POSTER: Bundled References: An Abstraction for Highly-Concurrent Linearizable Range Queries

Jacob Nelson
Lehigh University, USA
jjn217@lehigh.edu

Ahmed Hassan
Lehigh University, USA
ahh319@lehigh.edu

Roberto Palmieri
Lehigh University, USA
palmieri@lehigh.edu

Abstract

Bundled references are a new building block to provide linearizable range query operations for highly concurrent linked data structures. They enable range queries to traverse a path through the data structure that is consistent with the target atomic snapshot. The path consists of the minimal amount of nodes that should be accessed to preserve linearizability.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**.

Keywords: Concurrent Data Structures, Linearizable Range Queries

1 Introduction

In this paper we introduce *bundled references*, a new building block to design linearizable concurrent linked data structures (e.g., skip lists) optimized to scale up performance of range query operations executing concurrently with update operations. The core innovation behind bundled references lies in adopting the design principles of Multi Version Concurrency Controls (MVCC) [8] and persistent data structures [3] to generic linked data structures.

To summarize the strategy, we enable linearizable range queries by augmenting each link in a data structure with a historical record of its previous values, each of which is tagged with a timestamp reflecting the point in (logical) time when the operation that generated that link occurred. In other words, we associate timestamps to references connecting data structure elements instead of to the pointed elements. A range query simply follows the links that comply with its starting point in order to collect its result set.

The bundled reference building block equips a data structure with the following properties:

- Range query operations are linearized when they start, reducing interference with concurrent updates;
- Each thread performing a range query only traverses the *minimal* amount of nodes in the range, regardless of concurrent updates.
- Data structure traversals are left unaffected by bundles, permitting optimizations such as wait-free [4] traversals;
- State-of-art reclamation techniques (e.g., EBR [16]) can be easily integrated to reclaim data structure elements, minimizing the space overhead of bundling and making it practical.

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8294-6/21/02.
https://doi.org/10.1145/3437801.3441614

2 The Bundle Building Block

Overview. Figure 1 shows an example on how bundles are deployed in a linked list. As shown in the figure, the next pointer of each node is replaced by a *bundle* object that encapsulates the history of this next pointer. The figure shows the state of the linked list and its bundles after the following sequence of operations (assuming an initially empty linked list): `insert(20)`, `insert(30)`, `insert(10)`, `remove(20)`.

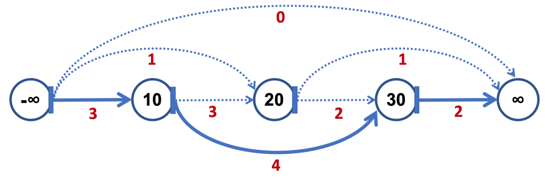

Figure 1. An example of using bundled references in a linked list. The path made of solid lines represents the state of the linked list after all update operations take place. Edges are labeled with their respective timestamps.

To understand how this state is generated, we assume that the list is initialized with a single bundle reference whose timestamp is "0" (the initial value of a global timestamp `globalTs`), which connects its head and tail sentinel nodes. Inserting `20` does not replace this reference. Instead, it creates a new entry in the head's bundle with timestamp "1" pointing to the newly inserted node as well as an entry with the same timestamp in this new node pointing to the tail node. Similarly, inserting `30` and `10` adds new bundle entries with timestamps "2" and "3", respectively. The last operation that removes `20` also does not replace any reference. Instead, it creates a new bundle entry in `10`'s bundle (with timestamp "4") that points to `30`, which reflects physically deleting `20` by making its predecessor node point to its next node.

Now, assume that different range queries start at different times concurrently with those update operations. Each range query (R_t) is always able to traverse the proper snapshot of the list that reflects the time (t) it started. For example, a query R_0 will skip any links whose timestamp greater than "0". Importantly, R_3 will observe `20` even if it reaches `10` after `20` is deleted because it will use the bundle entry whose timestamp is "3", which points to `20`. Furthermore, since a range query determines its result set by simply following the most recent link corresponding to its timestamp, it visits the minimum number of nodes necessary to collect its range.

Listing 1. Bundle.

```
timestamp_t globalTs;
class BundleEntry {
    Node * ptr;
    timestamp_t ts;
    BundleEntry * next;
}
class Bundle {
    BundleEntry * head;
}
```

Listing 2. Linked List Node.

```
class Node {
    key_t key;
    val_t val;
    lock_t lock;
    bool deleted;
    // The bundled reference
    Node * newestNextPtr;
    Bundle nextPtrBundle;
}
```

A bundle (see Listing 1) consists of a linked list of *bundle entries*, ordered by timestamp, and exists alongside a standard reference (e.g., `newestNextPtr`). Together, they encode a data structure link. By decoupling the bundle from the regular pointer, operations can retain certain optimizations such as wait-free traversals. In fact, range queries are the only operations that will ever traverse the structure using bundles. We demonstrate this particular optimization by showing nodes of a bundled linked list in Listing 2.

Updates. Bundles are updated whenever there is a corresponding change to the link. Care must be taken so that point operations and range queries observe a consistent view of the data structure. The following steps ensure the above requirement. ① Bundles corresponding to any links to be changed by the update are prepared by inserting a new bundle entry with `PENDING` timestamp. ② A fetch-and-add on `globalTs` reserves a timestamp with which to label bundle entries. ③ The linearization point is executed (e.g., linking a newly inserted node, logical deletion). ④ The update is made visible to range queries by replacing the `PENDING` timestamp with the one read from `globalTs`.

Range Queries. Range queries first read `globalTs` into a local variable (i.e., `ts`) to set their linearization point. A traversal can then simply use bundles to ensure that the visited nodes are consistent with this timestamp. If a `PENDING` entry is found, the range query must wait until the entry's timestamp is set since the update increments the global timestamp before performing its linearization point.

3 Bundling a Data Structure

Bundling a data structure is not automatic but the steps are well defined. It entails augmenting links in nodes with bundled references, surrounding the respective linearization points of update operations with bundle maintenance, and implementing range queries. The first two steps are straightforward and mostly data structure independent, while the bulk of the effort lies in the range query itself, as traversals are data structure dependent.

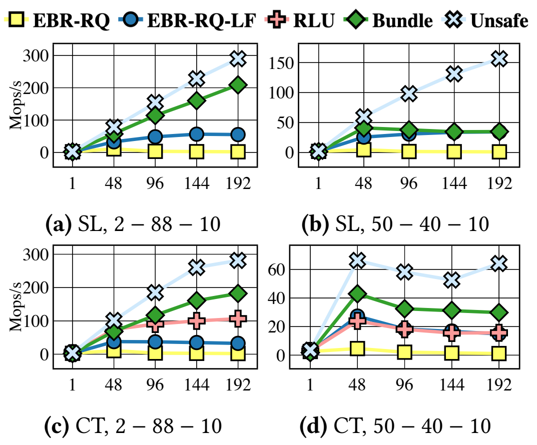

Figure 2. Throughput (Mops/s) under various workload configurations for the skip list (SL) and Citrus tree (CT), with the number of threads on the x-axis. Workloads are written as $U - C - RQ$, corresponding to the percentages of update (U), contains (C) and range queries (RQ).

However, regardless of data structure, we can break range queries into three phases. The *pre-range* phase reaches the node immediately preceding the range. Next, the *range-entry* phase traverses to the start of the range. Finally, the *range-collect* phase collects the result set to be returned.

Naturally, a traversal using bundles incurs overhead because it involves finding the correct link to follow. As an optimization, the pre-range phase leverages standard references used by point operations. Then, both the range-entry and range-collect phases leverage the bundles' `DereferenceBundle` function, which returns the most recent bundle entry that is no newer than the operation's linearization timestamp, `ts`. Note that with this strategy the range-entry phase is necessary to avoid missing a concurrent update, such as a deletion that removes a node belonging to the range. During this phase, if `DereferenceBundle` does not find a satisfying entry, then the operation restarts from the beginning. However, once the range-collect phase begins, the operation is destined to finish.

Complete details of how to bundle a data structure can be found in our companion technical report [7], which includes a description of a linked list, skip list and binary search tree.

4 Evaluation

We evaluate our technique by applying it to three data structures, integrating it in an existing benchmark, and comparing against the following three competitors:

- EBR-RQ [1] is a lock-based linearizable range query strategy based on epoch-based memory reclamation [2].
- EBR-RQ-LF is a lock-free variant of EBR-RQ.
- Read-log-update (RLU) [5] is an extension of the well-known RCU [6] strategy that supports concurrent writers.

Initial microbenchmark results (Figure 2) show our technique (i.e., Bundle) outperforms the competitors when the workload is mixed. This is because normal traversals are not instrumented (as in RLU) and range queries need not visit more nodes than necessary (as in EBR-RQ and EBR-RQ-LF). This confirms that bundling manages the trade-off between update-intensive and read-only workloads effectively. On the other hand, our competitors perform best under extreme workloads. Hence, performance stability across different workloads is an important byproduct of bundled data structures. EBR-RQ and EBR-RQ-LF outperfom bundling in 100% update workloads because range queries increment the global timestamp instead of updates. Conversely, RLU performs better in read-only workloads since its instrumentation of dereferences is lighter weight. More results and analysis, including the integration of our data structures in the DBx1000 database [9] running TPC-C, are presented in our technical report [7].

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1814974.

References

[1] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing epoch-based reclamation for efficient range queries. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018.* 14–27.

[2] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015.* 261–270.

[3] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. 1986. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing.* 109–121.

[4] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.* 13, 1 (1991), 124–149. https://doi.org/10.1145/114005.102808

[5] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* 168–183.

[6] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems.* 509–518.

[7] Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. 2020. Bundled References: An Abstraction for Highly-Concurrent Linearizable Range Queries. arXiv:cs.DS/2012.15438 https://arxiv.org/abs/2012.15438

[8] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. *Proceedings of the VLDB Endowment* 10, 7 (2017), 781–792.

[9] 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. *PVLDB* 8, 3 (2014), 209–220. https://doi.org/10.14778/2735508.2735511

A Artifact Description

Here we give a high-level overview of the artifact supporting this paper. For more detailed instructions on how to configure, run and plot the results, we refer the reader to the comprehensive README file included in the root directory of the source code (https://zenodo.org/record/4402298).

Organization. Our primary contributions are located in the directories prefixed with "bundle". Importantly, the "microbenchmark" and "macrobenchmark" directories are used for evaluating the approach. The *microbenchmark* consists of two experiments designed to understand the implications of design choices between competitors. The first runs various workload configurations while fixing the range query size to 50. The second uses a workload of 50% updates and 50% range queries, while adjusting the range query size. The artifact is pre-configured to execute experiments for only the skip list and Citrus tree since traversal time dominates the performance of lazy list; their results are more illustrative of algorithm behavior. The *macrobenchmark* integrates the data structures as indexes in the DBx1000 in-memory database and executes the TPC-C benchmark. Results are saved to a "data" directory in each benchmark while plots are saved to the "figures" directory at the project root folder.

Requirements. The project is primarily C++ and is compiled using g++ 7.3 with the compilation flags `-std=c++11 -mcx16 -O3`. The two primary dependencies are the `jemalloc` scalable memory allocation library and the `libnuma` NUMA library. The plotting scripts use Python (v3.6) and the Plotly library (v4.12). NUMA must enabled to run properly.

Configuration. Configuring the above benchmarks relies on five parameters that must be adjusted according to the testbed. The following can be found in "config.mk".

- `maxthreads`, the max number of concurrent threads
- `maxthreads_powerof2`, `maxthreads` rounded up to a power of 2
- `threadincrement`, the thread step size
- `cpu_freq_ghz`, processor frequency for measuring program execution time
- `pinning_policy`, the thread affinity mapping

The README contains a command to automatically generate a pinning policy that mirrors those used in our experiments, which fills each NUMA zone before moving on.

Execution. After compiling the project according to the instructions in the README, the benchmarks can be run by navigating to their respective directories and executing the script named `runscript.sh`. The results can then be plotted via the `plot.py` script located in the root directory. Plots are generated as interactive HTML files, which can be opened via a browser window.