Concurrent Data Structures for Near-Memory Computing

Zhiyu Liu
Computer Science Department
Brown University
zhiyu_liu@brown.edu

Irina Calciu
VMware Research Group
icalciu@vmware.com

Maurice Herlihy
Computer Science Department
Brown University
mph@cs.brown.edu

Onur Mutlu
Computer Science Department
ETH Zurich
onur.mutlu@inf.ethz.ch

ABSTRACT

The performance gap between memory and CPU has grown exponentially. To bridge this gap, hardware architects have proposed near-memory computing (also called processing-in-memory, or PIM), where a lightweight processor (called a PIM core) is located close to memory. Due to its proximity to memory, a memory access from a PIM core is much faster than from a CPU core. New advances in 3D integration and in die-stacked memory make PIM viable in the near future. Prior work has shown significant performance improvements by using PIM for embarrassingly parallel and data-intensive applications, as well as for pointer-chasing traversals in *sequential* data structures. However, current server machines have hundreds of cores; algorithms for concurrent data structures exploit these cores to achieve high throughput and scalability, with significant benefits over sequential data structures. Unlike prior work, we focus on *concurrent* data structures for PIM and we show two main results: (1) naive PIM data structures cannot outperform state-of-the-art concurrent data structures such as pointer-chasing data structures and FIFO queues, (2) novel designs for PIM data structures, using techniques such as combining, partitioning and pipelining, can outperform traditional concurrent data structures, with a significantly simpler design.

KEYWORDS

concurrent data structures; parallel programs; processing-in-memory; near-memory computing

ACM Reference format:
Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of SPAA '17, Washington DC, USA, July 24-26, 2017,* 11 pages.
DOI: http://dx.doi.org/10.1145/3087556.3087582

1 NEAR-MEMORY COMPUTING

The performance gap between memory and CPU has grown exponentially. Memory vendors have focused mainly on improving memory capacity and bandwidth, sometimes even at the cost of increased memory access latencies [10]. To provide higher bandwidth with lower access latencies, hardware architects have proposed near-memory computing (also called *processing-in-memory*, or PIM), where a lightweight processor (called a PIM core) is located close to memory. A memory access from a PIM core is much faster than from a CPU core. Near-memory computing is an old idea, that has been intensely studied in the past (e.g., [13, 16, 17, 27, 29, 32, 33, 37]), but so far has not yet materialized. However, new advances in 3D integration and in die-stacked memory likely make near-memory computing viable in the near future. For example, one PIM design [1, 2, 39] assumes that memory is organized in multiple vaults, each having an in-order PIM core to manage it. These PIM corescan communicate through message passing, but do not share memory, and cannot access each other's vaults.

This new technology promises to revolutionize the interaction between computation and data, as it enables memory to become an active component in managing the data. Therefore, it invites a fundamental rethinking of basic data structures and promotes a tighter dependency between algorithmic design and hardware characteristics.

Prior work has already shown significant performance improvements by using PIM for embarrassingly parallel and data-intensive applications [1, 3, 24, 39, 40], as well as for pointer-chasing traversals [25] in *sequential* data structures. However, current server machines have hundreds of cores; algorithms for concurrent data structures exploit these cores to achieve high throughput and scalability, with significant benefits over sequential data structures (e.g., [15, 22, 34, 38]). Unlike prior work, we focus on *concurrent* data structures for PIM and we show that naive PIM data structures cannot outperform state-of-the-art concurrent data structures. In particular, the lower-latency access to memory provided by PIM cannot compensate for the loss of parallelism. For example, we show that if a CPU core takes 2X time to access memory than a PIM core, a PIM-managed linked-list is still slower than a concurrent linked-list serving requests in parallel by only three CPU cores. To be competitive with traditional concurrent data structures, PIM data structures need new algorithms and new approaches to leverage parallelism.

But how do we design and optimize data structures for PIM? And how do these algorithms compare to traditional CPU-managed concurrent data structures? To answer these questions, even before the hardware becomes available, we develop a simplified model of

SPAA '17, Washington DC, USA

DOI: http://dx.doi.org/10.1145/3087556.3087582

the expected performance of PIM. Using this model, we investigate two classes of data structures.

First, we analyze *pointer chasing data structures* (Section 4), which have a high degree of inherent parallelism and low contention, but incur significant overhead due to hard-to-predict memory access patterns. We propose using techniques such as combining and partitioning the data across vaults to reintroduce parallelism for these data structures.

Second, we explore *contended data structures* (Section 5), such as FIFO queues, which can leverage CPU caches to exploit their inherent high locality. Therefore, FIFO queues might not seem to be able to leverage PIM's faster memory accesses. Nevertheless, these data structures exhibit a high degree of contention, which makes it difficult even for the most advanced algorithms to obtain good performance when many threads access the data concurrently. We use pipelining of requests, which can be done very efficiently in PIM, to design a new FIFO queue suitable for PIM that can outperform state-of-the-art concurrent FIFO queues [20, 31].

The contributions of this paper are as follows:

- We propose a simple and intuitive model to analyze performance of PIM data structures and concurrent data structures based on the latency of a memory access and an estimated number of accesses served from the cache, as well as the number of atomic operations used.
- Using this model, we show that the lower-latency memory accesses provided by PIM are not sufficient for PIM data structures to outperform efficient concurrent algorithms[1].
- We propose new designs for PIM data structures using techniques such as combining, partitioning and pipelining. Our evaluations show that these new PIM data structures can outperform traditional concurrent data structures, with a significantly simpler design.

The paper is organized as follows. In Section 2 we briefly describe our assumptions about the hardware architecture. In Section 3 we introduce a simplified performance model that we use throughout this paper to predict performance of our algorithms using the hardware architecture described in Section 2. Next, in Sections 4 and 5, we describe and analyze our PIM data structures and use our model to compare them to prior work. We also use current DRAM architectures to simulate the behavior of our algorithms and evaluate compared to state-of-the-art concurrent data structure algorithms. Finally, we present related work in Section 6 and conclude in Section 7.

2 HARDWARE ARCHITECTURE

In an example architecture utilizing PIM memory [1, 2, 39], multiple CPUs are connected to the main memory, via a shared crossbar network, as illustrated in Figure 1. The main memory consists of two parts—one is a standard DRAM accessible by CPUs and the other, called the *PIM memory*, is divided into multiple partitions, called *PIM vaults* or simply vaults. According to the *Hybrid Memory Cube* specification 1.0 [11], each HMC consists of 16 or 32 vaults and has a total size of 2GB or 4 GB (so each vault has size roughly 100MB).[2] We assume the same specifications in our PIM model, although the size of a PIM memory and the number of its vaults can be bigger. Each CPU core also has access to a hierarchy of L1 and L2 caches backed by DRAM, and a last level cache shared among multiple cores.

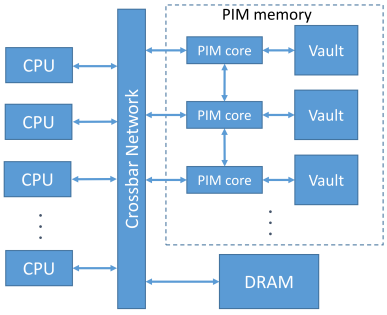

Figure 1: An example PIM architecture

Each vault has a *PIM core* directly attached to it. We say a vault is *local* to the PIM core attached to it, and vice versa. A PIM core is a lightweight CPU that may be slower than a full-fledged CPU with respect to computation speed [1]. A PIM core can be thought of as an in-order CPU with a small private L1 cache. A vault can be accessed only by its local PIM core.[3] Recent work has proposed efficient cache coherence mechanisms between PIM cores and CPUs (e.g., [2, 8]), but this introduces additional complexity. We show that we design efficient concurrent data structures even if there is no coherence. Although a PIM core has lower performance than a state-of-the-art CPU core, it has fast access to its local vault.

A PIM core communicates with other PIM cores and CPUs via messages. Each PIM core, as well as each CPU, has buffers for storing incoming messages. A message is guaranteed to eventually arrive at the buffer of its receiver. Messages from the same sender to the same receiver are delivered in FIFO order: the message sent first arrives at the receiver first. However, messages from different senders or to different receivers can arrive in an arbitrary order.

We assume that a PIM core can only perform read and write operations to its local vault, while a CPU also supports more powerful atomic operations, such as *Compare-And-Swap (CAS)* and *Fetch-And-Increment (F&A)*. Virtual memory can be realized efficiently if each PIM core maintains its own page table for the local vault [25].

3 PERFORMANCE MODEL

We propose the following simple performance model to compare our PIM-managed algorithms with existing concurrent data structure algorithms. For read and write operations, we assume

$$\mathcal{L}_{cpu} = r_1 \mathcal{L}_{pim} = r_2 \mathcal{L}_{llc}$$

where $\mathcal{L}_{cpu}$ is the latency of a memory access by a CPU, $\mathcal{L}_{pim}$ is the latency of a local memory access by a PIM core, and $\mathcal{L}_{llc}$ is the latency of a last-level cache access by a CPU. Based on the latency numbers in prior work on PIM memory, in particular on the

[1] We will use algorithms and data structures interchangeably in the rest of the paper.

[2] These small sizes are preliminary, and it is expected that each vault will be bigger when the PIM memory will be commercialized.

[3] Alternatively, we could assume that a PIM core has direct access to remote vaults, at a larger cost.

Hybrid Memory Cube [6, 11], and on the evaluation of operations in multiprocessor architectures [12], we may further assume

$$r_1 = r_2 = 3.$$

The latencies of operations may vary significantly on different machines. Our assumption that $r_1 = r_2 = 3$ is mainly to make the performance analysis later in the paper more concrete with certain latency numbers. We ignore the costs of cache accesses of other levels in our performance model, as they are negligible in the concurrent data structure algorithms we will consider.

We assume that the latency of a CPU performing an atomic operation, such as a CAS or a F&A, to a cache line is

$$\mathcal{L}_{atomic} = r_3 \mathcal{L}_{cpu}$$

where $r_3 = 1$, even if the cache line is currently in cache. This is because an atomic operation hitting in the cache is usually as costly as a memory access by a CPU [12]. When there are k atomic operations competing for a cache line concurrently, we assume that they are executed sequentially, that is, they complete in times $\mathcal{L}_{atomic}, 2\mathcal{L}_{atomic}, ..., k \cdot \mathcal{L}_{atomic}$, respectively.

We also assume that the size of a message sent by a PIM core or a CPU core is at most the size of a cache line. Given that a message transferred between a CPU and a PIM core goes through the crossbar network, we assume that the latency for a message to arrive at its receiver is

$$\mathcal{L}_{message} = \mathcal{L}_{cpu}$$

We make the conservative assumption that the latency of a message transferred between two PIM cores is also $\mathcal{L}_{message}$. Note that the message latency we consider here is the transfer time of a message through a message passing channel, that is, the elapsed time between when a PIM or a CPU core finishes sending off the message and when the message arrives at the buffer of its receiver. We ignore the time spent in other parts of a message passing procedure, such as *preprocessing and constructing the message*, and *a PIM or a CPU core sending off the message*, as it is negligible compared to the time spent in the message transfer [6].

4 LOW CONTENTION DATA STRUCTURES

In this section we consider data structures with low contention. Pointer chasing data structures, such as linked-lists and skip-lists, fall in this category. These are data structures whose operations need to de-reference a non-constant sequence of pointers before completing. We assume these data structures support operations such as add(x), delete(x) and contains(x), which follow "next node" pointers until reaching the position of node x. When these data structures are too large to fit in CPU caches and access uniformly random keys, they incur expensive memory accesses, which cannot be easily predicted, making the *pointer chasing* operations the dominating overhead of these data structures. Naturally, these data structures have provided early examples for the benefits of near-memory computing [18, 25], as the entire pointer chasing could be performed by a PIM core with fast memory access, and only the final result returned to the application.

However, these data structures have inherently low contention. Lock-free algorithms [15, 22, 34, 38] have shown that these data structures can scale to hundreds of cores under low contention [9]. Unfortunately, each vault in PIM memory has a single core. As a consequence, prior work has only compared PIM data structures with sequential data structures, not with carefully crafted concurrent data structures.

We analyze linked-lists and skip-lists, and show that the naive PIM data structure in each case cannot outperform the equivalent CPU-managed concurrent data structure even for a small number of cores. Next, we show how to use state-of-the art techniques from the concurrent computing literature to optimize algorithms for near-memory computing to outperform well-known concurrent data structures designed for multi-core CPUs.

4.1 Linked-lists

We now describe a naive PIM linked-list. The linked-list is stored in a vault, maintained by the local PIM core. Whenever a CPU[4] wants to perform an operation on the linked-list, it sends a request to the PIM core. The PIM core then retrieves the message, executes the operation, and sends the result back to the CPU. The PIM linked-list is sequential, as it can only be accessed by one PIM core.

Performing pointer chasing on sequential data structures by PIM cores is not a new idea. Prior work ([1, 18, 25]) has shown that pointer chasing can be done more efficiently by a PIM core for a sequential data structure. However, we are not aware of any prior comparison between the performance of PIM-managed data structures and concurrent data structures, for which CPUs can make operations in parallel. In fact, our analytical and experimental results show that the naive PIM-managed linked-list is not competitive with the concurrent linked-list with fine-grained locks [19].

We use the *combining optimization* proposed by flat combining [20] to improve this data structure: a PIM core can execute all concurrent requests by CPU cores using a single traversal over the linked-list.

The role of the PIM core in our PIM-managed linked-list is very similar to that of the combiner in a concurrent linked-list implemented using *flat combining* [20], where, roughly speaking, threads compete for a "combiner lock" to become the combiner, and the combiner takes over all operation requests from other threads and executes them. Therefore, we consider the performance of the flat-combining linked-list as an indicator of the performance of our PIM-managed linked-list.

Based on our performance model, we can calculate the approximate expected throughput (i.e., number of operations per second) of each of the linked-list algorithms mentioned above, when there are p CPUs making operation requests concurrently. We assume that a linked-list consists of nodes with integer keys in the range of $[1, N]$. Initially a linked-list has n nodes with keys generated independently and uniformly at random from $[1, N]$. The keys of the operation requests are generated the same way. To simplify the analysis, we assume that the size of the linked-list does not fluctuate much. This is achieved when the number of add() requests is similar to the number of delete() requests. We assume that a CPU makes a new operation request immediately after its previous

[4] We use the term CPU to refer to a CPU core, as opposed to a PIM core.

one completes. Assuming that $n \gg p$ and $N \gg p$, the approximate expected throughput (per second) of each of the concurrent linked-lists is presented in Table 1, where $\mathcal{S}_p = \sum_{i=1}^{n} (\frac{i}{n+1})^p$.

Algorithm	Throughput
Linked-list with fine-grained locks	$\frac{2p}{(n+1)\mathcal{L}_{cpu}}$
Flat-combining linked-list without combining	$\frac{2}{(n+1)\mathcal{L}_{cpu}}$
PIM-managed linked-list without combining	$\frac{2}{(n+1)\mathcal{L}_{pim}}$
Flat-combining linked-list with combining	$\frac{p}{(n-\mathcal{S}_p)\mathcal{L}_{cpu}}$
PIM-managed linked-list with combining	$\frac{p}{(n-\mathcal{S}_p)\mathcal{L}_{pim}}$

Table 1: Throughput of linked-list algorithms.

We calculate the throughput values in Table 1 in the following manner. In the linked-list with fine-grained locks, which has $(n+1)$ nodes including a dummy head node, each thread (CPU) executes its own operations to the linked-list. Since, the key of a request is generated uniformly at random, the average number of memory accesses by one thread for one operation is $(n+1)/2$ and hence the throughput of one thread is $2/((n+1)\mathcal{L}_{cpu})$. Since there are p threads running in parallel, the total throughput is $2p/((n+1)\mathcal{L}_{cpu})$. The throughput of the flat-combining and the PIM-managed linked-lists without combining optimization is calculated similarly. For the flat-combining and the PIM-managed linked-lists with combining, it suffices to prove that the average number of memory accesses by a PIM core (or a combiner) batching and executing p random operation requests in one traversal is $n - \mathcal{S}_p$, which is essentially the expected number of pointers a PIM core (or a combiner) needs to go through to reach the position for the request with the largest key among the p requests. Note that we have ignored certain communication costs incurred in some algorithms, such as the latency of a PIM core sending a result back to a waiting thread, and the latency of a combiner maintaining the combiner lock and the publication list in the flat-combining linked-list (we will discuss the publication list in more detail in Section 5), as they are negligible compared to the dominant costs of traversals over linked-lists.

It is easy to see that the PIM-managed linked-list with combining outperforms the linked-list with fine-grained locks, which is the best one among other algorithms, if $\frac{\mathcal{L}_{cpu}}{\mathcal{L}_{pim}} = r_1 > \frac{2(n-\mathcal{S}_p)}{n+1}$. Given that $0 < \mathcal{S}_p \leq \frac{n}{2}$, the PIM-managed linked-list can beat the linked-list with fine-grained locks as long as $r_1 \geq 2$. If we assume $r_1 = 3$, as estimated by prior work, the throughput of the PIM-managed linked-list with combining should be at least 1.5 times the throughput of the linked-list with fine-grained locks. Without combining, however, the PIM-managed linked-list *cannot* outperform the linked-list with fine-grained locks run by $p \geq r_1$ threads. On the other hand, the PIM-managed linked-list is expected to be r_1 times better than the flat-combining linked-list, with or without combining optimization applied to both.

We implemented the linked-list with fine-grained locks and the flat-combining linked-list with and without the combining optimization. We tested them on a Dell server with 512 GB RAM and 56 cores on four Intel Xeon E7-4850v3 processors running at 2.2 GHz. To eliminate NUMA access effects, we ran experiments with only one processor, which is a NUMA node with 14 cores, a 35 MB shared L3 cache, and a private L2/L1 cache of size 256 KB/64 KB per core. Each core has 2 hyperthreads, for a total of 28 hyperthreads.

The throughput of each of the algorithms is presented in Figure 2. The results confirmed the validity of our analysis in Table 1. The throughput of the flat-combining algorithm without the combining optimization is worse than the algorithm with fine-grained locks. Since the performance of the flat-combining linked-list is a good indicator of the performance of the PIM-managed linked-list, we triple the throughput of the flat-combining linked-list to obtain the expected throughput of the PIM-managed linked-list, based on the assumption that $r_1 = 3$. As we can see, it is still below the throughput of the one with fined-grained locks. However, with the combining optimization, the performance of the flat-combining algorithm improves significantly and the estimated throughput of our PIM-managed linked-list with combining optimization now outperforms the throughput of all other data structures.

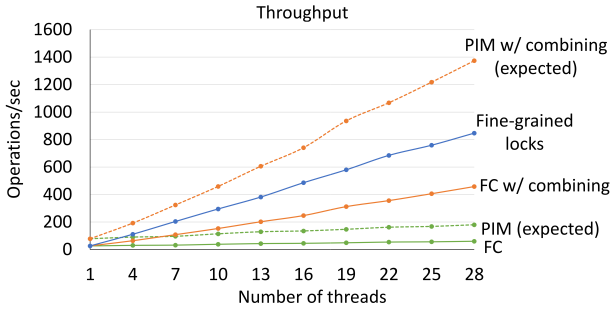

Figure 2: Experimental results of linked-lists. We evaluate the linked-list with fine-grained locks and the flat-combining linked-list (FC) with and without the combining optimization.

4.2 Skip-lists

Like the naive PIM-managed linked-list, the naive PIM-managed skip-list keeps the skip-list in a single vault and CPUs send operation requests to the local PIM core that executes those operations. As we will see, this algorithm is less efficient than some existing algorithms.

Unfortunately, the combining optimization *cannot* be applied to skip-lists effectively. The reason is that for any two nodes not close enough to each other in the skip-list, the paths threads traverse to reach such nodes do not largely overlap.

On the other hand, PIM memory usually consists of many vaults and PIM cores. For instance, the first generation of Hybrid Memory Cube [11] has up to 32 vaults. Hence, a PIM-managed skip-list can achieve much better performance if we can exploit the parallelism of multiple vaults. Here we present our PIM-managed skip-list with a *partitioning optimization*: A skip-list is divided into partitions of

disjoint ranges of keys, stored in different vaults, so that a CPU sends its operation request to the PIM core of the vault to which the key of the operation belongs.

Figure 3 illustrates the structure of a PIM-managed skip-list. Each partition of a skip-list starts with a *sentinel node* which is a node of the max height. For simplicity, assume the max height H_{max} is predefined. A partition covers a key range between the key of its sentinel node and the key of the sentinel node of the next partition. CPUs also store a copy of each sentinel node in the normal DRAM and this copy has an extra variable indicating the vault containing the sentinel node. Since the number of nodes of the max height is very small with high probability, those copies of those sentinel nodes very likely stay in cache if CPUs access them frequently enough.

When a CPU performs an operation for a key to the skip-list, it first compares the key with those of the sentinels, discovers which vault the key belongs to, and then sends its operation request to that vault's PIM core. Once the PIM core retrieves the request, it executes the operation in the local vault and sends the result back to the CPU.

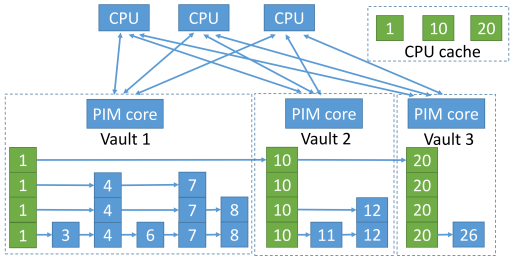

Figure 3: A PIM-managed skip-list with three partitions

We now discuss how we implement the PIM-managed skip-list when the key of each operation is an integer generated uniformly at random from range $[0, n]$ and the PIM memory has k vaults available. Initially we can create k partitions starting with fake sentinel nodes with keys 0, $1/k$, $2/k$,..., $(n-1)/k$, respectively, and allocate each partition in a different vault. The sentinel nodes are never deleted. If a new node to be added has the same key as a sentinel node, we insert it immediately after the sentinel node.

We compare the performance of our PIM-managed skip-list with partitions to the performance of a flat-combining skip-list [20] and a lock-free skip-list [22], where p CPUs keeps making operation requests. We also apply the partitioning optimization to the flat-combining skip-list, so that k combiners are in charge of k partitions of the skip-list. To simplify the comparison, we assume that all skip-lists have the same initial structure (expect that skip-lists with partitions have extra sentinel nodes) and all the operations are contains() operations (or the number of add() requests is the same as the number of delete() requests such that the size of each skip-list nearly doesn't change). The keys of requests are generated uniformly at random.

The approximate throughput of each of these skip-lists is presented in Table 2, where β is the average number of nodes an operation has to go through in order to find the location of its key in a skip-list ($\beta = \Theta(\log N)$, where N is the size of the skip-list). In the lock-free skip-list, p threads execute their own operations in parallel, so the throughput is roughly $p/(\beta \mathcal{L}_{cpu})$. Without the partitioning optimization, a combiner in the flat-combining skip-list and a PIM core in the PIM-managed skip-list both have to execute operations one by one sequentially, leading to throughput of roughly $\frac{1}{(\beta \mathcal{L}_{cpu})}$ and $\frac{1}{(\beta \mathcal{L}_{pim}+\mathcal{L}_{message})}$ respectively, where $\mathcal{L}_{message}$ is incurred by the PIM core sending a message with a result back to a CPU. After dividing these two skip-lists into k partitions, we can achieve a speedup of k for both of them, as k PIM cores and k combiners can serve requests in parallel now. Note that we have ignored certain costs in the lock-free skip-list and the two flat-combining skip-lists, such as the cost of a combiner's operations on the publication list in a flat-combining skip-list and the cost of CAS operations in the lock-free skip-list, so their actual performance could be even worse than what we show in Table 2.

Algorithm	Throughput
Lock-free skip-list	$\frac{p}{\beta \mathcal{L}_{cpu}}$
Flat-combining skip-list	$\frac{1}{\beta \mathcal{L}_{cpu}}$
PIM-managed skip-list	$\frac{1}{(\beta \mathcal{L}_{pim}+\mathcal{L}_{message})}$
Flat-combining skip-list with k partitions	$\frac{k}{\beta \mathcal{L}_{cpu}}$
PIM-managed skip-list with k partitions	$\frac{k}{(\beta \mathcal{L}_{pim}+\mathcal{L}_{message})}$

Table 2: Throughput of skip-list algorithms.

The results in Table 2 imply that the PIM-managed skip-list with k partitions is expected to outperform the second best algorithm, the lock-free skip-list, when $k > \frac{(\beta \mathcal{L}_{pim}+\mathcal{L}_{message})p}{\beta \mathcal{L}_{cpu}}$. Given that $\mathcal{L}_{message} = \mathcal{L}_{cpu} = r_1 \mathcal{L}_{pim}$ and $\beta = \Theta(\log N)$, $k > p/r_1$ should suffice. It is also easy to see that the performance of the PIM-managed skip-list should be $\frac{\beta r_1}{\beta + r_1} \approx r_1$ times better than the flat-combining skip-list, when they have the same number of partitions.

Our experiments have revealed similar results, as presented in Figure 4. We have implemented and run the flat-combining skip-list with different numbers of partitions and compared them with the lock-free skip-list. As the number of partitions increases, the performance of the flat-combining skip-list gets better, attesting to the effectiveness of the partitioning optimization. Again, we believe the performance of the flat-combining skip-list is a good indicator to the performance of our PIM-managed skip-list. Therefore, according to the analytical results in Table 2, we can triple the throughput of a flat-combining skip-list to estimate the expected performance of a PIM-managed skip-list. As Figure 4 illustrates, when the PIM-managed skip-list has 8 or 16 partitions, it is expected to outperform the lock-free skip-list with up to 28 hardware threads.

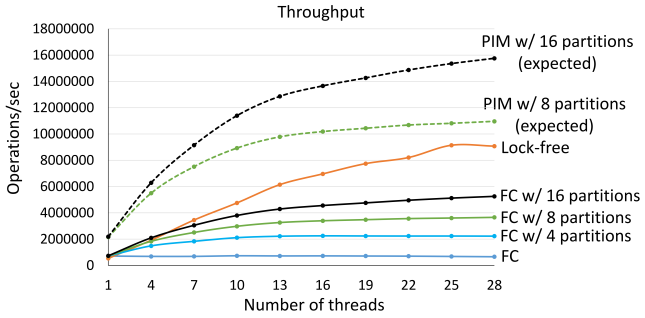

Figure 4: Experimental results of skip-lists. We evaluated the lock-free skip-list and the flat-combining skip-list (FC) with different numbers (1, 4, 8, 16) of partitions.

4.2.1 Skip-list Rebalancing. The PIM-managed skip-list performs well with a uniform distribution of requests. However, if the distribution of requests is *not* uniform, a static partitioning scheme will result in unbalanced partitions, with some PIM cores being idle, while others having to serve a majority of the requests. To address this problem, we introduce a non-blocking protocol for migrating consecutive nodes from one vault to another.

The protocol works as follows. A PIM core p that manages a vault v' can send a message to another PIM core q, managing vault v, to request some nodes to be moved from v' to v. First, p sends a message notifying q of the start of the migration. Then p sends messages to q for adding those nodes into v one by one in ascending order according to the keys of the nodes. After all the nodes have been migrated, p sends notification messages to CPUs so that they can update their copies of sentinel nodes accordingly. After p receives acknowledgement messages from all CPUs, it notifies q of the end of migration. To keep the node migration protocol simple, we don't allow q to move those nodes to another vault again until p finishes its node migration.

During the node migration, p can still serve requests from CPUs. Assume that a request with key k_1 is sent to p when p is migrating nodes in a key range containing k_1. If p is about to migrate a node with key k_2 at the moment and $k_1 \geq k_2$, p serves the request itself. Otherwise, p must have migrated all nodes in the subset containing key k_1, and therefore p forwards the request to q which will serve the request and respond directly to the requesting CPU.

The algorithm is correct, because a request will eventually reach the vault that currently contains nodes in the key range that the request belongs to. If a request arrives to p which no longer holds the partition the request belongs to, p simply replies with a rejection to the CPU and the CPU will resend its request to the correct PIM core, because it has already updated its sentinels and knows which PIM core it should contact now.

Using this node migration protocol, the PIM-managed FIFO queue can support two rebalancing schemes: 1) If a partition has too many nodes, the local PIM core can move nodes in a key range to a vault that has fewer nodes; 2) If two consecutive partitions are both small, we can merge then by moving one to the vault containing the other.

In practice, we expect that rebalancing will not happen very frequently, so its overhead can be ameliorated by the improved efficiency resulting from the rebalanced partitions.

5 HIGH CONTENTION DATA STRUCTURES

In this section, we consider data structures that are often contended when accessed by many threads concurrently. In these data structures, operations compete for accessing one or more locations, creating a contention spot, which can become a performance bottleneck. Examples include head and tail pointers in queues or the top pointer of a stack.

These data structures have good locality and the contention spots are often found in shared CPU caches, such as the last-level cache in a multi-socket non-uniform memory access machine when accessed by threads running only on one socket. Therefore, these data structures might seem to be a poor fit for near-memory computing: the advantage of faster memory access provided by PIM cannot be exercised because the frequently accessed data might stay in the CPU cache. However, such a perspective does not consider the overhead introduced by contention in a concurrent data structure where *many* threads try to access the same locations.

As a representative example of this class of data structures, we consider a FIFO queue, where concurrent enqueue and dequeue operations compete for the head and the tail of the queue, respectively. Although a naive PIM FIFO queue is not a good replacement for a well crafted concurrent FIFO queue, we show that, counterintuitively, PIM can still have benefits over a traditional concurrent FIFO queue. In particular, we exploit the *pipelining* of requests from CPUs, which can be done very efficiently in PIM, to design a PIM FIFO queue that can outperform state-of-the-art concurrent FIFO queues, such as the one using flat combining [20] and the one using fetch and add [31].

5.1 FIFO queues

The structure of our PIM-managed FIFO queue is shown in Figure 5. A queue consists of a sequence of *segments*, each containing consecutive nodes of the queue. A segment is allocated in a PIM vault, with a head node and a tail node pointing to the first and the last nodes of the segment, respectively. A vault can contain multiple (likely non-consecutive) segments. There are two special segments—the *enqueue segment* and the *dequeue segment*. To enqueue a node, a CPU sends an enqueue request to the PIM core of the vault containing the enqueue segment. The PIM core then inserts the node to the head of the segment. Similarly, to dequeue a node, a CPU sends a dequeue request to the PIM core of the vault holding the dequeue segment. The PIM core then pops out the node at the tail of the dequeue segment and sends the node back to the CPU.

Initially, the queue consists of an empty segment that acts as both the enqueue segment and the dequeue segment. When the length of enqueue segment exceeds some threshold, the PIM core maintaining it notifies another PIM core to create a new segment as the new enqueue segment.[5] When the dequeue segment becomes empty and the queue has other segments, the dequeue segment

[5] Alternative designs are also possible and the decision can also be made by CPUs, based on more complex criteria. We omit these alternatives for briefness.

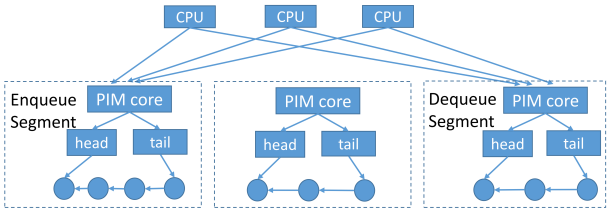

Figure 5: A PIM-managed FIFO queue with three segments

is deleted and the segment that was created first among all the remaining segments is designated as the new dequeue segment. This segment was created when the old dequeue segment acted as the enqueue segment and exceeded the length threshold. If the enqueue segment is different from the dequeue segment, enqueue and dequeue operations can be executed by two different PIM cores in parallel, which doubles the throughput compared to a straightforward queue implementation held in a single vault.

The pseudocode of the algorithm is presented in Algorithm 1. Each PIM core has local variables *enqSeg* and *deqSeg* that are references to local enqueue and dequeue segments. When *enqSeg* (or *deqSeq*) is not null, it indicates that the PIM core is currently holding the enqueue (or dequeue) segment. Each PIM core also maintains a local queue *segQueue* for storing local segments. CPUs and PIM cores communicate via message(*cid*, *content*) calls, where *cid* is the unique core ID (CID) of the receiver and *content* is either a request or a response to a request.

Once a PIM core receives an *enqueue* request enq(*cid*, *u*) of node *u* from a CPU whose CID is *cid*, it first checks if it is holding the enqueue segment (line 2 of Procedure enq(*cid*, *u*)). If so, the PIM core enqueues *u* (lines 5-12), and otherwise sends back a message informing the CPU that the request is rejected (line 3) so that the CPU can resend its request to the right PIM core holding the enqueue segment (we will explain later how the CPU can find the right PIM core). After enqueuing *u*, the PIM core may find that the enqueue segment is longer than the threshold (line 13). If so, it sends a message with a newEnqSeg() request to the PIM core of another vault that is chosen to create a new enqueue segment. The PIM core then sets its *enqSeq* to null, indicating that it no longer deals with enqueue operations. Note that the CID *cid* of the PIM core chosen for creating the new segment is recorded in *enqSeg.nextSegCid* for future use in dequeue requests. As Procedure newEnqSeg() in Algorithm 1 shows, The PIM core receiving this newEnqSeg() request creates a new enqueue segment and enqueues the segment into its *segQueue* (line 3). Finally, it notifies the CPUs of the new enqueue segment (we will discuss this notification in more detail later in this section).

Similarly, when a PIM core receives a *dequeue* request deq(*cid*) from a CPU with CID *cid*, it first checks whether it still holds the dequeue segment (line 2 of Procedure deq(*cid*)). If so, the PIM core dequeues a node and sends it back to the CPU (lines 5-7). Otherwise, it informs the CPU that this request has failed (line 3) and the CPU will have to resend its request to the right PIM core. If the dequeue segment is empty (line 8) and the dequeue segment is not the same as the enqueue segment (line 11), which indicates that the FIFO queue is not empty and there exists another segment, the PIM core sends a message with a newDeqSeg() request to the PIM core with CID *deqSeg.nextSegCid*. (We know that this PIM core must hold the next segment, according to how we create new segments in enqueue operations, as shown in lines 14-16 of Procedure enq(*cid*, *u*).) Upon receiving the newDeqSeg() request, as illustrated in Procedure newDeqSeg(), the PIM core retrieves from its *segQueue* the oldest segment it has created and makes it the new dequeue segment (line 2). Finally the PIM core notifies the CPUs that it is holding the new dequeue segment now.

We now explain how CPUs and PIM cores coordinate to make sure that the CPUs can find the right enqueue and dequeue segments, when their previous attempts have failed due to changes in where those segments are located. We only discuss how to deal with enqueue segments, since the same methods can be applied to dequeue segments. A straightforward way to inform the CPUs is to have the owner PIM core of the new enqueue segment send notification messages to them (line 4 of newEngSeg()) and wait until all the CPUs send back acknowledgement messages. However, if there is a slow CPU core that doesn't reply in time, the PIM core has to wait for it and therefore other CPUs cannot have their requests executed. A more efficient, non-blocking method is to have the PIM core start serving new requests immediately after it has sent off the notifications to all CPUs. A CPU does not have to reply to those notifications in this case, but if its request later fails, it needs to send messages to (sometimes all) PIM cores to ask which PIM core is currently in charge of the enqueue segment. In either case, the correctness of the algorithm is guaranteed: at any time, there is only one enqueue segment and only one dequeue segment, and only requests sent to them will be executed.

We would like to mention that the PIM-managed FIFO can be further optimized. For example, the PIM core holding the enqueue segment can combine multiple pending enqueue requests and store the nodes to be enqueued in an array as a "fat" node of the queue, in order to reduce memory accesses. This optimization is also used in the flat-combining FIFO queue [20]. Even without this optimization, our algorithm still performs well, as we will show next.

5.2 Pipelining and Performance Analysis

We compare the performance of three concurrent FIFO queue algorithms—our PIM-manged FIFO queue, a flat-combining FIFO queue and a F&A-based FIFO queue [31]. The F&A-based FIFO queue is the most efficient concurrent FIFO queue we are aware of, where threads make F&A operations on two shared variables, one for enqueues and the other for dequeues, to compete for slots in the FIFO queue to enqueue and dequeue nodes (see [31] for more details). The flat-combining FIFO queue we consider is based on the one proposed by [20], with a modification that threads compete for two "combiner locks", one for enqueues and the other for dequeues. We further simplify it based on the assumption that the queue is always non-empty, so that it doesn't have to deal with synchronization issues between enqueues and dequeues when the queue is empty.

Let us first assume that a queue is long enough such that the PIM-managed FIFO queue has more than one segment, and enqueue and dequeue requests can be executed separately. Since changes in where enqueue and dequeue segments are located happen very

Algorithm 1 PIM-managed FIFO queue

```
1:  procedure enq(cid, u)
2:      if enqSeg == null then
3:          send message(cid, false);
4:      else
5:          if enqSeg.head ≠ null then
6:              enqSeg.head.next = u;
7:              enqSeg.head = u;
8:          else
9:              enqSeg.head = u;
10:             enqSeg.tail = u;
11:         enqSeg.count = enqSeg.count + 1;
12:         send message(cid, true);
13:         if enqSeg.count > threshold then
14:             cid′ = the CID of the PIM core chosen to maintain the new segment;
15:             send message(cid′, newEnqSeg());
16:             enqSeg.nextSegCid = cid′;
17:             enqSeg = null;

1:  procedure newEnqSeg()
2:      enqSeg = new Segment();
3:      segQueue.enq(engSeg) ;
4:      notify the CPUs of the new enqueue segment;
```

```
1:  procedure deq(cid)
2:      if deqSeg == null then
3:          send message(cid, false);
4:      else
5:          if deqSeg.tail ≠ null then
6:              send message(cid, deqSeg.tail);
7:              deqSeg.tail = deqSeg.tail.next;
8:          else
9:              if deqSeg == enqSeg then
10:                 send message(cid, null);
11:             else
12:                 send message(deqSeg.nextSegCid, newDeqSeg());
13:                 deqSeg = null;
14:                 send message(cid, false);

1:  procedure newDeqSeg()
2:      deqSeg = segQueue.deq();
3:      notify the CPUs of the new dequeue segment;
```

infrequently, the overhead of such changes is negligible and therefore ignored to simplify our analysis. (If the threshold of segment length in line 13 of enq(*cid*, *u*) is a large integer *n*, then, in the worst case, changing an enqueue or dequeue segment happens only once every *n* requests, and the cost is only the latency of sending one message and a few steps of local computation.) Since enqueues and dequeues are isolated in all the three algorithms when queues are long enough, we will focus on dequeues. The analysis of enqueues is almost identical.

Assume there are p concurrent dequeue requests by p threads. Since each thread needs to perform a F&A operation on a shared variable in the F&A-based algorithm and F&As on a shared variable are essentially serialized, the execution time of p requests is at least $p\mathcal{L}_{atomic}$. If we assume that each CPU makes a request immediately after its previous request completes, the throughput (per second) of the algorithm is at most $\frac{1}{\mathcal{L}_{atomic}}$.

The flat-combining FIFO queue maintains a sequential FIFO queue and threads submit their requests into a *publication list*. The publication list consists of slots, one for each thread, to store their requests. After writing a request into the list, a thread competes with others for acquiring a lock to become the "*combiner*", which incurs one last-level cache access. The combiner then goes through the publication list to retrieve requests, executes operations for those requests, and writes results back to the list, while other threads with pending requests spin on their own slots, waiting for the results. The combiner therefore makes two last-level cache accesses to each slot other than its own slot, one for reading the request and one for writing the result back. Thus, the execution time of p requests in the algorithm is at least $(2p-1)\mathcal{L}_{llc}$ and the throughput (per second) of the algorithm is at most $\frac{1}{2\mathcal{L}_{llc}}$ for large enough p.

Note that our analysis of the F&A-based and the flat-combining algorithms is quite optimistic, as it counts only the costs in part of their executions. We have ignored the latency of accessing and modifying queue nodes in the two algorithms. For dequeues, this latency can be high: since nodes to be dequeued in a long queue is unlikely to be cached, the combiner has to perform a sequence of memory accesses to dequeue them one by one. Moreover, the F&A-based algorithm may also suffer performance degradation under heavy contention, because contended F&A operations may perform worse in practice [12].

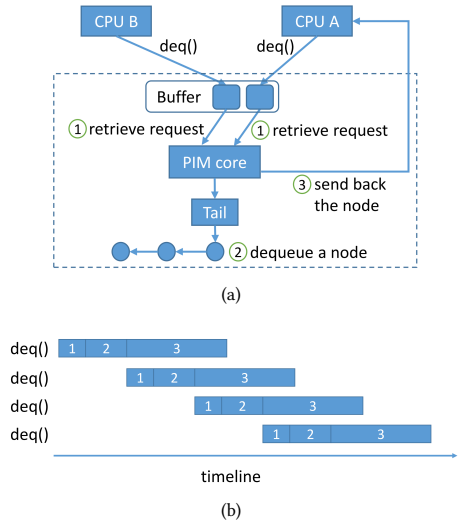

Figure 6: (a) The pipelining optimization, where a PIM core can start executing a new deq() (step 1 of deq() for CPU B), without waiting for the dequeued node of the previous deq() to return to CPU A (step 3). (b) The timeline of pipelining four deq() requests.

The performance of our PIM-managed FIFO queue seems poor at first sight: although a PIM core can update the queue efficiently, it takes a lot of time for the PIM core to send results back to CPUs one by one. To improve its performance, the PIM core can *pipeline* the execution of requests, as illustrated in Figure 6(a). Suppose p CPUs send p dequeue requests concurrently to the PIM core. The

PIM core then retrieves a request from its message buffer (step 1 in the figure), dequeues a node (step 2) for the request, and sends the node back to the CPU (step 3). We can hide the message latency in step 3 as follows. After sending off the message containing the node in step 3, the PIM core *immediately* retrieves the next request to execute, without blocking to wait for that message to arrive at its receiver. This way, the PIM core *pipelines* requests by overlapping the latency of message transfer in step 3 and the latency of memory accesses and local computations in steps 1 and 2 across multiple requests (see Figure 6(b)). Note that the PIM core still executes everything sequentially, as it always first sends off the message for the current request before serving the next.

The throughput of the PIM core is eventually decided by the costs of its memory accesses and local computations, as long as it has enough bandwidth to keep sending off messages, which is very likely the case for this algorithm, since only one message is sent per request. More specifically, Figure 6(b) illustrates that the total execution time of p requests is the sum of the execution times of the first two steps for the p requests, plus the message transfer time of step 3 for the last request. Since the PIM core in the execution of a dequeue only makes one memory access to read the node to be dequeued, and two L1 cache accesses to read and modify the tail node of the dequeue segment, the total execution time of p requests, including the time $\mathcal{L}_{message}$ spent by CPUs to their requests to the PIM core concurrently, is only $\mathcal{L}_{message}+p(\mathcal{L}_{pim}+\epsilon)+\mathcal{L}_{message}$, where ϵ is the total latency of the PIM core making two L1 cache accesses and sending off one message, which is negligible in our performance model.

Assume that each CPU makes another request immediately after it receives the result of its previous request and that there are enough (at least $2\mathcal{L}_{message}/\mathcal{L}_{pim}$) CPUs sending requests. We can prove that the PIM core can always find another request in its buffer after it executes one. Let x be the throughput of the PIM core in one second. By the same analysis as above, we have $\mathcal{L}_{message}+x(\mathcal{L}_{pim}+\epsilon)+\mathcal{L}_{message} = 1$. Therefore, the throughput (per second) of the PIM-managed FIFO queue is approximately

$$x = \frac{1-2\mathcal{L}_{message}}{\mathcal{L}_{pim}+\epsilon} \approx \frac{1-2\mathcal{L}_{message}}{\mathcal{L}_{pim}} \approx \frac{1}{\mathcal{L}_{pim}},$$

since $\mathcal{L}_{message}$ is usually only hundreds of nanoseconds and much smaller than 1 (second).

Comparing the throughput values of the three FIFO queue algorithms, we can conclude that the PIM-managed FIFO queue with pipelining outperforms the other two algorithms when $2r_1/r_2 > 1$ and $r_1 r_3 > 1$. If we assume $r_1 = r_2 = 3$ and $r_3 = 1$, then the throughput of our FIFO queue is expected to be twice the throughput of the flat-combining queue and three times that of the F&A queue.

When the PIM-managed FIFO queue is short, it may contain only one segment which deals with both enqueue and dequeue requests. In this case, its throughput is only half of the throughput shown above, but it should still be at least as good as the throughput of the other two algorithms.

6 RELATED WORK

The PIM model is undergoing a renaissance. Studied for decades (e.g., [13, 16, 17, 27, 29, 32, 33, 37]), this model has recently re-emerged due to advances in 3D-stacked techology that can stack memory dies on top of a logic layer [7, 26, 28, 30]. For example, a 3D-stacked memory prototype called the Hybrid Memory Cube [11] was recently released by industry, and the model has again become the focus of architectural research. Different PIM-based architectures have been proposed, either for general purposes or for specific applications [1–6, 8, 18, 24, 25, 35, 36, 39–41].

The PIM model has several advantages, including low energy consumption and high bandwidth (e.g., [1, 4, 39, 40]). Here, we focus on one more: low memory access latency [6, 18, 25, 30]. To our knowledge, we are the first to utilize PIM memory for designing efficient *concurrent data structures*. Although some researchers have studied how PIM memory can help speed up concurrent operations to data structures, such as parallel graph processing [1] and parallel pointer chasing on linked data structures [25], the applications they consider require very simple, if any, synchronization between operations. In contrast, operations to concurrent data structures can interleave in arbitrary orders, and therefore have to correctly synchronize with one another in all possible execution scenarios. This makes designing concurrent data structures with correctness guarantees, like linearizability [23], very challenging.

No prior work compares the performance of data structures in the PIM model with that of state-of-the-art concurrent data structures in the classical shared memory model. We analyze and evaluate concurrent linked-lists and skip-lists, as representatives of pointer-chasing data structures, and concurrent FIFO queues, as representatives of contended data structures. For linked-lists, we compare our PIM-managed implementation with well-known approaches such as fine-grained locking [19] and flat combining [14, 20, 21]. For skip-lists, we compare our implementation with the lock-free skip-list [22] and a skip-list with flat combining and the partitioning optimization. For FIFO queues, we compare our implementation with the flat-combining FIFO queue [20] and the F&A-based FIFO queue [31].

7 CONCLUSION

In this paper, we study how to exploit the low access latency of PIM memory to design efficient concurrent data structures for the PIM memory. We analyze and compare the performance of our PIM-managed data structures with concurrent data structures in the literature. To this end, we propose a simplified performance model for PIM memory. We show that naive PIM-managed data structures *cannot* outperform traditional concurrent data structures, due to the lack of parallelism and the high communication cost between the CPUs and the PIM cores. To improve the performance of PIM data structures, we propose novel designs for low contention pointer-chasing data structures, such as linked-lists and skip-lists, and for high-contention data structures, such as FIFO queues. We show that our PIM-managed data structures can outperform state-of-the-art concurrent data structures, making PIM memory a promising platform for managing data structures.

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 *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[2] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 336–348. https://doi.org/10.1145/2749469.2750385

[3] Berkin Akin, Franz Franchetti, and James C. Hoe. 2015. Data Reorganization in Memory Using 3D-stacked DRAM. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 131–143. https://doi.org/10.1145/2749469.2750397

[4] Erfan Azarkhish, Christoph Pfister, Davide Rossi, Igor Loi, and Luca Benini. 2017. Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube. *IEEE Trans. VLSI Syst.* 25, 1 (2017), 210–223. https://doi.org/10.1109/TVLSI.2016.2570283

[5] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2015. High Performance AXI-4.0 Based Interconnect for Extensible Smart Memory Cubes. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE '15)*. EDA Consortium, San Jose, CA, USA, 1317–1322. http://dl.acm.org/citation.cfm?id=2757012.2757119

[6] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube. In *Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*. Springer-Verlag New York, Inc., New York, NY, USA, 19–31. https://doi.org/10.1007/978-3-319-30695-7_2

[7] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen, and Clair Webb. 2006. Die Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 469–479. https://doi.org/10.1109/MICRO.2006.18

[8] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. 2016. LazyPIM: An efficient Cache Coherence Mechanism for Processing-in-memory. *IEEE Computer Architecture Letters* (2016).

[9] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[10] Kevin K. Chang, Abhijith Kashyap, Hasan Hassan, Saugata Ghose, Kevin Hsieh, Donghyuk Lee, Tianshi Li, Gennady Pekhimenko, Samira Khan, and Onur Mutlu. 2016. Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science (SIGMETRICS '16)*. ACM, New York, NY, USA, 323–336. https://doi.org/10.1145/2896377.2901453

[11] Hybrid Memory Cube Consortium. 2013. Hybrid Memory Cube Specification 1.0. (2013). http://hybridmemorycube.org/files/SiteDownloads/HMC_Specification%201_0.pdf

[12] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 33–48. https://doi.org/10.1145/2517349.2522714

[13] Duncan G. Elliott, W. Martin Snelgrove, and Michael Stumm. 1992. Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP. In *Proceedings of the IEEE 1992 Custom Integrated Circuits Conference (CICC '92)*. IEEE Press, Piscataway, NJ, USA, 30.6.1–30.6.4.

[14] Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849

[15] Keir Fraser. 2004. *Practical Lock-freedom*. Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

[16] Maya Gokhale, Bill Holmes, and Ken Iobst. 1995. Processing in Memory: The Terasys Massively Parallel PIM Array. *Computer* 28, 4 (April 1995), 23–31. https://doi.org/10.1109/2.375174

[17] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman, Apoorv Srivastava, William Athas, Vincent Freeh, Jaewook Shin, and Joonseok Park. 1999. Mapping Irregular Applications to DIVA, a PIM-based Data-intensive Architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing (SC '99)*. ACM, New York, NY, USA, Article 57. https://doi.org/10.1145/331532.331589

[18] Milad Hashemi, Khubaib, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2016. Accelerating Dependent Cache Misses with an Enhanced Memory Controller. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 444–455. https://doi.org/10.1109/ISCA.2016.46

[19] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2006. A Lazy Concurrent List-based Set Algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS'05)*. Springer-Verlag, Berlin, Heidelberg, 3–16. https://doi.org/10.1007/11795490_3

[20] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '10)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[21] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Scalable Flat-combining Based Synchronous Queues. In *Proceedings of the 24th International Conference on Distributed Computing (DISC'10)*. Springer-Verlag, Berlin, Heidelberg, 79–93. http://dl.acm.org/citation.cfm?id=1888781.1888793

[22] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[23] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[24] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-transparent Near-data Processing in GPU Systems. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*. IEEE Press, Piscataway, NJ, USA, 204–216. https://doi.org/10.1109/ISCA.2016.27

[25] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating Pointer Chasing in 3D-stacked Memory: Challenges, Mechanisms, Evaluation. In *IEEE 34th International Conference on Computer Design, ICCD 2016*. IEEE, 25–32.

[26] Joe Jeddeloh and Brent Keeth. 2012. Hybrid Memory Cube New DRAM Architecture Increases Density and Performance. In *Symposium on VLSI Technology, VLSIT 2012*. IEEE, 87–88.

[27] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenzhou Ge, Vinh Vi Lam, Josep Torrellas, and Pratap Pattnaik. FlexRAM: Toward an Advanced Intelligent Memory System. In *Proceedings of the IEEE International Conference On Computer Design (ICCD '99)*.

[28] Joonyoung Kim, Younsu Kim, undefined, undefined, undefined, and undefined. 2014. HBM: Memory solution for bandwidth-hungry processors. *2014 IEEE Hot Chips 26 Symposium (HCS)* 00 (2014), 1–24. https://doi.org/doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2014.7478812

[29] Peter M. Kogge. 1994. EXECUBE-A New Architecture for Scaleable MPPs. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01 (ICPP '94)*. IEEE Computer Society, Washington, DC, USA, 77–84. https://doi.org/10.1109/ICPP.1994.108

[30] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 453–464. https://doi.org/10.1109/ISCA.2008.15

[31] Adam Morrison and Yehuda Afek. 2013. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 103–112. https://doi.org/10.1145/2442516.2442527

[32] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)*. IEEE Computer Society, Washington, DC, USA, 192–203. https://doi.org/10.1145/279358.279387

[33] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17, 2 (March 1997), 34–44. https://doi.org/10.1109/40.592312

[34] W. Pugh. 1990. *Concurrent Maintenance of Skip Lists*. Technical Report. University of Maryland at College Park.

[35] Vivek Seshadri, Kevin Hsieh, Amirali Boroumand, Donghyuk Lee, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2015. Fast Bulk Bitwise AND and OR in DRAM. *IEEE Comput. Archit. Lett.* 14, 2 (July 2015), 127–131. https://doi.org/10.1109/LCA.2015.2434872

[36] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 185–197. https://doi.org/10.1145/2540708.2540725

[37] Harold S. Stone. 1970. A Logic-in-Memory Computer. *IEEE Trans. Comput.* 19, 1 (Jan. 1970), 73–78. https://doi.org/10.1109/TC.1970.5008902

[38] J. Valois. 1996. *Lock-free Data Structures*. Ph.D. Dissertation. Rensselaer Polytechnic Institute, Troy, NY, USA.

[39] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*. ACM,

New York, NY, USA, 85–98. https://doi.org/10.1145/2600212.2600213

[40] Qiuling Zhu, Berkin Akin, H. Ekin Sumbul, Fazle Sadi, James C. Hoe, Larry T. Pileggi, and Franz Franchetti. 2013. A 3D-stacked Logic-in-memory Accelerator for Application-specific Data Intensive Computing. In *IEEE International 3D Systems Integration Conference, 3DIC 2013, San Francisco, CA, USA, October 2-4, 2013*. 1–7. https://doi.org/10.1109/3DIC.2013.6702348

[41] Qiuling Zhu, Tobias Graf, H. Ekin Sumbul, Larry T. Pileggi, and Franz Franchetti. 2013. Accelerating Sparse Matrix-matrix Multiplication with 3D-stacked Logic-in-memory hardware. In *IEEE High Performance Extreme Computing Conference, HPEC 2013, Waltham, MA, USA, September 10-12, 2013*. 1–6. https://doi.org/10.1109/HPEC.2013.6670336