PIM-Managed Concurrent Data Structure

1 The Model

1.1 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 **??**. Each CPU also has access to a hierarchy of cache and the last-level cache is shared among all CPUs. The main memory is a piece of DRAM divided into multiple partitions, called *vaults*. Each vault has a *PIM core* attached to it directly. we say a vault is *local* to the PIM core attached to it, and vice versa. A PIM core is a lightweight CPU that is slower than a full-fledged CPU with respect to computation speed.[1] A CPU can access any vault, but a PIM core can only access its local vault.[2] 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 destination. 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 the vault.

A PIM core can only make read and write operations to its local vault, while a CPU also supports more powerful primitives, such as CAS and F&A.

1.2 Other assumptions

(TBD: virtual memory support and cache coherence)

1.3 Performance model

Based on the latency numbers in prior work on PIM memory [], we have the following simple performance model that we will use to compare our PIM-managed algorithms with

[1] We can assume a PIM core is an in-order CPU with small local L1 cache and it does't have some optimizations or hierarchical cache that full-fledged CPUs usually have.

[2] Some papers (e.g., []) assume a PIM core can have direct access to a remote vault, 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 as we will show in the paper.

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. When there are k atomic operations competing for a cache line concurrently, we assume 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 2 and 3, 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.

Given that a message 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 destination is $\mathcal{L}_{message} = \mathcal{L}_{cpu}$. We make a conservative assumption that the latency of a message 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 destination. 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.

2 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.

2.1 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 accessed directly only by the PIM core.

Doing pointer chasing on sequential data structures by PIM cores is not a new idea. 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[].

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 [], where, roughly speaking, CPUs (i.e., threads) 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: $\frac{2p}{(n+2)\mathcal{L}_{cpu}}$
- Flat-combining linked-list without combining optimization: $\frac{2}{(n+2)\mathcal{L}_{cpu}}$
- PIM-managed linked-list without combining optimization: $\frac{2}{(n+2)\mathcal{L}_{pim}}$
- Flat-combining linked-list with combining optimization: $\frac{p}{(n+1-\mathcal{S}_p)\mathcal{L}_{cpu}}$
- PIM-managed linked-list with combining optimization: $\frac{p}{(n+1-\mathcal{S}_p)\mathcal{L}_{pim}}$

where $\mathcal{S}_p = \sum_{i=1}^{n} (\frac{i}{n+1})^p$.[3]

Since $0 < \mathcal{S}_p \leq \frac{n}{2}$ and $\mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, it is not hard to figure out that the PIM-managed linked-list with combining optimization outperforms all other algorithms (in fact its throughput is at least 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 server and their throughputs is presented in Figure **??**. 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 fined-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'.

2.2 PIM-managed skip-list

2.2.1 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 Hybrid Memory Cube released by Micron [] 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.

[3] 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 each pointer chasing prosedure.

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 have access to a copy of each sentinel node 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 compares the key with those of the sentinel copies, 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 **??** illustrates structure of such a PIM-managed skip-list.

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 flat-combining skip-list and lock-free skip-list [], 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 analytical results above, we ignored some overheads in the flat-combining algorithms, such as maintaining combiner locks and combining arrays (we will discuss combining arrays in more detail in Section 3). 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 **??**. 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.

2.2.2 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 show how to migrate consecutive nodes from one vault to another without blocking requests, which is the key to achieving rebalancing.

To move consecutive nodes from its local vault to another vault v, a PIM core p can simply send messages of requests of adding those nodes to the local PIM core of v. The messages are sent in an ascending order according to the keys of the nodes, and p can delete a node locally once the request of adding the node has been sent off. After the local PIM core of vault v has acknowledged the completion of adding those nodes, p can send notification messages to CPUs so that CPUs can update their copies of sentinel nodes accordingly. To keep the node migration protocol simple, we don't allow the local PIM core of v to move those nodes to another vault again until v finishes its node migration.

Assume a request with key k_1 in the range of those nodes is sent to p during the node migration when p is about to send a message to the PIM core of vault v for adding a node with k_2. If $k_1 \geq k_2$, p serves the request itself, and otherwise forwards it to the other PIM core. In either case, p can then resume its node migration. It is not hard to see the correctness of the node migration procedure with the presence of a request: a request can always reach the vault that contains the range of nodes the request is in.

Using this node migration protocol, our FIFO queue can support the following 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 them are modified to have the max height in order to serve as sentinels) and migrate one 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.

2.2.3 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.

[] proposed how to modify a PIM core to support making 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 [] for more detail). 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 request 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 [] for more details, where the search procedure is denoted as the findNode 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 findNode(x) function in []). 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. □

3 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.

3.1 PIM-managed FIFO queue

The structure of our PIM-managed FIFO queue is shown in Figure **??**. A queue consists of a sequence of segments of consecutive nodes. Each segment has a unique ID (SID)—initially the single empty segment of the empty queue has SID=0 and each new segment has SID one greater than the id of the segment prior to it. Therefore, at any time of an execution, a queue consists of a sequence of segments of SIDs $k, k+1, ..., k+l$ (from right to left in the figure), where k and l are non-negative integers. Each 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 segments.

The segment with the largest SID of the queue is called the *enqueue segment* and the segment with the smallest SID 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 by making the head node point to it. Similarly, to dequeue a node, a CPU sends a dequeue request to the PIM core of the

```
   input  : enq(pid, u)
 1 begin
 2 |  if enqSeg == null then
 3 |  |  send message(pid, false);
 4 |  else
 5 |  |  if enqSeg.head ≠ null then
 6 |  |  |  enqSeg.head.next = u;
 7 |  |  |  enqSeg.tail.next = u;
 8 |  |  else
 9 |  |  |  enqSeg.head.next.next = u;
10 |  |  |  enqSeg.head.next = u;
11 |  |  end
12 |  |  enqSeg.count = enqSeg.count + 1;
13 |  |  send message(pid, true);
14 |  |  if enqSeg.count > threshold then
15 |  |  |  pid′ = the PID of the PIM core chosen to maintain the new segment;
16 |  |  |  send message(pid′, newEnqSeg(enqSeg.vid + 1));
17 |  |  |  enqSeg.nextSegPid = pid′;
18 |  |  |  enqSeg = null;
19 |  |  end
20 |  end
21 end
```

Figure 1: enq()

vault containing the dequeue segment and the PIM core will dequeue and send back to the CPU the node pointed to by the tail node of the dequeue segment.

When the length of enqueue segment exceeds some threshold, the PIM core maintaining it can notify another PIM core to create a new segment with a bigger SID as the new enqueue segment and that is why a long queue can consist of multiple segments.[4] As we can see, enqueue and dequeue operations can therefore be executed by two different PIM cores in parallel when the queue has more than one segment, which can double the throughput compared to a straightforward PIM-managed queue implementation maintained in a single vault.

[4]When and how to create a new enqueue segment can be decided in other ways. For example, a new enqueue segment can be created when the queue meets requirements other than a fixed threshold of length. Also, CPUs can decide which vault to create the new enqueue segment based on more complex criteria and send notifications to PIM cores by themselves (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  : deq(pid)
 1 begin
 2 |   if deqSeg == null then
 3 |   |   send message(pid, false);
 4 |   else
 5 |   |   if deqSeg.tail.next == null then
 6 |   |   |   node = null;
 7 |   |   else
 8 |   |   |   node = deqSeg.tail.next;
 9 |   |   |   deqSeg.tail.next = node.next;
10 |   |   end
11 |   |   send message(pid, node);
12 |   |   if deqSeg.tail.next == null then
13 |   |   |   if enqSeg == null || enqSeg.vid ≠ deqSeg.vid then
14 |   |   |   |   send message(deqSeg.nextSegPid, newdeqSeg(deqSeg.vid + 1));
15 |   |   |   |   deqSeg = null;
16 |   |   |   end
17 |   |   end
18 |   end
19 end
```

Figure 2: deq()

```
   input  : newEnqSeg(sid)
 1 begin
 2 |   enqSeg = new Segment() ;
 3 |   enqSeg.sid = sid ;
 4 |   segMap.add(sid, enqSeg) ;
 5 |   notify CPUs of the new enqueue segment;
 6 end
```

```
   input  : newDeqSeg(sid)
 1 begin
 2 |   deqSeg = segMap.get(sid) ;
 3 |   notify CPUs of the new dequeue segment;
 4 end
```

Figure 3: Functions to create new segments.

The pseudocode of our FIFO queue algorithm is presented in Figures 1-3. Each PIM core maintains local variables enqSeg and deqSeg that are references to segments of the queue. When enqSeg (respectively deqSeq) is not null, it is a reference to the enqueue (respectively dequeue) segment and indicates the PIM core is currently holding the enqueue (respectively dequeue) segment in its local vault. CPUs and PIM cores communicate via message(pid, content) calls, where pid is the unique processor ID (PID) of the receiver and the content can be any information like an operation request or a response to an operation request.

Once a PIM core receives an enqueue request enq(pid, u) of node u from a CPU whose PID is pid, the PIM core first checks if it is holding the enqueue segment (line 2 in Figure 1). If so, the PIM core enqueues node u (lines 5-13), and otherwise sends back a message informing the CPU that the enqueue operation has failed (line 3) so that the CPU can later resend its request to the right PIM core currently holding the enqueue segment (we will explain how the CPU can find the right PIM core). After enqueuing u, the PIM core may find the enqueue segment is longer than some threshold (line 14). If so, it chooses the PIM core of another vault for creating and maintaining a new enqueue segment with SID (enqSeg.sid + 1) by sending a message of a newEnqSeg(enqSeg.sid + 1) request, where enqSeg.sid is the SID of the current enqueue segment. Finally it sets its enqSeq to null indicating it no longer deals with enqueue operations. Note that the PID pid' of the PIM core chosen for creating the new segment is recorded in enqSeg.nextSegPid for future use in dequeue requests. As Figure 3 shows, The PIM core receiving this newEnqSeg request will create a new enqueue segment and insert the segment into its local hashmap segMap so that it can later retrieve the segment using the SID of the segment. Finally the PIM core 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(pid) from a CPU with PID pid, it first checks if it still holds the dequeue segment of the queue (line 2 of Figure 2). If so, the PIM core dequeues a node and sends it back to the CPU (lines 5-11). Otherwise, the PIM core informs the CPU that this operation has failed (line 3) and the CPU will have to resend its request to the right PIM core. If the PIM core finds the dequeue segment empty (line 12) after dequeuing a node and the dequeue segment is currently not the same as the enqueue segment (line 13), which indicates the FIFO queue is not empty, the PIM core sends a message of a newDeqSeg request to the PIM core with PID deqSeg.nextSegPid which must hold the next segment, according to how we create new segments in enqueue operations (see lines 15-17 in Figure 1). Upon receiving the newDeqSeg request, as illustrated in Figure 3, the PIM core retrieves the the segment with the right SID from segMap and notifies CPU that it is holding the new dequeue segment now.

Now we explain how CPUs and PIM cores coordinate, so that CPUs can find the right enqueue and dequeue segments after they have failed in their previous attempts due to the change of those segments. We only discuss how to deal with the enqueue segment in the paper, since the method for updating the information of dequeue segment is almost

identical. A straightforward way to inform CPUs is to have the owner PIM core of the new enqueue segment send notification messages to them (line 5 of newEngSeg in Figure 3). A CPU that failed its previous enqueue request can make another attempt once it receives the notification. CPUs can also help one another by writing the SIDs of the new enqueue segments, together with owners of those segments, into a shared variable using CAS operations. If CPUs only try to write SIDs bigger than the one in the shared variable, the SIDs of newer segments are guaranteed not to be overwritten by the obsolete ones of older segments, as they are bigger than older ones'. It is not hard to see the shared variable can finally contain the information of the latest enqueue segment. Thus, a CPU that doesn't receive a latest notification in time can still get the information by checking the variable and then make its request again.

It is easy to see why our PIM-managed FIFO queue algorithm is linearizable. The linearization point of an enq(u) is the moment when node u is enqueued, either at line 7 or at line 10 in Figure 1. Similarly, the linearization point of a deq() is the moment when an attempt to dequeue a node is completed, either at line 6 or at line 9 in Figure 2. Note that we have also covered the requests sent to wrong PIM cores that are not holding the enqueue and dequeue segments, since those requests will eventually be sent to the right PIM cores before they can complete.

When the enqueue segment is also the dequeue segment, the PIM core maintaining it executes requests one by one sequentially, so obviously requests are linearizable. When the enqueue segment is different from the dequeue segment, enq() and deq() requests are executed by different PIM cores on different segments. Therefore, an enq() and a deque() are isolated and can be linearized in an arbitrary order. In this case, enq() requests are still sequentially executed by a PIM core, and so are the deque() requests. Obviously, the linearization pointers of requests can linearize requests correctly.

3.2 Performance analysis

We will compare the performance of three concurrent FIFO algorithms—our PIM-manged FIFO queue, a flat-combining FIFO queue and a F&A-based FIFO queue proposed by []. The F&A-based FIFO queue is the best 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 [] for more detail). The flat-combing FIFO queue we consider is proposed by [], where threads compete for two combiner locks, one for enqueues and the other for dequeues, and we simplify it by assuming the queue is always non-empty so that it doesn't have to deal with the synchronization between an enqueue and a dequeue when it is empty.

We start with the performance analysis of the algorithms when a queue is long enough such that the PIM-managed FIFO queue has more than one segment and hence enqueue and dequeue requests are executed separately. Since creating and changing enqueue and dequeue segments happens infrequently, we ignore its overhead and assume it never happens

in order to simplify our performance analysis. (If the threshold of segment length at line 14 in Figure 1 is some large integer n, then, in the worst case, changing enqueue segment can happen only once every n enqueues and changing dequeue segment can happen only once every n dequeues. Also, the cost of do so is only the latency of sending one message and a few steps of local computation.)

Since enqueues and dequeues are isolated, we focus on dequeues here, and the analysis of enqueues is almost identical. Assume there are p concurrent dequeue requests by p CPUs. Since each CPU 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 $\frac{1}{\mathcal{L}_{atomic}}$.

In the flat-combining FIFO queue, the flat combiner has to go through the combining array to retrieve pending requests and later write results back to the combining array. Since other CUPs with pending requests spin on this shared combining array, each access by the flat combiner to a slot of the combining array incurs at least a last-level cache access. Therefore, 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 have made very conservative analysis for the F&A-based and flat-combining algorithm by counting only the costs in part of their executions and ignoring their performance degrading under contention. For example, under heavy contention, the F&A-based algorithm may suffer significant performance degradation because contended atomic operations, such as F&A and CAS, usually perform much worse in practice. Also, the latency of accessing and modifying nodes by a flat combiner is ignored here. For dequeue requests, this latency can be very high: since nodes to be dequeued in a long queue is unlikely to be cached, a flat combiner has to make a sequence of memory accesses to dequeue them one by one.

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. However, its performance can be much improved, if we have the PIM core *pipeline* the executions of requests as follows. 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, dequeues a node, and sends the node back to a CPU. Note that, after the PIM core sends off the message containing the node, it can immediately retrieve the next request in its buffer, without waiting for the message to arrive at its destination (see Figure **??**). Therefore, the PIM core can pipeline requests by overlapping the latency of message transfer and the latency of memory accesses and local computations in multiple request. 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 when the tail node is accessed frequently and hence cached by the PIM core. It is easy to show that the execution time of p

requests is only $\mathcal{L}_{message} + p(\mathcal{L}_{pim} + \epsilon) + \mathcal{L}_{message}$, where ϵ is the total latency of two L1 cache accesses, local computation and sending off the message, which is negligible in our performance model. If we assume a CPU makes a request immediately after its previous request is completed, we can prove that the throughput of the PIM-managed FIFO queue is

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

Hence, the throughput of the PIM-managed FIFO queue is expected at least twice the throughput of each of the other two algorithms in our performance model assuming $\mathcal{L}_{atomic} = 3\mathcal{L}_{llc} = 3\mathcal{L}_{pim}$, when the PIM-managed FIFO queue has more than one segment.

When a queue is short, the PIM-managed FIFO queue may always have only one segment which deals with both enqueue and dequeue requests. In this case, the throughput of the PIM-managed FIFO queue is only half of that in the case when the queue consists of more than one segment, but still at least as good as the other two algorithms.