Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation

Jiwon Choe
Brown University
jiwon_choe@brown.edu

Amy Huang
Brown University
amy_huang1@brown.edu

Tali Moreshet
Boston University
talim@bu.edu

Maurice Herlihy
Brown University
mph@cs.brown.edu

R. Iris Bahar
Brown University
iris_bahar@brown.edu

ABSTRACT

Recent advances in memory architectures have provoked renewed interest in *near-data-processing (NDP)* as way to alleviate the "memory wall" problem. An NDP architecture places logic circuits, such as simple processors, in close proximity to memory.

Effective use of NDP architectures requires rethinking data structures and their algorithms. Here, we provide an empirical evaluation of several NDP-aware algorithms for general-purpose concurrent data structures such as linked-lists, skiplists, and FIFO queues. The empirical analysis reveals that the potential benefits of NDP-based concurrent data structures are less than what had been expected in earlier studies. In turn, we introduce lightweight NDP hardware modifications, inspired by initial observations on data access patterns and underlying DRAM activity. Even the minimal changes to hardware significantly improve the performance and energy consumption of NDP-based concurrent data structures, and in many cases, the resulting data structures outperform state-of-the-art concurrent data structures.

ACM Reference Format:
Jiwon Choe, Amy Huang, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2019. Concurrent Data Structures with Near-Data-Processing: an Architecture-Aware Implementation. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3323165.3323191

1 INTRODUCTION

The increasing discrepancy between processor speeds and memory access speeds (often referred to as the *memory wall* [33]) causes memory access to be the principal performance bottleneck in many of today's data-intensive applications. Until recently, most architectures have relied on multi-level caches to reduce data access latency. However, caches have become less and less effective over time. As data-intensive applications increasingly use data sets much larger than cache sizes and exhibit irregular and unpredictable memory access patterns, it is hard to simply rely on caches to improve application performance. Moreover, frequent data movement between host processors and memory causes high energy consumption, a growing concern with data-intensive applications.

Recent architectural advances – in particular, 3D die-stacking technology – have renewed the interest in *near-data-processing* (*NDP*) (also referred to as *near-memory-computing*) as a way around the memory wall. This technology allows logic circuits, such as simple processors, to be placed in close proximity to memory, using high-bandwidth *Through-Silicon Via* (TSV) interconnects for communication between the near-memory processor and memory.

Today, commercially available devices that exploit the 3D die-stacking technology, such as Hybrid Memory Cube (HMC) [20] or High-Bandwidth Memory (HBM) [4], implement only simple memory controller logic near memory. However, we expect that soon simple processors will be placed near memory, enabling near-data-processing. In this paper, we investigate how near-memory accelerators can be combined with novel data structures and algorithms to exploit the low-latency, high-bandwidth memory access of future NDP architectures, while also preserving the high concurrency of conventional systems.

Prior works [1, 2, 7, 11, 15, 24, 28, 29, 31, 37] have investigated using near-data-processing to improve the performance and energy efficiency of specific applications. Here, however, we focus on software libraries and architectural support for *general-purpose concurrent data structures* with near-data-processing architectures. These data structures are used in many applications, and adapting them to NDP architectures is a key step toward making these architectures useful. In conventional architectures, "pointer-chasing" data structures with poor cache locality and high-contention concurrent data structures are often bottlenecks, while near-data-processing architectures have the promise to alleviate or even eliminate these problems.

Liu *et al.* [26] observed that naïve implementations of data structures on near-data-processing architecture will serialize data struture operations and will be outperformed by highly-concurrent state-of-the-art data structures on conventional architectures. As an algorithmic solution, they proposed using *flat-combining* techniques [17] to add concurrency to NDP-based data structures. In the absence of a specific architecural model, they provided a "back-of-the-envelope" analysis based on simple hardware latency assumptions and suggested that this approach had promise.

In this paper, we implement and test those data structure algorithms on a full-system NDP architecture framework with realistic

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323191

hardware constraints. Through this more realistic and detailed analysis, we find that Liu *et al.*'s work is incomplete. Although the results show that flat-combining does indeed enhance concurrency in NDP-based data structures, the resulting performance falls short compared to what was expected in the theoretical analysis. We identify that the theoretical analysis had two major pitfalls: (1) it ignored the cache impacts in host-based concurrent data structure performance, and (2) it had overly optimistic assumptions on near-data-processing memory access latencies. These pitfalls led to the overestimated relative performance of NDP-based concurrent data structures.

To address these shortcomings, we show that lightweight changes to hardware can reduce the performance gap while using the same algorithm. The hardware changes were inspired by observations on data structures' access patterns and underlying DRAM activity. The changes show significant improvements in not only performance but also energy consumption. While the improvements are still somewhat less than what had been expected in prior theoretical analysis, in many cases the resulting NDP-based data structures exhibit better performance and energy consumption than state-of-the-art concurrent data structures.

This paper makes the following contributions:

- We define a generic near-data-processing (NDP) architecture that is well-suited for concurrent data structures.
- We implement actual software kernels of the NDP-based concurrent data structures on a cycle-accurate full-system NDP architecture framework, yielding a more realistic and detailed analysis in terms of performance, energy, and power.
- Using our architecture framework, we identify the shortcomings of prior theoretical analyses that led to overestimated relative performance of NDP-based concurrent data structures.
- The findings from this evaluation suggest lightweight adjustments to hardware design. We show that minimal hardware changes can significantly improve the performance and energy consumption of NDP-based conurrent data structures.

2 ARCHITECTURE BACKGROUND

2.1 DRAM Basics

DRAM array: In DRAM, data is stored in two-dimensional arrays of storage cells, referred to as *DRAM arrays*. Each storage cell holds a very small electric charge representing one bit of data. In order to read a value, the entire *row* specified by the row address is sent to *sense amplifiers*, where the small charges get amplified for reading. The specified *column* is selected and read from the sense amplifiers.
DRAM bank: A *DRAM bank* refers to a set of DRAM arrays that act in unison to allow accessing multiple bits of data at once. The number of bits accessed with a single command is equal to the number of DRAM arrays in the bank and is referred to as the *column width*. Each bank operates independently of another.
DRAM access process: All memory read and write requests pass through the memory controller, where the memory request is translated into one or more DRAM access requests and put on the read queue or write queue. The number of DRAM access requests depends on the location and size of requested data. The DRAM accesses are processed one at a time. Usually, the read access requests are serviced before the write access requests.

A single DRAM access proceeds by taking the following steps:

(1) Bitlines are *precharged* so that sense amplifiers can detect the small bit data charges coming from DRAM rows. (t_{RP})
(2) The row (also called *page*) specified by the row address is *activated* and sent to sense amplifiers via bitlines. (t_{RCD})
(3) The column specified by the column address is read from the sense amplifiers and is put on the data bus. (t_{CL})
(4) Data is transferred to the memory controller. (t_{BURST})

t_{RP}, t_{RCD}, t_{CL}, and t_{BURST} refer to the time it takes for each of the steps to complete.

In modern DRAM devices, the column access in step (3) may fetch more than one column of data at once. The data fetched and transferred on a single DRAM access is referred to as a *burst*, and t_{BURST} depends on the burst size.
Row-buffer-management policy: Row-buffer-management policies define how the memory controller manages steps (1) and (2) in the data access process. As explained by Jacob *et al.* [22], *open-page policy* and *close-page policy* are the two primary row-buffer-management policies.

In the open-page policy, an activated row is kept open until another memory access requires activating a different row in the same bank or until a refresh operation is issued on the bank. The open-page policy is advantageous if the data access pattern often shows consecutive accesses to the same row, for it removes duplicate row activations and thereby reduces delay (t_{RCD}) and energy associated with row activation.

In the close-page policy, a row is activated for a single access only, and the DRAM bank is precharged immediately after an access. This not only hides precharge latency (t_{RP}) for the next row activation but also reduces background energy, for DRAM banks draw more current when a row is activated than when the rows are all precharged. This is effective when data is accessed randomly. Modern DRAM devices used in high-performance multicore systems usually adopt the close-page row-buffer-management policy because the highly concurrent memory accesses, coming from different contexts, are likely to require random row activations.

Note that in either row-buffer-management policy, all requests in the request queue that access the currently activated row are serviced before closing the row.

2.2 Generic Near-Data-Processing

Three-dimensional (3D) die-stacked memory consists of multiple DRAM dies stacked on top of a single logic die. The memory is divided into vertical sections called *vaults*, and each vault contains multiple DRAM banks, connected to the logic die using Through-Silicon Via (TSV) interconnect. The near-data compute units, which we refer to as *NDP cores*, are placed in the logic die. We refer to the memory vaults connected to NDP cores as *NDP vaults*. The memory controller that manages memory requests to the NDP vault, which we refer to as the *NDP controller*, is also contained in the logic die. Each NDP core is tightly coupled with an NDP vault and has exclusive access to the vault's data (see Figure 1).

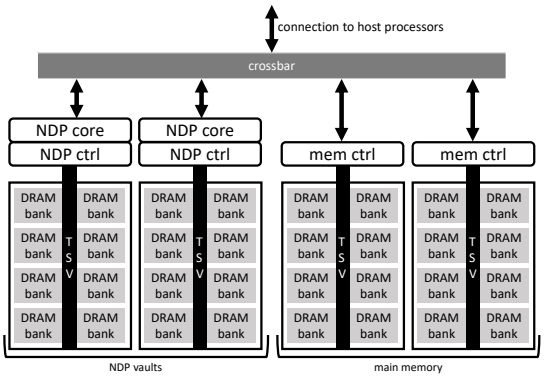


Figure 1: Generic near-data-processing architecture. We assume that each vault's data is private to the coupled NDP core.

NDP cores: While some have proposed using specialized accelerators as NDP cores [2, 11, 19, 23, 24], we simply assume that NDP cores are lightweight in-order, single-cycle processors. Sophisticated functionalities, such as pipelining or caching, are inappropriate in NDP cores for two reasons: (1) it is difficult to implement complex logic on the near-memory logic die, and (2) data structures and applications that are well-suited for NDP are often those that are bottlenecked by memory and benefit little from sophisticated processors.

NDP vaults: While some have proposed making NDP vaults shared among all NDP cores [36] or even with all host processors [3, 8, 19]. We treat NDP vaults as private to the coupled NDP core, removing the need for complex hardware and software to manage data races, coherence, and virtual address translation.

3 NDP-BASED CONCURRENT DATA STRUCTURES

We implemented the NDP-based linked-list, skiplist, and FIFO queue data structures of Liu *et al.* [26] on a cycle-accurate, full-system architecture framework, with the goal of deriving more realistic performance and energy evaluations.

Concurrency for these data structures is provided by the *flat-combining (FC)* [17] synchronization scheme. In a non-NDP context, a flat-combining data structure consists of a *publication list* and a *combiner lock*. Each thread posts its operation request, such as `contains(X)`, `add(X)`, `remove(X)`, in a dedicated slot in the data structure's publication list. After posting the request, threads compete for the combiner lock. The thread that acquires the combiner lock becomes the *combiner thread*, and it is the only thread that accesses the actual data structure. The combiner thread goes through the publication list, executes the posted operations, writes return values to corresponding slots in the publication list, and releases the combiner lock. Other threads that fail to acquire the combiner lock simply spin on their own slots in the publication list until the return value is set, which indicates that the operation has been applied to the data structure.

Flat-combining is well-suited to NDP-based concurrent data structures. The data structure itself resides in one or more NDP vaults, and each NDP core acts as a dedicated combiner thread for the portion of the data structure contained in its coupled NDP vault. Host processor threads simply send data structure operation requests to the corresponding NDP core.

We now describe these data structures in more detail.

3.1 Linked-List

A linked-list is a *pointer-chasing* data structure in which data is represented as a sequence of nodes: each node holds a data item, as well as a pointer to the next node. Operations on the data structure require chasing through pointers (*i.e.*, iteratively accessing the memory location specified by the next node pointer). Linked-lists, like most pointer-chasing data structures, typically have poor cache locality because each step in a traversal jumps to an unpredictable memory location.

In our examples, linked-list nodes are sorted in ascending order of integer keys. For the NDP-based implementation, we assume that the entire linked-list is contained in a single NDP vault.[1]

The NDP core handles concurrent operations by a combination of flat-combining and sorting operations in the order of their keys. Since the linked-list is a sorted list, sorting the operation requests according to requested keys allows the NDP core to execute all combined operations over a single traversal through the linked-list. Figure 2a shows an example of the NDP-based flat-combining and operation sorting linked-list.

3.2 Skiplist

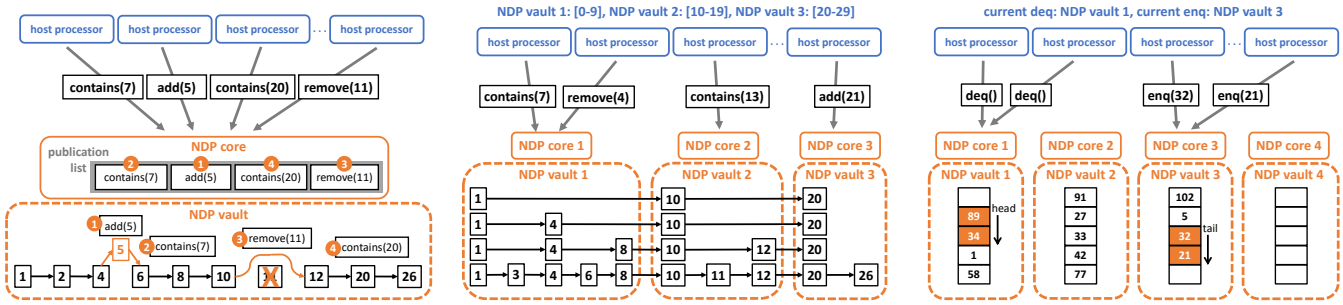
The skiplist [30] is another type of pointer-chasing data structure. Skiplist nodes hold multiple levels of pointers, and the pointer at each level points to the following node at that same level. Each node is assigned a random maximum level, taken from a particular distribution, to provide balanced tree-like characteristics. The nodes in the skiplist are also ordered in ascending order of integer keys.

The NDP-based skiplist is optimized by partitioning the skiplist across multiple NDP vaults based on pre-defined disjoint ranges of keys, as shown in Figure 2b. We assume that the host processors are provided with the range of keys belonging to each NDP vault. Host processors send operation requests to the appropriate NDP core, based on the requested operation key. Each NDP core acts as the combiner for its designated partition, which takes care of synchronization. Partitioning also enables multiple NDP cores to execute operations in parallel.

3.3 FIFO Queue

The FIFO queue is a representative example of a *contended* data structure. Unlike pointer-chasing data structures, FIFO queue operations access only the queue head or tail, and do not require extensive memory accesses. Moreover, consecutive items are often stored in contiguous memory locations, so operations benefit from good cache locality, especially in the single-threaded case. However, in a concurrent FIFO queue, multiple threads access the same head and tail locations. Locks or atomic operations are used on the head and tail to ensure thread safety, but operations suffer greatly from the contention at these locations.

[1] If the linked-list size exceeds that of a single NDP vault, it can be partitioned into multiple vaults based on ranges of keys. The basic functionality of each NDP core-vault pair remains the same; the hosts simply send operation requests to the NDP core that corresponds to the requested key.



(a) Flat-combining and operation sorting linked-list.

(b) Partitioned and flat-combining skiplist.

(c) Partitioned and flat-combining FIFO queue.

Figure 2: NDP-based concurrent data structure design.

The NDP-based FIFO queue is based on flat-combining and partitioning, as shown in Figure 2c. The NDP core effectively removes contention, for it is the only thread that operates on the data structure. Furthermore, partitioning the queue across multiple NDP vaults allows for separate enq and deq partitions, which adds parallelism among NDP cores.

The host processors maintain a shared record of the current enq and deq partition and send operation requests to NDP cores based on this record. When an enq partition is filled up or when a deq partition is emptied out, the corresponding NDP core sends a notification with the last operation that it completed. By this notification, the host processor that issued the last completed operation is designated as the thread to safely update the host-side record. The host processors, while spinning on their own slots in the publication list for the operation's return value, also check the shared enq/deq partition record to see if the operating partition has changed. If the operating partition changes before the operation completes, the host processor needs to clear out the existing operation request and reissue the operation to the new partition. Note that this has little impact on performance, for the active partition changes infrequently.

4 HARDWARE MODIFICATIONS

Our proposed NDP architecture is based on the generic NDP architecture described in Section 2.2. Lightweight modifications — inspired by data structure memory access patterns and underlying hardware behavior — are made to the NDP controller.

We observe that data structure operations exhibit temporal and spatial locality at DRAM row granularity. For list traversal in linked-lists or skiplists, the NDP core reads two words of information from each node, back-to-back: the node's key value and the pointer to the next node. A single node is stored in contiguous memory (*i.e.*, in the same DRAM row). This implies that data in a single row is accessed twice consecutively. The FIFO queue has an even higher rate of row hits. Queue items are stored successively in memory, according to queued order, so consecutive deq operations access all items in a DRAM row before moving onto the next row.

However, because the NDP core is a simple in-order processor without cache, it requests for only one word of data from memory at a time. In a typical memory controller, every memory access request is translated into a separate DRAM access operation, regardless

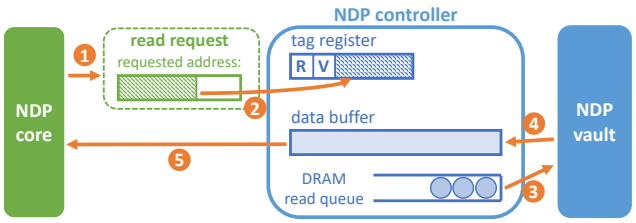


Figure 3: Proposed NDP controller design. The tag register and data buffer are added to the existing NDP controller. Steps 3 and 4 are skipped if the tag portion of the requested read address matches the tag register content.

of the row locality. If close-page row-buffer-management policy is used, this will require repetitive row activations; if open-page policy is used, this will reduce row activation latencies but increase background energy by leaving DRAM rows open. Moreover, data is tranferred from a DRAM bank to the memory controller in units of *bursts* (consecutive columns of data), which is often larger than one word. If the NDP core requests for only one word of data, the unused extra data is discarded, when oftentimes the next request would need data from the very same burst.

To address these issues, a small data buffer is added to the NDP controller. This buffer can be thought of as a single-block cache placed in the memory controller. The buffer size depends on the data structure: for the linked-list and skiplist, the buffer size is equal to the node size, and for the FIFO queue, it is equal to the DRAM row size. With the buffer, using the close-page policy is sufficient because the buffer serves the same purpose as an open row, except with additional benefits in access latency and power.

Figure 3 shows our modified design. Only two new hardware components needed for the buffer design: the buffer itself, which holds the most recently accessed block of data, and a *tag register*, which holds the *tag* portion of the buffered block's memory address. Upon receiving a data read request (step 1), the NDP controller first checks if the tag of the requested data address matches the tag register (step 2). If so, the request is responded to immediately (step 5). This eliminates the DRAM data access process (described in Section 2.1) entirely, which thereby reduces data access delays and energy consumption.

If requested data is not in the data buffer, the NDP controller creates DRAM access operations to fill the buffer (steps 3 and 4). If multiple DRAM accesses are necessary, the DRAM access operation that retrieves data for the original request is issued first. This ensures that the data buffer mechanism does not increase the delay in responding to the original data request, while it also allows effective latency hiding for the extra DRAM accesses needed for filling the buffer.

Bits in the tag register are used as *valid* and *ready* flags. Once the first DRAM access operation returns with its portion of data, the tag register is updated with the new address, and the buffer is marked *valid*. However, the *ready* flag is not set until the buffer has been filled with the corresponding data block. When a memory access request with an address tag that matches the tag register arrives while the buffer is being updated, the request is blocked until the buffer is set to *ready*. If a write request to the NDP controller modifies data in the buffer, the buffer is simply marked as invalid.

5 EVALUATION SETUP

In this section, we describe how we measure the performance and energy of the proposed data structure implementations, along with the details of our architecture framework.

5.1 Benchmarks

We use simple benchmarks to evaluate the proposed NDP-based concurrent data structures. The data structure is first initialized with random items according to the given initial size. We measure the performance, energy, and power of host threads concurrently issuing random operations on the data structure. The total number of random operations is fixed and is divided equally among the specified number of host threads. Performance is measured in terms of *operation throughput*, which refers to the total number of data structure operations completed in a given period of time.

We implement each of the NDP-based data structures from Section 3 and run them on the NDP architecture with and without the NDP controller modifications described in Section 4, in order to evaluate the impact of lightweight hardware modifications. We compare the results against the host-based flat-combining data structure, as a baseline for how the NDP-based algorithm would perform in a non-NDP context, and also against the host-based state-of-the-art concurrent data structure.

5.2 System Framework

We evaluate the benchmarks on a cycle-accurate, full-system architecture framework with real hardware constraints. In particular, we use SMCSim [5], a gem5 [6]-based full-system simulator, which includes the software stack and architecture support for near-data-processing. In order to build a framework that conforms to the NDP architecture design described in Section 2.2, we made the following modifications to SMCSim.

The default for the SMCSim simulator originally had 16 memory vaults, all used as host-accessible main memory and shared with a single NDP core. We split the memory vaults and use eight vaults as host-accessible main memory (referred to as main memory vaults), and the other eight vaults as NDP vaults. Each NDP core is connected directly to a memory controller and to the host, and each NDP vault is visible only to the directly connected NDP core. The configuration shown in Figure 1 is modeled after our resulting architecture.

Host Configuration	
Host cores	8 in-order processors (ARMv7 Cortex-A15), 1 thread/core
L1 cache	32kB icache, 64kB dcache, private, 2-way set-associative 0.8 ns dcache access latency, 256B/block
L2 cache	2MB, shared, 8-way set associative 1.8ns access latency, 256B/block
NDP configuration	
NDP cores	1 in-order processor/vault (ARMv7 Cortex-A15)
scratchpad memory	40kB/NDP core, 8kB reserved for host memory-map stores instructions and program stack
Memory Configuration	
vaults	16 vaults, 128MB/vault (total 2GB), 8 DRAM banks/vault for host-only baseline: 16 main memory vaults for NDP: 8 main memory vaults, 8 NDP vaults
DRAM parameters	row buffer size: 256B, burst size: 32B t_{RP}: 13.75ns, t_{RCD}: 13.75ns, t_{CL}: 13.75ns, t_{BURST}: 3.2ns

Table 1: Evaluation framework configuration.

We also implement the data buffer described in Section 4 on the NDP controllers. For evaluations without the NDP controller change, an unmodified generic memory controller is used as the NDP controller.

The NDP core used in the framework is a simple in-order processor modeled after the ARM Cortex-A15 processor and does not have any cache. A small program binary, referred to as the *NDP kernel*, is offloaded to the NDP core for execution. Each NDP core is equipped with a 40kB scratchpad memory, which stores the instructions and the program stack for the NDP kernel. A portion of the scratchpad memory (8kB) is reserved for communication: this region is memory-mapped into host memory and allows host processors to communicate with the NDP cores through this interface. We use the memory-mapped interface as slots for the data structure publication list.

Table 1 summarizes the configuration details of our evaluation framework. Although we use simple in-order processors for the host, this will have little impact on host-based data structure performance because data structure operations are primarily performance-bounded by memory accesses or successful execution of atomic operations.

5.3 Power Model

We take power measurements for core, cache, interconnect and memory controller from McPAT [25]. Configuration parameters were primarily drawn from the simulation framework settings, as well as Endo *et al.* [12] and the official ARMv7 Cortex-A15 manual [10] for both the NDP and host cores. Peripherals (*i.e.*, flash controllers, PCIE, network interface units) were not included. For DRAM energy and power, we use statistics available through SMCSim per simulation. They are calculated by DRAMPower [9] as an integrated library of gem5, and separate outputs are produced for precharge, read, write, activation, refresh, and background energy.

6 LINKED-LIST ANALYSIS

We implement the NDP-based linked-list described in Section 3.1 on our NDP architecture with the NDP controller modification proposed in Section 4 (*NDP ctrl buffer*) and on a generic, unmodified

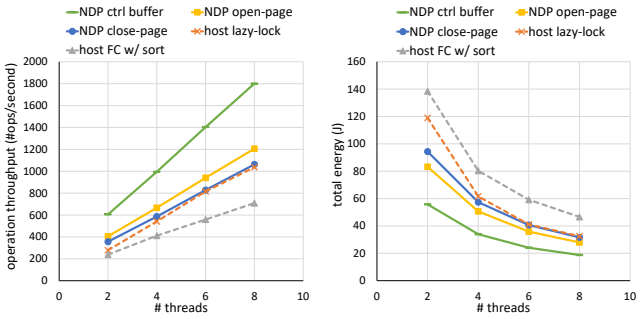
NDP architecture. For the latter, we evaluate using both the open-page and close-page row-buffer-management policies (*NDP open-page* and *NDP close-page*, respectively). *NDP ctrl buffer* uses the close-page policy, as described in Section 4.

For host-based references, we implement the flat-combining and operation sorting linked-list (*host FC w/ sort*) [17], which is algorithmically the host-based equivalent of the NDP linked-list, and the lazy-lock linked-list (*host lazy-lock*) [16], a state-of-the-art concurrent linked-list algorithm. In *host lazy-lock*, locking is required only on nodes that are affected by add or remove operations, so in most cases list traversal can run completely in parallel, unaffected by locking.

For all experiments, we set the initial linked-list to be approximately 5MB in total size (larger than the L2 cache). We set 90% of the random operations to be read-only, and the remaining 10% of the random operations are divided equally among add and remove operations, in order to maintain a constant list size.

6.1 Performance

Figure 4a shows the operation throughput for each linked-list implementation across varying numbers of concurrent threads. In general, the NDP-based implementations perform better than the host-based implementations, although *NDP close-page* barely does better than *host lazy-lock*, with only 2.5% improvement in throughput at eight concurrent threads. However, *NDP open-page* shows



(a) operation throughput (b) total energy

Figure 4: Linked-list evaluations. Solid lines show results from NDP-based implementations, and dashed lines show results from host-based implementations.

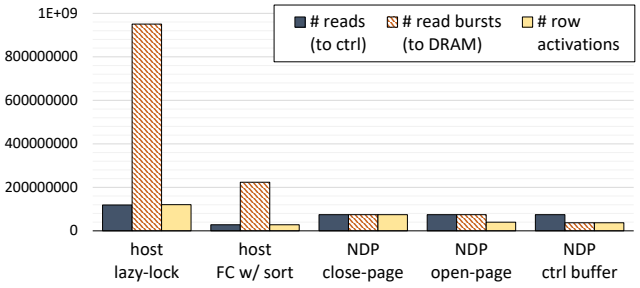


Figure 5: Number of memory accesses, DRAM bursts, and DRAM row activations in each linked-list implementation (based on 8-thread execution).

significant improvement and *NDP ctrl buffer* even more so. At eight concurrent threads, *NDP open-page* and *NDP ctrl buffer* each have 16.4% and 73.7% higher operation throughput than *host lazy-lock*. Note also that the operation throughput of *NDP ctrl buffer* scales better than that of all other implementations with an increasing number of threads.

NDP open-page has 13.5% higher throughput than *NDP close-page*, and *NDP ctrl buffer* has 49.3% higher throughput than *NDP open-page*. The performance difference among NDP-based implementations results from delays associated with DRAM access. For NDP-based implementations, accessing a node in list traversal requires the following DRAM delays: (1) t_{RCD} to activate the row containing the node's key, (2) t_{CL} to read the column that contains the node's key, and (3) t_{BURST} to move the burst containing the node's key to the NDP controller. After accessing the node's key value, the NDP core also needs to read the node's next node pointer. This next node pointer is stored in the same row – better yet, in the same burst – as the node's key. However, with *NDP close-page*, the row activated in step (1) is already precharged, so reading the node's next node pointer requires repeating steps (1), (2), and (3). On the other hand, *NDP open-page* leaves the row activated, so the DRAM access for the next node pointer requires only steps (2) and (3). *NDP ctrl buffer* caches the burst from step (3), so no additional DRAM accesses are needed. Therefore, DRAM delays associated with a single node traversal sums up to:

$$\text{(NDP close-page delay)} = 2 \times (t_{RCD} + t_{CL} + t_{BURST})$$
$$\text{(NDP open-page delay)} = t_{RCD} + 2 \times t_{CL} + 2 \times t_{BURST}$$
$$\text{(NDP ctrl buffer delay)} = t_{RCD} + t_{CL} + t_{BURST}$$

To get a better idea of how performance, power, and energy are affected by the various NDP and host schemes, we also looked into the memory accesses to the DRAMs. Figure 5 shows the number of memory read requests that arrive at memory controllers, the number of DRAM read accesses (read bursts) that the read requests are translated to, and the number of DRAM row activations that happen in order to serve these bursts. In the NDP-based implementations, the number of DRAM bursts are at most equal to the number of memory requests because each request asks for only one word of data. Although all NDP-based implementations make the same number of read accesses to the NDP controller, the number of DRAM row activations is reduced to half with *NDP open-page*, and the number of DRAM read bursts is reduced to half with *NDP ctrl buffer*.

In host-based implementations, the memory read request to the memory controller originates from a cache miss and therefore requests for a cache block size chunk of data. A cache block size is normally a multiple of the DRAM burst size, so the number of DRAM read bursts is much larger than the number of read requests. In our case, the cache block size is equal to a DRAM row size and is eight times the burst size. Because of the sheer number of DRAM read bursts, host-based linked-lists perform worse than the NDP-based linked-lists. However, although *host lazy-lock* has much more read bursts than *host FC w/ sort*, *host lazy-lock* shows higher throughput because the memory requests and DRAM bursts are issued and processed in parallel, whereas the memory requests in *host FC w/ sort* are issued sequentially by the combiner thread.

# threads	average total power (W)				
	host lazy-lock	host FC w/ sort	NDP close-page	NDP open-page	NDP ctrl buffer
2	7.8965	7.8710	8.0127	8.0300	8.0460
4	7.9424	7.8704	8.0125	8.0299	8.0455
6	7.9837	7.8710	8.0149	8.0319	8.0473
8	8.0177	7.8718	8.0156	8.0329	8.0485

Table 2: Average power dissipation for linked-list implementations across varying numbers of concurrent threads.

6.2 Energy & Power

Table 2 shows the average power dissipation for various linked list implementations, across increasing number of threads. It shows that NDP-based implementations in general dissipate more power than host-based implementations. First of all, the addition of NDP cores to the system increases power dissipation. Secondly, the NDP-based implementations require more power in the memory controllers than their host-based equivalent, *host FC w/ sort*, because the memory controllers process more read requests (2.66x), as shown in Figure 5.

Because extensive memory access is the limiting factor in linked-list operations, the time intervals between read request arrivals at the memory controller, DRAM row activations, and DRAM reads also impacts the average power dissipation. In other words, if the memory controllers and DRAM banks have less idle time and are more frequently in dynamic operation, the average power increases. For this very reason, *NDP ctrl buffer* dissipates more average power than *NDP open-page*, and *NDP open-page* more than *NDP close-page*. Another reason *NDP open-page* dissipates more power than *NDP close-page* is that leakage power is higher when a DRAM row is activated.

However, as shown in Figure 4b, performance gains allow for lower total energy consumption with *NDP open-page* and *NDP ctrl buffer*, despite the higher average power consumption. At eight concurrent threads, *NDP open-page* and *NDP ctrl buffer* each show 13.9% and 42.2% decrease in energy consumption compared to *host lazy-lock*.

In addition, we see from Table 2 that the increase in concurrent memory accesses also impacts power dissipation. Whereas the average power increases minimally with added threads in NDP-based implementations and *host FC w/ sort*, the average power for *host lazy-lock* increases at a much faster rate, and we expect that with more threads, the power consumption of *host lazy-lock* would exceed that of even *NDP ctrl buffer*.

7 SKIPLIST ANALYSIS

We implemented the NDP-based skiplist as described in Section 3.2 with 1, 2, 4, and 8 NDP vaults (*NDP 1part, 2part, 4part, 8part*). We also implemented host-based partitioned and flat-combining skiplists (*host FC 1part, 2part, 4part, 8part*) [17] as references on how the NDP-based algorithm would perform in a host-only setting. As described in Section 3.2, partitioning achieves higher concurrency by allowing multiple NDP cores (or in the case of *host FC* skiplists, multiple combiner threads) to safely operate on different parts of the skiplist in parallel.

In particular, the NDP-based skiplists are evaluated both with and without the NDP controller modifications described in Section 4.

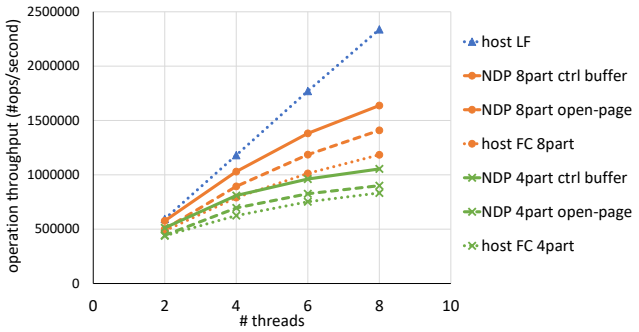


Figure 6: Baseline skiplist operation throughput. Solid lines and dashed lines show results from NDP-based implementations with and without NDP controller changes, respectively, and dotted lines show results from host-based implementations.

For evaluations without the NDP controller modifications, we use the open-page row-buffer-management policy for NDP vaults. We refer to the NDP-based implementations with and without NDP controller modifications as *NDP ctrl buffer* and *NDP open-page*.

We also implement the state-of-the-art host-based lock-free skiplist (*host LF*) [13] for comparison. *Host LF* allows complete concurrency among host threads by using only atomic compare-and-swap operations to apply changes to the skiplist.

7.1 Performance

7.1.1 Baseline. For the baseline skiplist evaluation, we use an initial skiplist that is approximately 12MB in size. We set 90% of the random operations to be read-only (contains) and divide the remaining 10% of the operations equally among add and remove operations, in order to maintain a consistent skiplist size.

Figure 6 shows the operation throughput of different skiplist implementations across varying number of concurrent threads. Results from 1 and 2 partition skiplists were omitted in order to highlight the most relevant and interesting results.

Just as it had been in the linked-list case, adding a node-size data buffer to the NDP controller improves NDP-based skiplist performance. Skiplist node traversal is subject to DRAM access delays similar to the linked-list node traversal: (1) t_{RCD} to activate the row containing the node, (2) t_{CL} to read the column containing the node's key, and (3) t_{BURST} to move the burst containing the key. After reading the key, the node's next node pointer is accessed, which requires no additional DRAM access delay with *NDP ctrl buffer*, whereas an additional ($t_{CL} + t_{BURST}$) delay is required with *NDP open-page*.

However, skiplist traversal sometimes requires backtracking to the previous node in order to move down a level. In this case, even with *NDP ctrl buffer*, the previous node needs to be accessed from DRAM again, for the buffer has already been overwritten. This results in an increased rate of buffer misses, compared to the linked-list.

Figure 7 shows the number of memory read requests, DRAM read bursts, DRAM row activations, and NDP controller buffer hits for various skiplist implementations. Because a skiplist node contains multiple levels of pointers, a skiplist node is much larger than a linked-list node and requires multiple DRAM bursts to fill

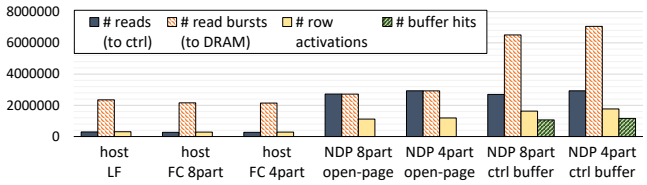


Figure 7: Number of memory read requests, DRAM read bursts, DRAM row activations, and NDP controller buffer hits in each skiplist implementation (based on 8-thread execution).

the buffer (4 bursts in our experiments). Nevertheless, the latency-hiding mechanism described in Section 4 allows *NDP ctrl buffer* to still yield higher performance than *NDP open-page*.

At eight concurrent threads, the partitioned *NDP ctrl buffer* skiplists on average have 17.2% higher operation throughput than the *NDP open-page* counterparts. Note that the improvement is greater with more threads, for execution on NDP cores is increasingly bound by memory accesses with more threads.

Figure 6 also reveals interesting results that were not captured in the theoretical analysis of [26]. At eight concurrent threads, with 4 partitions, *NDP open-page* and *NDP ctrl buffer* have 8.1% and 26.5% higher throughput respectively than *host FC 4part*; with 8 partitions, we observe 19% and 38.3% higher throughput respectively than *host FC 8part*. Although the NDP-based skiplists perform better than their host-based counterparts, the scale of improvement is not as great as what was expected in [26] (3x throughput). Moreover, contrary to the projection in [26], *host LF* completely outperforms the NDP-based implementations.

We find that cache effects account for the relatively high performance of host-based skiplists. The skiplist is inherently a balanced tree-like data structure, so an operation on the skiplist always begins at the few high-level nodes and traverses through only $O(\log_2 N)$ nodes (where N is the total number of nodes in the skiplist). Therefore, higher-level nodes are likely to remain in cache, and only a small number of accesses actually go out to main memory, even with a skiplist that is much larger than the last-level cache.

As shown in Figure 7, based on an 8-thread execution, the total number of read requests that go out to memory in NDP-based implementations is nearly 10x more than the number in host-based implementations. Even if NDP controller buffer hits are not counted towards memory read requests, *NDP 4part ctrl buffer* and *NDP 8part ctrl buffer* still have nearly 6x more memory read requests than host-based skiplists.

Yet, increasing the number of partitions has a greater impact on NDP-based skiplists than on host-based flat-combining skiplists: *host FC 8part, NDP 8part open-page,* and *NDP 8part ctrl buffer* each have 42.1%, 56.5%, and 55.3% higher throughput than its 4-partition counterpart. When the number of partitions doubles, the size of each skiplist partition is reduced in half. This decreases the height of each skiplist partition, which in turn slightly reduces the number of node accesses required per operation. Even this slight reduction significantly reduces the total number of memory accesses for NDP-based skiplists. On the other hand, in the *host FC* case, because the total number of nodes across all partitions remains the same, the

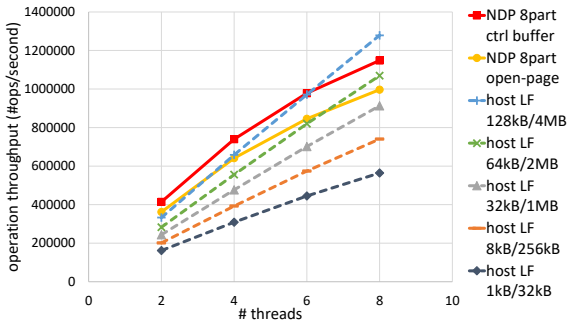


Figure 8: Operation throughput of host lock-free skiplist and NDP-based 8-partition skiplist with varying cache sizes. Numbers in the legend refer to cache sizes (L1 dcache/L2).

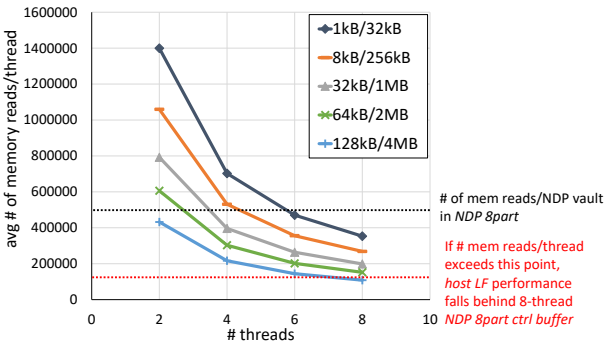


Figure 9: Average number of memory read requests per thread across varying cache sizes for host lock-free skiplists. Numbers in the legend refer to cache sizes (L1 dcache/L2).

nodes that consistently remain in cache are the same regardless of the partitioning. The same lower-level nodes that had to be read from memory still need to be read from memory, and therefore the number of data accesses that go out to memory remains relatively constant for host-based implementations.

Architecture constraints restrict the number of memory partitions in a memory space to be a power of 2.[2] We were limited by configurations in the architecture framework (16 memory vaults total) and could not empirically verify the performance with more than 8 partitions. Nonetheless, because increasing the number of partitions greatly improves NDP-based skiplist performance, we expect the NDP-based skiplists to outperform *host LF* with more partitions and enough concurrent host threads to saturate the NDP cores.

7.1.2 Cache & Skiplist Size. We do a sweep on cache sizes to find the skiplist size relative to the cache size that makes using the NDP 8-partition skiplist more advantageous than using the host lock-free skiplist.[3] The initial skiplist size is set to approximately 850MB, and the ratio of random operations is set to 90% contains, 5% add, and 5% remove.

[2]Bits in the address are used in order to determine the partition in which the data is placed.

[3] In reality, the skiplist size could be much larger than the size we experiment with, but we were restricted by memory address space limitations in the architecture framework. We therefore simulated the effects of having a larger data structure by reducing the cache size.

As expected, Figure 8 shows that cache size severely impacts the performance of host lock-free skiplists. On the other hand, the NDP-based skiplist performance is insensitive to cache size, so any cache configuration below 64kB/2MB shows clear advantages for the NDP schemes. Figure 9 shows the average number of memory read requests made by each thread in *host LF*. The black dotted line marks the average number of memory read requests per NDP vault in *NDP 8part* (both *open-page* and *ctrl buffer*), and the red dotted line marks the cutoff point where an 8-thread execution of *NDP 8part ctrl buffer* shows higher operation throughput than *host LF*. When the number of memory read requests per thread exceeds 26.5% of the memory read requests per vault, *host LF* performs worse than the 8-thread *NDP 8part ctrl buffer*.

7.2 Energy & Power

The skiplist energy and power analysis is based on the results from Section 7.1.2. We compare the energy and power of *NDP 8part open-page*, *NDP 8part ctrl buffer*, and *host LF* with 64kB L1 data cache and 2MB L2 cache (baseline cache size configurations).

Table 3 shows the average power dissipation for the different skiplist implementations across varying number of threads. The average power for NDP-based implementations are notably high compared to the power dissipated with *host LF*. Just as it had been in the linked-list case (Section 6.2), for *NDP 8part open-page*, the difference in average power compared to *host LF* comes from the NDP cores and the 3.23x more read requests processed at memory controllers (as shown in Figure 10b). Although *host LF* has higher average read power than *NDP 8part open-page*, it evens out with the higher leakage power with activated DRAM rows in *NDP 8part open-page*. *NDP 8part ctrl buffer* requires even higher average power than *NDP 8part open-page* because of the sheer number of extra DRAM reads (2.44x compared to *NDP 8part open-page*) that happen in order to fill the data buffer.

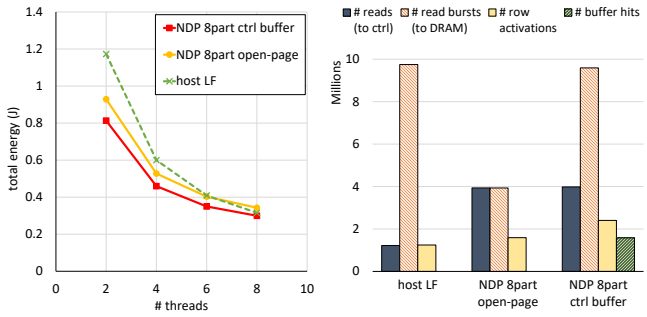
	average total power (W)		
# threads	host LF	NDP open-page	NDP ctrl buffer
2	7.897	8.005	8.023
4	7.938	8.060	8.094
6	7.984	8.107	8.153
8	8.030	8.140	8.193

Table 3: Average power dissipation for host lock-free and NDP-based 8-partition skiplists across varying numbers of concurrent threads.

Despite the significantly higher power dissipation, *NDP 8part ctrl buffer* still shows lower energy consumption than *host LF* due to the improvement in performance. Figure 10a shows the total energy consumption of the three different skiplist implementations across varying numbers of concurrent threads. At eight concurrent threads, *NDP 8part ctrl buffer* has 7.4% higher operation throughput and consumes 5% less energy compared to *host LF*.

8 FIFO QUEUE ANALYSIS

For brevity, we refer to the FIFO queue simply as the *queue*. Queues are inherently different from the pointer-chasing data structures we considered earlier. Items are usually stored in contiguous memory,



(a) total energy **(b) memory accesses**

Figure 10: Total energy consumption and memory access analysis of host lock-free and NDP-based 8-partition skiplists.

so they do not suffer from the same poor cache locality. Nevertheless, they do suffer from *contention* among multiple threads trying to access and modify the head and tail of the data structure.

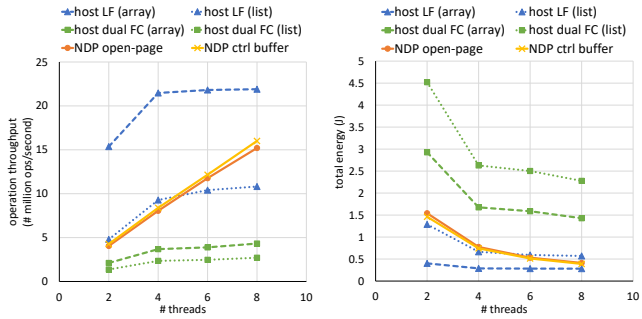
We implement the NDP-based queue as described in Section 3.3 on NDP architecture with and without the NDP controller data buffer (*NDP ctrl buffer* and *NDP open-page*, respectively). The NDP core effectively removes contention, for it is the only thread that operates on the data structure. We also implement the host-based dual flat-combining queue (*host dual FC*), which has separate combiner threads for the enq and deq operations, as the host-based counterpart. As the state-of-the-art concurrent queue, we implement the host-based lock-free queue (*host LF*) [27], in which concurrent threads use atomic compare-and-swap operations to safely execute enq and deq operations on the queue.

There are two ways to implement a queue. One is implementing it as a circular array buffer with indices specifying the head and tail (*array-based*), and another is implementing it as a linked-list with pointers to the head and tail (*list-based*). The array-based implementation uses fewer atomic operations for each enq or deq operation compared to the list-based implementation, but it requires setting aside a block of memory that is larger than the maximum possible size of the queue ahead of time. We considered both implementations for the host-based queues. However, the NDP-based queues were implemented array-based only, for the NDP vault is already memory dedicated to store the data structure.

For the evaluation, we set the initial queue size to be 2MB (4MB for the list-based implementations because each queue item needs to store a pointer to the next item, which is unnecessary in the array-based implementations). Random operations on the queue are divided as 50% enq and 50% deq.

8.1 Performance

Figure 11a shows the operation throughput of various queue implementations across varying number of concurrent threads. At eight concurrent threads, *NDP open-page* and *NDP ctrl buffer* each yield 3.5x and 3.7x higher throughput than the array-based *host dual FC*, just by removing contention. Compared to the list-based *host LF*, *NDP open-page* and *NDP ctrl buffer* show 40.5% and 48% higher throughput.



(a) operation throughput (b) total energy

Figure 11: FIFO queue evaluations. Solid lines show NDP-based implementations, dashed lines show host array-based implementations, and dotted lines show host list-based implementations.

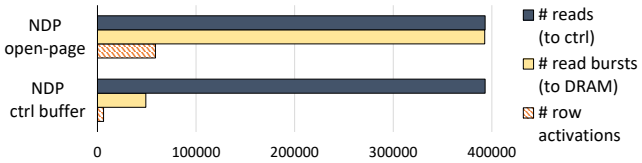


Figure 12: Number of memory read requests, DRAM read bursts, and DRAM row activations in NDP-based queues.

As mentioned earlier, the array-based *host LF* does fewer atomic operations for each queue operation than the list-based *host LF* and thus yields higher operation throughput than the NDP-based queues. However, while the operation throughput for host-based queues flatten out due to contention, the throughput for NDP-based queues scale linearly, and we expect the NDP-based queues to outperform even the array-based *host LF* with more concurrent threads. Also, depending on the variance in queue size, it may be wasteful to set aside a contiguous block of memory to store queue items in a host-based queue implementation.

At eight concurrent threads, *NDP ctrl buffer* has 5.3% higher operation throughput compared to *NDP open-page*. Figure 12 shows the number of memory read requests, DRAM read bursts, and DRAM row activations for the two NDP-based queues, based on an 8-thread execution. Among all the data structures that we considered, the queue has the highest controller data buffer hit rate (98.3%, while the linked-list and skiplist have approximately 50% and 40% hit rates, respectively), and this greatly reduces the number of DRAM bursts and row activations.

However, the impact of the data buffer on overall performance is not as dramatic compared to the pointer-chasing data structures because of Amdahl's law. In pointer-chasing data structures, successive memory accesses consume a large portion of the execution time in a data structure operation. On the other hand, each queue operation requires only one memory access, and because this is such a small portion of the total operation execution time, reducing the memory access latency improves the overall performance only slightly.

	average total power (W)			
# threads	host LF	host dual FC	NDP open-page	NDP ctrl buffer
2	7.848	7.844	7.936	7.923
4	7.850	7.843	7.936	7.927
6	7.850	7.844	7.940	7.930
8	7.851	7.845	7.942	7.931

Table 4: Average power dissipation for various implementations across varying numbers of concurrent threads. (List-based queue results omitted for simplicity.)

8.2 Energy & Power

Despite the limited performance improvement from the data buffer, the significant reduction in number of DRAM read bursts and DRAM row activations reduces average power consumption. Fewer DRAM row activations lead to lower NDP vault activation energy, and fewer DRAM read bursts lead to lower NDP vault read energy. Also, in *NDP ctrl buffer*, activated DRAM rows can be precharged immediately after the buffer has been filled, whereas in *NDP open-page* rows must remain activated. This leads to a lower background power, for precharged rows draw slightly less leakage current than activated rows. As shown in Table 4, *NDP ctrl buffer* consumes less power on average than *NDP open-page*, which had not been the case with linked-lists or skiplists. In fact, for queues, the main factor in the power difference between NDP-based and host-based implementations is power drawn from the NDP cores.

FIgure 11b shows the total energy consumption of various queue implementations. Although the energy consumption of NDP-based queues are higher than that of the array-based *host LF*, at eight concurrent threads, *NDP open-page* and *NDP ctrl buffer* each consume 28% and 31.8% less energy than the list-based *host LF*. *NDP ctrl buffer* consumes 5.2% less energy than *NDP open-page*.

9 RELATED WORK

While Liu *et al.* [26] proposed ways to refactor data structures to exploit NDP, they did not attempt any empirical tests of the resulting data structures, relying instead on a theoretical analysis based on a simple model of hardware latencies. Here, we implement actual software kernels of the NDP-based concurrent data structures on a full-system NDP architecture framework and provide empirical performance and energy analysis based on real hardware constraints.

Several other works have designed and evaluated near-data-processing mechanisms that improve data structure performance and energy consumption [18, 19, 32], but all are focused only on pointer-chasing data structures. Moreover, while [18] does evaluate the scalability of their design, the design targets only linked-list traversals. The works of [19] and [32] do not scrutinize the impact of increased concurrency on performance and energy, even though their designs support parallelism.

To our knowledge, our work is the first to implement and empirically evaluate *contended* concurrent data structures with near-data-processing. Nai *et al.* [28] offload atomic operations to near-data computation units to address contention issues in graph processing, but our work focuses on more conventional data structures in which contention is the sole performance bottleneck.

Many of the prior work also support concurrency with near-data-processing. To this end, Zhang *et al.* [36] implement GPU compute units as the NDP cores, but the compute units share data in all the vaults and do not leverage the benefits of vault-level locality. Other works [1, 2, 11, 15, 24, 31] implement NDP cores coupled with NDP vaults, which not only take advantage of data locality in NDP vaults but also enable concurrent execution among NDP cores. However, the parallelization techniques and data partitioning schemes to exploit provided concurrency in these works are very specific to the application they target.

There are works that provide more generic enhancements with near-data-processing. The work of Yikbarek *et al.* [34] builds near-data accelerators for representative data-intensive operations that are part of many big-data workloads. Other work [23] implements the multiply-accumulate operation in the logic die of 3D-stacked memory, which is a commonly used operation in many digital signal processing systems. The host processor ISA is extended in [3] to provide data-locality-aware general instruction execution. The work of [35] provides power management schemes that are applicable to any near-data-processing system. Finally, there are also works that utilize new memory technologies such as non-volatile memory [14, 21] to implement simple yet massively parallel arithmetic or bitwise logic directly into the memory circuitry. While these proposed schemes have been shown to provide performance and/or power advantages, they are orthogonal to what we are proposing in this paper.

10 CONCLUSION & FUTURE WORK

In this work, we provide the architectural support and actual software implementations of general-purpose concurrent data structures that are adapted to exploit near-data-processing hardware. We show through empirical evidence that thorough understanding of hardware limitations is essential in order to maximize performance and energy gains with NDP-based concurrent data structures. Our work in identifying and addressing the discrepancy between ideal and realistic gains of near-data-processing serves as a valuable guide for future investigations.

Based on our findings, we propose adding a small data buffer to the NDP controller. The NDP-based concurrent data structures – without any additional algorithmic modifications and with minimal hardware support – yield better performance and energy consumption than state-of-the-art concurrent data structures in many cases, particularly when the data structures are bottlenecked by frequent memory accesses.

We achieved good performance gains and energy savings with the simplest hardware and software designs. More sophisticated hardware-software co-designs are needed in order to further improve the data structures implemented in this work or to implement other concurrent data structures. The problems that need to be addressed in the hardware-software co-design include but are not limited to: referencing data across different NDP vaults, dynamically reallocating data among vaults to balance workloads, and designing specialized near-memory accelerators that further increase the benefits of NDP.

We plan to publicly release the data structure software libraries that we implemented, as well as our extensions to the SMCSim framework.

11 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments. We would also like to thank Erfan Azarkhish, who provided the original SMCSim simulator, and students Jonathan Lister, Ayako Shimizu, and Zhuohao (Jack) Yang, who helped in the early stages of this work. This work was supported by National Science Foundation Grant 1561807.

REFERENCES

[1] Paula Aguilera, Dong Ping Zhang, Nam Sung Kim, and Nuwan Jayasena. 2016. Fine-Grained Task Migration for Graph Algorithms using Processing in Memory. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 489–498.

[2] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 105–117.

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 336–348.

[4] JEDEC Solid State Technology Association. 2013. High Bandwidth Memory (HBM) DRAM. *Standard JESD235* (2013).

[5] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. 2016. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *International Conference on Architecture of Computing Systems*. Springer, 19–31.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[7] Amirali Boroumand, Saugata Ghose, Youngsok Kim, Rachata Ausavarungnirun, Eric Shiu, Rahul Thakur, Daehyun Kim, Aki Kuusela, Allan Knies, Parthasarathy Ranganathan, and Onur Mutlu. 2018. Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 316–331. https://doi.org/10.1145/3173162.3173177

[8] Amirali Boroumand, Saugata Ghose, Brandon Lucia, Kevin Hsieh, Krishna Malladi, Hongzhong Zheng, and Onur Mutlu. 2017. LazyPIM: An efficient cache coherence mechanism for processing-in-memory. *IEEE Computer Architecture Letters* (2017).

[9] Karthik Chandrasekar, Benny Akesson, and Kees Goossens. 2011. Improved power modeling of DDR SDRAMs. In *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 99–108.

[10] ARM Cortex. 2013. A15 MPCore processor technical reference manual. *ARM Limited, Jun* 24 (2013), 12.

[11] Palash Das, Shivam Lakhotia, Prabodh Shetty, and Hemangee K Kapoor. 2018. Towards Near Data Processing of Convolutional Neural Networks. In *VLSI Design and 2018 17th International Conference on Embedded Systems (VLSID), 2018 31st International Conference on*. IEEE, 380–385.

[12] Fernando A. Endo, Damien Courousse, and Henri-Pierre Charles. 2015. Micro-architectural Simulation of Embedded Core Heterogeneity with Gem5 and McPAT. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '15)*. ACM, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2693433.2693440

[13] Mikhail Fomitchev and Eric Ruppert. 2004. Lock-free Linked Lists and Skip Lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing (PODC '04)*. ACM, New York, NY, USA, 50–59. https://doi.org/10.1145/1011767.1011776

[14] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171

[15] M. Gao, G. Ayers, and C. Kozyrakis. 2015. Practical Near-Data Processing for In-Memory Analytics Frameworks. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 113–124. https://doi.org/10.1109/PACT.2015.22

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

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

[18] Byungchul Hong, Gwangsun Kim, Jung Ho Ahn, Yongkee Kwon, Hongsik Kim, and John Kim. 2016. Accelerating Linked-list Traversal Through Near-Data Processing. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/2967938.2967958

[19] Kevin Hsieh, Samira Khan, Nandita Vijaykumar, Kevin K Chang, Amirali Boroumand, Saugata Ghose, and Onur Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*. IEEE, 25–32.

[20] 2014. Hybrid memory cube specification 2.1. *Last Revision Nov 2015* (2014). http://www.hybridmemorycube.org/download20/, accessed 9/18/2017.

[21] Mohsen Imani, Saransh Gupta, and Tajana Rosing. 2018. GenPIM: Generalized Processing In-Memory to Accelerate Data Intensive Applications. (2018).

[22] Bruce Jacob, Spencer Ng, and David Wang. 2010. *Memory systems: cache, DRAM, disk*. Morgan Kaufmann.

[23] Dong-Ik Jeon, Kyeong-Bin Park, and Ki-Seok Chung. 2018. HMC-MAC: Processing-in Memory Architecture for Multiply-Accumulate Operations with Hybrid Memory Cube. *IEEE Comput. Archit. Lett.* 17, 1 (Jan. 2018), 5–8. https://doi.org/10.1109/LCA.2017.2700298

[24] Jeremie S Kim, Damla Senol Cali, Hongyi Xin, Donghyuk Lee, Saugata Ghose, Mohammed Alser, Hasan Hassan, Oguz Ergin, Can Alkan, and Onur Mutlu. 2018. GRIM-Filter: Fast seed location filtering in DNA read mapping using processing-in-memory technologies. *BMC genomics* 19, 2 (2018), 89.

[25] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. 469–480. https://doi.org/10.1145/1669112.1669172

[26] Zhiyu Liu, Irina Calciu, Maurice Herlihy, and Onur Mutlu. 2017. Concurrent Data Structures for Near-Memory Computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/3087556.3087582

[27] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*. ACM, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

[28] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. 2017. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 457–468.

[29] Lifeng Nai and Hyesoon Kim. 2015. Instruction Offloading with HMC 2.0 Standard: A Case Study for Graph Traversals. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS '15)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2818950.2818982

[30] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[31] Seth H Pugsley, Jeffrey Jestes, Huihui Zhang, Rajeev Balasubramonian, Vijayalakshmi Srinivasan, Alper Buyuktosunoglu, Al Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+ logic devices on MapReduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE, 190–200.

[32] Paulo C Santos, Geraldo F Oliveira, João P Lima, Marco AZ Alves, Luigi Carro, and Antonio CS Beck. 2018. Processing in 3D memories to speed up operations on complex data structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*. IEEE, 897–900.

[33] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[34] Salessawi Ferede Yitbarek, Tao Yang, Reetuparna Das, and Todd Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 1449–1452.

[35] Chao Zhang, Tong Meng, and Guangyu Sun. 2018. PM3: Power Modeling and Power Management for Processing-in-Memory. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 558–570.

[36] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: throughput-oriented programmable processing in memory. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 85–98.

[37] Mingxing Zhang, Youwei Zhuo, Chao Wang, Mingyu Gao, Yongwei Wu, Kang Chen, Christos Kozyrakis, and Xuehai Qian. 2018. GraphP: Reducing Communication for PIM-based Graph Processing with Efficient Data Partition. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*. IEEE, 544–557.