Concurrent Data Structures for Near-Memory Computing

ABSTRACT

The process-in-memory (near-memory or 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. We consider 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.

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

1. INTRODUCTION

The performance gap between memory and CPU has grown exponentially. Memory vendors have focused mostly on improving memory capacity and bandwidth, sometimes even at the cost of increased memory access latencies. To provide higher bandwidth with lower access latencies, hardware architects have proposed near-memory computing (NMC), where a lightweight processor is located close to memory. [details?] This is an old idea, that has been intensely studied in the past, but so far has not materialized. However, new advances in 3D integration [check?] and in die stacked memory [check?] make near-memory computing viable in the near future. [details?] This new technology promises to revolutionize the interaction between compute and data, as memory becomes an active component in managing the data. Therefore, it invites a fundamental rethinking of basic data structures and promotes a tighter dependency between algorithmic design and hardware characteristics.

But how do we design and optimize data structures for near-memory computing? And how do these algorithms compare to traditional CPU-managed concurrent data structures? To answer these questions, we develop a simplified model of the expected performance of NMC using the flat combining technique []. Using this model, we show that a naive NMC-managed data structure cannot outperform a carefully crafted CPU-managed *concurrent* data structure. In particular, the lower latency access to memory cannot compensate for the loss of parallelism. Server machines nowadays have hundreds of cores; algorithms for concurrent data structures exploit these cores to achieve high throughput and scalability. Therefore, to be competitive with traditional concurrent data structures, NMC data structures need new approaches to leverage parallelism. We investigate two classes of data structures. First, pointer chasing data structures, which have a high degree of inherent parallelism. We propose using techniques such as combining [] and partitioning the data across vaults to reintroduce parallelism for these data structures. Secondly, we evaluate contended data structures, such as FIFO queues, which are more likely to have the data in hardware caches, therefore not benefiting that much from NMC. We design a new NMC FIFO queue that exploits pipelining to outperform prior concurrent FIFO queues, despite their high locality benefit from caches.

Prior work has focused on using the increased bandwidth; the work that focused on the reduced latency ignored parallelism. For example, cite [] has shown that pointer chasing data structures can benefit from the shorter latency access, but did not evaluate the performance loss from the loss of parallelism.

2. PROCESSING IN MEMORY

2.1 Hardware model

In the PIM hardware model, multiple CPUs are connected to the main memory, via a shared crossbar network, as illustrated in Figure 1. The main memory consists of two parts—one is a normal DRAM accessible by CPUs and the other, called the *PIM memory*, is divided into multiple partitions, called *PIM vaults* or simply vaults. According to the Hybrid Memory Cube specification 1.0 [9], each HMC consists of 16 or 32 vaults and has total size 2GB or 4 GB (so each vault has size roughly 100MB). We assume the same

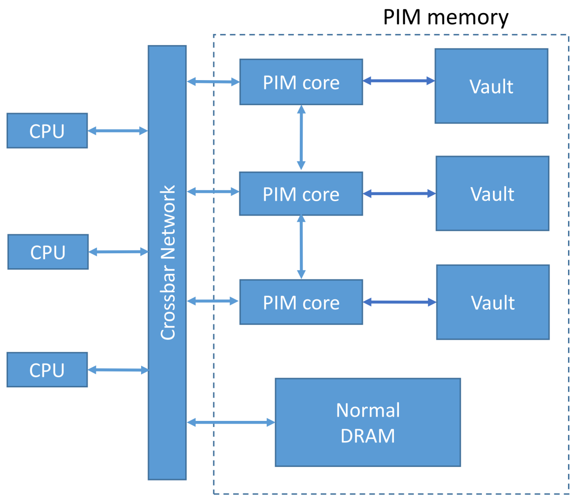


Figure 1: The PIM model

specifications in our PIM model, although the size of a PIM memory and the number of its vaults can be greater in theory. Each CPU also has access to a hierarchy of cache, coherent to the normal DRAM, and the last-level cache is shared among CPUs.

Each vault has a *PIM core* directly attached to it. we say a vault is *local* to the PIM core attached to it, and vice versa. A PIM core is a lightweight CPU that may be slower than a full-fledged CPU with respect to computation speed.[1] A vault can be accessed only by its local PIM core.[2] Although a PIM core is relatively slow computationally, it has fast access to its local vault.

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

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

2.2 Performance model

Based on the latency numbers in prior work on PIM memory, in particular on the Hybrid Memory Cube [9, 6], and on

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

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

the evaluation of operations in multiprocessor architectures [10], we propose the following simple performance model to compare our PIM-managed algorithms with existing concurrent data structure algorithms. For read and write operations, we assume

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

where $\mathcal{L}_{cpu}$ is the latency of a memory access by a CPU, $\mathcal{L}_{pim}$ is the latency of a local memory access by a PIM core, and $\mathcal{L}_{llc}$ is the latency of a last-level cache access by a CPU. We ignore the costs of cache accesses of other levels in our performance model, as they are negligible in the concurrent data structure algorithms we will consider. We assume that the latency of a CPU making an atomic operation, such as a CAS or a F&A, to a cache line is

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

even if the cache line is currently in cache. This is because an atomic operation hitting the cache is usually as costly as a memory access by a CPU, acorrding to [10]. When there are k atomic operations competing for a cache line concurrently, we assume that they are executed sequentially, that is, they complete in times $\mathcal{L}_{atomic}, 2\mathcal{L}_{atomic}, ..., k \cdot \mathcal{L}_{atomic}$, respectively.

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

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

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

3. POINTER-CHASING DATA STRUCTURE

Now we discuss how we design efficient PIM-managed pointer-chasing data structures. In a pointer-chasing data structure, an operation (e.g., an insertion) has to traverse over items of the data structure by following a sequence of pointers, until it finds the right place to apply itself. The examples we consider 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. The performance bottleneck of such an operation is usually the pointer chasing procedure. Here 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 reside in memory at any time. Thus, the performance bottleneck becomes the memory accesses incurred during pointer chasing.

3.1 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[3] wants to make an operation to the linked-list, it sends the PIM core a message containing a request of the operation. The PIM core will retrieve the message, execute the operation, and send the result back to CPU. The concurrent linked-list can actually be implemented as a sequential one, 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 apply the following *combining optimization* to it: the PIM core retrieves all pending requests from its buffer and executes all of them during only one traversal over the linked-list. It is not hard to see that the role of the PIM core in our PIM-managed linked-list is very similar to that of the combiner in a concurrent linked-list implemented using *flat combining* [14], where, roughly speaking, threads compete for a "combiner lock" to become the combiner, and the combiner will take over all operation requests from other threads and execute them. Therefore, we think the performance of the flat-combining linked-list is a good indicator 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 that a linked-list consists of nodes with integer keys in the range of $[1, N]$. Initially a linked-list has n nodes with keys generated independently and uniformly at random from $[1, N]$. The keys of the operation requests are generated the same way. To simplify the analysis, we assume that CPUs only make *contains*() requests (or the number of *add*() requests is the same as the number of *delete*() so that the size of each linked-list nearly doesn't change). We also assume that a CPU makes a new operation request immediately after its previous one completes. The approximate expected throughputs (per second) of the concurrent linked-lists are listed below, given $n \gg p$ and $N \gg p$:

- 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: $\frac{p}{(n-\mathcal{S}_p)\mathcal{L}_{pim}}$

[3]From now on, we use threads and CPUs interchangeably.

where $\mathcal{S}_p = \sum_{i=1}^{n}(\frac{i}{n+1})^p$.[4] It is easy to see that the PIM-managed linked-list with combining outperforms the linked-list with fine-grained locks, which is the best one among other algorithms, as long as $\frac{\mathcal{L}_{cpu}}{\mathcal{L}_{pim}} > \frac{2(n-\mathcal{S}_p)}{n+1}$. Given that $0 < \mathcal{S}_p \leq \frac{n}{2}$ and $\mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, the throughput of the PIM-managed linked-list with combining should be at least 1.5 times the throughput of the linked-list with fine-grained locks. Without combining, however, the PIM-managed linked-list cannot beat the linked-list with fine-grained locks when $p > 6$.

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

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

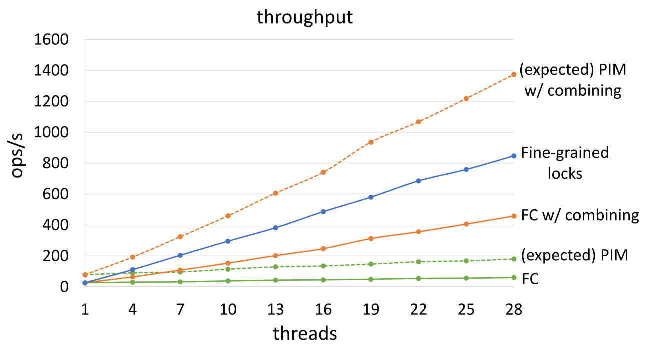


Figure 2: Experimental results of linked-lists

3.2 PIM-managed skip-list

3.2.1 *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 that executes those operations. As we will see, this algorithm is less efficient than some existing algorithms.

Unfortunately, the combining optimization cannot be applied to skip-lists effectively. The reason is that for any two nodes not close enough to each other in the skip-list, the paths we traverse through to reach them don't largely overlap.

On the other hand, PIM memory usually consists of many vaults and PIM cores. For instance, the first generation of Hybrid Memory Cube [9] has up to 32 vaults. Hence, a PIM-managed skip-list may achieve much better performance if we can exploit the parallelism of multiple vaults. Here we present our PIM-managed skip-list with a *partitioning optimization*: A skip-list is divided into partitions of disjoint ranges of keys, stored in different vaults, so that a CPU sends its operation request to the PIM core of the vault to which the key of the operation belongs.

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

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

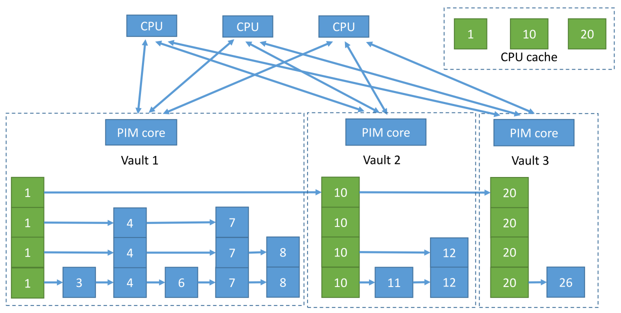


Figure 3: A PIM-managed FIFO queue with three partitions

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

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

node, we insert it immediately after the sentinel node.

We compare the performance of our PIM-managed skip-list with partitions to the performance of a flat-combining skip-list [14] and a lock-free skip-list [15], where p CPUs keeps making operation requests. We also apply the partitioning optimization to the flat-combining skip-list, so that k combiners are in charge of k partitions of the skip-list. To simplify the comparison, we assume that all skip-lists have the same initial structure (expect that skip-lists with partitions have extra sentinel nodes) and all the operations are contains() operations (or the number of *add*() requests is the same as the number of *delete*() so that the size of each skip-list nearly doesn't change). Their approximate expected throughputs are 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). Note that we have ignored some overheads in the flat-combining algorithms, such as maintaining combiner locks and publication lists (we will discuss publication lists in more detail in Section 4). We also have overestimated the performance of the lock-free skip-list by not counting the CAS operations used in add() and delete() requests, as well as the cost of retries caused by conflicts of updates. Even so, our PIM-managed linked-list with partitioning optimization is still expected to outperform the second best algorithm, the lock-free skip-list when $k > \frac{(\beta \mathcal{L}_{pim} + \mathcal{L}_{message})p}{\beta \mathcal{L}_{cpu}}$. Given that $\mathcal{L}_{message} = \mathcal{L}_{cpu} = 3\mathcal{L}_{pim}$, $k > p/3$ should suffice.

Our experiments have revealed similar results, as presented in Figure 4. We have implemented and run the flat-combining skip-list with different numbers of partitions and compared them with the lock-free skip-list. As the number of partitions increases, the performance of the flat-combining skip-list gets better, implying the effectiveness of the partitioning optimization. Again we believe the performance of the flat-combining skip-list is a good indicator to the performance of our PIM-managed skip-list. Therefore, according to the analytical results 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.

3.2.2 *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.

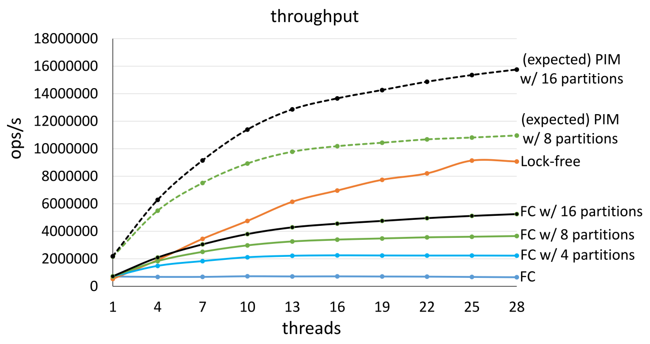


Figure 4: Experimental results of skip-lists

To do so, we can 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 requesting those nodes to be added to the local PIM core q of v as follows. First, p sends a message notifying q of the start of the migration. Then p sends messages of adding those nodes to q one by one in an ascending order according to the keys of the nodes. After all 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 p finishes its node migration.

During this node migration, p can still 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 containing k_1. If p is about to migrate a node with k_2 at the moment and $k_1 \geq k_2$, 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. This algorithm is correct in the presence of requests, because a request will eventually reach the vault that currently contains nodes in the key range that the request belongs to: If a request arrives to p which no longer holds the partition the request belongs to, p can simply reply with a rejection to the CPU and the CPU will resend its request to the correct PIM core, because it has already updated its sentinels and knows which PIM core it should contact now.

Using this node migration protocol, 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 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.

4. CONTENDED DATA STRUCTURE

In this section, we will discuss techniques of designing efficient contended data structures with the PIM memory. 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 compete for the head and tail of the queue, respectively. 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 pointers can stay in cache if they are accessed frequently by CPUs, and each enqueue or dequeue operation only needs to access and update one or two pointers before completing its operation. One may think that 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.

4.1 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 containing the enqueue segment. The PIM core will then insert the node to the head of the segment. Similarly, to dequeue a node, a CPU sends a dequeue request to the PIM core of the vault holding the dequeue segment. The PIM core will then pop out the node at the tail of the dequeue segment and send the node back to the CPU.

Initially the queue consists of an empty segment which acts as both the enqueue segment and the dequeue segment. When the length of enqueue segment exceeds some threshold, the PIM core maintaining it notifies another PIM core to create a new segment as the new enqueue segment.[5] When the dequeue segment becomes empty and the queue has other segments, the dequeue segment is deleted and the segment that 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.) If the enqueue segment is different from the dequeue segment, enqueue and dequeue operations can be executed by two different PIM cores in parallel, which doubles the throughput compared to a straightforward queue implementation held in a single vault.

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

[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.

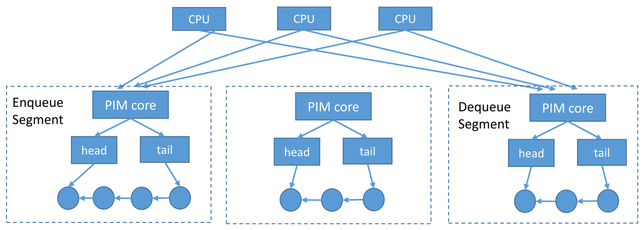


Figure 5: A PIM-managed FIFO queue with three segments

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 **??**). 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 sends a message with a newEnqSeg() request to the PIM core of another vault that is chosen to create a new enqueue segment. Finally the PIM core sets its enqSeq to null indicating it no longer deals with enqueue operations. Note that the CID cid' of the PIM core chosen for creating the new segment is recorded in enqSeg.nextSegCid for future use in dequeue requests. As Figure **??** shows, The PIM core receiving this newEnqSeg() request creates a new enqueue segment and enqueues the segment into its segQueue (line 3). Finally it notifies CPUs of the new enqueue segment (we will get to it in more detail later).

Similarly, when a PIM core receives a dequeue request deq(cid) from a CPU with CID cid, it first checks whether it still holds the dequeue segment (line 2 of Figure **??**). If so, the PIM core dequeues a node and sends it back to the CPU (lines 5-7). Otherwise, it informs the CPU that this request has failed (line 3) and the CPU will have to resend its request to the right PIM core. If the dequeue segment is empty (line 8) and the dequeue segment is not the same as the enqueue segment (line 11), which indicates that the FIFO queue is not empty and there exists another segment, the PIM core sends a message with a newDeqSeg() request to the PIM core with CID deqSeg.nextSegCid. (We know that this PIM core must hold the next segment, according to how we create new segments in enqueue operations, as shown at lines 15-17 in Figure **??**.) Upon receiving the newDeqSeg() request, as illustrated in Figure **??**, the PIM core retrieves from its segQueue 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 **??**) and wait until CPUs all send back acknowledgement messages. However, if there is a slow CPU that doesn't reply in time, the PIM core has to wait for it and therefore other CPUs cannot have their requests executed. A more efficient, non-blocking method is to have the PIM core start working for new requests immediately after it has sent off those notifications. A CPU does not have to reply to those notifications in this case, but if its request later fails, it needs to send messages to (sometimes all) PIM cores to ask whether a PIM core is currently in charge of the enqueue segment. In either case, the correctness of the algorithm is guaranteed: at any time, there is only one enqueue segment and only one dequeue segment, and only requests sent to them will be executed.

We would like to mention that the PIM-managed FIFO can be further optimized. For example, the PIM core holding the enqueue segment can combine multiple pending enqueue requests and store the nodes to be enqueued in an array as a "fat" node of the queue, so as to reduce memory accesses. This optimization is also used in the flat-combining FIFO queue [14]. Even without this optimization, our algorithm still performs well, as we will show next.

4.2 Pipelining and Performance analysis

We compare the performance of three concurrent FIFO queue algorithms—our PIM-manged FIFO queue, a flat-combining FIFO queue and a F&A-based FIFO queue [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 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 modification that threads compete for two "combiner locks", one for enqueues and the other for dequeues. We further simplify it based on the assumption that the queue is always non-empty, so that it doesn't have to deal with synchronization issues between enqueues and dequeues when the queue is empty.

Let us first assume that a queue is long enough such that the PIM-managed FIFO queue has more than one segment, and enqueue and dequeue requests can be executed separately. Since changes of enqueue and dequeue segments happen very infrequently, its overhead is negligible and therefore ignored to simplify our analysis. (If the threshold of segment length at line 14 in Figure **??** is a large integer n, then, in the worst case, changing an enqueue or dequeue segment happens only once every n requests, and the cost is only the latency of sending one message and a few steps of local computation.) Since enqueues and dequeues are isolated in all the three algorithms when queues are long enough, we will focus on dequeues, and the analysis of enqueues is almost identical.

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

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

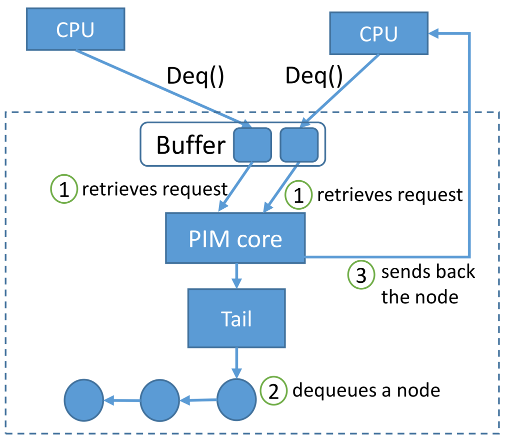
Note that we have made quite optimistic analysis for the F&A-based and flat-combining algorithms by counting only the costs in part of their executions. The latency of accessing and modifying queue nodes in the two algorithms is ignored here. For dequeues, this latency can be high: since nodes to be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

The performance of our PIM-managed FIFO queue seems poor at first sight: although a PIM core can update the queue efficiently, it takes a lot of time for the PIM core to send results back to CPUs one by one. To improve its performance, the PIM core can *pipeline* the executions of requests, as illustrated in Figure 6(a). Suppose p CPUs send p dequeue requests concurrently to the PIM core, which takes time $\mathcal{L}_{message}$. The PIM core fist retrieves a request from its message buffer (step 1 in the figure), dequeues a node (step 2) for the request, and sends the node back to the CPU (step 3). After the PIM core sends off the message containing the node, it immediately retrieves the next request, without waiting for the message to arrive at its receiver. This way, the PIM core can pipeline requests by overlapping the latency of message transfer (step 3) and the latency of memory accesses and local computations (steps 1 and 2) in multiple requests (see Figure 6(b)). During the execution of a dequeue, the PIM core only makes one memory access to read the node to be dequeued, and two L1 cache accesses to read and modify the tail node of the dequeue segment. It is easy to prove that the execution time of p requests, including the time CPUs send their requests to the PIM core, is only $\mathcal{L}_{message} + p(\mathcal{L}_{pim} + \epsilon) + \mathcal{L}_{message}$, where ϵ is the total latency of the PIM core making L1 cache accesses and sending off one message, which is negligible in our performance model. If each CPU makes another request immediately after it receives the result of its previous request, we can prove that the throughput of the PIM-managed FIFO queue is

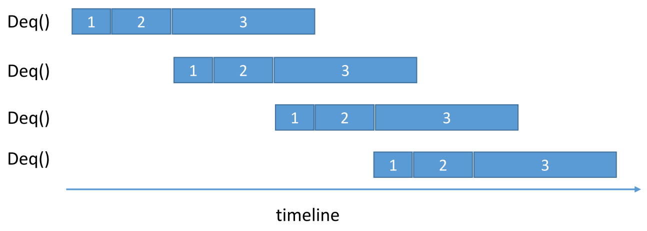
$$\frac{1-2\mathcal{L}_{message}}{\mathcal{L}_{pim}+\epsilon} \approx \frac{1-2\mathcal{L}_{message}}{\mathcal{L}_{pim}} \approx \frac{1}{\mathcal{L}_{pim}},$$

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

When the PIM-managed FIFO queue is short, it may con-



(a)



(b)

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

tain only one segment which deals with both enqueue and dequeue requests. In this case, its throughput is only half of the throughput shown above, but it is still at least as good as the throughput of the other two algorithms.

We have 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 combiner does not make real enqueue or dequeue operations for requests. Instead, after retrieving a request from the publication list, the combiner simply waits time t_1 and then writes a fake result back, where t_1 is the average execution time of an enqueue 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 7. In each experiment, each thread made 10^7 requests and we counted the average number of last-level cache accesses a request incurred.

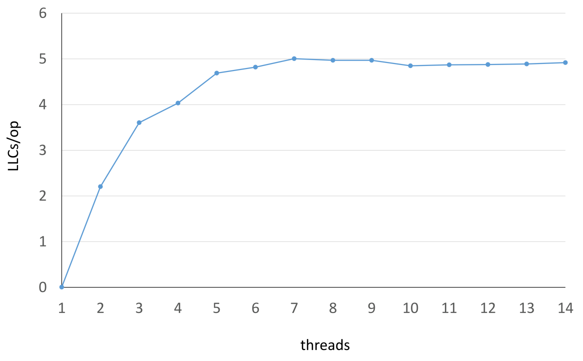
In the the Flat-combining queue without hyperthreading (Figure 7(a)), each request incurred on average roughly 4.8-4.9 last-level cache accesses. Note that a thread submitting a request usually makes at most 3 last-level cache accesses—one when it writes the request into the publication list, one when it tries to acquires the lock, one when it reads the result returned by the combiner. (A thread may retry to acquire the lock, incurring more last-level cache accesses, if its request was missed by the previous combiner. However, this situation happened rarely in our experiments and therefore its impact can be ignored.) Thus, we can conclude that the combiner incurred at least 1.8-1.9 last-level cache accesses on average per request, and this result is very close to our expectation (i.e., $2\mathcal{L}_{llc}$ per request). With hyperthreading(Figure 7(b)), each request incurred 4.0-4.5 last-level cache accesses when we had more than 10 threads, and therefore the 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 7(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 7(d)), each request made roughly 0.55-0.8 last-level cache access by F&A. Even if we think F&A on L1 and L2 caches by F&A is cheap and negligible, the performance of the algorithm improves only by 82%-25%, which is still worse than the PIM-managed queue's expected performance.

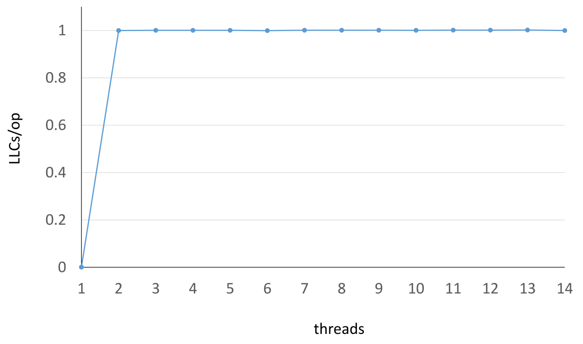
5. RELATED WORK

The PIM model is undergoing a renaissance. Studied for decades (e.g., [25, 20, 11, 24, 23, 19, 12]), this model has recently re-emerged due to advances in 3D-stacked techology that can stack memory dies on top of a logic layer [18, 21, 7]. For example, Micron and others have recently released a PIM prototype called the *Hybrid Memory Cube* [9], and the model has again become the focus of architectural research. Different PIM-based architectures have been proposed, either for general purposes or for specific applications [2, 1, 26, 17, 6, 3, 5, 4, 8, 27, 28].

The PIM model has many advantages, including low energy consumption and high bandwidth (e.g., [1, 26, 27, 4]). Here, we focus on one more: low memory access latency [21, 17, 6]. To our knowledge, however, we are the first to utilize PIM memory for designing efficient concurrent data structures. Although some researchers have studied how PIM memory can help speed up concurrent operations to data structures, such as parallel graph processing [1] and parallel pointer chasing on linked data structures [17], the



(a)



(b)

Figure 7: (a) Flat-combining queue. (b) F&A-based queue without hyperthreading.

applications they consider require very simple, if any, synchronization between operations. In contract, operations to concurrent data structures can interleave in arbitrary orders, and therefore have to correctly synchronize with one another in all possible situations. This makes designing concurrent data structures with correctness guarantees like linearizability [16] very challenging.

Moreover, no one has ever compared the performance of data structures in the PIM model with that of state-of-the-art concurrent data structures in the classic shared memory model. We analyze and evaluate concurrent linked-lists and skip-lists, as representatives of pointer-chasing data structures, and concurrent FIFO queues, as representatives of contended data structures. For linked-lists, we compare our PIM-managed implementation with well-known approaches such as fine-grained locking [13] and flat combining [14].

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].

6. 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.