A. Artifact description

An artifact associated with this paper includes

- implementations of the four different queues evaluated in the paper's experiments—our `wfqueue`, Morrison and Afek's `lcrq`, Fatourou and Kallimanis' `ccqueue`, and Michael and Scott's `msqueue`,
- `faa`—a synthetic queue benchmark used to evaluate the performance of `fetch-and-add` (FAA), and
- a benchmark framework to evaluate their performance.

This benchmark framework was used to produce the experiment results presented in this paper.

Since we are interested in queue implementations for an execution environment without a garbage collector, we augmented `lcrq` and `msqueue` to use Michael's hazard pointer scheme to reclaim memory. In an execution environment without garbage collection, comparing the performance of the algorithms without memory reclamation to others with memory reclamation wouldn't be a fair comparison.

The benchmark framework uses the `pairwise` benchmark to evaluation the throughput of different queues. In `pairwise`, all threads repeatedly execute pairs of enqueue and dequeue operations. Between two operations, `pairwise` uses a delay routine that adds an arbitrary delay (up to 150ns) to avoid artificial *long run* scenarios, as described in the paper.

The framework uses GNU `make` for compilation. It uses POSIX threads for concurrency and allocates memory using `jemalloc`. The framework uses `sched_setaffinity` to pin each threads to a specific hardware thread; this primitive was introduced in Linux kernel 2.5.8 and `glibc 2.3`. The queue implementations employ GCC's atomic primitives, supported by GCC versions 4.1 or later. `lcrq` requires the hardware support of `CAS2`, which is a 16-Byte wide `compare-and-swap`. For a wait-free implementation, `wfqueue` requires hardware support for `fetch-and-add`; emulating `fetch-and-add` with a traditional load-linked and store-conditional retry loop violates the wait-free property, even though it can work well in practice.

The experimental results in this paper can be reproduced by compiling and running the benchmarks in the framework on instances of x86_64 or IBM Power7 platforms described in the paper. The expected performance of different queue implementations on a large scale are: `faa` $\geq$ `wfqueue` $\approx$ `lcrq` $>$ `wfqueue0` $>$ `ccqueue` $>$ `msqueue`, where `wfqueue0` is our wait-free queue with `PATIENCE` $= 0$.

A.1 Description

A.1.1 Artifact informal meta information

- **Algorithm:** `wfqueue`, `wfqueue0`, `lcrq`, `ccqueue`, `msqueue`, `faa`.
- **Compilation:** GCC 4.1 or later. Recommand GCC 4.8.2.
- **Run-time environment:** Linux kernel 2.5.8 or later.
- **Hardware:** x86_64 or IBM Power7.
- **Run-time state:** Sensitive to cache contentions.
- **Execution:** sole user, process pinning.
- **Output:** Console.
- **Publicly available?:** Yes.

A.1.2 How delivered

The source code for the artifact is available on Github at `https://github.com/chaoran/fast-wait-free-queue`. The artifact associated with this paper is tagged as version 1.0.2 at `https://github.com/chaoran/fast-wait-free-queue/releases/tag/v1.0.2`.

A.1.3 Hardware dependencies

`lcrq` requires a processor that supports a double-width (16 byte) compare-and-swap.

A.1.4 Software dependencies

The framework uses GCC atomic primitives, which are included in GCC 4.1 or later. We **strongly** recommend using a GCC that is newer than GCC 4.7.3, which implements a new set of atomic primitives that may yield better performance.

The framework only compiles on Linux 2.5.8 (or later) with glibc 2.3 (or later) that supports POSIX threads. The scripts in the framework uses a `bash` shell with GNU `bc` 1.0.6.

Our experiments use the `jemalloc` memory allocator. Using `jemalloc` is optional. `jemalloc` is available at: `https://github.com/jemalloc/jemalloc/releases`.

A.2 Installation

Download the latest released version. Then unzip the tar file into a directory and execute `make` in the directory.

A.3 Experiment workflow

Map threads to cores By default, the framework will pin threads to cores in the order listed in `/proc/cpuinfo` on the experimental platform. In particular, the framework maps a thread with id i to the core with id ($i \bmod P$), where P is the total number of cores available in the system. One can add a `cpumap` function to the `cpumap.h` file to change this mapping. `cpumap` takes a thread's id and the total number of threads as its arguments and returns the corresponding core id for the thread. `cpumap.h` contains several examples of the `cpumap` function. A conditional macro should guard the added `cpumap` function; when the macro is defined, the added `cpumap` will be compiled. The macro guarding the new `cpumap` implementation should be added to `CFLAGS` in the provided makefile.

Compile code Executing `make` without any argument will compile all the binaries of the framework; they are `wfqueue`, `wfqueue0`, `lcrq`, `ccqueue`, `msqueue`, `faa`, and `delay`. `delay` is a `pairwise` benchmark with `enqueue` and `dequeue` operations removed, executing only delay operations.

Before executing `make`, one should make sure that `which gcc` returns the location of the correct version of GCC. When compiling code for a system that does not support `CAS2`, one can remove `lcrq` from the list of binaries to compile in the makefile.

Run benchmarks Directly invoking a binary with the number of threads as an argument will run the benchmark up to 20 times until it collects 5 consecutive measurements with a coefficient of variation less than 0.02. For example, `./wfqueue 72` runs the `pairwise` benchmark using the proposed wait-free queue on 72 threads.

We provide a bash script `driver` to run a binary repeatedly to collect the mean running time across several invocations. For each run, `driver` reports the mean of previous running times, the running time of the current run, the standard deviation, and the margin of error of the 95% confidence interval (both in absolute value and percentage). `driver` runs a benchmark at least 5 times and terminates when the margin of error drops below 2% or executes the benchmark 10 times. For example, `./driver ./wfqueue 72` will run `./wfqueue 72` repeatedly.

We also provide a bash script `benchmark`. Given a list of binaries and a list of numbers of threads to run, it invokes `driver` on all combinations of binaries and numbers of threads. The `benchmark` script reports the mean running time and the margin of error of each combination. The list of binaries and the list of numbers of threads are specified using environment variables `TESTS` and `PROCS` respectively. For example, `TESTS=wfqueue:lcrq:faa PROCS=1:2:4 ./benchmark` runs each of `wfqueue`, `lcrq`, and `faa` using 1, 2, 4 threads.

Compute throughput To compute the throughput of each run of a benchmark, one should subtract the time spent in the delay routines that are injected between enqueue and dequeue operations to avoid artificially long run scenarios. The time spent in the delay routines are measured using the synthetic benchmark `delay`. For example, to compute the time spent in the delay routines in `wfqueue` on 72 threads, we subtract the running time of `delay` on 72 threads.

By default, each benchmark executes 10^7 pairs of enqueue and dequeues, which is 2×10^7 operations. We calculate the throughput in $Mops/second$ using $20000/T_{mean}$, where T_{mean} is the mean of the running time of a benchmark measured in milliseconds.

A.4 Evaluation and expected result

It is unrealistic to replicate the exact experiment results in the paper without running the benchmarks on equivalent systems. To reproduce the results on a machine, perform the following steps:

1. download and unzip the framework on an x86_64 system or an IBM Power7 system,
2. compile all code with GCC 4.8.2 with -O2 optimization level and link all binaries with `jemalloc`,
3. execute `benchmark` with all benchmarks and number of threads from 1 to the maximum available threads, and
4. compute the throughput of each run.

The precise performance of each queue varies when running on different machines. However, the performance trend of each queue should remain the same. On a single core, we expect the tested queues' throughput to be: `faa` $>$ `ccqueue` $>$ `wfqueue` $\approx$ `wfqueue0` $>$ `lcrq` $>$ `msqueue`. On a large number of cores, we expect the tested queues' throughput to be: `faa` $\geq$ `wfqueue` $\approx$ `lcrq` $>$ `wfqueue0` $>$ `ccqueue` $>$ `msqueue`.