Concurrent Data Structures for Near-Memory Computing

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

Irina Calciu
VMware Research Group
icalciu@vmware.com

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

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

(This is a regular paper.)

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, 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.

In this paper, we show two results: (1) naive PIM data structures cannot outperform state-of-the-art concurrent data structures such as pointer-chasing data structures and FIFO queue, (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.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

1. NEAR-MEMORY COMPUTING

The performance gap between memory and CPU has grown exponentially. Memory vendors have focused mostly on improving memory capacity and bandwidth, sometimes even at the cost of increased memory access latencies. 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., [28, 22, 12, 26, 25, 21, 13]), but so far has not yet materialized. However, new advances in 3D integration and in die stacked memory make near-memory computing viable in the near future. For example, one PIM design assumes memory is organized in multiple vaults, each having an in-order PIM core to manage it. These PIM cores can 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 memory becomes 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 [30, 1, 31, 3], as well as for pointer-chasing traversals [19] 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.

In this paper, we show that naive PIM data structures cannot outperform state-of-the-art *concurrent* data structures. In particular, the lower latency access to memory cannot compensate for the loss of parallelism. 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 the expected performance of PIM. Using this model, we investigate two classes of data structures.

First, in Section 4 we analyze pointer chasing data structures, which have a high degree of inherent parallelism and low contention, but incur significant overhead due to unpredictable memory accesses. 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, such as FIFO queues (Section 5), 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 for many threads accessing the data oncurrently. 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 [24, 15].

The contributions of this paper are summarized below.

- We propose a very simple 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 latencies are not sufficient for PIM data structures to outperform efficient concurrent algorithms.
- We propose new designs for PIM data structures using techniques such as combining, partitioning and pipelining, that 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 algorithms and use our model to compare them to prior work. We also use current architectures to simulate the behavior of our algorithms and evaluate compared to state-of-the-art concurrent algorithms. Finally, we present related work in Section 6 and conclude in Section 7.

2. HARDWARE ARCHITECTURE AND MODEL

In the PIM hardware model, 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 normal 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 [9], each HMC consists of 16 or 32 vaults and has total size 2GB or 4 GB (so each vault has size roughly 100MB). We assume the same specifications in our PIM model, although the size of a PIM memory and the number of its vaults can be greater. Each CPU also

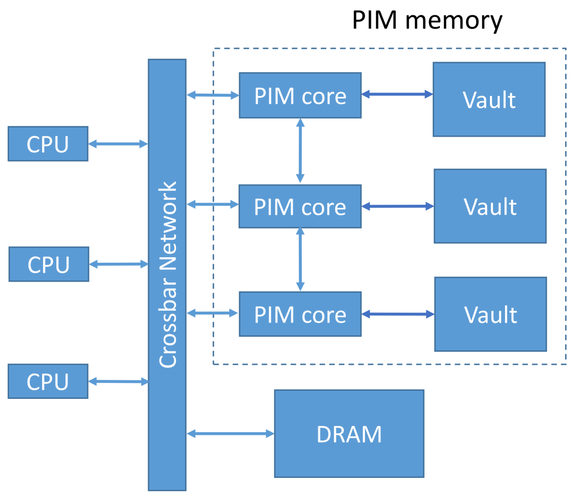


Figure 1: The PIM model

has access to a hierarchy of caches backed by DRAM, and there can be last-level caches shared among multiple CPUs.

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 vault can be accessed only by its local PIM core.[2] Although a PIM core is relatively slow computationally, 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.

To keep the PIM memory simple, we assume that a PIM core can only make read and write operations to its local vault, while a CPU also supports more powerful atomic operations, such as CAS and F&A. Virtual memory is cheap to be achieved in this model, by having each PIM core maintain its own page table for its local vault [19].

[1]A PIM core can be thought of as an in-order CPU with only small private L1 cache and without some optimizations that full-fledged CPUs usually have.

[2]We may alternatively assume that a PIM core has direct access to remote vaults, at a larger cost. We may also assume that vaults are accessible by CPUs as well, but at the cost of dealing with cache coherence between CPUs and PIM cores. Some cache coherence mechanisms for PIM memory claim to be not costly (e.g., [8, 2]). However, we prefer to keep the hardware model simple and we will show that we are still able to design efficient concurrent data structure algorithms with this simple, less powerful PIM memory.

3. PERFORMANCE MODEL

Based on the latency numbers in prior work on PIM memory, in particular on the Hybrid Memory Cube [9, 6], and on the evalutation of operations in multiprocessor architectures [10], 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

$$\mathcal{L}_{cpu} = 3\mathcal{L}_{pim} = 3\mathcal{L}_{llc},$$

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. 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 making an atomic operation, such as a CAS or a F&A, to a cache line is

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

even if the cache line is currently in cache. This is because an atomic operation hitting the cache is usually as costly as a memory access by a CPU, acorrding to [10]. 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 assume that the size of a message sent by a PIM core or a CPU 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 a 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 period between the moment when a PIM or a CPU sends off the message and the moment 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, as it is negligible compared to the time spent in message transfer.

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 they 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 the dominating overhead of these data structures. Naturally, these data structures have been early examples of the benefit of near-memory computing [19], as the entire pointer chase could be performed by the PIM core, and only the final result returned to the application.

However, under the same conditions, these data structures have inherently low contention. Lock-free algorithms [11, 27, 29, 16] have shown that these data structures can scale to hundreds of cores under low contention. 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 concurrent computing literature to optimize algorithms for near-memory computing to outperform well-known concurrent data structures.

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[3] wants to perform an operation on the linked-list, it sends a request to the PIM core. The PIM core will retrieve the message, execute the operation, and send the result back to the CPU. The PIM linked-list is sequential, as it can only be accessed by one PIM core.

Doing pointer chasing on sequential data structures by PIM cores is not a new idea (e.g., [19, 1]). It is obvious that for a sequential data structure like a sequential linked-list, replacing the CPU with a PIM core to access the data structure will largely improve its performance due to the PIM core's much faster memory access. However, we are not aware of any prior comparison between the performance of PIM-managed data structures and concurrent data structures in which CPUs can make operations in parallel. In fact, our analytical and experimental results will show that the performance of the naive PIM-managed linked-list is much worse than that of the concurrent linked-list with fine-grained locks [14].

To improve the performance of the PIM-managed linked-list, we apply the following *combining optimization* to it: the PIM core retrieves all pending requests from its buffer and executes all of them during only one traversal over the linked-list. It is not hard to see that 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* [15], where, roughly speaking, threads compete for a "combiner lock" to become the combiner, and the combiner will take over all operation requests from other threads and execute them. Therefore, we think the performance of the flat-combining linked-list is a good indicator of the performance of our PIM-managed linked-list.

Based on our performance model, we can calculate the approximate expected throughputs 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 CPUs only make *contains*() requests (or the number of *add*() requests is the same as the number of *delete*() so that the size of each linked-list nearly doesn't change). We also assume that a CPU makes a new operation request immediately after its previous one completes. Assuming that $n \gg p$ and $N \gg p$, the approximate expected throughputs (per second) of the concurrent linked-lists are presented in Table 1, where $\mathcal{S}_p = \sum_{i=1}^{n} (\frac{i}{n+1})^p$.[4]

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: Throughputs of linked-list algorithms.

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, as long as $\frac{\mathcal{L}_{cpu}}{\mathcal{L}_{pim}} > \frac{2(n-\mathcal{S}_p)}{n+1}$. Given that $0 < \mathcal{S}_p \leq \frac{n}{2}$ and $\mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, 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-

[3]We use the term CPU to refer to CPU cores, as opposed to PIM cores.

[4]We define the rank of an operation request to a linked-list as the number of pointers it has to traverse until it finds the right position for it in the linked-list. $\mathcal{S}_p$ is the expected rank of the operation request with the biggest key among p random requests a PIM core or a combiner has to combine, which is essentially the expected number of pointers a PIM core or a combiner has to go through during one pointer chasing procedure.

grained locks. Without combining, however, the PIM-managed linked-list cannot beat the linked-list with fine-grained locks when $p > 6$.

We implemented the linked-list with fine-grained locks and the flat-combining link-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 at 2.2 GHz. To get rid of 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. Cache lines have 64 bytes.

The throughputs of the algorithms are presented in Figure 2. The results confirmed the validity of our analysis in Table 1. The throughput of the flat-combining algorithm without combining optimization is much worse than the algorithm with fine-grained locks. Since we believe the performance of the flat-combining linked-list is a good indicator of that of the PIM-managed linked-list, we triple the throughput of the flat-combining algorithm without combining optimization to get the estimated throughput of the PIM-managed algorithm. As we can see, it is still far 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 beats all others'.

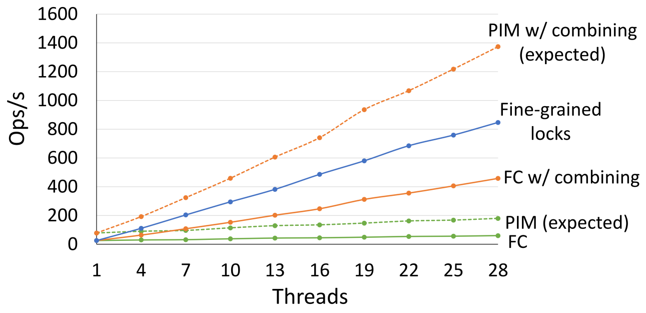


Figure 2: Experimental results of linked-lists. We evaluated the linked-list with Fine-grained locks and the flat-combining linked-list (FC) with and without the combining optimization.

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 we traverse through to reach them don't 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 [9] has up to 32 vaults. Hence, a PIM-managed skip-list may 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 the 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 can almost certainly stay in cache if CPUs access them frequently.

When a CPU applies 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 finally sends the result back to the CPU.

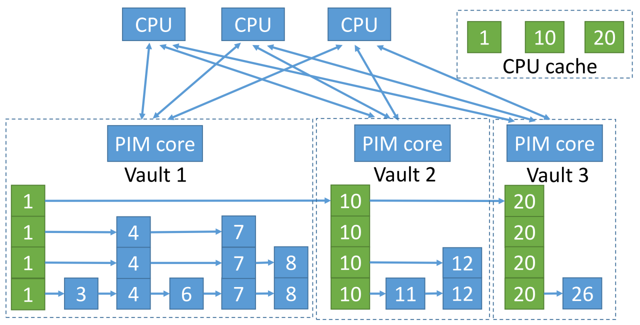


Figure 3: A PIM-managed FIFO queue with three partitions

Now let us 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 will never be 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 [15] and a lock-free skip-list [17], 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 $delete()$ so that the size of each skip-list nearly doesn't change). Their approximate expected throughputs are 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). Note that we have ignored some overheads in the flat-combining algorithms, such as maintaining combiner locks and publication lists (we will discuss publication lists in more detail in Section 5). We also have overestimated the performance of the lock-free skip-list by not counting the CAS operations used in add() and delete() requests, as well as the cost of retries caused by conflicts of updates. Even so, our PIM-managed linked-list with partitioning optimization is still 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 $\mathcal{L}_{message} = \mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, $k > p/3$ should suffice.

Algorithm	Throughput
Look-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: Throughputs of skip-list algorithms.

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, implying 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 get 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.

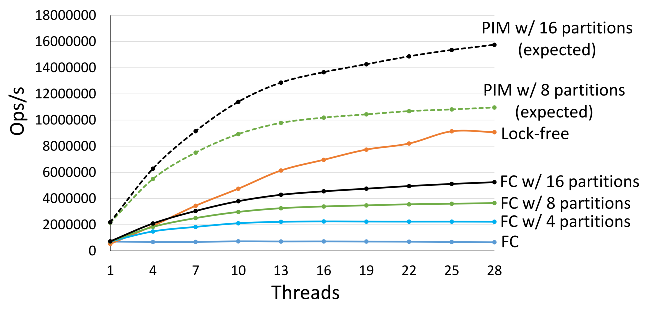


Figure 4: Experimental results of skip-lists. We evaluated the lock-free skip-list and the flat-combining skip-list (FC) with different numbers (1, 4, 8, 16) of partitions.

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 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 that some nodes are moved from v' to v. First, p sends a message notifying q of the start of the migration. Then p sends messages of adding those nodes to q one by one in an 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 can simply reply 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 send 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 a rebalance.

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 several 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, because the advantage of the faster access to memory is muted by having the frequently accessed data in the cache. However, such a perspective does not consider the overhead introduced by contention in a concurrent data structure where all 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 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, counter-intuitively, 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 [15] and the one using Fetch And Add [24].

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 (mostly 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 will then insert 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 will then pop out the node at the tail of the dequeue segment and send the node back to the CPU.

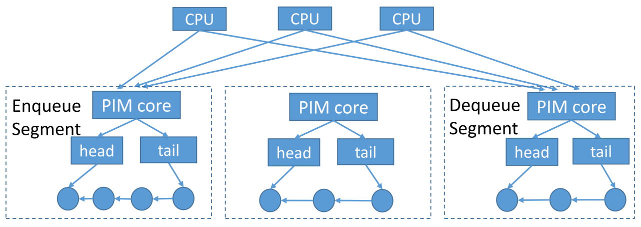


Figure 5: A PIM-managed FIFO queue with three segments

Initially the queue consists of an empty segment which 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 is deleted and the segment that was created first among all the remaining segments is designated as the new dequeue segment. (It is not hard to see that the new dequeue segment were 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 (respectively deqSeq) is not null, it indicates that the PIM core is currently holding the enqueue (respectively 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

[5]When and how to create a new segment can be decided in other ways. For example, CPUs, instead of the PIM core holding the enqueue segment, can decide when to create the new segment and which vault to hold the new segment, based on more complex criteria (e.g., if a PIM core is currently holding the dequeue segment, it will not be chosen for the new segment so as to avoid the situation where it deals with both enqueue and dequeue requests). To simplify the description of our algorithm, we omit those variants.

Algorithm 1 PIM-managed FIFO queue: PIM core's procedures upon receiving requests enq(cid, u), deq(cid), newEnqSeg(), and newEnqDeq()

```
 1: procedure enq(cid, $u$)
 2:     if enqSeg == null then
 3:         send message(cid, false);
 4:     else
 5:         if enqSeg.head $\neq$ 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 deq(cid)
 2:     if deqSeg == null then
 3:         send message(cid, false);
 4:     else
 5:         if deqSeg.tail $\neq$ 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 newEnqSeg()
2:     enqSeg = new Segment();
3:     segQueue.enq(engSeg) ;
4:     notify CPUs of the new enqueue segment;
```

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

unique core ID (CID) of the receiver and the 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 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. Finally the PIM core sets its enqSeq to null indicating 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 CPUs of the new enqueue segment (we will get to it in more detail later).

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 at 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 CPU that it is holding the new dequeue segment now.

Now we explain how CPUs and PIM cores coordinate to make sure that CPUs can find the right enqueue and dequeue segments, when their previous attempts have failed due to changes of those segments. We will only discuss how to deal with enqueue segments here, since the same methods can be applied to dequeue segments. A straightforward way to inform 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 CPUs all send back acknowledgement messages. However, if there is a slow CPU 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 working for new requests immediately after it has sent off those notifications. 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 whether a 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, so as to reduce memory accesses. This optimization is also used in the flat-combining FIFO queue [15]. 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 [24]. 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 [24] for more details). The flat-combining FIFO queue we consider is based on the one proposed by [15], 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.

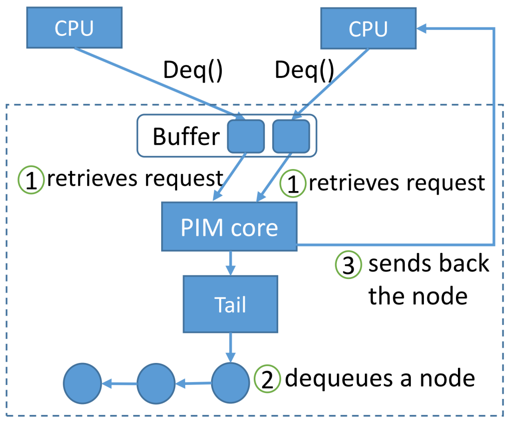
Let us first assume that a queue is long enough such that the PIM-managed FIFO queue has more than one segment, and enqueue and dequeue requests can be executed separately. Since changes of enqueue and dequeue segments happen very infrequently, its overhead is negligible and therefore ignored to simplify our analysis. (If the threshold of segment length at 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, and the analysis of enqueues is almost identical.

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

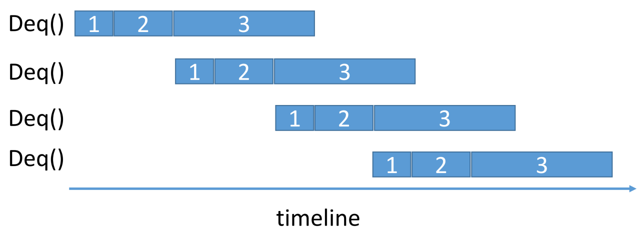
The flat-combining FIFO queue maintains a sequential FIFO queue and threads submit their requests into a publication list. The publication list consists of slots, one for each thread, to store those requests. After writing a request into the list, a thread competes with other threads for acquiring a lock to become the "combiner". 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 spin on their 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 of the algorithm is roughly $\frac{1}{2\mathcal{L}_{llc}}$ for large enough p.

Note that we have made quite optimistic analysis for the F&A-based and flat-combining algorithms by counting only the costs in part of their executions. The latency of accessing and modifying queue nodes in the two algorithms is ignored here. 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 make a sequence of memory accesses to dequeue them one by one. Moreover, the F&A-based algorithm may suffer performance degradation under heavy contention, because contended F&A operations may perform worse in practice.

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 executions of requests, as illustrated in Figure 6(a). Suppose p CPUs send p dequeue requests concurrently to the PIM core, which takes time $\mathcal{L}_{message}$. The PIM core fist 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). After the PIM core sends off the message containing the node, it immediately retrieves the next request, without waiting for the message to arrive at its receiver. This way, the PIM core can pipeline requests by overlapping the latency of message transfer (step 3) and the latency of memory accesses and local computations (steps 1 and 2) in multiple requests (see Figure 6(b)). During the execution of a dequeue, the PIM core 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. It is easy to prove that the execution time of p requests, including the time CPUs send their requests to the PIM core, is only $\mathcal{L}_{message} + p(\mathcal{L}_{pim} + \epsilon) + \mathcal{L}_{message}$, where ϵ is the total latency of the PIM core making L1 cache accesses and sending off one message, which is negligible in our performance model. If each CPU makes another



(a)



(b)

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

request immediately after it receives the result of its previous request, we can prove that the throughput of the PIM-managed FIFO queue is

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

which is expected twice the throughput of the flat-combining queue and three times that of the F&A queue, in our performance model assuming $\mathcal{L}_{atomic} = 3\mathcal{L}_{llc} = 3\mathcal{L}_{pim}$.

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., [28, 22, 12, 26, 25, 21, 13]), this model has recently re-emerged due to advances in 3D-stacked techology that can stack memory dies on top of a logic layer [20, 23, 7]. For example, Micron and others have recently released a PIM prototype called the Hybrid Memory Cube [9], 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 [2, 1, 30, 19, 6, 3, 5, 4, 8, 31, 32].

The PIM model has many advantages, including low energy consumption and high bandwidth (e.g., [1, 30, 31, 4]). Here, we focus on one more: low memory access latency [23, 19, 6]. To our knowledge, however, 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 [19], the applications they consider require very simple, if any, synchronization between operations. In contract, operations to concurrent data structures can interleave in arbitrary orders, and therefore have to correctly synchronize with one another in all possible situations. This makes designing concurrent data structures with correctness guarantees like linearizability [18] very challenging.

Moreover, no one has ever compared the performance of data structures in the PIM model with that of state-of-the-art concurrent data structures in the classic 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 [14] and flat combining [15].

For skip-lists, we compare our implementation with the lock-free skip-list [17] and a skip-list with flat combining and partitioning optimization. For FIFO queues, we compare our implementation with the flat-combining FIFO queue [15] and the F&A-based FIFO queue [24].

7. CONCLUSION

8. REFERENCES

[1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[2] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. 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, pages 336–348, New York, NY, USA, 2015. ACM.

[3] B. Akin, F. Franchetti, and J. C. Hoe. Data reorganization in memory using 3d-stacked dram. In *Proceedings of the 42Nd Annual International*

Symposium on Computer Architecture, ISCA '15, pages 131–143, New York, NY, USA, 2015. ACM.

[4] E. Azarkhish, C. Pfister, D. Rossi, I. Loi, and L. Benini. Logic-base interconnect design for near memory computing in the smart memory cube. *IEEE Trans. VLSI Syst.*, 25(1):210–223, 2017.

[5] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. 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, pages 1317–1322, San Jose, CA, USA, 2015. EDA Consortium.

[6] E. Azarkhish, D. Rossi, I. Loi, and L. Benini. 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*, pages 19–31, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[7] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3d) microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 469–479, Washington, DC, USA, 2006. IEEE Computer Society.

[8] A. Boroumand, S. Ghose, B. Lucia, K. Hsieh, K. Malladi, H. Zheng, and O. Mutlu. Lazypim: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters*, 2016.

[9] H. M. C. Consortium. Hybrid memory cube specification 1.0.

[10] T. David, R. Guerraoui, and V. Trigonakis. 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, pages 33–48, New York, NY, USA, 2013. ACM.

[11] K. Fraser. Practical lock-freedom. Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[12] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, Apr. 1995.

[13] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, A. Srivastava, W. Athas, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to diva, a pim-based data-intensive architecture. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM.

[14] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems*, OPODIS'05, pages 3–16, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[17] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[18] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[19] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 25–32. IEEE, 2016.

[20] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88. IEEE, 2012.

[21] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. V. Lam, J. Torrellas, and P. Pattnaik. Flexram: Toward an advanced intelligent memory system. In *Proceedings of the IEEE International Conference On Computer Design, VLSI in Computers and Processors, ICCD '99, Austin, Texas, USA, October 10-13, 1999*, pages 192–201, 1999.

[22] P. M. Kogge. Execube-a new architecture for scaleable mpps. In *Proceedings of the 1994 International Conference on Parallel Processing - Volume 01*, ICPP '94, pages 77–84, Washington, DC, USA, 1994. IEEE Computer Society.

[23] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society.

[24] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13,

pages 103–112, New York, NY, USA, 2013. ACM.

[25] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A computation model for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, ISCA '98, pages 192–203, Washington, DC, USA, 1998. IEEE Computer Society.

[26] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent ram. *IEEE Micro*, 17(2):34–44, Mar. 1997.

[27] W. Pugh. Concurrent maintenance of skip lists. Technical report, University of Maryland at College Park, 1990.

[28] H. S. Stone. A logic-in-memory computer. *IEEE Trans. Comput.*, 19(1):73–78, Jan. 1970.

[29] J. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, Troy, NY, USA, 1996.

[30] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014. ACM.

[31] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. T. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *2013 IEEE International 3D Systems Integration Conference (3DIC), San Francisco, CA, USA, October 2-4, 2013*, pages 1–7, 2013.

[32] Q. Zhu, T. Graf, H. E. Sumbul, L. T. Pileggi, and F. Franchetti. 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*, pages 1–6, 2013.