectures. In IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021. IEEE, 263–276. https://doi.org/10.1109/HPCA51647.2021.00031

[18] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F Oliveira, and Onur Mutlu. 2021. Benchmarking a new paradigm: An experimental analysis of a real processing-in-memory architecture. arXiv preprint arXiv:2105.03814 (2021).

[19] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. 1999. Using MPI: portable parallel programming with the message-passing interface. Vol. 1. MIT press.

[20] Peng Gu, Shuangchen Li, Dylan Stow, Russell Barnes, Liu Liu, Yuan Xie, and Eren Kursun. 2016. Leveraging 3D technologies for hardware security: Opportunities and challenges. In 2016 International Great Lakes Symposium on VLSI (GLSVLSI). IEEE, 347–352.

[21] Saransh Gupta and Tajana Šimunić Rosing. 2021. Invited: Accelerating Fully Homomorphic Encryption with Processing in Memory. In 2021 58th ACM/IEEE Design Automation Conference (DAC). 1335–1338. https://doi.org/10.1109/DAC18074.2021.9586285

[22] Yu Huang, Long Zheng, Pengcheng Yao, Jieshan Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A Heterogeneous PIM Hardware-Software Co-Design for Energy-Efficient Graph Processing. In 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 684–695. https://doi.org/10.1109/IPDPS47924.2020.00076

[23] Joe Jeddeloh and Brent Keeth. 2012. Hybrid memory cube new DRAM architecture increases density and performance. In 2012 Symposium on VLSI Technology (VLSIT). 87–88. https://doi.org/10.1109/VLSIT.2012.6242474

[24] Hongbo Kang, Phillip B Gibbons, Guy E Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures. 295–306.

[25] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. ACM Trans. Model. Perform. Eval. Comput. Syst. 3, 2, Article 9 (Mar 2018).

[26] Ji-Hoon Kim, Juhyoung Lee, Jinsu Lee, Jaehoon Heo, and Joo-Young Kim. 2021. Z-PIM: A Sparsity-Aware Processing-in-Memory Architecture With Fully Variable Weight Bit-Precision for Energy-Efficient Deep Neural Networks. IEEE Journal of Solid-State Circuits 56, 4 (2021), 1093–1104. https://doi.org/10.1109/JSSC.2020.3039206

[27] Dong Uk Lee, Kyung Whan Kim, Kwan Weon Kim, Hongjung Kim, Ju Young Kim, Young Jun Park, Jae Hwan Kim, Dae Suk Kim, Heat Bit Park, Jin Wook Shin, Jang Hwan Cho, Ki Hun Kwon, Min Jeong Kim, Jaejin Lee, Kun Woo Park, Byongtae Chung, and Sungjoo Hong. 2014. 25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV. In 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC). 432–433. https://doi.org/10.1109/ISSCC.2014.6757501

[28] Wen Li, Ying Wang, Huawei Li, and Xiaowei Li. 2019. P<sup>3</sup>M: A PIM-Based Neural Network Model Protection Scheme for Deep Learning Accelerator (ASPDAC '19). Association for Computing Machinery, New York, NY, USA, 633–638. https://doi.org/10.1145/3287624.3287695

[29] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-memory Computing. In ACM Symposium on Parallelism in Algorithms and Architectures (SPAA). 235–245.

[30] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2020. A Modern Primer on Processing in Memory. CoRR abs/2012.03112 (2020).

[31] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). 457–468. https://doi.org/10.1109/HPCA.2017.54

[32] Newton, Virendra Singh, and Trevor E. Carlson. 2020. PIM-GraphSCC: PIM-Based Graph Processing Using Graph's Community Structures. IEEE Comput. Archit. Lett. 19, 2 (jul 2020), 151–154. https://doi.org/10.1109/LCA.2020.3039498

[33] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 62–73. http://www.vldb.org/conf/1998/p062.pdf

[34] Peter Sanders. 1996. On the Competitive Analysis of Randomized Static Load Balancing. In Workshop on Randomized Parallel Algorithms (RANDOM).

[35] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. Proceedings of the VLDB Endowment 4, 11 (2011), 795–806.

[36] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). 531–543. https://doi.org/10.1109/HPCA.2018.00052

[37] Harold S. Stone. 1970. A Logic-in-Memory Computer. IEEE Trans. Comput. C-19, 1 (1970), 73–78.

[38] Yihan Sun, Guy E Blelloch, Wan Shen Lim, and Andrew Pavlo. 2019. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. 13, 2 (2019), 211–225.

[39] UPMEM. 2022. UPMEM Technology. https://www.upmem.com/technology/. Accessed March 15, 2022.

[40] Leslie G Valiant. 1990. A bridging model for parallel computation. Commun. ACM 33, 8 (1990), 103–111.

[41] Zhao Wang, Yijin Guan, Guangyu Sun, Dimin Niu, Yuhao Wang, Hongzhong Zheng, and Yinhe Han. 2020. GNN-PIM: A Processing-in-Memory Architecture for Graph Neural Networks. In Advanced Computer Architecture, Dezun Dong, Xiaoli Gong, Cunlu Li, Dongsheng Li, and Junjie Wu (Eds.). Springer Singapore, Singapore, 73–86.

[42] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. 2021. SpaceA: Sparse Matrix Vector Multiplication on Processing-in-Memory Accelerator. In 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA). 570–583. https://doi.org/10.1109/HPCA51647.2021.00055

[43] Fan Zhang, Shaahin Angizi, Naima Ahmed Fahmi, Wei Zhang, and Deliang Fan. 2021. PIM-Quantifier: A Processing-in-Memory Platform for mRNA Quantification. In 2021 58th ACM/IEEE Design Automation Conference (DAC). 43–48. https://doi.org/10.1109/DAC18074.2021.9586144

[44] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). 544–557. https://doi.org/10.1109/HPCA.2018.00053

[45] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (Columbus, OH, USA) (MICRO '52). Association for Computing Machinery, New York, NY, USA, 712–725. https://doi.org/10.1145/3352460.3358256

[46] Tobias Ziegler, Sumukha Tumkur Vani, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In ACM SIGMOD International Conference on Management of Data. 741–758.

[47] George Kingsley Zipf. 2016. Human behavior and the principle of least effort: An introduction to human ecology. Ravenio Books.