Hardware-Software Coordination for High Performance Concurrent Data Structures with Near-Data-Processing

Jiwon Choe[1] jiwon_choe@brown.edu
Amy Huang[1] amy_huang1@brown.edu
R. Iris Bahar[1] iris_bahar@brown.edu
Maurice Herlihy[1] mph@cs.brown.edu
Tali Moreshet[2] talim@bu.edu

[1]Brown University, [2]Boston University

Recent advances in three-dimensional (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 [8]. We investigate how near-memory accelerators can be combined with novel concurrent data structure algorithms in order to exploit the low-latency, high-bandwidth memory access of NDP architectures, while also preserving the high concurrency of conventional architectures. Concurrent data structures are used in many applications, and adapting them to NDP architectures is a key step towards making such architectures useful.

3D die-stacked memory consists of multiple DRAM dies stacked on top of a single logic die. The memory is divided into vertical sections: the memory dies of the vertical section form the *NDP vault*, which contains multiple DRAM banks, and the logic die of the vertical section contains a near-memory compute unit (*NDP core*) and a memory controller (*NDP controller*) that manages memory requests to the coupled NDP vault. We assume that the NDP core is a simple in-order, single-cycle processor without cache.

Liu *et al.* [6] described how the *flat-combining* (*FC*) synchronization scheme [5] can be applied to NDP-based linked-lists, skiplists, and FIFO queues to manage concurrency. However, these data structures were not empirically tested, relying instead on theoretical performance analysis based on simple hardware latency assumptions.

Figure 1 shows the various NDP-based data structures. The data structure resides in one or more NDP vaults, and each NDP core has exclusive access to the portion of the data structure contained in its coupled NDP vault. Host processor threads can simply send data structure operation requests (*e.g.* `contains(X)`, `add(X)`, `remove(X)`) to the corresponding NDP core. Each NDP core can *combine* the concurrent requests and execute them on behalf of host threads, while each host thread waits for the NDP core to complete its requested operation.

In our work, we implement and test these NDP-based data structures on a full-system NDP architecture framework with realistic hardware constraints. To this end, we extended SMC-Sim [1], a gem5 [2] cycle-accurate full-system simulator with the software stack and hardware support for NDP.

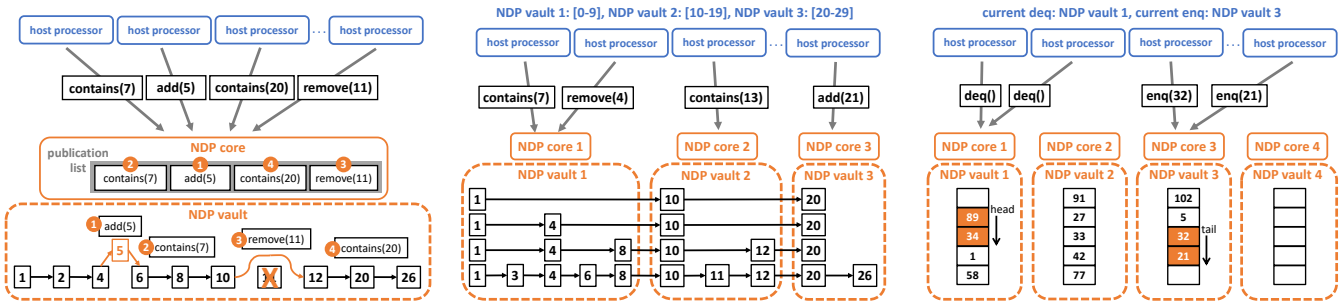
Through this more realistic and detailed analysis, we find that Liu *et al.*'s work had overestimated the performance benefits of NDP-based data structures. The theoretical analysis had two major pitfalls: 1) it ignored the cache impacts in host-based concurrent data structure performance, and 2) it had overly optimistic assumptions on near-data-processing memory access latencies. We address these shortcomings and show that lightweight changes to hardware – inspired by observations on data structures' data access patterns and underlying DRAM activity – can significantly improve NDP-based data structure performance while using the same algorithm.

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 of the row locality. With the close-page row-buffer-management policy, this leads to repetitive row activations and data movement. Even with the open-page policy, in which an activated row is kept open until another memory access requires activating a different row, delays associated with repetitive data movement are not removed. Moreover, data is transferred 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, we add a small buffer 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, in order to prevent unnecessary data movement. 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.

Figure 2 shows our modified design. Only two new hardware components are 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 removes delays associated with accessing data in DRAM banks. Only if the requested data is not in the data buffer, the NDP con-



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

(b) Partitioned and flat combining skiplist.

(c) Partitioned and flat combining FIFO queue.

Figure 1: NDP-based concurrent data structure design.

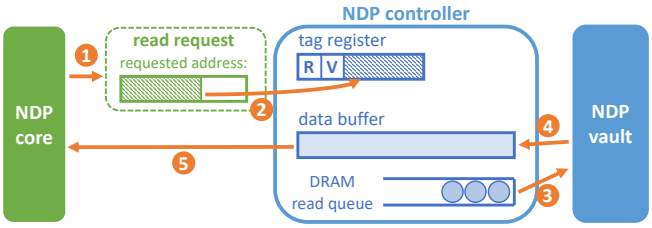


Figure 2: Proposed NDP controller design. The tag register and data buffer are added to the existing NDP controller. Steps 3 and 4 are skipped if the tag portion of the requested read address matches the tag register content.

troller creates DRAM access operations to fill the buffer (steps 3, 4).

For the NDP-based linked-list (Figure 1a), we assume that the entire list is contained in a single NDP vault. The NDP core combines and sorts received operations in order of the requested keys, which allows the NDP core to execute all combined operations over a single traversal through the list.

We set the initial linked-list to be approximately 5MB in total size (2.5x L2 cache size). Performance is measured in terms of operation throughput – number of data structure operations completed within a fixed time. We implement the NDP-based linked-lists with the NDP controller as described in Figure 2 (*NDP ctrl buffer*) and with the NDP controller as a generic, unmodified memory controller. For the latter, we evaluate using the close-page and open-page row-buffer-management policies (*NDP close-page* and *NDP open-page*, respectively). At eight concurrent threads, *NDP close-page* barely performs better than *host lazy-lock* [4], the state-of-the-art concurrent linked-list implementation. However, *NDP open-page* and *NDP ctrl buffer* have 16.4% and 73.7% higher operation throughput than *host lazy-lock*, respectively.

The NDP-based skiplist (Figure 1b) is optimized by partitioning the skiplist across multiple NDP vaults based on predefined disjoint range of keys. Host processors send operation requests to the appropriate NDP core based on the requested operation key. Increasing the number of partitions enhances concurrency and thereby improves performance; here we discuss the results from using eight skiplist partitions.

Based on the findings from the linked-list, we evaluated the NDP-based skiplists with the unmodified NDP controller using open-page row-buffer-management policy (*NDP open-page*) and with the modified NDP controller (*NDP ctrl buffer*). When the initial skiplist size is set to 850MB, at eight concurrent threads, *NDP open-page* has 6.8% *lower* operation throughput than *host lock-free* [3], the state-of-the-art host-based concurrent skiplist implementation. However, the hardware modification significantly improves performance, and *NDP ctrl buffer* shows 7.4% *higher* operation throughput than *host lock-free*.

Cache effects account for the relatively high performance of *host lock-free*. The skiplist is inherently a balanced tree-like structure, so a skiplist operation 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 memory, even with a skiplist that is much larger than last-level cache.

The NDP-based FIFO queue (Figure 1c) is optimized based on flat-combining and partitioning. The NDP core effectively removes contention at the head and tail, for it is the only thread that operates on the data structure. Partitioning the queue across multiple NDP vaults allows for separate enq and deq partitions and adds parallelism. Again, we implement the NDP-based queue with unmodified NDP controllers using open-page row-buffer-management policy (*NDP open-page*) and with the modified NDP controller (*NDP ctrl buffer*). We compare the results against *host lock-free* queue [7], the state-of-the-art concurrent queue, which can be implemented as a circular array with head and tail indices (array-based) or as a linked-list with head and tail node pointers (list-based).

At eight concurrent threads, *NDP open-page* and *NDP ctrl buffer* show 40.5% and 48% higher operation throughput than list-based *host lock-free*. However, because the array-based *host lock-free* uses fewer atomic operations for each enq or deq operation compared to the list-based *host lock-free*, array-based *host lock-free* completely outperforms either NDP-based implementation. Nevertheless, while the operation throughput for either *host lock-free* queue flattens out with increasing number of threads, the throughput for NDP-based queues scale linearly, and we expect the NDP-based queues to outperform even the array-based *host lock-free* with more concurrent threads.

References

[1] AZARKHISH, E., ROSSI, D., LOI, I., AND BENINI, L. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *International Conference on Architecture of Computing Systems* (2016), Springer, pp. 19–31.

[2] BINKERT, N., BECKMANN, B., BLACK, G., REINHARDT, S. K., SAIDI, A., BASU, A., HESTNESS, J., HOWER, D. R., KRISHNA, T., SARDASHTI, S., ET AL. The gem5 simulator. *ACM SIGARCH Computer Architecture News 39*, 2 (2011), 1–7.

[3] FOMITCHEV, M., AND RUPPERT, E. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2004), PODC '04, ACM, pp. 50–59.

[4] HELLER, S., HERLIHY, M., LUCHANGCO, V., MOIR, M., SCHERER, W. N., AND SHAVIT, N. A lazy concurrent list-based set algorithm. In *Proceedings of the 9th International Conference on Principles of Distributed Systems* (Berlin, Heidelberg, 2006), OPODIS'05, Springer-Verlag, pp. 3–16.

[5] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures* (2010), SPAA '10, ACM, pp. 355–364.

[6] LIU, Z., CALCIU, I., HERLIHY, M., AND MUTLU, O. Concurrent data structures for near-memory computing. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (2017), SPAA '17, ACM, pp. 235–245.

[7] MICHAEL, M. M., AND SCOTT, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1996), PODC '96, ACM, pp. 267–275.

[8] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news 23*, 1 (1995), 20–24.