Bundled References

In this paper we address the challenge of providing linearizablerange query operations for linked data structures by introducing a new building block; we call bundled references. Using bundled references provides range queries with a path through the data structure consistent with their linearization point. Our design guarantees that range queries only traverse nodes belonging to their snapshot and that they never block writers. With multi-version concurrency control (MVCC) in mind, we implement our technique in three data structures. The experimental evaluation of our bundled linkedlist, skip list, and binary search tree, including their integration as indexes in the DBx1000 in-memory database, shows up to 20% improvement over state-of-the-art techniques to provide linearizable range queries, as well as a more consistent performance profile across a variety of workloads

Implementation notes

This work graciously builds on a benchmark developed by Arbel-Raviv and Brown's work (https://bitbucket.org/trbot86/implementations/src/master/cpp/range_queries/) to provide linearizable range queries in linked data structures. We use their codebase as a starting point, and integrate our technique into their benchmark. The core of our bundling implementation is contained in the 'bundle' directory, which implements the global structures as well as necessary functions of bundling. The three data structures we implement are found in 'bundled_*' directories. The scripts necessary to produce the plots found in our paper are included under the 'microbench' directory.

Getting Started Guide

The remainder of this document intendeds to support an artifact evaluation for the paper "Bundled References: An Abstraction for Highly-Concurrent Linearizable Range Queries" by Jacob Nelson, Ahmed Hassan and Roberto Palmieri. As stated above, our contributions with respect to the paper are contained in the directories prefixed with "bundle".

`bundle` implements the bundling interface as a linked list of bundle entries. In addition to the linked list bundle, there is an experimental cirular buffer bundle (not included in the paper) as well as an unsafe version that eliminates the overhead of ensuring bundle consistency for comparison.

`bundle_lazylist`, `bundle_skiplistlock` and `bundle_citrus` each implement a data structure to which we apply bundling. Note that we do not apply our technique to the remaining data structures (which are lock-free) because our current bundling implementation would impose blocking.

The experiments that we report in the paper are located in the following directories.

- `microbench` tests each data structure in isolation.
- `macrobench` ports DBx1000 to include the implemented data structures.

Requirements

The experiments from the paper were executed on a 4-socket machine with Intel Xeon Platinum 8160 processors running Red Hat Enterprise 7.6. However, we also successfully tested on a dual-socket machine with Intel Xeon E5-2630 v3 processors running Ubuntu 18.04. The C++ libraries are requried to build and run the experiments, while the Python libraries are used for plotting results.

C++ Libraries:

- libnuma (e.g., `sudo apt install libnuma-dev`)
- libjemalloc (included in 'lib'

Python libraries:

- python (v3.6)
- plotly (v4.12.0)
- plotly-orca (v1.3.1)
- psutil (v5.7.2)
- requests (v2.24.0)
- pandas (v1.1.3)

The easiest way to install the Python libraries is by using Ananconda3. This is a somewhat annoying step because Linux distros must download and run the Anaconda installer. However, it is either this or building plotly-orca manually as it is not currently available with `pip`. After installing Anaconda3, use the following commands.

```
conda create -n bundledrefs python=3.6
conda activate bundledrefs
conda install -y -c plotly plotly plotly-orca
conda install -y psutil requests pandas
```

Kicking the Tires

Note: any warnings regarding hardware transactional memory (HTM) can be safely ignored since we do not compare with this competitor.

Once the C++ dependencies have been installed, you can begin to test the microbenchmark. First, configure the build with the `config.mk` file using the instructions provided there. Then, build the binaries for each of the data structures and range query techniques with the following:

```
cd microbench
make -j lazylist skiplistlock citrus rlu lbundle unsafe
```

The first three arguments to the `make` command (i.e., `lazylist`, `skiplistlock`, `citrus`) build the EBR-based approach from Arbel-Raviv and Brown. The next argument (i.e., `rlu`) builds the RLU-based lazy-list and Citrus tree. The fifth argument (i.e., `lbundle`) builds the bundled lazy-list, optimistic skip-list and Citrus tree. Finally, the last argument (i.e., unsafe) builds the three data structures of interest with no instrumentation for range queries. Unlike the unsafe implementation provided by Arbel-Raviv and Brown,

our implementation does not reclaim memory. We chose to do this because the RLU-based data structures do not utilize epoch-based memory reclamation. As such, our unsafe versions are an upper bound on all range query techniques and provides a more general reference.

Finally, run individual tests to obtain results for a given configuration. Note that this project uses jemalloc to replace the standard memory allocation. The following command runs a workload of 5% inserts (`-i 5`), 5% deletes (`-d 5`), 80% gets and 10% range queries (`-rq 10`) on a key range of 100000 (`-k 100000`). Each range query has a range of 50 keys (`-rqsize 50`) and is prefilled (`-p`) based on the ratio of inserts and deletes. The execution lasts for 1s (`-t 1000`). There are no dedicated range query threads (`-nrq 0`) but there are a total of 8 worker threads (`-nwork 8`) and they are bound to cores following the bind policy (`-bind 0-7,16-23,8-15,24-31`).

```
env LD_PRELOAD=../lib/libjemalloc.so TREE_MALLOC=../lib/libjemalloc.so \
./hostname.skiplistlock.rq_lbundle.out -i 5 -d 5 -k 100000 -rq 10 \
-rqsize 50 -p -t 1000 -nrq 0 -nwork 8 -bind 0-7,16-23,8-15,24-31
```

The above example assumes a 32-core system. A simple way to follow our bind policy is to execute `lscpu` and copy the NUMA node CPUs, appending each set of CPUs separated by a comma.

For more information on the input parameters to the microbenchmark itself see README.txt.old, which was written for the original implementation. We did not change any arguments.

Step-by-Step Instructions

Full Microbenchmark

Our results demonstrate that in mixed workload configurations, and in the presence of range queries, our implementation outperforms the competitors. This can be demonstrated by the running the full microbenchmark using `microbench/runscript.sh`.

Assuming that the binaries have already been built during the previous steps, let's generate the plots included in the paper (once the dependencies above are installed). From the root directory, run the following:

```
./runscript.sh
cd ..
python plot.py --save_plots --microbench
```

`runscript.sh` will run expeirments based on `experiment_list_generate.sh`, which will write a list of experiments to be run into a file. This generation script can be altered to try out new configurations. Note that `plot.py` defines which number of worker threads to include in the range query size speedup over unsafe plot. You should verify that the numbers of threads align with the configurations defined by `config.mk`.

WARNING: The experiments can take a long time to run because there are many competitors. We have preconfigured the run to execute a single trial (the paper uses 3), run for 1s (the paper uses 3s), and test the optimistic skip-list and Citrus tree (the paper tests all three data structures). The number of trials and runtime can be configured in `runscript.sh` and the data structures in `experiment_list_generate.sh`.

`experiment_list_generate.sh` includes two experiments. The first, saved under `microbench/data/workloads` fixes the range query size to 50 and tests various workload configurations. This corresponds to Figure 2 in the paper as well as additional experiments for get-only and update-only workloads. The second, whose results will be written to `microbench/data/rq_sizes`, executes a 50%-50% update-rq workload at various range query lengths (i.e., 1, 5, 10, 50, 100, 500). This corresponds to Figure 3.

The default is that both experiments will run, which is estimated at 20 minutes to complete but may take longer because that does not include prefill time.

Macrobenchmark

In addition to demonstrating better performance in mixed workloads, we also demonstrate improvements over competitors in index performance when integrated into a database. This can be observed by running the macrobenchmark.

To build and run the DBx1000 integration, run the following from the root directory:

```
cd ./macrobench
./compile.sh
./runscript.sh
cd ..
python plot.py --save_plots --macrobench
```

In comparison to the microbenchmark, this will take longer to run. We suggest going for a long walk, calling a friend, or taking a nap. Two plots will be generated, one for each of the data structures at various numbers of threads.

Memory Reclamation

The initial binaries are built without bundle entry reclamation to match the paper discussion. Whenever a node is deleted its bundle entries are reclaimed but stale bundle entries are not garbage collected for connected nodes. To enable reclamation of bundle entries, uncomment line 11 of `bundle.mk`. The following line defines the number of nanoseconds that elapse between iterations of the cleanup thread. It is currently set to 100ms.

Once `bundle.mk` is updated, remake the the bundled data structures using `make -j lbundle` and rerun the previously described microbenchmark. Be sure to move the original plots so they are not overwritten when regenerating them.