PIM-Managed Concurrent Data Structures

Abstract

The process-in-memory (PIM) model has reemerged and drawn a lot of attention recently, as breakthroughs on 3D die-stacked technology make PIM architectures viable. In the PIM model, some lightweight computing units, called PIM cores, are directly attached to the main memory, making memory access by PIM cores much faster than by CPUs. Researchers have already shown significant performance improvements on applications, such as embarrassingly parallel, data-intensive algorithms and pointer-chasing traversals in sequential data structures, in PIM architectures.

In this paper, we explore ways to design efficient concurrent data structures in the PIM model. The concurrent data structures we consider include linked-list and skip-list, as examples of pointer-chasing data structures, and FIFO queue, as an example of contended data structures. Designing and implementing concurrent data structures with the normal DRAM memory is known to be notoriously hard for non-experts. We show that in the PIM model, PIM-managed concurrent data structures can be much simpler. With the help of different optimizations, such as combining, partitioning and pipelining, our PIM-managed concurrent data structures can in theory be better than, or at least as good as, all other existing algorithms we are aware of, based on our performance model. Our preliminary experiments that indirectly compare our PIM-managed concurrent data structures with other algorithms also indicate that our algorithms are expected to outperform others.

1 Introduction

2 Related Work

Researchers have studied the PIM model for decades (e.g., [25, 20, 11, 24, 23, 19, 12]). Implementations of PIM memory have become much more feasible recently due to the advancements in 3D-stacked technology that can stack memory dies on top of a logic layer [18, 21, 7]. Based on this technology, Micron and other vendors together implemented a prototype of PIM memory called the Hybrid Memory Cube [9] a few years ago. Since then, the PIM model has drawn a lot of attention in the computer architecture community. Different PIM-based architectures have been proposed, either for general purposes or for specific applications [2, 1, 26, 17, 6, 3, 5, 4, 8, 27, 28].

Besides PIM memory's low energy consumption and high bandwidth (e.g., [1, 26, 27, 4]), researchers have also explored the benefits of PIM memory's low memory access latency [21, 17, 6], which is what we focus on in this paper. 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 [17], 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 they have to correctly synchronize with one another in all possible situations. This makes designing concurrent data structures with correctness guarantees like linearizability [16] very challenging.

Moreover, no one has ever compared the performance of data structures in the PIM model with that of existing 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 the concurrent linked-list with fine-grained locks [13], and the one implemented using flat combining [14], a generic technique to design concurrent data structures. For skip-lists, we compare our implementation with the lock-free skip-list [15] and a skip-list with flat combining and partitioning optimization. For FIFO queues, we compare our implementation with the flat-combining FIFO queue [14] and the F&A-based FIFO queue [22]. As we will show in the paper, our simple PIM-managed concurrent data structures can in theory outperform those concurrent data structures, making PIM memory a promising platform to run concurrent data structures.

3 The Model

3.1 Hardware model

In the hardware model called the *PIM model*, multiple CPUs are connected to the main memory, via a shared crossbar network, as illustrated in Figure 1. Each CPU has access to a hierarchy of cache and the last-level cache is shared by them. 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. 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

[1]We can assume a PIM core is an in-order CPU with only small private L1 cache and it doesn't have some of the optimizations that full-fledged CPUs usually have.

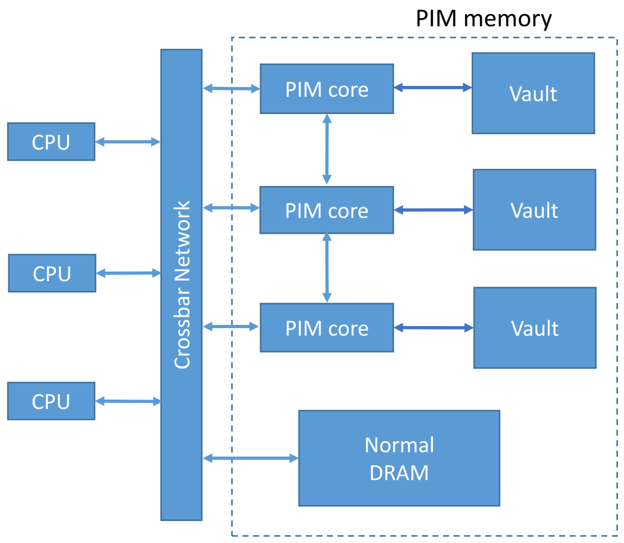

Figure 1: The PIM model

be accessed only by its local PIM core[2] and a PIM core can only access its local vault[3]. Although a PIM core is relatively slow computationally, it has much faster access to its local vault than a CPU does.

A PIM core communicates with other PIM cores and CPUs via messages. Each PIM, as well as each CPU, has a buffer 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 have the 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. Although A PIM core cannot access remote vaults directly, it can in theory send a message to the local PIM core of a remote vault to request for the data stored in that vault.

To keep the PIM memory simple, we assume 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, by having each PIM core maintain its own page table for its local vaults.

According to the Hybrid Memory Cube specification 1.0 [9], each HMC consists of 16 or 32 vaults and has size 2GB or 4 GB (so each vault has size roughly 100MB). Therefore, we assume the same specification in our PIM model, although the size of a PIM memory and the number of its vaults can be greater in theory.

3.2 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 that we will use to compare our PIM-managed algorithms

[2]We may alternatively assume vaults are accessible by CPUs as well, but at the cost of dealing with cache coherence between CPUs and PIM cores. Although some cache coherence mechanisms for PIM memory have be proposed and claimed to be not costly (e.g., [8]), we prefer to keep the hardware model simple and we will show that we are still able to design efficient concurrent data structure algorithms without CPUs accessing PIM vaults.

[3]We may alternatively assume a PIM core has direct access to remote vaults, at a larger cost. Even the PIM cores in our model are a little less powerful than that, we can still design efficient concurrent data structures with them.

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 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 in an arbitrary order, that is, they complete in times $\mathcal{L}_{atomic}, 2\mathcal{L}_{atomic}, ..., k\mathcal{L}_{atomic}$, respectively. Here we ignore the potential performance degrading of atomic operations when they compete for a contended cache line. As we will see in Sections 4 and 5, this is a conservative assumption in the sense that it sometimes overestimates the performance of some existing concurrent data structures in our analysis, making the comparisons a little unfair against our PIM-managed algorithms.

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 such 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 Pointer-Chasing Data Structure

In a pointer-chasing data structure, an operation to the data structure has to traverse over the data structure, following a sequence of pointers, before it completes. The examples we consider in the paper are linked-list and skip-list, where add(x), delete(x) and contains(x) operations have to go through a sequence of "next node" pointers until reaching the position of node x.

In a pointer-chasing data structure, the performance bottleneck is the procedure of the pointer chasing of operations. We focus on data structures much larger than the size of cache of CPUs such that a large portion of a data structure has to resides in memory at any time. Thus, the performance bottleneck of such a data structure is the memory accesses incurred during pointer chasing.

4.1 PIM-managed linked-list

The naive PIM-managed linked-list is simple: the linked-list is stored in a vault, maintained by the local PIM core. Whenever a CPU wants to make an operation to the linked-list, it sends the

PIM core a message containing a request of the operation and the PIM core will later retrieve the message and execute the operation. The concurrent linked-list can actually be implemented as a sequential linked-list, since it is accessible directly only by the PIM core.

Doing pointer chasing on sequential data structures by PIM cores is not a new idea (e.g., [17, 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[13].

To improve the performance of the PIM-managed linked-list, we can apply the following *combining optimization* to it: The PIM core retrieves all operation requests from its buffer and execute 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 [14], where, roughly speaking, threads (CPUs) compete for a "combiner lock" to become the combiner, and the combiner will take over all operation requests from other CPUs and execute them. Hence, we think the performance of the flat-combining linked-list is a good indicator to 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 a linked-list consists of nodes whose keys are integers 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 of the CPUs are generated the same way. To simplify the calculations, we assume CPUs only make contains() requests (or the number of *add*() requests is almost 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. The approximate expected throughputs (per second) of the concurrent linked-lists are as follows, given $n >> p$ and $N >> p$.

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

where $\mathcal{S}_p = \sum_{i=1}^{n}(\frac{i}{n+1})^p$.[4]

Since $0 < \mathcal{S}_p \leq \frac{n}{2}$ and $\mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, it is not hard to see that the PIM-managed linked-list with combining optimization outperforms all other algorithms (in fact its throughput is at least

[4]We define the rank of an operation request to a linked-list as the number of pointers we have to traverse until we find the right position for it in the linked-list. $\mathcal{S}_p$ is actually 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 the pointer chasing procedure.

1.5 times the throughput of the second best algorithm, the one with fine-grained locks). Without combining, however, the PIM-managed linked-list cannot beat the linked-list with fine-grained locks when $p > 6$.

We have implemented the linked-list with fine-grained locks and the flat-combining link-list with and without combining optimization. We have run them on a node of 28 hardware threads in a machine and their throughputs are presented in Figure 2. The results meet our expectation based on the analytical results above. The throughput of the flat-combining algorithm without combining optimization is much worse than the algorithm with fine-grained locks. According to our analytical results, 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 away from the throughput of the one with fined-grained locks. However, with combining optimization, the performance of the flat-combining algorithm improves significantly and the estimated throughput of our PIM-managed linked-list with combining optimization finally beats all others'.

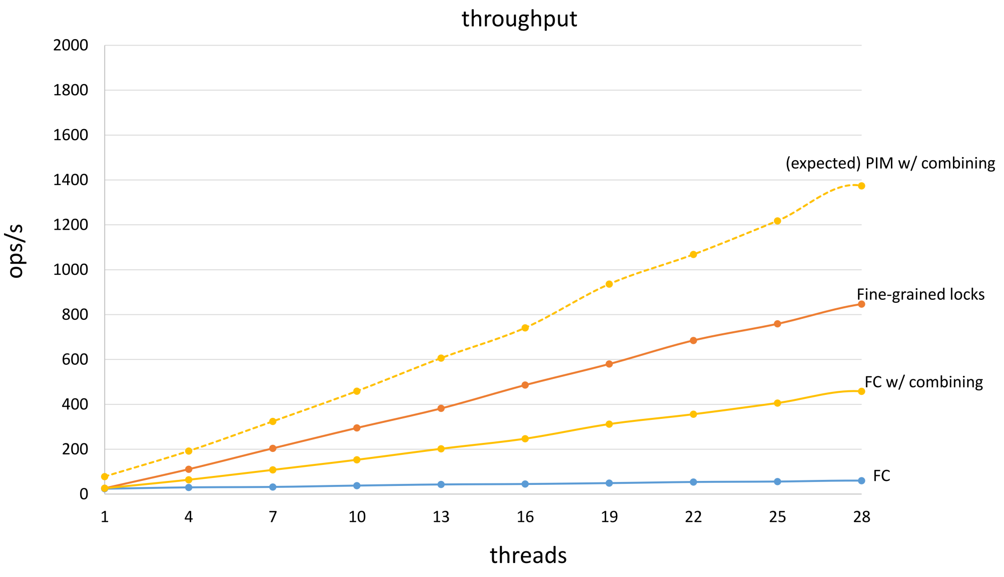

Figure 2: Experimental results of linked-lists

4.2 PIM-managed skip-list

4.2.1 Base algorithm

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 which executes those operations. As we will see, this algorithm is less efficient than some existing algorithms either.

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 consist of many vaults and PIM cores. For instance, the first generation of Hybrid Memory Cube [9] has 32 vaults. Hence, a PIM-managed skip-list may achieve much better performance if we can exploit the parallelism of multiple vaults and PIM cores. Here we present our PIM-managed skip-list with *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 request 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 and each node in a skip-list has height between 1 and H_{max}. The key range a partition covers is the 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 of the PIM memory 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, the copies of those sentinel nodes can almost for certain stay in cache if CPUs access them frequently.

When a CPU wants to make an operation of some key to the skip-list, it first compares the key with those of the sentinels, finds out the vault the key belongs to, and then sends its operation request to the PIM core of the vault. Once the PIM core retrieves the request, it executes the operation in the local vault and finally sends the result back to the CPU.

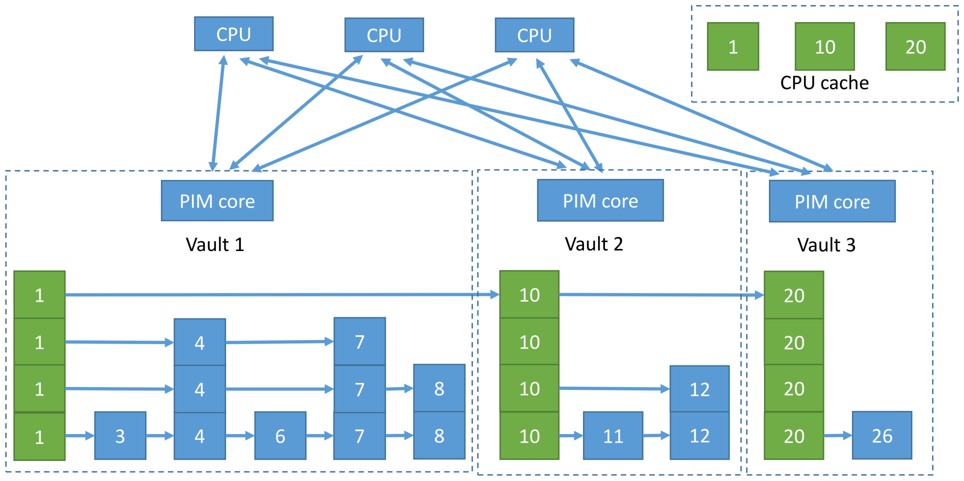

Figure 3: A PIM-managed FIFO queue with three partitions

Now let us discuss how to implement our 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. In this case, we can initialize k partitions starting with fake sentinel nodes $0, 1/k, 2/k, ..., (n-1)/k$, respectively. We allocate one partition in a different vault and hence k vaults cover k disjoint key ranges of size same. The sentinel nodes will never be deleted. If a new node to be added has the same key as a sentinel node, we add it immediately after the sentinel node.

We compare the performance of our PIM-managed skip-list with partitions with the performance of a flat-combining skip-list [14] with modifications and a lock-free skip-list [15], in a machine with p CPUs and k PIM vaults. 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 almost the same as the number of *delete*() so that the size of each skip-list nearly doesn't change). Their approximate expected throughputs are as follows:

- Look-free skip-list: $\frac{p}{\beta \mathcal{L}_{cpu}}$

- Flat-combining skip-list without partitioning: $\frac{1}{\beta\mathcal{L}_{cpu}}$
- PIM-managed skip-list without partitioning: $\frac{1}{(\beta\mathcal{L}_{pim}+\mathcal{L}_{message})}$
- Flat-combining skip-list with k partitions: $\frac{k}{\beta\mathcal{L}_{cpu}}$
- PIM-managed linked-list with k partitions: $\frac{k}{(\beta\mathcal{L}_{pim}+\mathcal{L}_{message})}$

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). The flat-combining skip-list with k partitions is one that has k combiner locks, each for a partition, so that CPUs compete for locks of partitions their operation keys belong to. Note that in the analysis above, we 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 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 outperformance other algorithms when $k > p/3$.

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 and better, implying the effectiveness of 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 we have shown, we can triple the throughput of a flat-combining skip-list to get the expected performance of a PIM-managed skip-list. As the figure 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.

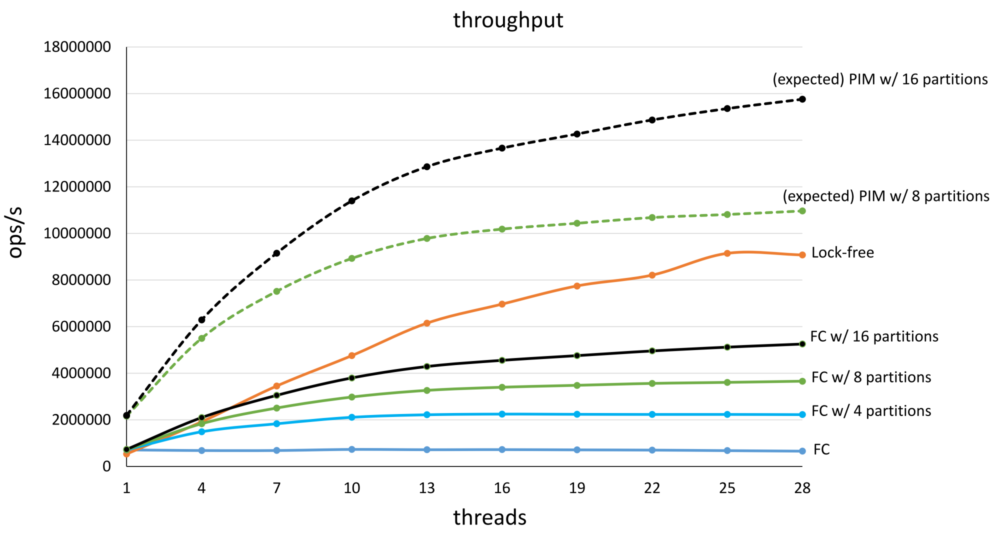

Figure 4: Experimental results of skip-lists

4.2.2 Rebalancing skip-list

We have shown that our PIM-managed skip-list performs well with uniform distribution of requests. With non-uniform distribution of requests, we may need to periodically rebalance the skip-list in order to maintain good performance. Here we discuss how to migrate consecutive nodes from one vault to another without blocking requests.

To move consecutive nodes from its local vault to another vault v, a PIM core p can send messages of requests of adding those nodes to the local PIM core q of v as follows. First, p sends a message notifying q of the start of the node 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 those nodes have been migrated, p sends notification messages to CPUs so that CPUs can update their copies of sentinel nodes accordingly. Once 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 v finishes its node migration.

During this node migration, p can periodically check its message buffer for requests from CPUs. Assume that a request with key k_1 is sent to p when p is migrating nodes in a key range that k_1 is in. If p is about to send a message to migrate a node with k_2 at the moment, where $k_1 \geq k_2$, p serves the request itself. Otherwise p forwards the request to q. In either case, p can then continue its node migration without blocking concurrent requests. It is not hard to see the correctness of the node migration procedure with the presence of requests: a request will eventually reach the vault that currently contains nodes in the key range that the request is in. If a request later arrives to p which no longer holds the partition the request belongs to, p can simply reply with a rejection to the sender CPU and the sender CPU will resend its request to the correct PIM core, because the CPU has already updated its sentinels and knows which PIM core it should contact.

Using this node migration protocol, our FIFO queue can support two rebalancing schemes: 1) If a partition has too many nodes, the local PIM core can divide it into two smaller partitions (the first nodes of two smaller partitions are modified to have the max height in order to serve as sentinels) and migrate one of them to another PIM vault; 2) If two consecutive partitions are both small, we can merge then by moving one to the vault containing the other. If rebalancing happens infrequently, its overhead is affordable.

5 Contended Data Structure

In a contended concurrent data structure, operations have to compete for accessing some contended spots and such contention is the performance bottleneck of the data structure. Examples are stack and different kinds of queues.

The contended data structure we focus on is FIFO queue, where concurrent enqueue and dequeue operations need to compete for the head and tail of the queue, respectively. In existing concurrent FIFO queue algorithms, enqueue and dequeue usually make CAS or F&A to compete for the head and tail pointers of the queue.

A contended data structure like a FIFO queue usually has good cache locality and doesn't need long pointer chasing to complete an operation. In a concurrent FIFO queue, for instance, the head and tail pointer can stay in cache if they are accessed frequently by CPUs directly, and each enqueue or dequeue operation only needs to access and update one or two pointers before completing its operation. One may think the PIM memory is therefore not a suitable platform for such a data structure, since now we cannot make good use of the fast memory access of PIM cores, but also lose the performance boosting provided by CPUs' cache. However, we are going to show a somewhat counterintuitive result that we can still design a PIM-managed FIFO queue that outperforms other existing algorithms.

5.1 PIM-managed FIFO queue

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 that contains 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 pop out the node at the tail of the dequeue segment and send the node back to the CPU.

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 one than one segment at the moment, the dequeue segment is deleted and the segment that were 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.) When 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 maintained in a single vault.

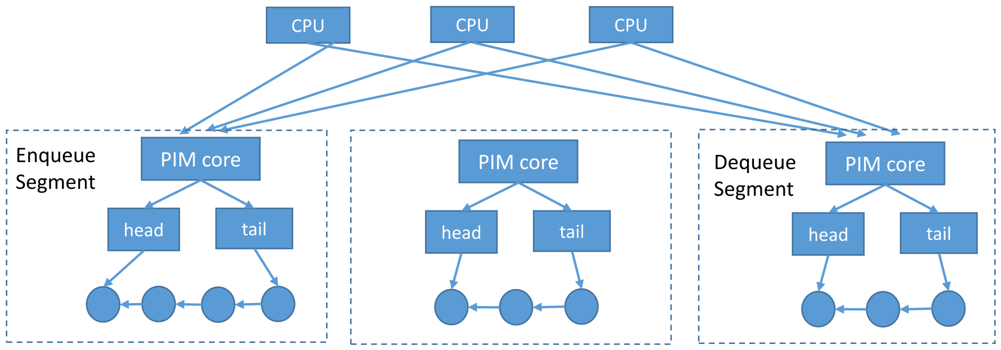

Figure 5: A PIM-managed FIFO queue with three segments

The pseudocode of our FIFO queue algorithm is presented in Figures 6-8. Each PIM core has local variables enqSeg and deqSeg that are references to segments. When enqSeg (respectively deqSeq) is not null, it is a reference to the enqueue (respectively dequeue) segment and 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

[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.

```
input  : enq(cid, $u$)
begin
    if $enqSeg == null$ then
        send message(cid, false);
    else
        if $enqSeg.head \neq null$ then
            enqSeg.head.next = $u$;
            enqSeg.head = $u$;
        else
            enqSeg.head = $u$;
            enqSeg.tail = $u$;
        end
        enqSeg.count = enqSeg.count + 1;
        send message(cid, true);
        if $enqSeg.count > threshold$ then
            cid$'$ = the CID of the PIM core chosen to maintain the new segment;
            send message(cid$'$, newEnqSeg());
            enqSeg.nextSegCid = cid$'$;
            enqSeg = null;
        end
    end
end
```

Figure 6: PIM core's execution upon receiving an enqueue request.

```
input  : deq(cid)
begin
    if $deqSeg == null$ then
        send message(cid, false);
    else
        if $deqSeg.tail \neq null$ then
            send message(cid, deqSeg.tail);
            deqSeg.tail = deqSeg.tail.next;
        else
            if $deqSeg == enqSeg$ then
                send message(cid, null);
            else
                send message(deqSeg.nextSegCid, newDeqSeg());
                deqSeg = null;
                send message(cid, false);
            end
        end
    end
end
```

Figure 7: PIM core's execution upon receiving a dequeue request.

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

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

Figure 8: Functions to create and retrieve new segments.

communicate via message(cid, content) calls, where cid is the 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 in Figure 6). If so, the PIM core enqueues u (lines 5-13), 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 14). If so, it chooses the PIM core of another vault for creating and maintaining a new enqueue segment and sends a message with a newEnqSeg() request. Finally it 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 Figure 8 shows, The PIM core receiving this newEnqSeg() request creates a new enqueue segment and enqueues the segment into 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 if it still holds the dequeue segment (line 2 of Figure 7). If so, the PIM core dequeues a node and sends it back to the CPU (lines 5-7). Otherwise, the PIM core 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 15-17 in Figure 6.) Upon receiving the newDeqSeg() request, as illustrated in Figure 8, the PIM core retrieves the oldest segment it has created and makes it the new dequeue segment. 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() in

Figure 8) and wait until they 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. CPUs don't have to reply to those notifications in this case, but they need to send messages to (all) PIM cores to ask whether they are in charge of the enqueue segment, if their requests have failed. In either case, the correctness of the algorithm is guaranteed: at any time, there is only one enqueue segment and only one dequeue segment, 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 [14]. Even without this optimization, our algorithm still performs well, as we will show next.

5.2 Performance analysis

We will compare the performance of three concurrent FIFO queue algorithms—our PIM-manged FIFO queue, a flat-combining FIFO queue and the F&A-based FIFO queue [22]. 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 the slots in the FIFO queue to enqueue and dequeue nodes (see [22] for more details). The flat-combining FIFO queue we consider is based on the one proposed by [14], with a modifications that threads compete for two "combiner locks", one for enqueues and the other for dequeues. We also simplify it by assuming the queue is always non-empty, so that it doesn't have to deal with the synchronization issue 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 therefore enqueue and dequeue requests can be executed separately. Since creating and changing enqueue and dequeue segments happens very infrequently, its overhead is negligible and therefore ignored to simplify our analysis. (If the threshold of segment length at line 14 in Figure 6 is a large integer n, then, in the worst case, changing enqueue or dequeue segments 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, 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 $p\mathcal{L}_{atomic}$. Similarly, if we assume a CPU makes a request immediately after its previous request is completed, 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 operation requests of the queue 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 "flat combiner". The flat 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 flat combiner therefore makes two last-level cache accesses to each 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 $2p\mathcal{L}_{llc}$ and the throughput of the algorithm is at most $\frac{1}{2\mathcal{L}_{llc}}$.

Note that we has made quite optimistic analysis for the F&A-based and flat-combining algorithms by counting only the costs in part of their executions and ignoring their performance degrading under contention. For example, the latency of accessing and modifying queue nodes by the flat combiner is ignored here. For dequeue operations, this latency can be high: since nodes to be dequeued in a long queue is unlikely to be cached, the flat combiner has to make a sequence of memory accesses to dequeue them one by one. Also, the F&A-based algorithm may suffer performance degradation under heavy contention, because contended atomic operations, such as F&A and CAS, usually perform worse in practice.

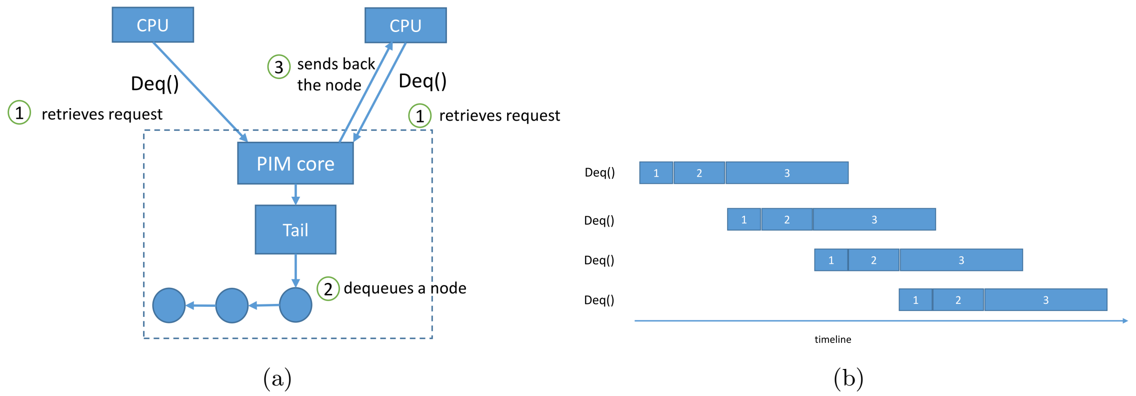

Figure 9: (a) illustrates the pipelining optimization, where a PIM core can start executing a new deq() (i.e., step 1 of deq() for the CPU on the left), without waiting for the dequeued node of the previous deq() to get back to a CPU (step 3 of the deq() for the CPU on the right). (b) shows the timeline of pipelining the three steps of 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 executions of requests, as illustrated in Figure 9(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 a CPU (step 3). After the PIM core sends off the message containing the node, it immediately retrieves the next request from its buffer, 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 9(b)). Note that, during the execution of a dequeue, the PIM core only needs 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 PIM core's cost of two L1 cache accesses, doing local computation and sending off one message, which is negligible in our performance model. If each CPU makes another request immediately after it receives the result of its previous request, we can prove that the throughput of the PIM-managed FIFO queue is

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

Therefore, the throughput of the PIM-managed queue is expected twice that 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 enqueue and dequeue segments are held by two different PIM cores.

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

We have run preliminary experiments for the flat-combining queue and the F&A-based queue. To be consistent with our analysis above, we have modified the two algorithms in order to measure the performance of the parts we focused in our analysis. More specifically, in the flat-combining queue, the flat combiner does not make real enqueue or dequeue operations for requests. Instead, after retrieving a request from the publication list, the flat combiner simply waits time t_1 and then writes a fake result back, where t_1 is the average execuion time of enqueuing or dequeuing a node in the original algorithm. This way, we can easily use Linux perf to count the last-level cache accesses incurred on the publication list, in which we are interested, while simulating the rest of the algorithm we omit. In the F&A-based queue, each thread only makes a F&A operation on one of the two shared objects to compete for a slot of the queue, and the rest of the algorithm, which does the real enqueue and dequeue operations, is omitted. Again, to simulate the latency of the operations omitted, each thread stays idle for time t_2 after the F&A operation, where t_2 is the average execution time of those operations in the original algorithm.

The experiments were run on a machine with 14 cores, each having two hyperthreads. To evaluate the algorithms with and without hyperthreading, we chose two settings for the experiments: 1) (without hyperthreading) for $1 \leq n \leq 14$, each of the n threads was pinned to a hyperthread of a different core, and 2) (with hyperthreading) for $1 \leq n \leq 28$ and any $1 \leq i \leq n/2$, the ith pair of threads were pinned to the two hyperthreads of the ith core respectively. The results of our experiments are presented in Figure 10. In each experiment, each thread made 10^7 requests and we counted the average number of last-level cache accesses a request incurred.

In the the Flat-combining queue without hyperthreading (Figure 10(a)), each request incurred on average roughly 4.8-4.9 last-level cache accesses. Note that a thread submitting a request can make at most 3 last-level cache accesses: one when it writes the request into the publication list, one when it tries to acquires the lock, one when it reads the result returned by the flat combiner. Thus, we can conclude that the flat combiner incurred at least 1.8-1.9 last-level cache accesses on average for a request, and this result is very close to our expectation (i.e., $2\mathcal{L}_{llc}$ per request). With hyperthreading(Figure 10(b)), each request incurred 4.0-4.5 last-level cache accesses when we had more than 10 threads, and therefore the flat combiner incurred at least 1.0-1.5 last-level cache accesses per request. Although this improves the performance by 90%-20%, our PIM-managed queue is still better in theory, given that the performance of our algorithm is expected twice the performance of the Flat-combining queue without hyperthreading in our performance model.

In the F&A-based queue without hyperthreading (Figure 10(c)), each request incurred almost one last-level cache access, which is a F&A operation, meeting our expectation (i.e., $\mathcal{L}_{atomic}$ per request) in earlier analysis. With hyperthreading (Figure 10(d)), each request made roughly 0.55-0.8 last-level cache access (F&A operation), improving the performance by 82%-25%. Given that the PIM-managed queue's performance is expected three times better than the F&A-based queue's in our analysis, the PIM-managed queue should still outperform the F&A-based queue with hyperthreading.

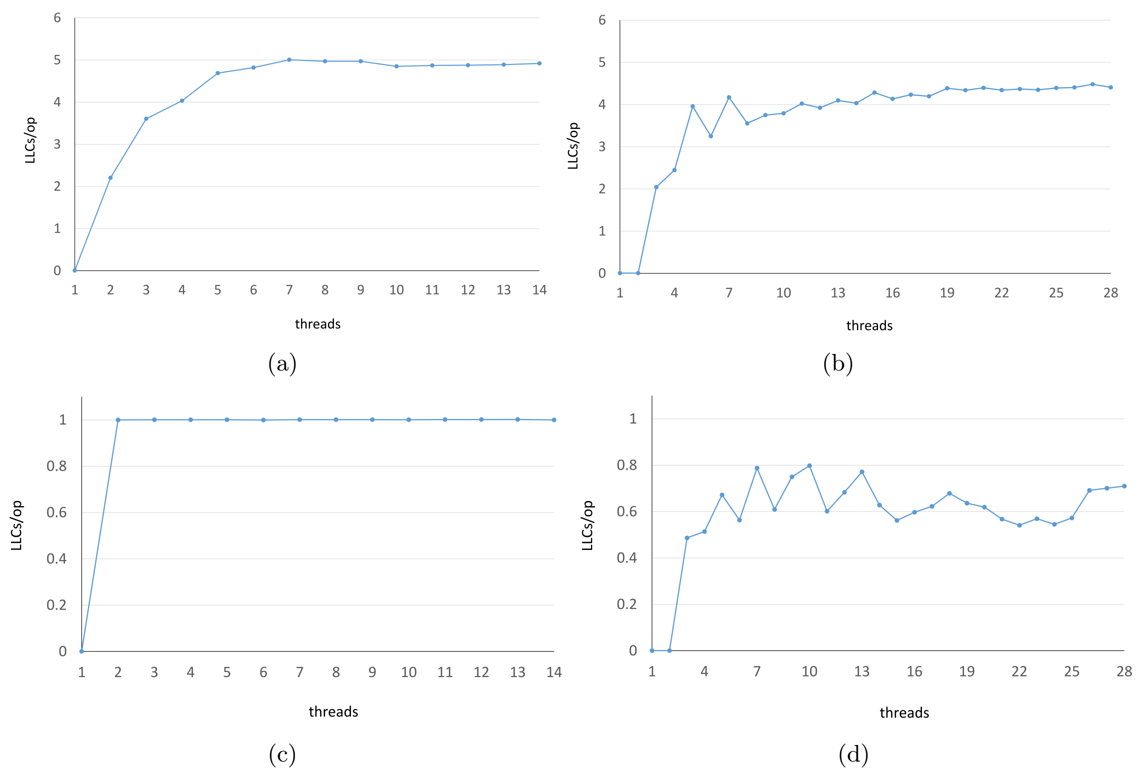

Figure 10: (a) Flat-combining queue without hyperthreading. (b) Flat-combining queue with hyperthreading. (c) F&A-based queue without hyperthreading. (d) F&A-based queue with hyperthreading.

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

A Parallelism of memory accesses

If the distribution of requests to the PIM-managed skip-list is very sharp, that is, a lot of requests hit a small range of keys, we may have to rebalance the skip-list too often, incurring significant performance overhead. However, if we don't rebalance the skip-list frequently, a large number of requests may be sent one PIM core, which becomes the performance bottleneck. On the other hand, even existing concurrent skip-list algorithms in which threads execute their operations by themselves in parallel cannot have good performance either, since concurrent updates (i.e., add() and delete()) within a small range conflict with one another. Thus, we believe a PIM-managed skip-list can achieve competitive throughput if each PIM core can execute multiple read-only requests (i.e., contains()) and at most one update request efficiently. We have assumed each PIM core can only deal with one request at a time, but in fact it has the potential to execute multiple requests in parallel.

Hsieh et al. [17] proposed how to design a PIM core that can make multiple memory accesses for multiple pointer-chasing based requests in parallel. The idea is that, after requiring a memory access for an operation request A, it can immediately retrieve another request B without waiting for the data of the memory access for A. The PIM core will later resume A once the data of the memory access is returned. In other words, the PIM core can hide its memory access latency by making multiple accesses in multiple requests in parallel. In theory, if the sum of the latency of doing the computation between two memory accesses for a pointer-chasing based request and the latency of switching from one request to another is only $1/k$ of the latency of a memory access by the PIM core, the PIM core can essentially execute k requests in parallel without delaying the pointer-chasing procedure of each request (see [17] for more details). Those requests can therefore be thought of as requests executed in parallel by different hardware threads of the PIM core.

Now we explain in detail how each kind of requests to a PIM-managed skip-list is executed. For a contains(x) request of key x, a PIM core simply does the search procedure for key x as in a normal sequential skip-list algorithm. For an add(x) request, The PIM core also does the same search procedure first. If a node of key x already exists in the skip-list, the execution is completed. Otherwise, the search procedure must has collected the predecessor and successor nodes between which a new node of key x will be inserted (see [15] for more details, where the search procedure is denoted as the find function). The new node of random height is then generated and inserted **from the bottom layer up**: at each layer, the PIM core first sets the "next node" pointer of the new node to point to its successor at the layer and then sets the "next node" pointer of its predecessor at the layer to point to the new node. Similarly, for a delete(x) request, the PIM core first does the search for key x. If a node with key x is in the skip-list, the PIM core removes it **from the top layer down**: at each layer, the PIM core sets the "next node" pointer of the node's predecessor to point to its successor. To free the memory space of the node, the PIM core can later physically deleted it, either in a quiescent state, or after the PIM core have completed another k requests, since there are at almost k contains() requests executed concurrently with the delete(x) request and requests after that will never reach the node.

Now we prove that this PIM-managed skip-list is linearizable.

Proof. What we need to prove is that each partition of the PIM-managed skip-list, which is essentially a skip-list within a smaller range of keys in a single PIM vault, is linearizable when there are at most one update request and multiple contains() executed concurrently.

We start with the linearization points of requests. For a contains(x) request that finds a node with key x in the skip-list at some layer, its linearization point is the moment the PIM core reads the predecessor of the node at that layer. For a contains(x) request that doesn't find a node with key x, its linearization point is the moment the PIM core reads the predecessor of the first node with key greater than x at the bottom layer (i.e., the node at the bottom layer in the "predecessor array" return by the find(x) function [15]). For an unsuccessful update to x, that is, a delete(x) that doesn't find an existing node with key x or an add(x) that finds an existing node with key x), its execution is in fact the same as that of a contains(x) and so does its linearization point. For a successful add(x) (which doesn't find an existing node with key x and needs to insert a new node), its linearization point is the moment the PIM core modifies the "next node" pointer of the predecessor of the new node at the bottom layer to point to the new node (that is, the step of inserting the new node at the bottom layer). For a successful delete(x) (which finds an existing node with key x), its linearization point is the moment the PIM core modifies the "next node" pointer of the predecessor of the node to point to the successor of the node at the bottom level (that is, the step of removing the node at the bottom layer).

Since update requests are executed sequentially and contains() requests are read-only, it is obvious that the results (i.e., responses) of update requests are consistent with the sequential history defined by the linearization points we just described. To see that a contains(x) request is also correctly linearized, we first consider the case where the last successful update to x linearized before contains(x) is a successful add(x) and the first successful update to x linearized after contains(x) is a successful delete(x). We can prove that contains(x) must find a node with key x, consistent with the sequential history. To prove it by contradiction, we assume contains(x) doesn't find such a node and hence is linearized at the moment it reads the predecessor of the first node with key greater than x at the bottom layer. However, since the last successful update to x linearized before contains(x) is a successful add(x), a node with key x must have been inserted between the node contains(x) is about to read and the first node with key greater than x at the first layer, according to the way we linearize a successful add(x). Therefore the node contains(x) is about to read cannot pointer to a node with key greater than x, a contradiction, and hence contains(x) must find a node with key x. By very similar proofs that are omitted here, we can show that a contains(x) is also correctly linearized when the last successful update to x linearized before contains(x) is a successful delete(x), or the first successful update to x linearized after contains(x) is a successful add(x). This completes the proof. □