How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures

Artifact Overview Document

Kåre von Geijer and Philippas Tsigas
{karev, tsigas}@chalmers.se
Chalmers University of Technology, Gothenburg, Sweden

Document Overview

This document complements the 2024 Euro-Par paper '*How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures*' by K. von Geijer and P. Tsigas [1], and provides a description of the artifact used as well as step-by-step instructions to replicate the experiments. The artifact contains a library of concurrent data structures and benchmarks for evaluating their scalability. It encompasses both the new elastically relaxed data structures introduced in [1] and previous relaxed and strict designs to facilitate comparisons.

This document starts with a *Getting Started Guide*. This describes the artifact dependencies, and outlines how to install them locally on a Ubuntu system or how to set up a prepared Docker container. Furthermore, it also covers how the library pins software threads to hardware threads, and how to set this up for a new computer.

Following the getting started guide, the *Step-by-Step Instructions* outlines how to use the system, and how to run a script to re-run the experiments of the paper. This section covers both how to run individual benchmarks, and also how to run Python scripts aggregating many runs and data structures into graphs. More general information about how to navigate the artifact is found in its `README.md` file.

Getting Started Guide

The artifact is mostly self-contained and written in C, with some python3 used to generate the plots. It is implemented for the memory model of x86-64, so its performance and linearizability are not guaranteed for other architectures. Furhermore, it is built for Linux. All experiments should have exclusive access to the machine, to guarantee no interference from other processes.

Dependencies

Re-running the experiments from [1] requires:

- *gcc* (preferably version 13.2.1),

- GNU *make* (preferably version 4.4.1),
- *bc* (preferably version 1.07.1),
- *python3* (preferably version 3.11.8, and at least 3.7),
- *pip3* (preferably version 23.3.2).

It also requires that gcc is set as the standard C compiler (e.g. by using `alias cc=gcc`). Furthermore, the plots need the following python packages:

- matplotlib (version 3.8.2),
- numpy (version 1.26.3),
- scipy (version 1.12.0).

Installing locally on Ubuntu

If running on Ubuntu, the above dependencies can all be installed as following:

```
$ apt install build-essential bc python3 python3-pip
$ pip3 install numpy==1.26.3 matplotlib==3.8.2 scipy==1.12.0
```

Similar other setups can be done for other OSs. If not done when installing `gcc`, make sure to set `cc` as an alias for it.

Setup with Docker

The artifact also includes a `Dockerfile` which can be used to set up an Ubuntu container with the dependencies outlined above. This requires `docker` to be available on your system.

First build the image, which installs the environment:

```
$ docker build -t relax_instantly .
```

Then create the folder `results`, which will be mounted to the container so that the graphs can be viewed from the outside. Finally, run the container in interactive mode, from which you can run experiments as described in the next section.

```
$ mkdir results
$ docker run -it --rm -v ./results:/app/results relax_instantly
```

Thread Pinning

Although not necessary to run the experiments and generate graphs, setting up thread pinning is essential for getting good quality results. Furthermore, this has to be configured individually per machine, as the hardware thread numbering varies between systems.

A common pinning strategy, which was used in [1], is to first pin one thread per core, and then pin with SMT when all cores all full, while only keeping to one NUMA node. Tools like `lscpu` (that shows which threads are on what NUMA node) and `lstopo` (which visualizes the core and thread layout) are very useful.

To pin the software threads to the hardware thread numbers found above, one has to add an entry to `common/Makefile.common` and `include/utils.h`. We have added example entries here that you can modify to set up your machine:

1. `Makefile.common`: Search for the entry starting with `ifeq ($(PC_NAME), example)`, and replace `example` with the current machine name (the output from `uname -n`). Then, you could change the compiler flag `-DEXAMPLE` to something more descriptive of your machine, but this is optional. Otherwise it should be fine.
    - Note, if running in a container, the perceived machine name will be changed from the real machine name, so you have to check it with `uname -n` within the container after starting it.
2. `utils.h`: Search for the entry starting with `#if defined(EXAMPLE)`. If you changed the compiler flag in the Makefile, change `EXAMPLE` here to what you changed the flag to (excluding the `-D`). Then, change the thread pinning order (now set to 0..256) to your desired order of how to allocate hardware threads.
    - There are three orders to specify here. One for alternating between NUMA nodes, one for directly utilizing SMT, and the final one (which we recommend starting with) is to avoid NUMA and SMT as long as possible.

To validate that the thread pinning works successfully, run `htop` during the run of your experiments and see that the threads are being filled up in the correct order, and all are being used in the experiments with maximum thread counts. For complementary information about thread pinning, read the included `README.md`.

Verify Setup

To ensure that the setup went smoothly, all data structures should be able to be compiled using the command `make all`. They should be present in `./bin`, and output some statistics when run. Furthermore, the following command should show (and save in `./results/`) a very simple plot.

```
$ python3 scripts/benchmark.py --initial 65536 --runs 2 -d 500 -k 5000 \
  -m 1  --width-ratio 2 -f 2 -t 10 -s 4 --ndebug 2Dd-queue queue-ms    \
  --title "Testing Queues: Thread Scalability"  --name queue_test --show
```

Step-by-Step Instructions

To compile the default test for any data structure in the `src/` folder, run `make <name>`, after which the binary with the same name will be found in `bin/`. This binary can be run with different options, such as number of threads or relaxation configuration, and outputs stats such as throughput. If instead compiling with `RELAXATION_ANALYSIS=1 make <name>`, then the binary will be instrumented with locks to measure relaxation errors (at the cost of very low throughput).

To re-run the experiments from [1], run the `./scripts/recreate-europar.sh` script. This has a few variables at the top which can be adjusted to vary the run, such as how many runs to use. The most important setting to change is the maximum number of threads to use, which we recommend to set equal to the number of hardware threads. When done, the plots can be found in `./results/`. The script is by default configured to do far fewer runs than the experiments in [1], and takes around 10 minutes on our machines.

The experiments in [1] were run on a 128-core 2.25GHz AMD EPYC 9754 with two-way SMT, and a similar system is recommended to get the most similar results. Using a system with fewer threads will result in not seeing the full scalability, and likely not seeing as large difference between the 2D structures and the rest. Similarly, if using a NUMA system, the extra inter-node communication will slow down different algorithms differently. None of the relaxed algorithms are currently optimized to handle NUMA settings, although heuristics probably could alleviate the communication penalty significantly.

Scalability Evaluation

The python script `scripts/benchmark.py` is recommended to use for benchmarking the thread and relaxation scalability of different data structures, as shown in Figure 3 in [1]. Use the `-help` flag to get detailed information of all the arguments to use for the script. To compare the throughput of the *2D queue* from [3] and *MS queue* from [2], one can run the following:

```
$ python3 scripts/benchmark.py --runs 5 --width-ratio 2 -l 16 -f 2 -t 16 \
  --step 2 -d 1000 2Dd-queue queue-ms --name scalability_test
```

Here we compare the throughput of the two data structures at thread counts of 2 to 16 with a stride of 2. For each thread count the test aggregates results from 5 runs, where each run runs for 1000 ms. For relaxation configuration, the 2D queue uses a width of twice the thread count, and a depth of 16. Finally, the test is saved in `results/scalability_test/`.

This script is used in `./scripts/recreate-europar.sh` to recreate the plots from Figure 3 in [1]. However, the low number of runs and test duration will likely make the results more volatile than in the paper. To ensure results corresponding to the paper, the 2D designs should outscale the other ones in throughput, and the elastic designs should be very comparable to the static one.

Variable Workload Plots

The other important testing script is `scripts/benchmark-variable-workload.py`, which is used to benchmark how data structures adapt to a variable workload in a producer-consumer scenario. This corresponds to Figure 4 in [1]. This script utilizes the C tests `test-variable-workload.c`, which are only implemented for the two elastic queues, but can rather easily be adapted for other data structures. Here is how to use it to run a simple example similar to in the paper:

```
$ python3 scripts/benchmark-variable-workload.py 2Dd-queue_elastic-law \
  --args "-i 8388608"  "-l 16" "-d 1000" "-w 128" "-n 128" --show      \
```

```
--ops-per-ts 500  --name variable_test
```

Here, the initial number of items is set to 8388608, a window depth of 16 and a starting width of 128 are used, it runs for one second width 128 total threads. It also specifies that the throughput and relaxation characteristics should be sampled every 500 operations (thread locally). Finally, the results are saved in `results/variable_test`.

The two plots corresponding to Figure 4 in [1] are generated by the two last (not commented out) lines in `recreate-europar.sh`. The test without dynamic relaxation should look very similar to the plot in the paper. However, the dynamic relaxation requires some fine tuning to work well. The controller can be tuned by changing the constants at the top of `controller.h` in the queue source directory, but the default values should work okay. Furthermore, if the *Rank error bound* significantly lags behind the *Tail error*, the number of initial items should be lowered. The important result is that the *Tail error* should dynamically adapt to the number of active producers, indirectly stabilizing the *Producer latency* compared to the static run.

References

[1] von Geijer, K., Tsigas, P.: How to Relax Instantly: Elastic Relaxation of Concurrent Data Structures, To appear in: Euro-Par 2024: Parallel Processing. Springer (2024)

[2] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. pp. 267–275 (1996)

[3] Rukundo, A., Atalar, A., Tsigas, P.: Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework. In: 33rd International Symposium on Distributed Computing (DISC 2019). vol. 146, pp. 31:1–31:15. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2019)