1. 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. We experimentally evaluate the performance of our bundled linkedlist, skip list, and binary search tree, and integrate them as indexes in the DBx1000 in-memory database.

2. 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 implement our technique on top of the existing framework. 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.

3. Getting Started Guide

a. Important files and directories

Implementation

`./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.

`./vcas_lazylist`, `./vcas_skiplist_lock`, and `./vcas_bst` each implement our porting of vCAS to lock-based data structures for the evaluation.

Experiments

`config.mk` is the primary configuration file and is used across all experiments. Users will need to update this file to match their system (more on this later).

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.

Both of the above experiments are run by using their respective `./runscript.sh` scripts.

Generated files

`experiment_list.txt` is generated by the microbenchmark, resides in `./microbench` and contains the list of experiments to run.

`./microbench/data` and `./macrobench/data` are the destination directories for all saved data during experiments. This is also where automatically generated .csv files will be saved when plotting the results.

`./figures` stores all generated plots. It is split first into the respective experiments and then into the data structures used. Note that the generated plots are saved as interactive HTML files and should be opened using a browser.

b. Requirements

The experiments from the paper were executed on a 4-socket machine with Intel Xeon Platinum 8160 processors running Ubuntu 20.04. However, we also verified our results on a dual-socket machine with Intel Xeon E5-2630 v3 processors running Ubuntu 18.04. The C++11 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 (e.g., `sudo apt install libjemalloc-dev`)

After installing jemalloc, **it is necessary to replace the jemalloc library** contained in the `lib` folder. To do so, simply copy the shared library to `lib` so that the scripts can locate it. Otherwise, update line 82 in `microbench/runscript.sh` and line 73 in `macrobench/runscript.sh` to point to the location of the library.

Python libraries:

- python (v3.10)
- plotly (v5.1.0)
- psutil (v5.8.0)
- requests (v2.26.0)
- pandas (v1.3.4)
- absl-py (v0.13.0)

The above libraries can be installed with Miniconda by running the following:

```
conda create -n paper63 python=3
conda activate paper63
conda install plotly psutil requests pandas absl-py
```

c. Configuration

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

Once the C++ dependencies have been installed, you can begin to test the microbenchmark. First, configure the build with the `config.mk` file. There are five configuration parameters.

- `maxthreads` is the maximum number of threads to be tested during the experiments
- `maxthreads_powerof2` this is used for bookkeeping and is the next largest power of two from `maxthreads`
- `threadincrement` is the sampling period of threads between 0 and `maxthreads` for each experiment
- `cpu_freq_ghz` is the system's CPU frequency in GHz (used by the macrobenchmark)
- `pinning_policy` is a string that starts with "-bind " (or left blank) and maps threads to cores during execution

Configuration Tips

1. Together, `maxthreads` and `threadincrement` determine the number of samples generated during experiments. For example, on a 44 core machine with `maxthreads=44` and `threadincrement=8` the resulting numbers of threads tested will be [1, 8, 16, 32, 40, 44]. Both 1 and `maxthreads` are always included, regardless of whether `maxthreads` is a multiple of `threadincrement`.

2. The easiest way to determine both `cpu_freq_ghz` and `pinning_policy` is to execute `lscpu` on the command line. The first is directly used from the line indicating CPU frequency. The latter is a comma separated list of the NUMA node mappings. Consider a hypothetical machine with NUMA zones of four cores each that has the folling mappings: `NUMA 0: 1,3,5,7` and `NUMA 1: 0,2,4,6`. The pinning policy that mimics our setup would then be `pinning_policy="-bind 1,3,5,7,0,2,4,6`. If `pinning_policy` is left blank then no specific policy is used.

3. The following command will extract the cores associated with each NUMA zone and make a comma deliminated list that follows our pinning policy of filling NUMA zones. The output can then be copy and pasted into `config.mk`.

```
lscpu | grep -P "NUMA node[[:digit:]]" | sed -E 's/.*:\W*([0-9]*)/\1/g' |
tr '\n' ',' | sed 's/,$/\n/g' | xargs echo "-bind"
```

d. Building the Project

Once configured, build the binaries for each of the data structures and range query techniques with the following:

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

The first three arguments to the `make` command (i.e., `lazylist`, `skiplistlock`, `citrus`) build the EBR-based approach from Arbel-Raviv and Brown, the vCAS approach of Wei et al., our bundling approach, and

an unsafe version of each that has not consistency guarantees for range queries. The next argument (i.e., `rlu`) builds the RLU-based lazy-list and Citrus tree.

e. Running Individual Experiments

Finally, run individual tests to obtain results for a given configuration. 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`). Do not forget to load jemalloc and replace `<hostname>` with the correct value.

```
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
```

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

4. Results Validation

Corresponding Figures

Each of the following experiments saves a set of raw data, which is then automatically converted into a .csv file for processing and graphing. For both, the plots generated by `plot.py` and saved to `./figures` are interactive HTML files, which can be opened using a browswer window. These files correspond to Figures 2-4 in the paper.

- Figure 2 is a compilation of plots generated under `./figures/microbench/workloads`, with each individual file (in its data structure subdirectory) representing a subplot (in the corresponding row) of the figure. Each of these plots are named `updateX_rqY_maxkeyZ.html` where X is the percentage of update operations, Y the percentage of range queries, and Z the key range. The percentage of get operations is calculated by 100 - (X + Y).
- Figure 3 is directly represented by `./figures/microbench/rq_sizes/rq_sizes.html`
- Figure 4 is represented by the two plots generated when passing the `--macrobench` argument to `plot.py`, which are stored in `./figures/macrobench/`.
- The remaining plots in the supplemental material can be generated using the `--workloads_urate` flag for `plot.py` and passing the appropriate value corresponding to the update rate.

Configuring `plot.py` (Advanced)

In general, there should be no need to perform any configuration for `plot.py` itself. We have arranged for it to pull information from other scripts. However, there are numerous flags that can be set if the user wants more control over the output (run `python plot.py --help` for details).

a. Microbenchmark

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

Assuming that the data structure libraries 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. `plot.py` pulls the configuration directly from `config.mk` so no changes to it should be necessary.

`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. This corresponds to Figure 3.

WARNING: The experiments can take a long time to run because there are many competitors. As was used for our results, have preconfigured the run to execute three trials, run for 3s, and test the lazy-list, skip-list and Citrus tree. Both `runscript.sh` and `experiment_list_generate.sh` contain some addtional configuration options, but *they are not required*.

- `runscript.sh` defines the length of experiments. Specifically, lines 9 and 36 are pertinent as they adjust the number of trials per-configuration and the length of each trial. If you do not wish to wait as long for the experiments to terminate, you may adjust these values knowing that the results may differ from those presented in the paper.
- `experiment_list_generate.sh` contains some other configuration options. The current configuration is an abridged version, which limits the nubmer of overall tests to run for the sake of time. Lines 16-18 indicate which competitors to test (`rqtechniques`), which data structures to run them on (`datastructures`), and the key ranges to use (`ksizes`).

Output

As stated previously, the microbenchmark saves data under `./microbench/data`. This raw data is used by the plotting script, but is first translated to a .csv file that is also stored in the subdirectory corresponding to each experiment in `experiment_list_generate.sh`. Upon running `plot.py` with the argument `--save_plots`, the generated graphs will be stored in `./figures` (again, in the corresponding subdirectories).

To further support the figures, there is also output generated to the console that prints the speedup of each competitor over the "unsafe" version.

b. 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.

Output

As with the microbenchmark, the macrobenchmark generates raw output in `./macrobench/data`. The last command in `./runscript.sh` automatically generates the .csv file that is stored in `./macrobench`. This file (i.e., `data.csv`) is then used by the plotting scripts, whose output is saved under `./figures/macrobench`.

c. Memory Reclamation

The initial binaries are built with memory reclamation enabled but do not include background bundle entry cleanup, which matches the paper discussion. In other words, when 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 lazylist.bundle skiplistlock.bundle citrus.bundle` and rerun the previously described microbenchmark. Be sure to move the original plots so they are not overwritten when regenerating them.