Brief Announcement: BatchBoost: Universal Batching for Concurrent Data Structures

Vitaly Aksenov
City, University of London, UK
ITMO University, Russia

Michael Anoprenko
Telecom SudParis, France

Alexander Fedorov
IST Austria, Austria

Michael Spear
Lehigh University, USA

Abstract

Batching is a technique that stores multiple keys/values in each node of a data structure. In sequential search data structures, batching reduces latency by reducing the number of cache misses and shortening the chain of pointers to dereference. Applying batching to concurrent data structures is challenging, because it is difficult to maintain the search property and keep contention low in the presence of batching.

In this paper, we present a general methodology for leveraging batching in concurrent search data structures, called BatchBoost. BatchBoost builds a search data structure from distinct "data" and "index" layers. The data layer's purpose is to store a batch of key/value pairs in each of its nodes. The index layer uses an unmodified concurrent search data structure to route operations to a position in the data layer that is "close" to where the corresponding key should exist. The requirements on the index and data layers are low: with minimal effort, we were able to compose three highly scalable concurrent search data structures based on three original data structures as the index layers with a batched version of the Lazy List as the data layer. The resulting BatchBoost data structures provide significant performance improvements over their original counterparts.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases Concurrency, Synchronization, Locality

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.12

Category Brief Announcement

37th International Symposium on Distributed Computing (DISC 2023).
Editor: Rotem Oshman; Article No. 12; pp. 12:1–12:7
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Motivation and Background

Batching is an increasingly important technique for maximizing the performance of concurrent data structures. Briefly, batching is the technique by which a linked data structure stores multiple elements in a single data node. The most well-known batched data structure is the B-tree [4], but batching has been applied to a variety of trees [17, 23], lists [5], and skip lists [3, 5]. The benefit of batching is that it co-locates multiple elements in a contiguous region of memory (e.g., a cache line). While batching typically does not improve asymptotic guarantees, it can reduce the total number of cache lines accessed by an operation.

The latency reductions that stem from batching are broadly beneficial. In data structures that provide `scan` operations and `range` queries [2,3,8,12,24], batching coarsens the granularity of synchronization metadata, so that it can be accessed less frequently. In data structures that use remote direct memory access (RDMA), Non-Uniform Memory Access (NUMA), or non-volatile byte-addressable memory (NVM), batching reduces the number of accesses to a memory that is slower than local DRAM. Batching can also benefit algorithms for GPUs [16] and emerging near-memory computing paradigms [11], where careful consideration of data placement is paramount.

Batching is not without its downsides, for both sequential and concurrent programs. For example, consider an ordered map implemented as a batched linked list (i.e., each node uses a sorted vector to represent a batch of N key/value pairs). While lookup operations within a batch take $O(\log N)$ time, it takes $O(N)$ work to insert or remove an element in a batch, in order to preserve sorting. If instead we used an unsorted batch, each operation would cost $O(N)$, but with lower constants. Similarly, if each batch is protected by a coarse lock, then when keys K_1 and K_2 are stored in the same batch, threads operating on those keys would not be able to proceed in parallel.

While it may seem difficult to find an ideal batch implementation, recent work has shown that it is not too difficult, especially for workloads that deal with large volumes of data and low rates of skew, so long as batch sizes remain modest. Examples of scalable, low-latency batched data structures include maps (e.g., Kiwi [3], CUSL [19], Skip Vector [21], OCC (a, b)-tree [22], Lock-Free B+Tree [6]), and queues [10, 20, 25]. These works tended to treat batching as a first-class design consideration, raising the question of whether it is possible to build a general methodology for adding batching to an existing concurrent data structure. We propose the BatchBoost methodology as a step toward this goal. BatchBoost is designed specifically for ordered maps. It provides programmers with a scalable batched doubly-linked list. The original data structure is then treated as an index to some node in the list. The key innovation is that an out-of-date index will always return a valid node, from which the "correct" node can be found by moving through the links of our doubly-linked links. In this way, BatchBoost lets programmers keep their existing, scalable index, while still benefitting from batching of key/value pairs.

2 Requirements and the BatchBoost Construction

Our goal is to emphasize orthogonality. It should be possible for a programmer to think of a data structure as consisting of an *index layer* and a *data layer*. The data layer should be batched, with as few configuration knobs as possible. The index layer should be decoupled from the data layer, and chosen based on workload and machine characteristics. At any time, it should be trivial to replace the index or data layer with a more suitable data structure, without changing the other layer's implementation.

In BatchBoost, data structure operations always linearize in the data layer. The index layer can be thought of as providing routing "hints." Given relatively straightforward requirements on the data layer, an operation proceeds in three steps. First, it queries the index layer to find a good starting position in the data layer. Second, it operates on the data layer. Finally, it might update the index. A key point is that the index layer need not be kept consistent with the data layer, so long as (1) data layer operations can recover from bad hints, and (2) the index and data layers agree on how to achieve safe memory reclamation.

Listing 1 Composition of index and data layer operations into BatchBoost operations

```
1  fn lookup(IndexLayer I, Key K) -> Option<V>
2      at = I.findApprox(K)
3      <ret, val, node> = at.lookup(K)
4      if ret == Found: return Some(val)†
5      if ret == NotFound: return None()†
6      if ret == DeletedNode: I.remove(node.key); goto 2
7
8  fn insert(IndexLayer I, Key K, Value V) -> bool
9      at = I.findApprox(K)
10     <ret, node> = at.insert(K, V)
11     if ret == InsertSuccess: return true†
12     if ret == AlreadyExists: return false†
13     if ret == DeletedNode: I.remove(node.key); goto 9
14     assert(ret == InsertSuccessAndSplit)
15     I.insert(node.key, node)
16     if node.deleted: I.remove(node.key)
17     return true
18
19 fn remove(IndexLayer I, Key K) -> bool
20     at = I.findApprox(K)
21     <ret, node> = at.remove(K)
22     if ret == RemoveSuccess: return true†
23     if ret == NotPresent: return false†
24     if ret == DeletedNode: I.remove(node.key); goto 20
25     assert(ret == RemoveSuccessAndMerge)
26     I.remove(node.key)
27     return true
```

Listing 1 presents a general BatchBoosted data structure. We model the `DataLayer` type as a collection of nodes, each of which stores a tuple $\langle pairs, lower, upper, size, capacity\rangle$, as well as links to other nodes. *pairs* is a collection of *size* key/value pairs ($size \leq capacity$), whose keys are in the range $[lower, upper)$. The range of the `DataLayer` is from $\perp$ to $\top$, which is also the union of all nodes' ranges. We require that from any node, there is a way to reach any other node (perhaps because nodes have predecessor and successor pointers, or because everything is reachable from some sentinel node). We also require that the node include a field indicating if it has been removed from the data layer (a `mark` or `deleted` bit). Each node in `DataLayer` supports three operations with a key argument: 1) `lookup` operation (line 3) traverses the doubly-linked list and returns the node that should contain the key; 2) `insert` operation (line 10) traverses the doubly-linked list, finds the node where the key should be inserted, and inserts there; 3) `remove` operation (line 21) traverses the doubly-linked list, finds the node where the key should be, and removes it from there.

The `IndexLayer` type is an ordered map from keys to `DataLayer::Node` objects. We do not specify its implementation, only that it allows the creation and removal of mappings, and supports some suitable `findApprox(k)` function that returns a value mapped to a key which is likely to be close to `k`. The precision of `findApprox()` does not affect correctness, but the performance of BatchBoosted data structures is likely to correlate with the precision of the index's `findApprox()` implementation.

Initially the data layer contains a single node, which is mapped to the index with key $\perp$. The index may store references to logically deleted nodes; it can also lack references to nodes that are in the data layer. `IndexLayer::findApprox(key)` represents these possibilities: when queried with a key, there is no guarantee that the returned node contains it or even be somewhere close. Note that for an ordered map, `findApprox(key)` can be implemented in many ways, including `ceil(key)` and `floor(key)`.

The index is updated lazily. Insertion of a key/value pair into a node may result in the creation of a new node in the data layer; removal of a pair may result in a node becoming "too small", in which case it can be unlinked once its contents are merged into an adjacent node. These conditions are returned on lines 10 and 21, respectively. If a node becomes deleted between when it is created and when it is added to the index, an `insert` operation is responsible for removing it (line 16). Coupled with standard assumptions about safe memory reclamation, this ensures a node pointed to by the index is still safe to access, even if it has been unlinked from the data layer. Similarly, removal of a merged node from the index layer can delay (line 24), in which case some other thread may remove it (e.g., line 6), and a subsequent insertion can put a different key/node mapping into the index. When this happens, the removal of a valid node is possible. Lines marked with † represent places where an operation may choose to remedy this situation by trying to insert `node` if `node` $\neq$ `at`.

For clarity, the code in Listing 1 skips other optimizations. We do not describe the exact implementation of the data layer because there are lots of them. For example, some data layer implementations may allow `lookup` to succeed even when the node returned by `findApprox` has been unlinked, avoiding the need for line 6.

3 Performance Evaluation

Description. We implemented BatchBoost in C++. We use three non-batched search structures as index layers: Fraser's skip list [13] and trees by Bronson et al. [7] and Natarajan et al. [18]. For all index layers we use the existing `floor` method for `findApprox`. The skip-list code is from SynchroBench [14], the trees are from SetBench [9]. For the data layer, we created a batched, doubly linked list based on the Lazy List [15]. While many configurations of the data layer are possible, we only consider a fixed-capacity array storing its key/value pairs in ascending order. We use epoch-based memory reclamation; threads enter the epoch at the beginning of an operation in Listing 1, and exit the epoch immediately before the operation returns.

All experiments were conducted on a machine with two Intel Xeon Gold 5218 CPUs at 2.30GHz (32 total cores / 64 threads), running Ubuntu 22.04 (Linux Kernel 5.15). We compiled all code with clang 15 (–O3 optimizations). Each data point is the average of five 5-seconds trials. Variance was typically low, and is indicated via error bars.

Experiments are parameterized by lookup ratio R and key range K. Each operation type is chosen randomly and is a lookup with R% probability, with remaining operations split equally between insert and remove. Data structures are pre-filled with 50% of keys, so that the data structure size stays roughly constant. Integer keys are chosen with uniform probability from $[1, K]$.

Sensitivity to Batch Size. The batch size is a critical configuration parameter. If it is too small, batching might increase latency. If it is too large, then contention on batches will be too high, hindering scalability. Figure 1 measures throughput at 32 threads as we vary the batch size ($K = 10^7$). We consider lookup ratios of 34% and 90%. The labels **sl**, **bro**, and **nat** refer to Fraser's skip list [13], Bronson's tree [7], and Natarajan's tree [18], respectively.

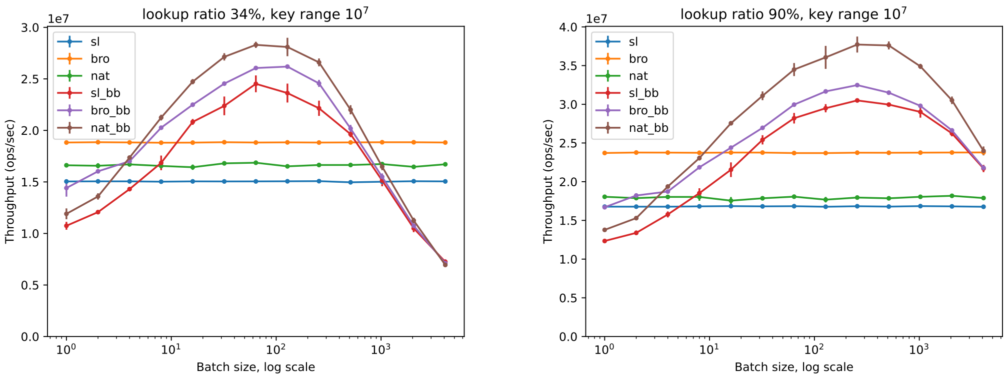

Figure 1 Impact of batch size on throughput at 32 threads

	4	64	1024
bro_bb	44.43	30.22	36.68
sl_bb	29.07	22.27	35.58
nat_bb	37.14	36.32	40.71

Table 1 Impact of batch size on cache miss ratio at 16 threads

The **_bb** suffix refers to a BatchBoost data structure composing the corresponding index with our doubly-linked list.

While the results confirm that there is a sensitivity to batch size, the expected performance plateau is surprisingly wide. Thus while there is more than 2× difference between good and bad batch sizes, the exact size does not seem to be particularly significant. We observe that sensitivity is lower than in nonblocking batched data structures [22]. This is due to our use of a lock-based list, which allows in-place modification instead of copy-on-write. Since the drop-off is worse when the batch size gets too large, we conservatively chose a batch size of 100 for all subsequent experiments.

Using the Linux `perf` tool, we were able to attribute these results directly to a reduction of cache misses. Table 1 shows cache miss ratio against the total number of cache loads for different batch sizes. In effect, BatchBoost shrinks the size of the index, thereby reducing pointer chasing. While the data layer has more cache accesses than a leaf of the unmodified data structure, the increase is less than the savings in the index layer. However, with the increasing batch the ratio of cache misses also increases, thus, we need to choose some ideal batch size.

Throughput and Scalability. Figure 2 measures throughput of our BatchBoost data structures with a fixed batch size as we vary the thread count. BatchBoost consistently improves the performance. The peak speedup depends on workload parameters and varies from $5 - 10\%$ to almost 2×.

Furthermore, we do not observe significant cache traffic due to contention. By the time threads reach the data layer, the index has dispersed them, reducing the likelihood of contention. Thus as long as the data layer has low latency, the window of contention is low, and threads are not likely to interfere with each other. Additionally, the data layer hides most mutations (insertions and removals) from the index layer. A smaller index, with fewer writes, is more likely to remain resident in most CPUs' caches. In essence, BatchBoost increases the likelihood that the index stays in its common (read-only) case.

4 Conclusions and Future Work

In this paper we introduced the BatchBoost methodology, and demonstrated that it simplifies the creation of scalable data structures with good locality. As discussed in Section 1, batching

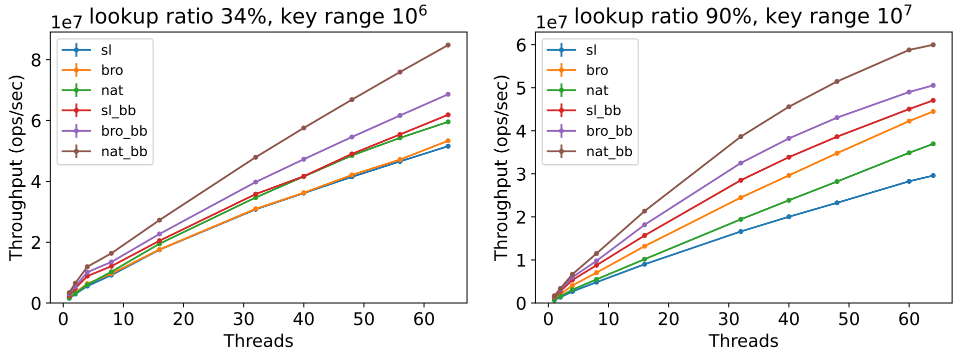

Figure 2 BatchBoost throughput and scalability for varied R and K

has broad potential. An important future research direction is to apply our BatchBoost construction in additional domains, as well as on more complex benchmarks. We also intend to compare against other batching techniques. Another important research question pertains to the data layer: We demonstrated that BatchBoost worked well with different index layer implementations, but what about alternate data layer implementations (especially nonblocking)? Further afield, our evaluation showed that BatchBoost amplified the "common case" in the index layer. This may motivate designing new index layers with an explicit and highly optimized `findApprox` operations. For example, we are interested whether we can use a fast sequential index data structure, e.g., Abseil B-trees [1], protected by a scalable readers/writer lock. This could allow concurrent updates and reads, since even under concurrent rebalancing, index lookup operations will give a good enough approximation in our doubly-linked list.

References

1. Abseil b-tree containers. URL: `https://abseil.io/about/design/btree`.
2. Maya Arbel-Raviv and Trevor Brown. Harnessing epoch-based reclamation for efficient range queries. *ACM SIGPLAN Notices*, 53(1):14–27, 2018.
3. Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. Kiwi: A key-value map for scalable real-time analytics. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 357–369, 2017.
4. Rudolf Bayer and Edward McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, pages 107–141, 1970.
5. Anastasia Braginsky and Erez Petrank. Locality-conscious lock-free linked lists. In *Distributed Computing and Networking: 12th International Conference, ICDCN 2011, Bangalore, India, January 2-5, 2011. Proceedings 12*, pages 107–118. Springer, 2011.
6. Anastasia Braginsky and Erez Petrank. A Lock-Free B+tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, Pittsburgh, PA, June 2012.
7. Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *ACM Sigplan Notices*, 45(5):257–268, 2010.
8. Trevor Brown and Hillel Avni. Range queries in non-blocking k-ary search trees. In *Principles of Distributed Systems: 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings 16*, pages 31–45. Springer, 2012.
9. Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–291, 2020.

10 Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In *Distributed Computing: 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings 28*, pages 406–420. Springer, 2014.

11 Jiwon Choe, Andrew Crotty, Tali Moreshet, Maurice Herlihy, and R Iris Bahar. Hybrids: Cache-conscious concurrent data structures for near-memory processing architectures. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 321–332, 2022.

12 Panagiota Fatourou, Elias Papavasileiou, and Eric Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, pages 275–286, 2019.

13 Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

14 Vincent Gramoli. More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2015.

15 Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems: 9th International Conference, OPODIS 2005, Pisa, Italy, December 12-14, 2005, Revised Selected Papers 9*, pages 3–16. Springer, 2006.

16 Kevin Hsieh, Eiman Ebrahimi, Gwangsun Kim, Niladrish Chatterjee, Mike O'Connor, Nandita Vijaykumar, Onur Mutlu, and Stephen W Keckler. Transparent offloading and mapping (tom) enabling programmer-transparent near-data processing in gpu systems. *ACM SIGARCH Computer Architecture News*, 44(3):204–216, 2016.

17 Justin J Levandoski, David B Lomet, and Sudipta Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.

18 Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 317–328, 2014.

19 Kenneth Platz, Neeraj Mittal, and S. Venkatesan. Concurrent Unrolled Skiplist. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, Dallas, TX, July 2019.

20 Anastasiia Postnikova, Nikita Koval, Giorgi Nadiradze, and Dan Alistarh. Multi-queues can be state-of-the-art priority schedulers. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 353–367, 2022.

21 Matthew Rodriguez, Ahmed Hassan, and Michael Spear. Exploiting locality in scalable ordered maps. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 998–1008. IEEE, 2021.

22 Anubhav Srivastava and Trevor Brown. Elimination (a, b)-trees with fast, durable updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 416–430, 2022.

23 Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, pages 473–488, 2018.

24 Yuanhao Wei, Naama Ben-David, Guy E Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. Constant-time snapshots with applications to concurrent data structures. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 31–46, 2021.

25 Chaoran Yang and John Mellor-Crummey. A Wait-Free Queue as Fast as Fetch-and-Add. *SIGPLAN Notices*, 51(8), February 2016. URL: `https://doi.org/10.1145/3016078.2851168`.