Lock Implementations

Minke Zhang

February 21, 2014

1 Modules

Locks

We wish to abstract away the specifics of the lock interface so that at lock invocation, we do not need to worry about how to handle each type of lock. As such, we will allow a lock struct to deal with the complexities of each lock, and to provide a standard interface for the experiments.

```
struct lock_t {
  int type;             // the type of lock being used
  int status;           // a copy of the lock status
  void *l;              // the lock
} lock;

lock *init_lock(int type);
void lock(lock *l) {
  switch(l->type) {
    ...
  }
  l->status = 1;
}

int try_lock(lock *l) {
  return(status);
}

void unlock(lock *l) {
  switch(l->type) {
    ...
  }
  l->status = 0;
}
```

Experiment Interfaces

What follows is the expected interface for experiments– the hope being that if all else is sufficient, we will simply alter the given variables and

```
int time_counter_serial(int M);                          // returns the counter value at
```

```
                                                    //   M seconds
int time_counter_parallel(int M, int n, int L);     // returns the counter value at
                                                    //   M seconds, with n threads
                                                    //   and lock type L

float work_counter_serial(int B);                   // returns the time taken for
                                                    //   the counter to reach B
float work_counter_parallel(int B, int n, int L);   // returns the time taken for
                                                    //   the counter to reach B,
                                                    //   with n threads and lock
                                                    //   type L

void packet_serial(int M, int n, long W, int        // same as the serial version
  uniform_flag, short experiment_number);           //   of the last project, but
                                                    //   without specifying T
void packet_parallel(int M, int n, long W, int      // parallel version, with (non)-
  uniform_flag, short experiment_number, int D,     //   locking dequeues and load
  int L, int S);                                    //   balancing strategy S
```

2 Hypotheses

TaS, exponential, mutex, anderson, CLH

Idle Overhead

We would expect that the overhead generated by the TAS and backoff lock will not significantly slow down the parallel counter– the length of the uncontested path is a simple straight walk through the `lock()` function, whereas the queue-based locks will require memorry allocation per lock call, and as such, should incur a higher overhead penalty. The POSIX lock implementation is, as per StackOverflow (http://bit.ly/MHnDdX), a userspace increment-and-test implementation, which is hardware instruction based; as such, we would expect the performance of the POSIX implemenation to lie closer to TAS at $n = 1$ than to the queue-based locks.

Lock Scaling

As was mentioned in class, we would expect that the TAS and backoff locks to behave rather poorly under heavy contention, whereas the queue-based locks are much better at adjusting to the heavier congestion, as the queue-based locks avoid in large part the expensive cache misses which occur for conventional locks.

Fairness

We would expect that the TAS to have a much higher deviation than the queue-based locks, due to possibilities of starvation of certain threads in cases of heavy contention, whereas the queue-based locks should have a small deviation due to the way in which lock requests are being made.

Packet Overhead

lock	R_T	R_W
test and set	0.108	0.020
backoff	0.100	0.021
pthread mutex	0.132	0.025
Anderson	0.221	0.045
CLH	0.200	0.044

Table 1: Performance ratios for the parallel time-limit based counter (R_T) and parallel work-limit based counter (R_W) for a variety of lock schemas, as compared to the serial counters.

We believe there will be more overhead generated due to the homequeue lock implementation– because the nature of locks, we are serializing a process which was before perfectly fine being modified concurrently. Thus, we would expect the worker rate to be lower for implementations of the firewall utilizing locks than using the lock-free queue. Moreover, we would expect that the relative overhead between lock implementations to mirror that of the **idle overhead**.

Packet Scaling

We do not believe there to be a significant speedup while using locks in the uniform packet case– the use of locks for the firewall is to provide a way for multiple threads to call `dequeue()` on a queue simultaneously. This is only useful for the case of load-balancing, which will only be a major consideration in the exponentially distributed packet case. We will therefore expect the *awesome* strategy to emphasize high number of threads using the queue-based lock, and with some work-stealing scheduling implemented.

3 Data & Analysis

Idle Overhead

We see from Fig. (1) that there is in fact a *better* ratio for the more intricate locks (i.e., Anderson and CLH) than for the simpler locks by a factor of ~ 2 (a figure which is consistent in the work-limited counter). Keeping in mind this is the uncontested lock, the shortest path in these implementations is the TAS lock, which we had expected to have the best performance for small n. We surmise this is due to the specfics of our implementation of the counters – after spawning in the worker thread(s), the parallel counter dispatcher then sits in a loop testing the completion status of the child threads – the spin here consists of a `sched_yield()`. It may be that due to the shortened length of the uncontested lock, `sched_yield()` is being called more than is optimal. To test this, we would need to rewrite the dispatcher to handle the task completion in another way – perhaps by calling `nanosleep()` in a fine-grained manner.

Lock Scaling

NB: The backoff lock is tuned to `MIN_DELAY` = 1ms, `MAX_DELAY` = 16ms.

We see a steep falloff by the TAS lock as n increases, and that, surprisingly, the backoff lock surpasses the mutex lock in performance in the time-limit counter – the pthreads mutex has performance peak at $n = 2$, dropping off as the backoff lock hits a performance peak at $n = 8$. This can be explained by the fact that the pthread lock is expected to work across a wide variety of architectures, whereas the backoff lock

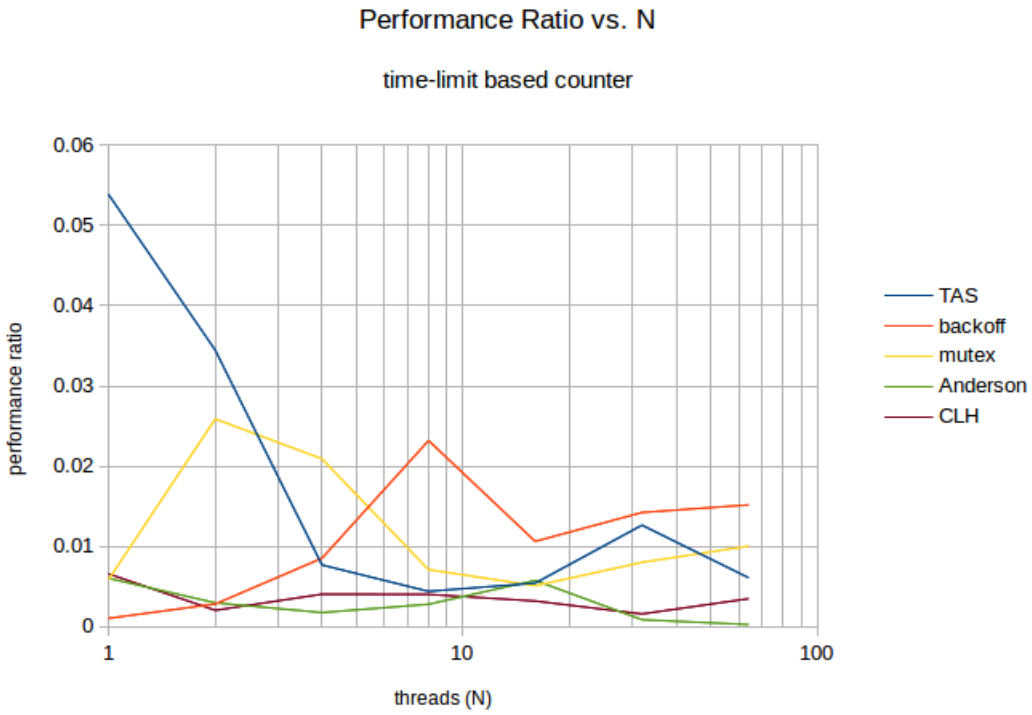

Figure 1: Performance ratios for the parallel time-limit based counter for different lock schemas, as compared to the serial implementation.

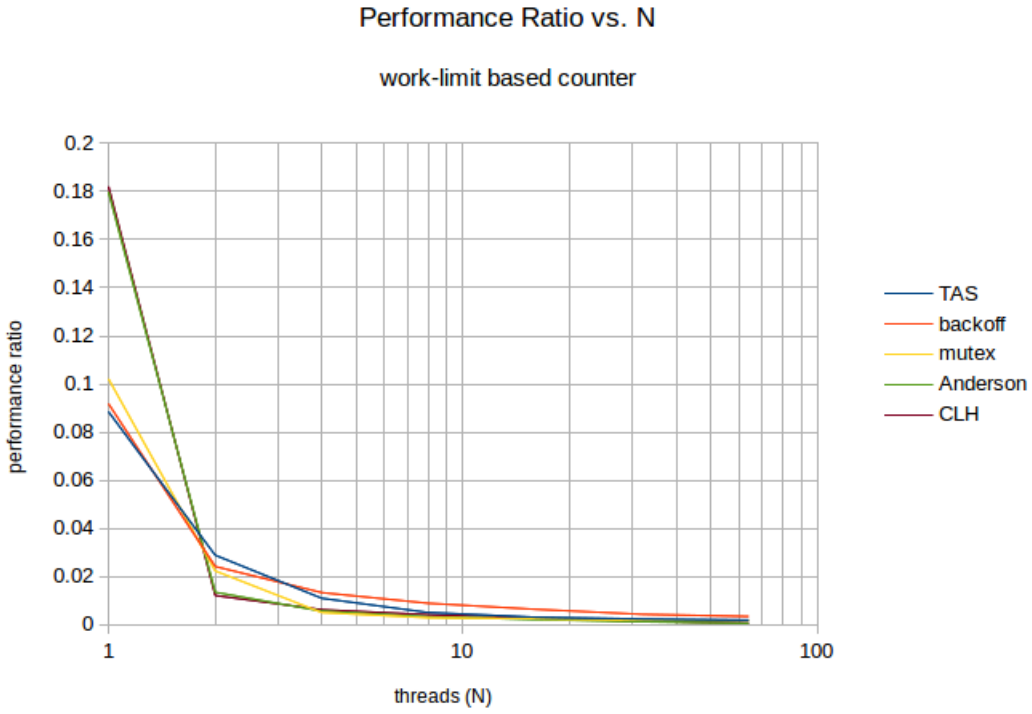

Figure 2: Performance ratios for the parallel work-limit based counter for different lock schemas, as compared to the serial implementation.

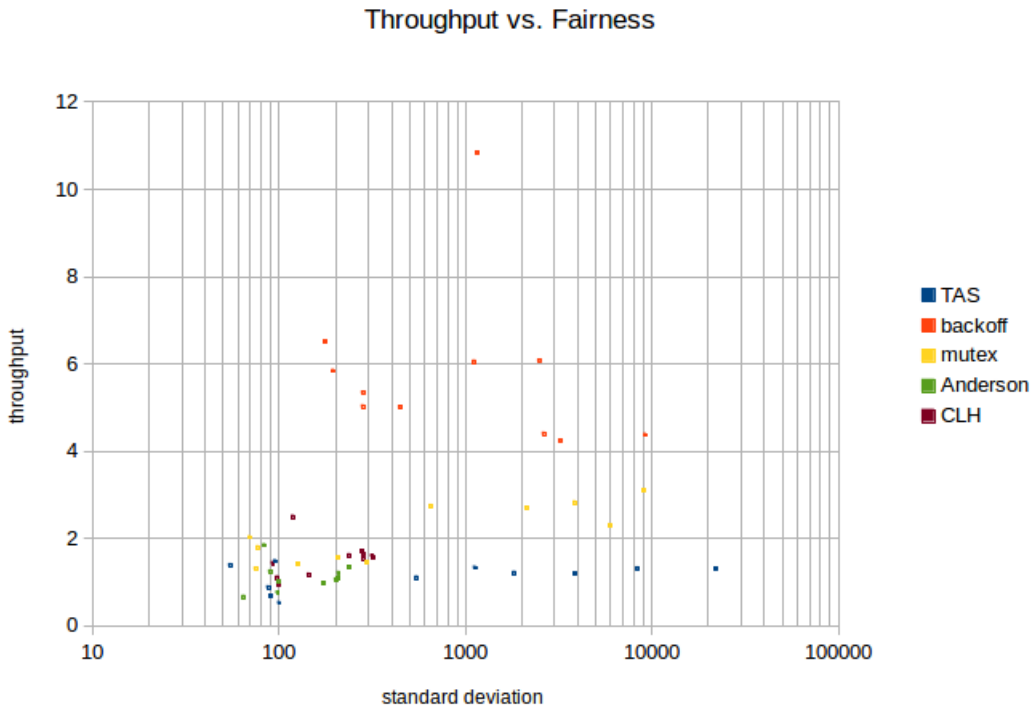

Figure 3: Throughput as a function of the standard deviation of worker contributions in the time-limit based counter.

is tuned per hardware – as a result, we would expect that the tuned backoff to perform favorably in this condition.

Fairness

As expected, we see that the Anderson and CLH lock implementations, though quite poor in performance, ensures that threads are accessed fairly – this is in contrast to spread of the standard deviation of the TAS, mutex, and backoff locks, which has a spread of standard deviation values across several orders of magnitude. We note here that, as was implicit in the previous experiments, the backoff lock hits the highest performance peaks (with a performance rati of ~ 11 at the function maxima). We also see that the cluster of mutex data points has a spread similar to that of the TAS lock (i.e., the wide, even distribution of standard deviations, as well as the relative constant throughput factor), suggesting that this particular pthread implementation may be an optimized TAS lock.

Packet Overhead

An interesting characteristic of our implementation of this project as seen in Fig. (4) is that the `HomeQueue` (i.e. uncontested lock) has a performance boost at low W when compared with the `LockFree` implementation of the firewall – as was the case with the **idle overhead** experiment results, we hypothesize that this is due to the way in which our firewall reacts to handles waiting by sleeping for a set amount of time – the extra overhead afforded by the locks may be enough of a processing buffer, such that the worker queue already is filled by the time that the worker attempts to `deq()`. We also see that as expected, the performance overhead created by the locks approaches zero as the time spent outside the locks (i.e. processing the packet fingerprint) increases due to the increased workload W.

Figure 4: Performance ratios for the `HomeQueue` firewall as compared to the `LockFree` implementation, as a function of the work involved per packet.

Packet Scaling

Qualtitatively, we see that as W increases, locks become much more feasible as n increases - we note in general the performance hit at low W for $n = 16$, resulting in the arch-like shape; this changes to the much more linear features at $W = 8000$, resulting in a peak speedup of ~ 12 for 15 workers in the uniform case.

4 Testing

The crux of the issue here, assuming the firewall was sufficiently tested in the last project, is the ability for each lock to serialize some set of operations. The simplest– and perhaps the most effective– method for testing this is to spawn n threads making m incrementing calls each on a shared resource, and test to see at the end if the shared resource has been altered $n \times m$ times.

In particular, we have implemented two test suites – one for testing locks, and one for testing the correctness of the the parallel packet model.

```
int test_strategy(int strategy) {
  ...
  int success = (serial->fingerprint == parallel->fingerprint);
  ...
}
```

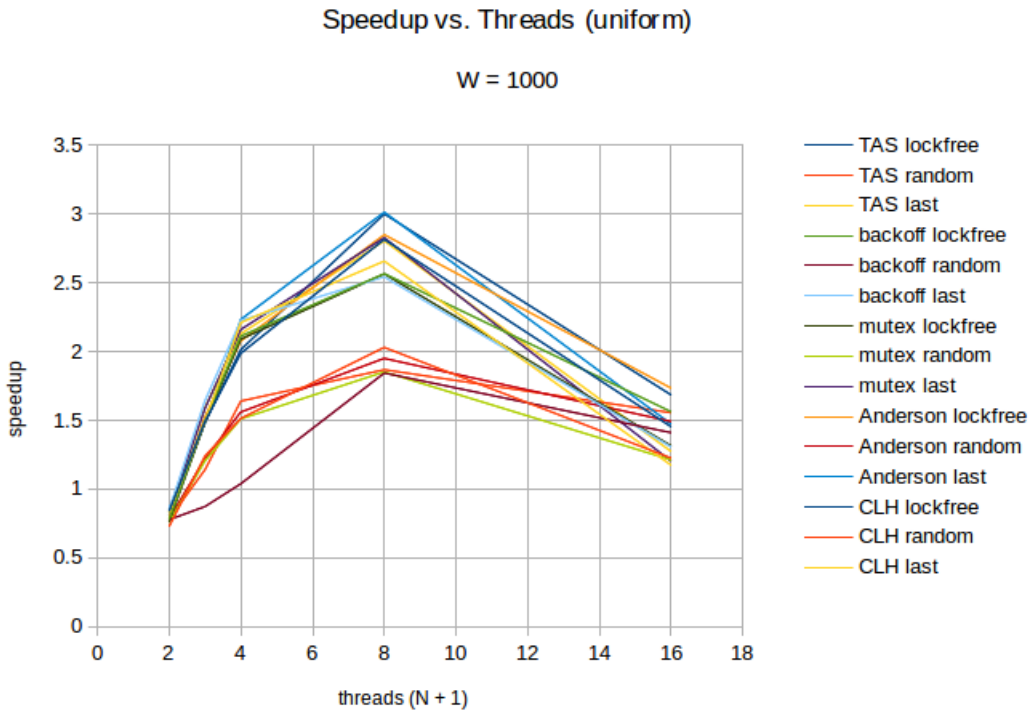

Figure 5: Performance ratios for parallel firewall, with uniform packet distribution at mean work $W = 1000$.

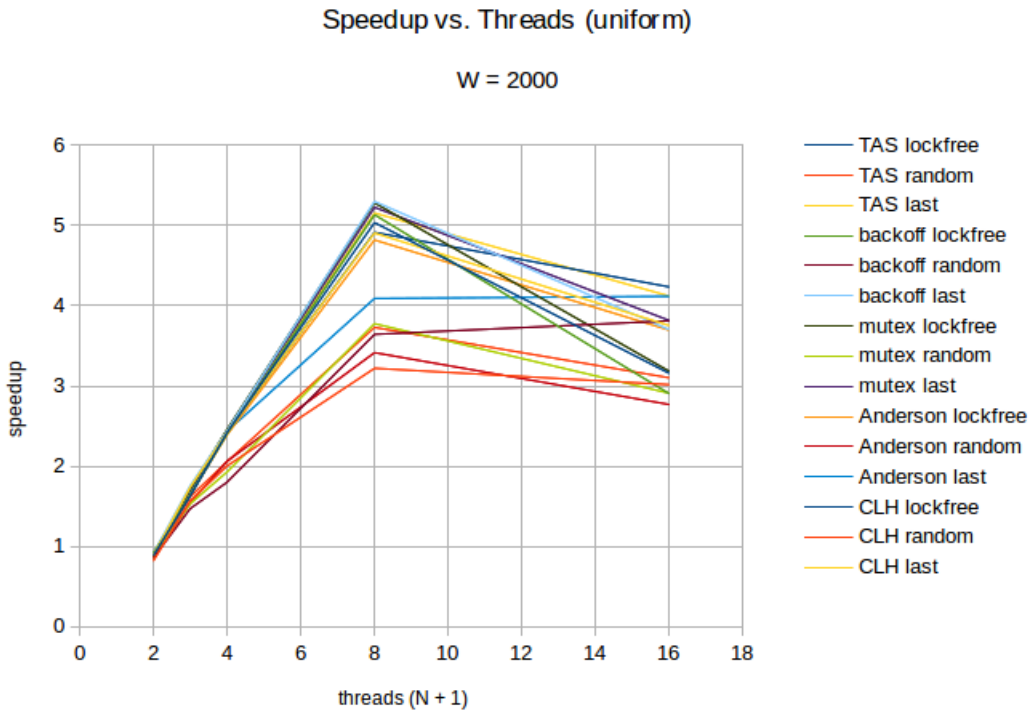

Figure 6: Performance ratios for parallel firewall, with uniform packet distribution at mean work $W = 2000$.

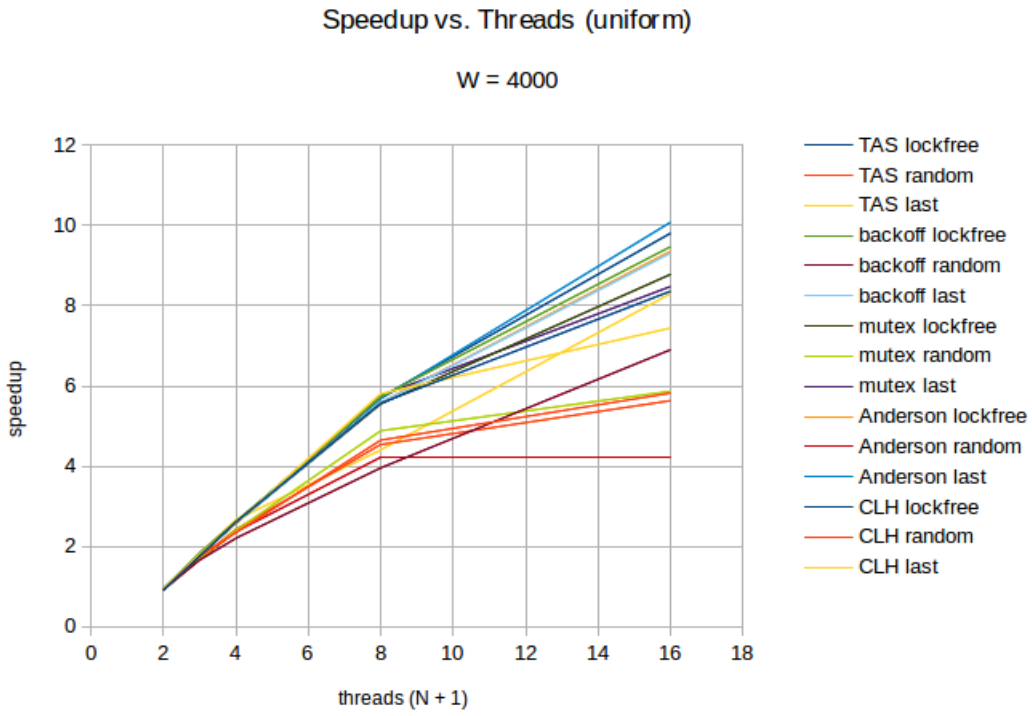

Figure 7: Performance ratios for parallel firewall, with uniform packet distribution at mean work $W = 4000$.

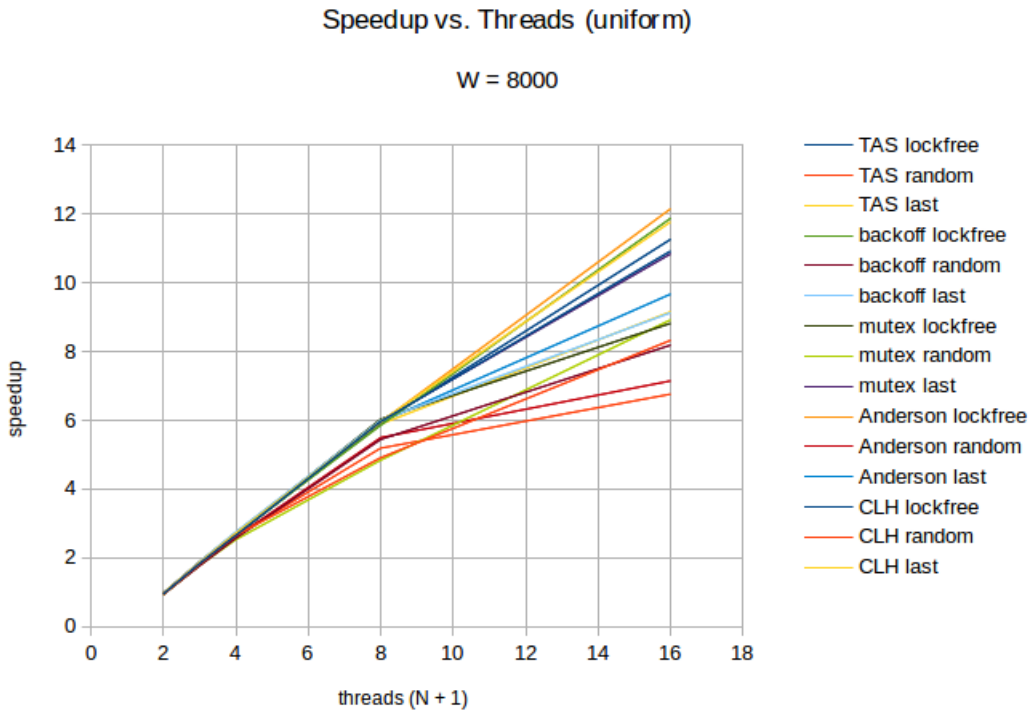

Figure 8: Performance ratios for parallel firewall, with uniform packet distribution at mean work $W = 8000$.

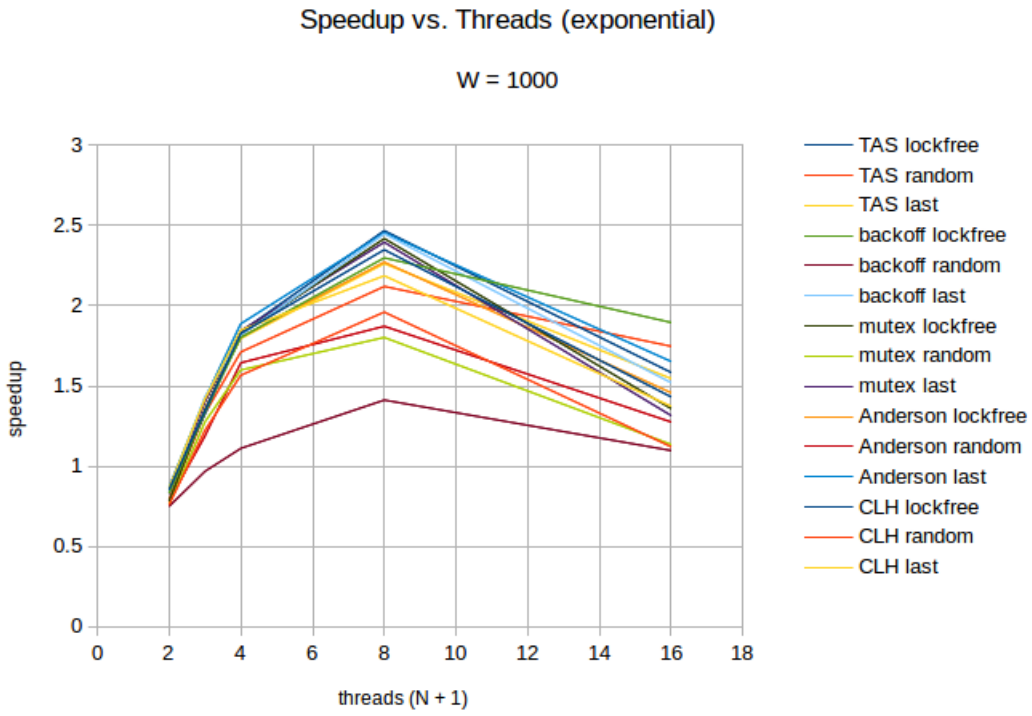

Figure 9: Performance ratios for parallel firewall, with exponential packet distribution at mean work $W = 1000$.

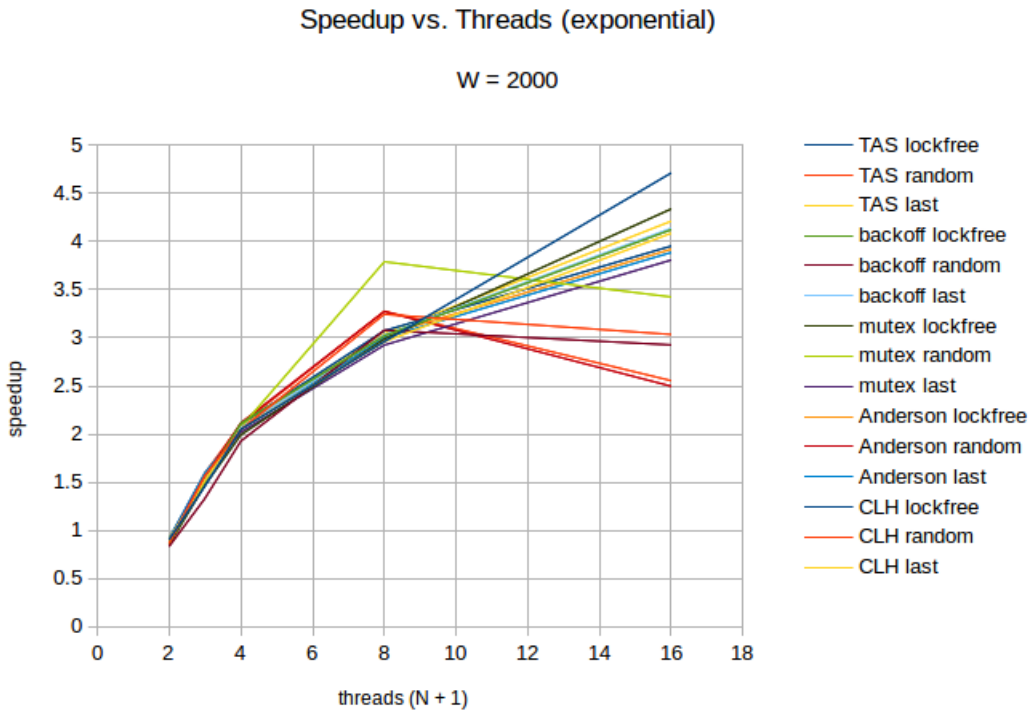

Figure 10: Performance ratios for parallel firewall, with exponential packet distribution at mean work $W = 2000$.

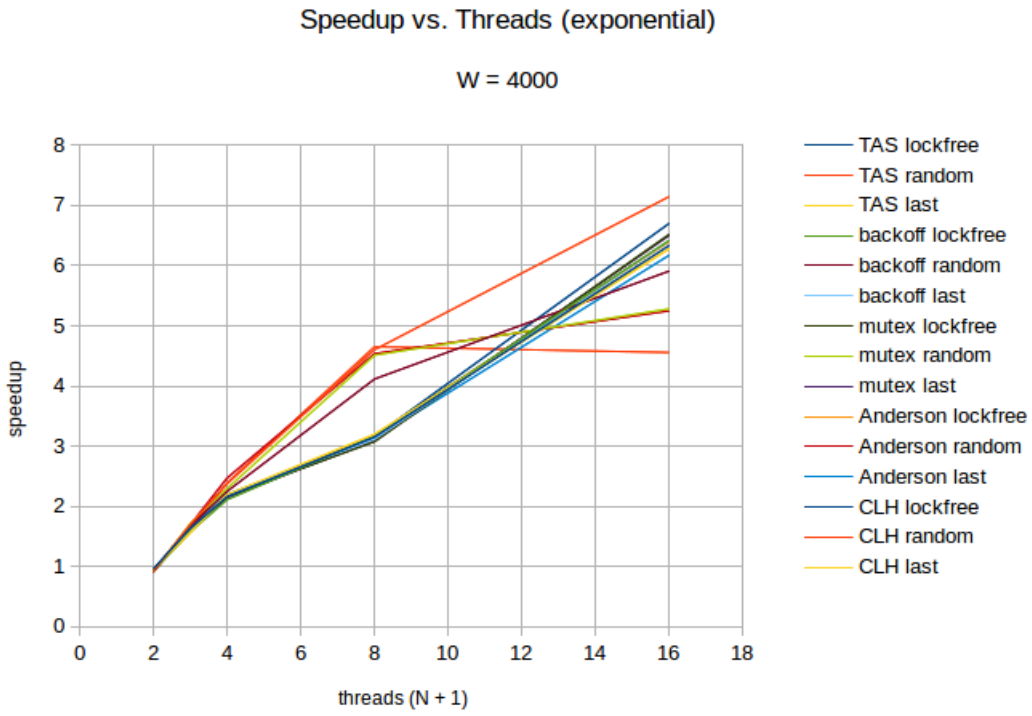

Figure 11: Performance ratios for parallel firewall, with exponential packet distribution at mean work $W = 4000$.

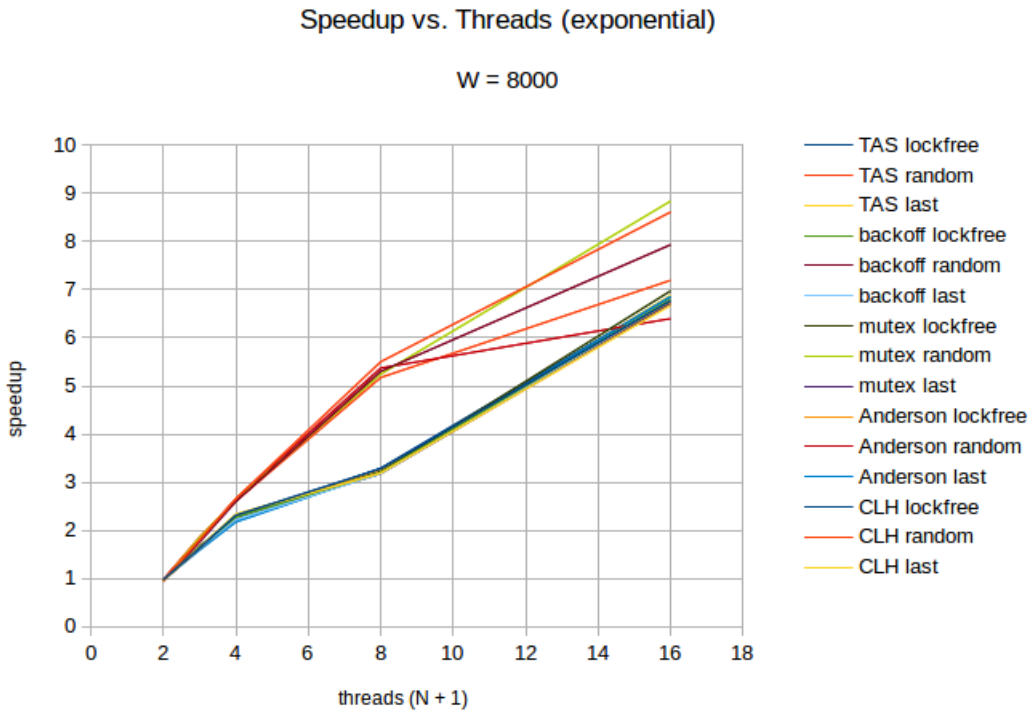

Figure 12: Performance ratios for parallel firewall, with exponential packet distribution at mean work $W = 8000$.

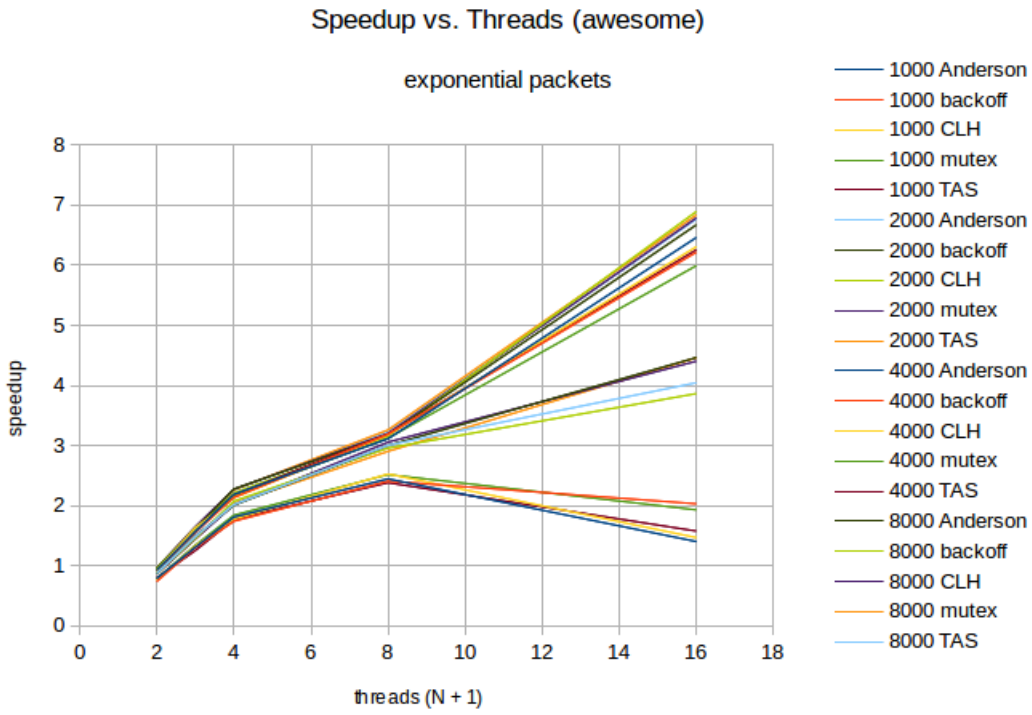

Figure 13: Performance ratios for parallel firewall, with exponential packet distribution and with the awesome scheduling strategy.

```
int test_lock(int type) {
  ...
  int success = (counter->total == COUNTERS * COUNTER_INCREMENT);
  ...
}
```