HybriDS: Cache-Conscious Concurrent Data Structures for Near-Memory Processing Architectures

Jiwon Choe
Brown University
Providence, RI, USA
jiwon_choe@brown.edu

Andrew Crotty
Northwestern University
Evanston, IL, USA
crottyan@northwestern.edu

Tali Moreshet
Boston University
Boston, MA, USA
talim@bu.edu

Maurice Herlihy
Brown University
Providence, RI, USA
mph@cs.brown.edu

R. Iris Bahar
Colorado School of Mines
Golden, CO, USA
ribahar@mines.edu

ABSTRACT

In recent years, the ever-increasing impact of memory access bottlenecks has brought forth a renewed interest in *near-memory processing* (*NMP*) architectures. In this work, we propose and empirically evaluate *hybrid data structures*, which are concurrent data structures custom-designed for these new NMP architectures.

We focus on cache-optimized data structures, such as skiplists and B+ trees, that are often used as index structures in online transaction processing (OLTP) systems to enable fast key-based lookups. These data structures are *hierarchical*, where lookups begin at a small number of top-level nodes and diverge to many different node paths as they move down the hierarchy, such that nodes in higher levels benefit more from caching. Our proposed hybrid data structures split traditional hierarchical data structures into a *host-managed* portion consisting of higher-level nodes and an *NMP-managed* portion consisting of the remaining lower-level nodes, thus retaining and further enhancing the cache-conscious optimizations of their conventional implementations. Although the idea might seem relatively simple, the splitting of the data structure prompts new synchronization problems, and careful implementation is required to ensure high concurrency and correctness.

We provide implementations of a *hybrid skiplist* and a *hybrid B+ tree*, and we empirically evaluate them on a cycle-accurate full-system architecture simulator. Our results show that the hybrid data structures have the potential to improve performance by more than 2× compared to state-of-the-art concurrent data structures.

CCS CONCEPTS

• **Theory of computation** → *Data structures design and analysis*; *Concurrent algorithms*; • **Hardware** → *Emerging architectures.*

KEYWORDS

near-memory processing; concurrent data structures

SPAA '22, July 11–14, 2022, Philadelphia, PA, USA.

ACM ISBN 978-1-4503-9146-7/22/07…$15.00
https://doi.org/10.1145/3490148.3538591

ACM Reference Format:
Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R. Iris Bahar. 2022. HybriDS: Cache-Conscious Concurrent Data Structures for Near-Memory Processing Architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '22), July 11–14, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3490148.3538591

1 INTRODUCTION

The *memory wall* [64], which refers to the increasing performance gap between processor speeds and memory access speeds, has been a well-known issue for nearly three decades. Moreover, its impact has become more pronounced than ever in recent years with the ever-increasing number of data-intensive applications. Designing systems and hardware that alleviate the memory bottleneck has since been a major concern, and this has sparked a renewed interest in *near-memory processing* (*NMP*) architectures that move computation physically close to memory. While NMP architectures do not cut down latencies inherent to DRAM, they reduce performance degradation and energy consumption caused by data movement on long and narrow off-chip interconnects.

In this work, we propose and empirically evaluate *hybrid data structures*, which are concurrent data structures custom-designed for new NMP architectures. We specifically focus on data structures that have already been heavily optimized to exploit on-chip cache locality of conventional systems, which have benefited little from prior NMP-based flat-combining implementations [16, 44]. However, our proposed hybrid data structures utilize the NMP architecture more effectively in order to retain and further enhance the data structures' cache-locality benefits, thereby providing significant performance gains.

Existing cache-optimized concurrent data structures have been particularly important for in-memory *online transaction processing* (*OLTP*) systems. Workloads on OLTP systems are typified by high volumes of short-lived queries on relatively few data items; in order to process such queries at scale and with minimal delay, OLTP systems employ cache-optimized concurrent data structures as *index structures* that enable key-based lookups on stored data. However, microarchitectural analyses of in-memory OLTP workloads have shown that they are still plagued by memory stalls: long-latency last-level cache misses due to random data accesses can account for more than half of the execution time in otherwise highly optimized OLTP systems [59, 60]. Despite their cache-conscious optimizations,

index structures are fundamentally *pointer-chasing* data structures, in which lookups involve iteratively accessing nodes at memory addresses referenced by next-node pointers stored at each node. This inevitably results in unpredictable memory access patterns, which is exacerbated by the fact that indexes in OLTP systems are typically significantly larger than the last-level cache [70].

Nonetheless, we make the observation that these index structures are often pointer-chasing yet *hierarchical* data structures, in which nodes are organized by levels in a top-down hierarchy. Importantly, the higher levels exhibit better cache locality than lower levels. Lookups on hierarchical data structures always begin at the few top-level nodes but diverge to many different node paths while moving down the hierarchy. For this reason, the access frequency of each node depends heavily on its position in the hierarchy; that is, over several lookups, nodes in the higher levels are accessed much more frequently (and thereby benefit more from caching) than nodes in the lower levels.

The cache-conscious hybrid data structures that we propose are designed based on this observation. We split a hierarchical data structure into (*i*) a *host-managed* portion consisting of higher levels of the data structure and (*ii*) an *NMP-managed* portion consisting of the remaining lower levels. This split effectively "pins" frequently accessed higher-level nodes to the on-chip cache without any hardware modifications, and it also prevents infrequently accessed lower-level nodes from polluting the cache.

Most prior works that utilize NMP architectures focus on alleviating memory bandwidth bottlenecks by exploiting the high internal memory bandwidth available to near-memory compute units [2–4, 20, 33–35, 39–41, 43, 51, 65]. However, since index lookups in OLTP systems are limited by memory *access latency* rather than memory bandwidth, our hybrid data structure design focuses on reinforcing cache-conscious optimizations to reduce expensive cache misses. Note that reducing the amount of cache misses and memory accesses improves not only the performance but also the energy consumption of index lookup operations.

Despite the seemingly simple concept of our idea, rigorous technical detail with respect to the architecture and algorithms is required to ensure high concurrency and correctness. Conventional concurrent data structures have been designed under the assumption that any operation will be applied onto a single coherent structure, but a hybrid data structure is split into two parts, each accessed and managed by different sets of threads (*i.e.*, threads in the host CPU vs. near-memory compute units). The data structure as a whole must nevertheless be kept coherent throughout the entire operation execution, even as multiple operations are applied concurrently. Careful coordination among the host threads, among the near-memory compute units, and between the host threads and near-memory compute units is required to meet high concurrency and correctness guarantees.

In summary, this paper makes the following contributions:

- We propose *hybrid data structures*, which are cache-conscious concurrent data structures designed for new NMP architectures. Importantly, our hybrid data structure design is applicable to any generic NMP architecture.
- We provide a generic design pattern for hybrid data structures and then describe concrete *hybrid* implementations of a *skiplist* and *B+ tree*, two data structures widely used in OLTP systems.

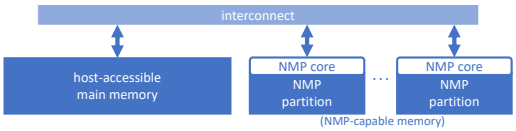


Figure 1: Baseline NMP architecture.

- We empirically evaluate the hybrid data structures on a cycle-accurate, full-system NMP architecture simulator. Depending on the workload, our NMP-hybrid approach improves performance by up to 3.12× and 2.11× for the skiplist and B+ tree, respectively, compared to non-NMP implementations.

2 NMP ARCHITECTURE BASELINE

In this work, we use a simple and generic NMP architecture (shown in Figure 1) so that we can focus on the broadly applicable algorithmic details of our proposed hybrid data structures. Specifically, we assume that memory is separated into *host-accessible main memory* and *NMP-capable memory*. By restricting host-accessible memory from having any NMP capabilities, we eliminate many challenges that NMP architectures would need to address, including data coherence and virtual address translation across host processors and NMP cores. However, we expect our hybrid data structure designs to be extensible to more sophisticated NMP architectures in which these architectural challenges are addressed.

The NMP-capable memory is further divided into physically separate partitions (*NMP partitions*). Each of these partitions is coupled with an *NMP core* that has exclusive access to data in the partition. Many recent proposals for NMP architectures—whether 2.5D/3D memory-based [3, 4, 24, 32, 41, 53, 56, 65, 66] or commodity DIMM-based [22, 39, 40]—hold similar assumptions about memory partitioning and per-partition processing units, in order to provide capacity-proportional scalability, to exploit memory-level parallelism, and to simplify hardware design with regard to shared memory access and data coherence.

Each NMP core is modeled as an in-order, single-cycle processor without any on-chip caches. The NMP core is instead equipped with a node-size hardware register that acts like a single-block cache, which was shown to be sufficient for exploiting locality exhibited at node granularity in NMP-based data structures [16].

3 HYBRID DATA STRUCTURE DESIGN

3.1 Overview

In this work, we focus on two data structures that are widely used in OLTP systems: (*i*) skiplists [52] and (*ii*) B+ trees [9, 18]. Both of these data structures are *hierarchical*, meaning that their nodes are organized by levels in a top-down hierarchy. The topmost level consists of either one node (B+ tree) or very few nodes (skiplist), and the number of nodes at each level below is greater than the number of nodes at the level above. Lookups on these data structures always begin at the topmost level, but they diverge to different node paths as the lookups move down the hierarchy. For this reason, the access frequency of each node in the data structure depends heavily on its position in the hierarchy; that is, nodes in the higher levels are accessed much more frequently (and thereby benefit more from caching) than nodes in the lower levels.

The key insight of our NMP-enabled hybrid data structures is to leverage this skew in access frequency that stems from the data

structure's topology. Essentially, we divide each data structure into a *host-managed* portion consisting of the higher levels of the data structure and an *NMP-managed* portion consisting of the lower levels. The *host-NMP split point* (*i.e.*, the level at which to divide the data structure) is chosen such that the entire host-managed portion can fit within the last-level cache of the CPU, effectively "pinning" higher-level nodes to the on-chip cache without any hardware modifications. The remaining lower levels are placed in NMP-capable memory accessed only by NMP cores, thus preventing infrequently accessed nodes from polluting the host cache.

Although a seemingly simple idea, rigorous technical detail at the algorithmic level is required to ensure high concurrency and correctness in these hybrid data structures. Conventional concurrent data structures assume that operations are applied on a single coherent structure, but a hybrid data structure is divided into two parts, each accessed and managed by different sets of threads (*i.e.*, host cores vs. NMP cores). Both parts must therefore be kept coherent throughout the entire operation, prompting new synchronization problems. Careful coordination among the host threads, among the NMP cores, and between host and NMP cores is essential. Moreover, while skiplists and B+ trees have similar hierarchical topologies, the two data structures by design have different correctness conditions for concurrent modifications, so each of the hybrid algorithms must be designed accordingly.

For concurrency in the NMP-managed portion, we build upon prior work that applies *flat-combining* [28] to NMP-based data structures [16, 44]. We first elaborate on the implementation of NMP-managed portions of the hybrid data structures (§3.2). The following subsections then provide details on the *hybrid skiplist* and *hybrid B+ tree* designs (§3.3 and §3.4, respectively). Lastly, §3.5 describes an optimization that further improves the performance of hybrid data structures.

3.2 NMP-Managed Portion

Conventional flat-combining data structures have a single designated *combiner thread* that actually applies operations to the data structure. Each concurrent thread simply posts its operation request to an assigned slot in the *publication list*, which the combiner thread iterates through to check for unserved operation requests. In NMP-based flat-combining data structures, each NMP core becomes the combiner thread for the portion of the data structure stored in its coupled NMP partition. We assume that each NMP core is equipped with a small scratchpad memory, and a portion of it is memory-mapped into the host address space so that the space can be used as the publication list.[1]

In order to offload a data structure operation to the NMP-managed portion, the host thread writes the following items to its assigned slot in the target NMP core's publication list:

(1) lookup key (4 bytes)
(2) associated value (for update/insert operations; 4 bytes)
(3) pointer to the node from which the NMP core should begin its traversal (referred to as *begin-NMP-traversal node*; 4 bytes)
(4) operation type (read, update, insert, remove, or other operations for maintenance and debugging; 3 bits)
(5) flag indicating that the slot contains a valid operation (1 bit)

[1]Note that our evaluation framework (elaborated in §4) accurately models latencies caused by memory-mapped I/O.

After offloading an operation, the host thread polls on the flag to check the status of the operation.

In previous NMP-based data structures [16, 44], *begin-NMP-traversal node* (3) was unnecessary because the NMP core traversed the entire data structure, such that all traversals began at a fixed sentinel node within the NMP partition. On the other hand, host-side traversals become shortcuts into NMP-managed lower levels in our approach, so NMP-side traversals can begin at any node referenced from the bottommost host-managed level.

However, because the NMP core processes operations in its publication list one at a time, the begin-NMP-traversal node of an operation may end up being modified by a concurrent operation that gets processed earlier in the NMP core. This is a new synchronization problem that affects correctness in hybrid data structures, and therefore the hybrid skiplist and the hybrid B+ tree provide tailored mechanisms for the NMP core to detect such issues, in which case the NMP core aborts the operation and notifies the host thread to retry the operation accordingly.

When the NMP core finishes processing an offloaded operation, it writes the following to the publication list before clearing the valid flag:

(1) retry flag if the NMP-side operation requires a retry (1 bit)
(2) return value indicating success/failure (1 bit)
(3) associated value (for read operations; 4 bytes)
(4) pointer address to node created in the NMP partition during an insert operation (if applicable; 4 bytes)

In addition to flat-combining, which handles the correct application of concurrent operations to the NMP-managed portion of a hybrid data structure, the NMP-managed portion is divided across multiple NMP partitions to increase parallelism. Because each NMP partition is an isolated portion of memory accessed and modified exclusively by its coupled NMP core, operations to different partitions can run in parallel without data races or coherence issues.

3.3 Hybrid Skiplist

The skiplist is an ordered, pointer-chasing data structure with multiple levels of pointers at each node. Each node is assigned a *height*, and a node holds next-node pointers to the succeeding node at each level up to its height. The height is taken from a particular distribution such that all nodes are linked together at level 0 (bottommost level), but each node at level i is only half as likely to appear at level $i+1$. The skiplist is usually configured with $\log_2 N$ total levels, where N is the number of key-value pairs at initialization.

A lookup on the skiplist is a top-down search through the levels. The lookup initially traverses through next-node pointers at the topmost level, but once it reaches a node with a key larger than the lookup key, it backtracks to the previous node on the same level and continues the traversal at the level immediately below. This process is repeated until the lookup key is found. The particular distribution of node heights allows for a logarithmic time execution, similar to a balanced binary search tree.

The cumulative size of the top x levels of a skiplist can be estimated as 2^x`sizeof(Node)`. In the hybrid skiplist (Figure 2), the number of levels to be placed in the host-managed portion is chosen such that this is approximately equal to the last-level cache size.

The host-managed portion of the hybrid skiplist is implemented according to the state-of-the-art *lock-free* skiplist [23, 29]. Nodes in

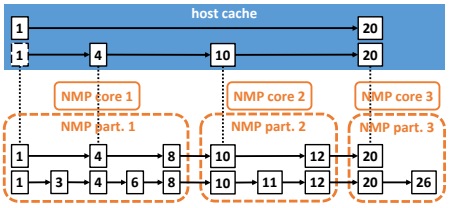


Figure 2: Hybrid skiplist design.

the host-managed portion hold references to their counterparts in the NMP-managed portion, in order to provide shortcuts (*i.e.*, references to begin-NMP-traversal nodes) for NMP-side traversals. The NMP-managed portion is implemented according to the NMP-based flat-combining data structures described in §3.2. Here, we simply assume that nodes in the NMP-managed portion are distributed across NMP partitions based on predefined, equal-size ranges of keys. The data structure stores information on the key ranges so that the target NMP partition for an operation can easily be determined with the lookup key. The algorithm can be extended to other partitioning schemes for reasons such as better load balancing, but we leave this to future work.

Guaranteeing correctness in the hybrid skiplist is relatively straightforward. For correctness, the skiplist data structure simply needs to always maintain the skiplist property [30], which is that the set of nodes at every level i is a subset of all nodes at level $i-1$. The lock-free skiplist and the NMP-based flat-combining skiplist both correctly maintain these properties by applying insertions from bottom to top and by applying removals from top to bottom. In the hybrid skiplist, we ensure the same by applying insertions in the NMP-managed portion first, and then in the host-managed portion; likewise, removals are applied in the host-managed portion first, and then in the NMP-managed portion.

Data modifications in the NMP-managed portion of the data structure that may affect correctness are cases where the begin-NMP-traversal node of an NMP-offloaded operation is removed by a concurrent operation processed earlier. To ensure that such modifications are detected, the NMP core always first marks a `remove` target node as logically deleted, and then proceeds to physically remove it. Because the NMP core is single-threaded, a logical and physical removal will not happen concurrently, but the logical deletion ensures that another offloaded operation does not begin its traversal at a stale, deleted node.

Listing 1 shows pseudocode for the `insert` operation on the host-side. After the initial traversal through the host-side (line 7), the begin-NMP-traversal node for the operation is identified (lines 14-15). Then, the `SEND_NMP_INSERT_OPERATION` (line 16) writes operation information to the thread's assigned slot in the target NMP partition's publication list, according to the description in §3.2. Note that most of the steps in Listing 1 apply to `read`, `update`, and `remove` operations as well.

Once the operation is offloaded to a target NMP core, the NMP core carries out its portion of the operation as described in Listing 2. Line 7 checks if the begin-NMP-traversal node has been marked as deleted by a concurrent operation processed earlier, in which case the retry flag is set for the host thread (lines 8-9). The remainder of the operation is carried out according to the single-threaded skiplist algorithm (lines 14-25). Note that the described process is identical across all skiplist operations.

```
1  bool insert(int key, int value, int thread_id) {
2    Node* preds[HOST_HEIGHT];
3    Node* succs[HOST_HEIGHT];
4    Node* newnode, foundnode, nmpnode = NULL;
5    int part_id, height = 0;
6    bool ret = false;
7    foundnode = find(key, head, preds, succs); // finds position
         of node associated with key and fills in preds, succs
         arrays with pointers to the predecessor and successor
         nodes at each level
8    if (foundnode != NULL) { return false; }
9    part_id = get_partition(key);
10   height = rand_height(); // height of new node
11   if (height > NMP_HEIGHT) {
12     newnode = new Node(key, value, height-NMP_HEIGHT);
13     newnode->nmp_ptr = NULL; }
14   if (part_id == get_partition(preds[0]->key)) {
15     nmpnode = preds[0]->nmp_ptr; }
16   SEND_NMP_INSERT_OP(); //according to Section 3.2
17   if (GET_NMP_RETRY_FLAG()) {
18     if (newnode != NULL) { delete newnode; }
19     // retry from beginning
20   } else {
21     ret = GET_NMP_RETURN_VALUE();
22     if (newnode != NULL) {
23       if (!ret) {
24         delete newnode;
25         return false; }
26       newnode->nmp_ptr = GET_NMP_NODE_PTR();
27       // From here, link newnode into host-side levels
28       // according to LF skiplist algorithm (code omitted)
29     } } return ret; }
```

Listing 1: Hybrid skiplist host-side `insert` operation.

```
Node* preds[NMP_HEIGHT];
Node* succs[NMP_HEIGHT];
Node* curr, foundnode, newnode;
// publist[i] contains operation information offloaded from
     thread i
if (publist[i].nmp_ptr != NULL) {
  curr = publist[i].nmp_ptr;
  if (MARKED(curr)) {
    // curr has been removed by a concurrent remove.
    // set RETRY flag and move on to next op (code omitted)
  }
} else {
  curr = partition_head; }
foundnode = find(key, curr, preds, succs);
if (foundnode != NULL) {
  publist[i].return_value = 0;
} else {
  newnode= new Node(publist[i].key, publist[i].value);
  if (publist[i].height > NMP_HEIGHT)
    newnode->height = NMP_HEIGHT;
  else
    newnode->height = publist[i].height;
  newnode->host_ptr = publist[i].host_ptr;
  // link newnode into skiplist according to single-threaded
     skiplist algorithm (code omitted)
  publist[i].nmp_ptr = newnode;
  publist[i].return_value = 1; }
```

Listing 2: Hybrid skiplist NMP-side `insert` operation.

The hybrid skiplist retains *linearizability* [31], a strong correctness guarantee for concurrent data structures. Linearizable data structures are identified by *linearization points*, which are single points when operations take instantaneous effect. The pseudocode in Listings 1 and 2 shows that an insertion in the hybrid skiplist takes instantaneous effect when the new node is linked into the NMP-managed portion. Although we omit pseudocode for other operations, removals also takes instantaneous effect when the removal target node is removed from the NMP-managed portion.

The linearization points for successful `read` and `update` operations are points where the value associated with the target node is successfully read or updated, whether in the host-managed or NMP-managed portion. However, to account for concurrent `insert` and `update` operations on the same key (where the `update` is offloaded to NMP after the new node is linked in the NMP-managed portion

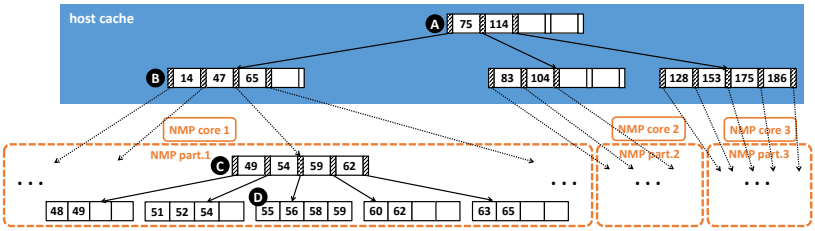


Figure 3: Hybrid B+ tree design.

but before it is linked in the host-side), the NMP-side `update` returns the target node's `host_ptr` through the publication list. Using this information, the host-side `update` ensures that the new value is updated in the host-side node as well. This process guarantees future `read` operations will read correctly updated values.

3.4 Hybrid B+ Tree

The B+ tree is an n-ary tree data structure consisting of a root, inner nodes, and leaf nodes. Leaf nodes store up to n key-value pairs in sorted order; non-leaf nodes (*i.e.*, the root and inner nodes) hold up to n child node pointers to subtrees, along with *dividing keys*. A subtree to the left of a dividing key must only contain keys that are less than or equal to the dividing key; similarly, a subtree to the right of a dividing key must only contain keys that are greater than the dividing key. Lookups on a B+ tree require top-down pointer-chasing from the root to the leaf containing the lookup key, with the traversal path guided by dividing keys in the inner nodes.

The B+ tree also maintains the following invariants: (*i*) each non-root inner node has at least $\lceil n/2 \rceil$ children, (*ii*) each leaf node holds at least $\lceil n/2 \rceil$ items, and (*iii*) the path from the root to any leaf node is the same length. Note that the value of n for in-memory OLTP systems is chosen such that the B+ tree nodes align with the cache block size (typically 64 or 128 bytes), which is to fully exploit spatial locality [54]. The resulting n is often fairly large—for example, n is 16 for the B+ tree used in the DBx1000 framework [67].

To simplify the hybrid B+ tree's configuration phase, we assume that the initial B+ tree is constructed over an existing database table. This is not uncommon when building index structures for in-memory OLTP systems, which is a major use case of B+ tree structures. Furthermore, for a sufficiently large B+ tree, the hybrid B+ tree does not require frequent reconfiguration: the initial setup becomes inappropriate only when the host-managed portion becomes significantly larger than the last-level cache, which requires the B+ tree to grow by several magnitudes in size.[2] We assume that enough memory (including the NMP partitions) is provisioned to account for the data structures and their expected growth.

Figure 3 illustrates the hybrid B+ tree design. The hybrid B+ tree is first constructed entirely in the host-managed region. The host-NMP split point for the hybrid B+ tree is determined based on the cumulative size of the top x levels. This is approximately $(1 + r\sum_{i=0}^{x-2} m^i)$`sizeof(Node)`, where r is the number of children from the root node and m is the average fanout of the non-root inner nodes. m ranges between $\frac{1}{2}n$ and $\frac{2}{3}n$, depending on the order in which initial keys are inserted into the tree [58].

[2]In such cases, the hybrid B+ tree will indeed need to be reconfigured. Because this would be an infrequent operation, we expect that it could be done during a maintenance phase, when no other concurrent operations are made on the B+ tree. However, we leave this to future work.

The lower levels for the NMP-managed portion are divided into NMP partitions at range-based boundaries chosen based on the root's grandchildren. For example, if the root has exactly rm grandchildren and the system has p NMP partitions, lower levels under the first $\frac{rm}{p}$ grandchildren nodes are pushed down to the first partition. Pointers from the bottommost level nodes of the host-managed portion are references to child nodes stored in the NMP partitions. The NMP partition information is stored with the pointer reference: since B+ tree nodes are aligned at 64 or 128 bytes, we exploit unused least significant bits of the NMP-side node pointer to store the corresponding NMP partition's ID. The initialization phase completes once all lower levels constituting the NMP-managed portion are pushed down to appropriate NMP partitions.

Once the initialization phase is complete, synchronizing concurrent operations becomes the challenge in hybrid B+ trees. In general, providing concurrency in B+ trees is nontrivial, for insertions may trigger structural changes across multiple levels to satisfy B+ tree invariants. Although infrequent, structural changes in upper levels are driven by changes in lower levels, and changes at all affected levels must be completed in order to bring the B+ tree to a correct state. This is a major difference from the skiplist, where the linking of an inserted node at each level is a stand-alone operation that does not interfere with the correctness of the overall structure. While the same applies to deletions, we relax the B+ tree invariant on the minimum number of items in the leaf node to simplify synchronization with regard to `remove` operations, which is an approach also taken in prior work [36, 49, 57, 69]. Even so, the hybrid B+ tree requires a more complicated synchronization method than the hybrid skiplist.

Furthermore, the hybrid B+ tree must be capable of synchronizing concurrent modifications in the begin-NMP-traversal node. For example, in the hybrid B+ tree of Figure 3, an `insert(key=57)` operation will split nodes D and C, and in turn, node B will be updated to hold a reference to node C′ that has been split from C. However, a concurrently offloaded operation may also have had the original node C as its begin-NMP-traversal node; if this operation gets processed by the NMP core after the `insert(57)`, it may be applied incorrectly, for some subtrees under the original node C cannot be accessed from node C at that point. The NMP-side of the hybrid B+ tree algorithm must be capable of detecting such cases to ensure correctness.

Sequence locks [11] are at the core of our hybrid B+ tree algorithm. The sequence lock is essentially a *sequence number* that is atomically incremented at the beginning and end of a critical section that involves writes. An operation that reads data protected by the sequence lock checks the sequence number at the beginning and end of its operation to verify that the data has remained unchanged

```
1  // host-managed portion:
2  struct InnerNode {
3    volatile int seqnum;
4    int level;
5    int slotuse; // # of slots in use
6    int dividing_keys[SLOTMAX];
7    Node *children[SLOTMAX+1]; };
8  //NMP-managed portion:
9  struct InnerNode {
10   int parent_seqnum; //for synchronization at host-NMP boundary
11   int level;
12   short lock; // 1 if locked, 0 otherwise
13   short slotuse;
14   int dividing_keys[SLOTMAX];
15   Node *children[SLOTMAX+1]; };
16 // LeafNodes have the same structure, except they hold data
17 // keys and values, instead of dividing keys and child ptrs.
```

Listing 3: Hybrid B+ tree node definitions.

in the meanwhile (if the data has changed, the operation can abort and restart). Because atomic operations on the sequence locks are issued only for writes, synchronization traffic is reduced compared to read-write locks. We also use the sequence numbers as version numbers for synchronization across the host-NMP boundary.

Listing 3 specifically shows how nodes are defined in our hybrid B+ tree implementation. Each node in the host-managed portion has a sequence number associated with it (line 3). However, because the NMP-managed portion is accessed by a single NMP core only, the sequence lock is unnecessary. Each node in the NMP-managed portion instead has a simple lock (line 12). It also keeps track of its parent node's latest sequence number (line 10), which is used to manage synchronization issues at the begin-NMP-traversal node.

To explain how these synchronization mechanisms are used to ensure correctness, we describe the `insert` algorithm of the hybrid B+ tree. The pseudocode in Listing 4 shows the host-side of the algorithm. As the host thread traverses down the tree, it records all nodes on the traversal path and their sequence numbers at the time of access (lines 6-7, 12-14). If a destination child node is being modified at the time of access, the traversal first waits for the write on the child to complete (lines 12-14). Afterwards, if the current node has remained unchanged, the traversal moves down the tree (lines 16-18); otherwise, the traversal moves back up the tree to the lowest ancestor node that has not been modified during the operation's execution (iterations of lines 19-22). Upon reaching the last host-side level of the B+ tree, the host thread offloads the `insert` operation to an appropriate NMP core (line 24). The appropriate NMP-side child node reference held in the last host-side node becomes the begin-NMP-traversal node.

Listing 5 shows pseudocode for the NMP-side `insert` algorithm. Lines 3-8 deal with synchronization at the host-NMP boundary, but we first describe the insertion process assuming that the begin-NMP-traversal node has not been split by a concurrent operation. In lines 9-11, the NMP core traverses down the tree, recording all nodes along the traversal path. Upon reaching the bottom, the NMP core traverses back up the path to lock each node that will be affected by the insert, starting from the leaf node (lines 13-19). Whether a node will be split is determined by the number of slots in use: if a node is currently full, the insertion will split the node, and in turn its parent node will be affected and must be locked as well. This is iteratively applied until a non-splitting node is reached. If all affected nodes are contained within the NMP-managed portion, the insertion takes place immediately, following the single-threaded B+ tree insert algorithm, and the path is unlocked (lines 25-29).

```
1  Node* path[TREE_HEIGHT]; int local_seqnum[TREE_HEIGHT];
2  Node* curr, child = NULL; int curr_level = 0;
3  bool retry_from_root = true;
4  while (retry_from_root) {
5    curr = root; curr_level = curr->level;
6    local_seqnum[curr_level] = curr->seqnum;
7    path[curr_level] = curr;
8    if (local_seqnum[curr_level] % 2 != 0) continue;
9    while (curr_level < TREE_HEIGHT) {
10     child = find_child(curr, key);
11     if (curr_level > LAST_HOST_LEVEL) {
12       do {
13         local_seqnum[curr_level-1] = child->seqnum;
14       } while (local_seqnum[curr_level-1] % 2 != 0);
15       path[curr_level-1] = child;
16       if (curr->seqnum == local_seqnum[curr_level]) {
17         curr_level--;
18         curr = child;
19       } else {
20         // curr has been modified, move back up the path
21         curr_level++;
22         curr = path[curr_level]; }
23     } else { // reached last host-side level
24       SEND_NMP_INSERT_OP(); // according to Section 3.2
25       if (GET_NMP_RETRY_FLAG()) break; // retry from root
26       if (GET_NMP_LOCK_PATH()) { // lock nodes on path
27         locked_all = false;
28         for (i = LAST_HOST_LEVEL; i < TREE_HEIGHT; i++) {
29           if (!CAS(path[i]->seqnum, local_seqnum[i],
       local_seqnum[i]+1)) {
30             // failed to lock node at level i:
31             // unlock locked nodes on path (code omitted)
32             SEND_NMP_UNLOCK_PATH_OP();
33             break; } // retry insert from root
34           if (path[i]->slotuse < INNER_SLOTMAX) {
35             locked_all = true; break; } }
36         if (locked_all) {
37           SEND_NMP_RESUME_INSERT_OP();
38           retry_from_root = false;
39           // RESUME_INSERT is guaranteed to succeed.
40           // From here, complete host-side insertion process
41           // following the single-threaded insert algorithm,
42           // unlock all nodes on path and return (code omitted)
43         }
44       } else { // of if GET_NMP_LOCK_PATH
45         retry_from_root = false;
46         // insert completed within NMP partition. return.
47       } break; } } }
```

Listing 4: Hybrid B+ tree host-side `insert` operation.

```
1  Node *path[TOP_NMP_LEVEL+1];
2  Node *curr = publist[i].nmp_ptr;
3  if (curr->parent_seqnum > publist[i].parent_seqnum) {
4    // curr node has been split by concurrent op.
5    // set RETRY flag and move on to next op. (code omitted)
6  } else if (curr->parent_seqnum < publist[i].parent_seqnum) {
7    // parent had been split by insertion in sibling node
8    curr->parent_seqnum = publist[i].parent_seqnum; }
9  while (curr->level > 0) {
10   path[curr->level] = curr;
11   curr = find_child(curr, publist[i].key); }
12 path[0] = curr;
13 locked_all = false;
14 for (i = 0; i < TOP_NMP_LEVEL; i++) {
15   if (path[i]->lock == 0) {
16     path[i]->lock = 1;
17     if (path[i]->slotuse < SLOTMAX) {
18       locked_all = true;
19       break; }
20   } else {
21     // path[i] has already been locked by a concurrent insert.
22     // unlock nodes on path locked by this operation,
23     // set RETRY flag, and move on to next op (code omitted)
24   } }
25 if (locked_all) {
26   // 1.follow single-threaded insert algorithm to complete
27   // the insertion, 2. unlock locked nodes, and 3. increment
28   // the parent_seqnum of begin-NMP-traversal node and its
29   // split-off sibling. (code omitted)
30 } else {
31   // set LOCK_PATH flag and move on to next op (code omitted)
32 }
```

Listing 5: Hybrid B+ tree NMP-side `insert` operation.

However, if even the topmost level node on the NMP-side path requires a node split, the NMP core notifies the host thread (via the publication list slot) to lock nodes on the host-side path accordingly (lines 30-32). In this case, the NMP-side nodes remain locked, in order to prevent concurrent `insert` or `remove` operations from modifying any of the affected nodes in the meanwhile (lines 20-24).

If the host thread receives a `LOCK_PATH` command from the NMP core, it proceeds to lock (*i.e.,* increment the sequence numbers of) affected nodes on the host-side path, based on the traversal path and sequence numbers that it recorded during the initial traversal down the tree (Listing 4: lines 26-35). Once all affected nodes are locked, the host thread offloads a `RESUME_INSERT` operation to the NMP core (line 37). However, if the locking fails at any point (line 29), the host thread unlocks all acquired locks on the host side (lines 30-31), notifies the NMP core to unlock the path accordingly (line 32), and then retries the operation from the beginning (line 33).

Once the NMP core receives the `RESUME_INSERT` operation, the NMP core completes the insertion and unlocks the NMP-side path, for the insertion is guaranteed to succeed at this point. Before notifying the host thread of the operation completion, the NMP core increments the parent sequence number of the topmost level nodes on the operation path (including the newly split-off node) to reflect the eventual unlocked sequence number of their parent nodes.[3] After the NMP-side process completes, the host thread completes its portion of the insertion process and unlocks the affected nodes (*i.e.*, increments their sequence numbers).

The parent sequence number recorded in the begin-NMP-traversal node (hereby referred to as *recorded parent#*) is used for synchronization at the host-NMP boundary. When a host thread offloads an operation to the NMP core, it writes the sequence number of the last host-side node (*i.e.*, parent of the begin-NMP-traversal node) to the publication list slot, in addition to other operation information mentioned in §3.2. Before the NMP-side tree traversal begins, the NMP core compares the offloaded parent sequence number (referred to as *offloaded parent#*) against the recorded parent# (Listing 5: lines 2-8). If the recorded parent# is greater than the offloaded parent#, this indicates that the host-side parent node has been modified after the operation offload, due to a node split in the begin-NMP-traversal node. This implies that the corresponding leaf node for the operation may have become unreachable from the begin-NMP-traversal node, so the NMP core notifies the host thread to retry this operation. However, there could also be cases where the offloaded parent# is greater than the recorded parent#. This indicates that the host-side parent node had been modified due to a split in a sibling node; in this case, the recorded parent# is simply updated to the offload parent# for consistency.

Synchronizing `read`, `update`, and `remove` operations is relatively straightforward. These operations follow the `insert` operation's initial host-side and NMP-side traversal process (Listing 4: lines 4-22, Listing 5: lines 2-11). Once the NMP core reaches the leaf node, `read` or `update` operations can be applied immediately. However, the `remove` operation cannot be applied if the leaf node is in a locked state, for the removal affects the number of slots in use at the node,

[3]When a node splits, the split-off node replicates the sequence number (in NMP-side nodes, the parent sequence number) of the original node. This ensures sequence number consistency between the host-side parent and the NMP-side children, even after node splits.

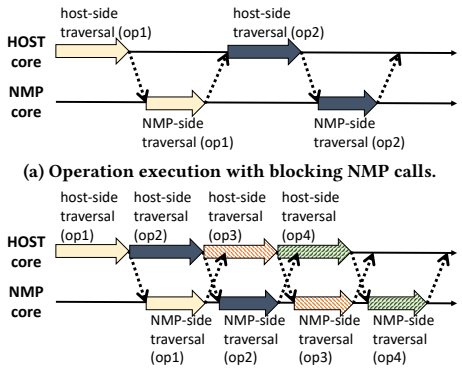


Figure 4: The throughput of data structure operations when NMP calls are (a) blocking and (b) non-blocking.

which in turn affects the node split process being prepared by a concurrent insertion. In this case, the `remove` operation is aborted and retried from the beginning.

3.5 Optimization: Non-blocking NMP Calls

In our proposed hybrid data structures, once a host thread executing a data structure operation reaches the bottom level of the host-managed portion, it sends the operation request to an appropriate NMP core. These NMP calls are assumed to be *blocking* operations; that is, the host thread actively waits until the NMP-side of the operation is done. This poorly utilizes compute resources and limits throughput, as shown in Figure 4a.

In order to increase concurrency and overall throughput, we modify the NMP calls to be *non-blocking*. With non-blocking NMP calls, a host thread can offload an operation to an NMP core and immediately move on to process the next pending operation while the NMP core does its portion of the work (Figure 4b). Host threads can maintain a list of ongoing operations to later retrieve results returned by NMP cores.

Data structure operations with non-blocking NMP calls involve only minor changes to the high-level API. In typical data structure APIs, a high-level function call simply returns a Boolean flag indicating the success of an operation. With non-blocking NMP calls, the function call should instead return an operation ID number as an acknowledgment that the operation is now in progress. The API should also be extended with a separate function that takes the operation ID as input in order to check on the operation's status and retrieve any return values.

4 EVALUATION METHODOLOGY

Because NMP-capable memory devices are not readily available yet, we rely on architecture simulations to evaluate our proposed hybrid data structures. We use the gem5-based [10] SMCSim simulator [6, 14], which is a cycle-accurate, full-system NMP architecture simulator that was used for prior NMP-based concurrent data structure evaluations [16].

Table 1 outlines the simulator configurations. Cycle-accurate architecture simulations by nature result in long execution times, especially as more architecture components (*e.g.*, CPUs, caches, memory devices) are added to the simulation. Simulating even one second of real execution time may take more than 24 hours,

Table 1: Evaluation framework configuration.

Host Configuration	
Host cores	8 out-of-order processors (ARMv7 Cortex-A15) 2GHz frequency, 1 thread/core
L1 cache	32kB icache, 64kB dcache, private 2-way set-associative LRU 2-cycle latency, 128B/block
L2 cache	1MB, shared, 8-way set associative LRU 20-cycle latency, 128B/block
Memory Configuration	
1 HMC w/ 16 memory vaults total (8 main memory vaults, 8 NMP vaults) 128MB/vault (total 2GB), 8 DRAM banks/vault t_{RP}: 13.75ns, t_{RCD}: 13.75ns, t_{CL}: 13.75ns, t_{BURST}: 3.2ns	
NMP Core Configuration	
NMP cores	1 in-order single-cycle processor/vault (gem5 TimingSimpleCPU), 2GHz frequency 128B buffer in memory controller
scratchpad memory	40kB/NMP core, 8kB reserved for host memory-map, stores instructions and program stack.

depending on the configuration. To run experiments in a reasonable time frame, the simulator was configured with a relatively small memory size (1GB host-accessible main memory, 1GB NMP-capable memory) and 8 host and NMP cores each. Only two levels of cache were used in order to make cache sizes proportional to the memory size. Using the relatively small baseline system, we show that our hybrid data structures can improve performance, and we project that the improvements will carry through to larger systems.

For results and analysis in this paper, we assess data structure performance in terms of *operation throughput*, which is the number of data structure operations completed across all available threads in a given period of time. For a more complete analysis (including energy consumption), please see [15].

5 RESULTS & ANALYSIS

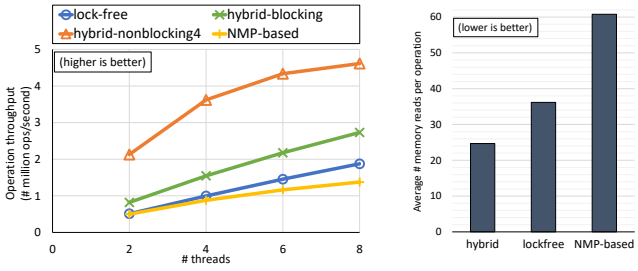
5.1 Baseline Evaluation

For our baseline analysis, we used a predefined core workload from the *Yahoo! Cloud Serving Benchmark* (*YCSB*) [19] framework, which provides (*i*) keys to populate the data structure and (*ii*) operations to perform, both based on distributions that appear in realistic cloud OLTP applications. The core workload we used is YCSB-C, a read-only workload with a zipfian distribution in accessed keys.

Skiplists. We compare the operation throughputs of hybrid skiplists with blocking and non-blocking NMP calls (*hybrid-blocking* and *hybrid-nonblocking*, respectively) against the *NMP-based* skiplist of prior work [16] and the *lock-free* skiplist [23, 29] as a non-NMP reference. In particular, *hybrid-nonblocking4* denotes that we allowed each host thread to have up to 4 NMP calls in-flight at a time.

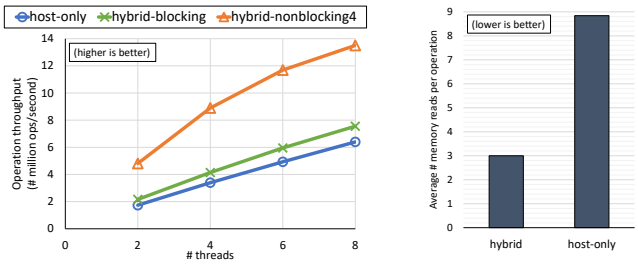
We initialized each skiplist with 2^{22} key-value pairs (generated by the YCSB framework according to core workload characteristics), which results in a skiplist of size 0.5GB with 22 total levels. We configured the hybrid skiplists to hold the top 13 levels in the host-managed portion, which makes its size roughly equivalent to the last-level cache (LLC) size.

Figure 5a shows the operation throughput of various skiplist implementations with YCSB-C. We plot the throughput across varying numbers of host threads to show scalability with increasing numbers of threads. Although the skiplist is significantly larger than the LLC size (512×), *lock-free* still shows higher operation throughput than *NMP-based*, which is similar to the results seen in prior work [16]. However, **our *hybrid-blocking* skiplist increases the throughput by 99% over *NMP-based* and by 46% over *lock-free* at 8 concurrent threads.**



(a) operation throughput (b) memory reads

Figure 5: Skiplist baseline evaluation with YCSB-C.



(a) operation throughput (b) memory reads

Figure 6: B+ tree baseline evaluation with YCSB-C.

The performance improvement with *hybrid* skiplists comes from significantly reducing the number of DRAM reads. As shown in Figure 5b, *hybrid* skiplists incur only two-thirds the amount of DRAM reads incurred by the *lock-free* skiplist, and only 40% the amount incurred by the *NMP-based*.

The performance improvement with *hybrid* skiplists becomes even more substantial with non-blocking NMP calls, which hide NMP operation offload costs and reduce the idle time of both host and NMP cores. (Note that the average number of memory reads remains the same across *hybrid-blocking* and *hybrid-nonblocking*.) **At 8 concurrent threads, *hybrid-nonblocking4* shows 2.46× the throughput of *lock-free*.**

B+ trees. We compare the performance of our *hybrid* B+ trees with blocking and non-blocking NMP calls (*hybrid-blocking* and *hybrid-nonblocking*, respectively) against a non-NMP, *host-only* B+ tree. Like the host-managed portion of the hybrid B+ tree, the host-only B+ tree uses sequence locks for concurrency.

We configured the B+ tree so that each node is 128 bytes, which is a typical B+ tree node size for in-memory OLTP systems [54] where B+ trees are commonly used. In our implementation, the 128-byte node size allowed each leaf node to hold up to 14 key-value pairs and each non-leaf node to hold up to 15 children. We populated each B+ tree with approximately 30M key-value pairs and inserted the items in sorted order. This generated a balanced initial B+ tree of 9 total levels and approximately 0.57GB total size. The top 6 levels (summing up to 1.14MB in size) were placed in the host-managed portion for the *hybrid* implementations.

Figure 6a shows that **the *hybrid-blocking* B+ tree yields 18% higher throughput than *host-only*.** While 18% increase is not insignificant, the improvement is relatively small compared to the hybrid skiplists, where *hybrid-blocking* yielded nearly 50% higher throughput than *lock-free* (Figure 5a).

Table 2: Host-NMP communication costs.

Event	Delay
Operation valid flag written from host reaching NMP core	65–80ns
Operation completion flag written from NMP core reaching host	88–96ns
Last-level cache miss	100–120ns

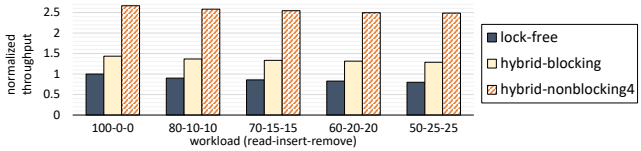


Figure 7: Skiplist sensitivity evaluation: normalized operation throughput (lock-free 100-0-0 throughput as baseline).

The reason for the relatively small increase in operation throughput is the small number of memory reads needed for each operation relative to the cost of offloading operation requests to NMP cores. Figure 6b shows the average number of DRAM reads per operation. The *host-only* B+ tree makes approximately 9 DRAM reads per operation, and the *hybrid* B+ trees make 3 DRAM reads. While the reduction in the number of memory reads is nearly 3×, the absolute number of reads is still small, particularly compared to the skiplists, where *lock-free* and *hybrid* made 36 and 24 memory reads per operation, respectively.

On a related note, the delays in offloading operation requests to NMP cores are shown in Table 2. These delays were measured across multiple iterations of a single operation offload request in otherwise same settings as the baseline B+ tree evaluation (*i.e.*, same initial B+ tree size, same number of host-side levels, and same architectural configurations). Note that the delays just for communicating operation information to and from the NMP core sum up to be comparable to 1-2 LLC miss delays. When the data structure operation incurs only a few memory reads, this overhead can dominate the performance of *hybrid-blocking* data structure implementations. However, nonblocking NMP calls effectively hide this overhead and significantly improve the *hybrid* B+ tree's performance. **At 8 concurrent threads, *hybrid-nonblocking4* yields 2.11× the throughput of *host-only*.** Moreover, larger, taller B+ trees will have longer node traversals, in which case the *hybrid* implementations will be more advantageous.

5.2 Sensitivity Analysis

We also provide a sensitivity analysis to show how the hybrid data structures scale with concurrent modifications. We generated our own workloads with varying ratios of insertions and removals and uniform distribution of accessed keys. In Figures 7 and 8, *X-Y-Z* of the x-axis denotes the ratio of read-insert-remove operations in the workload. The initial configurations of the skiplists and the B+ trees remain the same as our baseline evaluation experiments.

In each experiment, we fixed the number of concurrent host threads to 8 (the maximum number of host threads available in the simulated system). Also, for faster simulation, the experiments were executed with in-order host cores. Note that the general trends of the results are still the same as using out-of-order host cores, since memory access (not CPU cycles) is the main performance-limiting factor in these data structures.

Skiplists. Figure 7 shows the operation throughput of the various skiplist implementations across the different workloads, normalized against the throughput of the *lock-free* skiplist with the 100-0-0 workload. Although increasing concurrent modifications reduces the throughput across all skiplist implementations, it has relatively smaller impact on the *hybrid* skiplists. With the 50-25-25 workload, *lock-free*, *hybrid-blocking* and *hybrid-nonblocking4*'s throughputs are 80%, 90%, and 93% of their throughputs with the 100-0-0 workload, respectively. In other words, **the *hybrid* skiplists have relative advantage with more concurrent modifications.** At 50-25-25, *hybrid-blocking* and *hybrid-nonblocking4* each have 1.61× and 3.12× the throughput of *lock-free*.

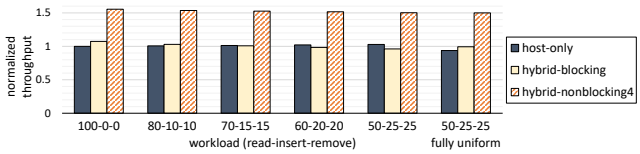


Figure 8: B+ tree sensitivity evaluation: normalized operation throughput (host-only 100-0-0 throughput as baseline).

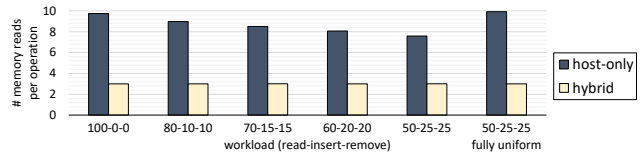


Figure 9: B+ tree sensitivity evaluation: average number of memory reads per operation.

B+ trees. The workloads for the B+ tree sensitivity analysis were generated with the following additional characteristics. While the keys accessed for `read` and `remove` operations were uniformly distributed across the entire key space, the `insert` keys were chosen so that insertions in the hybrid B+ trees would happen at the last leaf node (in terms of incrementing keys) of each NMP partition. This was in order to forcefully incur maximum possible node splits, while still evenly distributing the insertions across the NMP partitions.[4] To verify how the node splits impact performance, we also used a *50-25-25 fully uniform* workload, in which even the insertions were uniformly distributed across all leaf nodes. This workload did not incur any node splits, despite the high ratio of insert operations.

Figure 8 shows the operation throughput of the various B+ tree implementations, normalized against the throughput of the *host-only* B+ tree with the 100-0-0 workload. First, we note that the throughput of *hybrid-blocking* decreases slightly with increasing amounts of modifications and node splits. With the 50-25-25 workload, *hybrid-blocking* yields 10% less throughput than the same implementation with the 100-0-0 workload; with the 50-25-25 fully uniform workload that does not incur node splits, the throughput is 7.5% less. Even so, ***hybrid-blocking* shows performance comparable to *host-only* across all workloads.** With the 50-25-25 workload, where *hybrid-blocking* is least advantageous, *hybrid-blocking* still yields 93.5% of the throughput of *host-only*.

On the other hand, the performance of *host-only* slightly increases by up to 2.7% with more modifications. This slight improvement in performance is due to better cache locality. Because the insertions in these workloads are targeted at specific leaf nodes to incur maximum node splits, nodes on the insertion target paths

[4] If operations are heavily targeted to a single NMP partition, operations will serialize in the specific partition and become the performance bottleneck. This is a limitation of the hybrid data structures that could be addressed in future work.

are likely to be accessed from the on-chip cache. Figure 9 shows that this is indeed true: with the *host-only* B+ tree, the number of memory reads per operation decreases with increasing insertion ratios. The 50-25-25 fully uniform workload eliminates the cache locality benefits, and in this case, *host-only*'s operation throughput decreases by 6% compared to the read-only workload. In fact, this workload causes *hybrid-blocking* to have a relative advantage (6% higher throughput) over *host-only*.

Lastly, we note that **regardless of the workload, *hybrid-nonblocking4* yields approximately 50% higher throughput than *host-only***. More specifically, the improvements in throughput range from 46%, with the 50-25-25 workload where *hybrid* is least advantageous, to 60%, with the 50-25-25 fully uniform workload where *hybrid* is most advantageous.

6 RELATED WORK

Data structures with NMP. Liu *et al.* [44] proposed NMP-based concurrent data structure algorithms, which Choe *et al.* [16] extensively evaluated using a full-system architecture simulator. Other prior work [32, 34, 45, 56] designed near-memory hardware accelerators for pointer-chasing data structures. However, none of these works considered the on-chip cache locality of the data structures, since pointer chasing occurs in the near-memory compute units.

Kang *et al.* [38] presented an NMP-based skiplist that provides better load-balancing among NMP modules. Their algorithm also divided the skiplist into upper-level and lower-level nodes, but the upper-level nodes were replicated in each NMP partition, unlike our work in which they are placed in the on-chip cache.

Graph processing applications with NMP [3, 20, 48, 71, 72] may seem similar to data structures in their pointer-chasing aspect. However, they are fundamentally different applications with different data traversal patterns and data partitioning strategies; thus, the optimizations are not transferable.

Locality-aware NMP work offloading. Some works suggested locality-aware offloading of NMP operations at instruction granularity, either through programmer input and runtime detection [4] or compile-time techniques [27]. Tsai *et al.* [61] dynamically mapped threads to host processors or NMP cores based on per-thread cache miss rates and cache resource contention. Livia [46] goes a step further and schedules tasks (defined by a programming model) to compute units placed throughout the entire memory hierarchy, including at each cache level. Kandemir *et al.* [37] also presented compiler-based techniques to optimally co-locate data and computation throughout the memory hierarchy.

Other works [33, 51] focused on GPU-based NMP architectures and devised techniques to divide data and work between main GPU cores and near-memory GPU cores. To the best of our knowledge, our work is the first to take an algorithm-based approach with data structures to provide locality- and architecture-aware data placement and work division with NMP.

Latency-aware data structures. Other data structure designs have also treated particular levels of hierarchical data structures differently to improve performance. NUMASK [21] constructed different index layers (upper levels) for each NUMA zone, which allowed index traversals to access NUMA-local data only. DLTree [17] exploited the fact that higher levels of a tree change much less frequently than lower levels to compact the upper levels together for better cache locality. FPTree [50] is a persistent B+ tree that persists only the bottommost level containing the actual data in non-volatile memory and maintains the upper levels in faster DRAM. To the best of our knowledge, our work is the first to exploit the data structure hierarchy in the context of NMP architectures.

There are also many other cache-conscious data structure designs (*e.g.*, [8, 12, 42, 47, 55, 62]), but these mostly attempt to reduce cache invalidation traffic or improve data layout for better spatial locality. Our work focuses on adapting hierarchical data structures to NMP architectures in a way that enhances their cache locality.

Some recent concurrent data structure designs leveraged synchronization techniques similar to the ones used for our NMP-hybrid data structures. Node Replication for NUMA-aware data structures [13] used flat-combining to batch operations and reduce synchronization overhead across NUMA nodes. MEDS [63] separated data into elastic partitions based on ranges of keys, and the data structure maintained separate layers to navigate operations to partitions and synchronize operations within each partition.

7 CONCLUSION & FUTURE WORK

This paper proposed a new, *hybrid* approach to using NMP architectures to improve the performance of cache-conscious concurrent data structures. Specifically, we focused on *hierarchical* data structures where cache locality is high when traversing higher-level nodes but deteriorates as traversals descend into lower levels. Our hybrid data structures place higher-level nodes in the host's on-chip cache and lower-level nodes in NMP-capable memory, rendering the cache more effective. NMP calls can also be non-blocking in hybrid data structures, enabling even greater concurrency. Our empirical evaluation showed that hybrid skiplists and hybrid B+ trees can yield significant performance gains compared to state-of-the-art implementations on conventional architectures.

An important next step is to implement and evaluate the proposed hybrid data structures on real NMP hardware. Although not yet commercialized as an off-the-shelf product, UPMEM has recently made its NMP-capable memory device [22] available to interested users [25, 26]. Another potential avenue for future work involves extending hybrid data structures to work with more sophisticated NMP architectures, such as when host and NMP cores share the memory space. Moreover, since hybrid data structures focus on reducing cache miss latencies rather than exploiting high internal memory bandwidth offered by certain NMP-enabling technologies, our algorithmic optimizations inspired by NMP can also be complemented with non-NMP accelerators that support generic data structures (*e.g.*, QEI [68]).

Lastly, one noteworthy limitation of our current approach arises with highly skewed workloads. In such cases, frequently accessed nodes of traditional data structures are likely to remain in the on-chip cache, resulting in better performance than our hybrid versions that always force lower-level nodes to remain in NMP-managed memory. One possible solution is to enhance hybrid data structures with self-adjusting algorithms (*e.g.*, biased skiplists [7], splay-lists [5], CBTree [1]) to dynamically push frequently accessed nodes to the host-managed region. Nonetheless, our work has laid the foundations for many promising opportunities for future work, and we look forward to seeing follow-up works that further enhance the effectiveness of NMP-hybrid data structures.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. This work was supported by NSF grant 1909715.

REFERENCES

[1] Yehuda Afek, Haim Kaplan, Boris Korenfeld, Adam Morrison, and Robert E. Tarjan. 2012. CBTree: A Practical Concurrent Self-Adjusting Search Tree. In *Distributed Computing*, Marcos K. Aguilera (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–15.

[2] H. Ahmed, P. C. Santos, J. P. C. Lima, R. F. Moura, M. A. Z. Alves, A. C. S. Beck, and L. Carro. 2019. A Compiler for Automatic Selection of Suitable Processing-in-Memory Instructions. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 564–569. https://doi.org/10.23919/DATE.2019.8714956

[3] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[4] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-Enabled Instructions: A Low-Overhead, Locality-Aware Processing-in-Memory Architecture. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 336–348. https://doi.org/10.1145/2749469.2750385

[5] Vitaly Aksenov, Dan Alistarh, Alexandra Drozdova, and Amirkeivan Mohtashami. 2020. The Splay-List: A Distribution-Adaptive Concurrent Skip-List. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference (LIPIcs, Vol. 179)*, Hagit Attiya (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:18. https://doi.org/10.4230/LIPIcs.DISC.2020.3

[6] 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.

[7] Amitabha Bagchi, Adam L Buchsbaum, and Michael T Goodrich. 2005. Biased skip lists. *Algorithmica* 42, 1 (2005), 31–48.

[8] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2020. KiWi: A Key-Value Map for Scalable Real-Time Analytics. 7, 3, Article 16 (jun 2020), 28 pages. https://doi.org/10.1145/3399718

[9] R. Bayer and E. M. Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.* 1, 3 (Sept. 1972), 173–189. https://doi.org/10.1007/BF00288683

[10] 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.

[11] Hans-J. Boehm. 2012. Can Seqlocks Get along with Programming Language Memory Models?. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (Beijing, China) *(MSPC '12)*. Association for Computing Machinery, New York, NY, USA, 12–20. https://doi.org/10.1145/2247684.2247688

[12] Anastasia Braginsky and Erez Petrank. 2011. Locality-Conscious Lock-Free Linked Lists. In *Distributed Computing and Networking*, Marcos K. Aguilera, Haifeng Yu, Nitin H. Vaidya, Vikram Srinivasan, and Romit Roy Choudhury (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–118.

[13] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-Box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[14] Jiwon Choe. 2019. Brown-SMCSim. https://github.com/jiwon-choe/Brown-SMCSim.

[15] Jiwon Choe. 2022. *Concurrent Data Structures with Near-Memory Processing: Software-Hardware Co-Design*. Ph. D. Dissertation. Brown University.

[16] 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 *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (Phoenix, AZ, USA) *(SPAA '19)*. ACM, New York, NY, USA, 297–308. https://doi.org/10.1145/3323165.3323191

[17] Nachshon Cohen, Arie Tal, and Erez Petrank. 2017. Layout Lock: A Scalable Locking Paradigm for Concurrent Data Layout Modifications. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Austin, Texas, USA) *(PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 17–29. https://doi.org/10.1145/3018743.3018753

[18] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (June 1979), 121–137. https://doi.org/10.1145/356770.356776

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[20] G. Dai, T. Huang, Y. Chi, J. Zhao, G. Sun, Y. Liu, Y. Wang, Y. Xie, and H. Yang. 2019. GraphH: A Processing-in-Memory Architecture for Large-Scale Graph Processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 4 (2019), 640–653. https://doi.org/10.1109/TCAD.2018.2821565

[21] Henry Daly, Ahmed Hassan, Michael F Spear, and Roberto Palmieri. 2018. NUMASK: high performance scalable skip list for NUMA. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[22] F. Devaux. 2019. The true Processing In Memory accelerator. In *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–24. https://doi.org/10.1109/HOTCHIPS.2019.8875680

[23] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[24] 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

[25] Christina Giannoula, Ivan Fernandez, Juan Gómez-Luna, Nectarios Koziris, Georgios Goumas, and Onur Mutlu. 2022. SparseP: Towards Efficient Sparse Matrix Vector Multiplication on Real Processing-In-Memory Architectures. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 21 (2022). https://doi.org/10.1145/3508041

[26] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2021. Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture. arXiv:2105.03814 [cs.AR]

[27] Ramyad Hadidi, Lifeng Nai, Hyojong Kim, and Hyesoon Kim. 2017. CAIRO: A Compiler-Assisted Technique for Enabling Instruction-Level Offloading of Processing-In-Memory. *ACM Trans. Archit. Code Optim.* 14, 4, Article 48 (Dec. 2017), 25 pages. https://doi.org/10.1145/3155287

[28] 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* (Thira, Santorini, Greece) *(SPAA '10)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

[29] M Herlihy, Y Lev, and N Shavit. 2007. A lock-free concurrent skiplist with wait-free search. *Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts* 32 (2007), 35.

[30] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2020. *The art of multiprocessor programming*. Newnes.

[31] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[32] 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* (Haifa, Israel) *(PACT '16)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/2967938.2967958

[33] Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W. Keckler. 2016. Transparent Offloading and Mapping (TOM): Enabling Programmer-Transparent near-Data Processing in GPU Systems. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 204–216. https://doi.org/10.1109/ISCA.2016.27

[34] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu. 2016. Accelerating pointer chasing in 3D-stacked memory: Challenges, mechanisms, evaluation. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. 25–32. https://doi.org/10.1109/ICCD.2016.7753257

[35] Jaeyoung Jang, Jun Heo, Yejin Lee, Jaeyeon Won, Seonghak Kim, Sung Jun Jung, Hakbeom Jang, Tae Jun Ham, and Jae W. Lee. 2019. Charon: Specialized Near-Memory Processing Architecture for Clearing Dead Objects in Memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 726–739. https://doi.org/10.1145/3352460.3358297

[36] Theodore Johnson and Dennis Shasha. 1993. B-trees with inserts and deletes: Why free-at-empty is better than merge-at-half. *J. Comput. System Sci.* 47, 1 (1993), 45–76. https://doi.org/10.1016/0022-0000(93)90020-W

[37] Mahmut Taylan Kandemir, Jihyun Ryoo, Xulong Tang, and Mustafa Karakoy. 2021. Compiler Support for near Data Computing. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 90–104. https://doi.org/10.1145/3437801.3441600

[38] Hongbo Kang, Phillip B. Gibbons, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, and Charles McGuffey. 2021. The Processing-in-Memory Model. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. Association

for Computing Machinery, New York, NY, USA, 295–306. https://doi.org/10.1145/3409964.3461816

[39] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. 2020. RecNMP: Accelerating Personalized Recommendation with near-Memory Processing. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE Press, 790–803. https://doi.org/10.1109/ISCA45697.2020.00070

[40] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. TensorDIMM: A Practical Near-Memory Processing Architecture for Embeddings and Tensor Operations in Deep Learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 740–753. https://doi.org/10.1145/3352460.3358284

[41] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *Proceedings of the 48th International Symposium on Computer Architecture (ISCA '21)*. Association for Computing Machinery, New York, NY, USA.

[42] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 302–313. https://doi.org/10.1109/ICDE.2013.6544834

[43] Jie Li, Xi Wang, Antonino Tumeo, Brody Williams, John D. Leidel, and Yong Chen. 2019. PIMS: A Lightweight Processing-in-Memory Accelerator for Stencil Computations. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia, USA) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 41–52. https://doi.org/10.1145/3357526.3357550

[44] 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* (Washington, DC, USA) *(SPAA '17)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/3087556.3087582

[45] Scott Lloyd and Maya Gokhale. 2017. Near Memory Key/Value Lookup Acceleration. In *Proceedings of the International Symposium on Memory Systems* (Alexandria, Virginia) *(MEMSYS '17)*. Association for Computing Machinery, New York, NY, USA, 26–33. https://doi.org/10.1145/3132402.3132434

[46] Elliot Lockerman, Axel Feldmann, Mohammad Bakhshalipour, Alexandru Stanescu, Shashwat Gupta, Daniel Sanchez, and Nathan Beckmann. 2020. Livia: Data-Centric Computing Throughout the Memory Hierarchy. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 417–433. https://doi.org/10.1145/3373376.3378497

[47] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/2168836.2168855

[48] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. 2017. GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 457–468. https://doi.org/10.1109/HPCA.2017.54

[49] Michael A. Olson, Keith Bostic, and Margo Seltzer. 1999. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Monterey, California) *(ATEC '99)*. USENIX Association, USA.

[50] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

[51] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. 2016. Scheduling techniques for GPU architectures with processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 31–44. https://doi.org/10.1145/2967938.2967940

[52] 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

[53] 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 *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*. IEEE Computer Society, 190–200. https://doi.org/10.1109/ISPASS.2014.6844483

[54] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 78–89.

[55] Matthew Rodriguez, Ahmed Hassan, and Michael Spear. 2021. Exploiting Locality in Scalable Ordered Maps. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 998–1008. https://doi.org/10.1109/ICDCS51616.2021.00099

[56] P. C. Santos, G. F. Oliveira, J. P. Lima, M. A. Z. Alves, L. Carro, and A. C. S. Beck. 2018. Processing in 3D memories to speed up operations on complex data structures. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 897–900. https://doi.org/10.23919/DATE.2018.8342135

[57] Siddhartha Sen and Robert E. Tarjan. 2014. Deletion without Rebalancing in Multiway Search Trees. *ACM Trans. Database Syst.* 39, 1, Article 8 (jan 2014), 14 pages. https://doi.org/10.1145/2540068

[58] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2020. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company. https://www.db-book.com/db7/index.html

[59] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. 2016. Micro-Architectural Analysis of In-Memory OLTP. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 387–402. https://doi.org/10.1145/2882903.2882916

[60] Utku Sirin, Pınar Tözün, Danica Porobic, Ahmad Yasin, and Anastasia Ailamaki. 2021. Micro-architectural analysis of in-memory OLTP: Revisited. *The VLDB Journal* (2021), 1–25.

[61] Po-An Tsai, Changping Chen, and Daniel Sanchez. 2018. Adaptive Scheduling for Systems with Asymmetric Memory Hierarchies. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture* (Fukuoka, Japan) *(MICRO-51)*. IEEE Press, 641–654. https://doi.org/10.1109/MICRO.2018.00058

[62] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 473–488. https://doi.org/10.1145/3183713.3196895

[63] Kevin Williams, Joe Foster, Athicha Srivirote, Ahmed Hassan, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. On Building Modular and Elastic Data Structures with Bulk Operations. In *International Conference on Distributed Computing and Networking 2021* (Nara, Japan) *(ICDCN '21)*. Association for Computing Machinery, New York, NY, USA, 237–238. https://doi.org/10.1145/3427796.3433932

[64] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News* 23, 1 (March 1995), 20–24. https://doi.org/10.1145/216585.216588

[65] Y. Xiao, S. Nazarian, and P. Bogdan. 2018. Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1387–1392. https://doi.org/10.23919/DATE.2018.8342229

[66] S. F. Yitbarek, T. Yang, R. Das, and T. Austin. 2016. Exploring specialized near-memory processing for data intensive operations. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1449–1452.

[67] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[68] Yifan Yuan, Yipeng Wang, Ren Wang, Rangeen Basu Roy Chowhury, Charlie Tai, and Nam Sung Kim. 2021. QEI: Query Acceleration Can be Generic and Efficient in the Cloud. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 385–398. https://doi.org/10.1109/HPCA51647.2021.00040

[69] Bin Zhang and Meichun Hsu. 1989. Unsafe operations in B-trees. *Acta Informatica* 26 (1989), 421–438.

[70] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1567–1581. https://doi.org/10.1145/2882903.2915222

[71] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian. 2018. GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 544–557. https://doi.org/10.1109/HPCA.2018.00053

[72] Youwei Zhuo, Chao Wang, Mingxing Zhang, Rui Wang, Dimin Niu, Yanzhi Wang, and Xuehai Qian. 2019. GraphQ: Scalable PIM-Based Graph Processing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 712–725. https://doi.org/10.1145/3352460.3358256