CMSC 23000 Project 3: Final Design and Results

Rachel Hwang

February 20, 2014

1 Final Design

My implementation draws from the following modules:

- **locks.c/locks.h**
  A library containing all lock functions
- **serial_time_counter.c**
  An application taking a single time argument as run duration, spawning a single thread to increment a counter.
- **parallel_time_counter.c**
  The parallel version, spawning n threads which each increment under the supervison of a given lock type.
- **time_counter.c/time_counter.h**
  The substance of all time counter programs.
- **serial_work_counter.c**
  An application taking a single argument as a counter target value (BIG), spawning a single thread to increment up to that value.
- **parallel_work_counter.c**
  The parallel version, spawning n threads, each of which increment the counter (BIG)/n times.
- **work_counter.c/work_counter.h**
  The substance of all work counter programs.
- **serial_packet.c**
  Runs the firewall checksum application from project 2.
- **parallel_packet.c**
  The parallel version, again similar to project 2, but taking lock and queue picking stategy arguments.
- **packets.c/packets.h**
  Th substance of all packet programs, build off the firewall program, including new enqueue and dequeue functions.

Programs will also use a number of the provided utility modules, such as the stopwatch and packet gen.

1.1 Locks

My implementation of the five locks changed significantly in that each lock takes only a single lock argument. Given that this project requires many variations on similar ideas, I attempted to introduce greater abstration into my program. The new lock functions are as follows, for each type, a lock, unlock and a try method:

```
/* From "locks.c" */
#include "locks.h"

#define MAX_DELAY 256

/* TAS lock functions */
void tas_lock(volatile lock_t *lock) {
  while (__sync_lock_test_and_set(lock->tas, 1)) {}
}

void tas_unlock(volatile lock_t *lock) {
  *(lock->tas) = 0;
}

// Attempts to acquire lock. Returns 0 on success.
int tas_try(volatile lock_t *lock) {
  return __sync_lock_test_and_set(lock->tas, 1);
}



/* Exponential Backoff Lock functions */
void backoff_lock(volatile lock_t *lock) {
  double time;
  double backoff = 0;

  while(1) {
    if (!__sync_lock_test_and_set(lock->tas, 1)) {
      return;
    }
    else {
      time = fmin((rand()/(double)RAND_MAX)*pow(2, backoff), MAX_DELAY);
      backoff++;
      usleep(time);
    }
  }
}

void backoff_unlock(volatile lock_t *lock) {
  *(lock->tas) = 0;
}

// Attempts to acquire lock. Returns 0 on success.
int backoff_try(volatile lock_t *lock) {
  return __sync_lock_test_and_set(lock->tas, 1);
}



/* Mutex lock functions */
void mutex_lock(volatile lock_t *lock) {
  pthread_mutex_lock(lock->m);
}

void mutex_unlock(volatile lock_t *lock) {
  pthread_mutex_unlock(lock->m);
```

```
}

// Attempts to acquire lock. Returns 0 on success.
int mutex_try(volatile lock_t *lock) {
  return pthread_mutex_trylock(lock->m);
}



/* Anderson lock functions */
void anders_lock(volatile lock_t *lock) {
  int idx = __sync_fetch_and_add((lock->a).tail, 4) \% (lock->a).max;
  while(!((lock->a).array)[idx]) {}
  *((lock->a).head) = idx;
}

void anders_unlock(volatile lock_t *lock) {
  int idx = *((lock->a).head);
  ((lock->a).array)[idx] = 0;
  ((lock->a).array)[(idx + 4) \% (lock->a).max] = 1;
}

// Attempts to acquire lock. Returns 0 on success.
int anders_try(volatile lock_t *lock) {
  int max = (lock->a).max;
  volatile long *tail = (lock->a).tail;

  if ((lock->a).array[*tail \% max]) {
    anders_lock(lock);
    return 0;
  }
  return 1;
}


/* CLH lock functions */
node_t *new_clh_node() {
  return (node_t *)malloc(sizeof(node_t));
}

void clh_lock(volatile lock_t *lock) {
  (lock->clh).me->locked = 1;
  (lock->clh).pred = __sync_lock_test_and_set((lock->clh).tail, (lock->clh).me);
  while (((lock->clh).pred)->locked) {}
}

void clh_unlock(volatile lock_t *lock) {
  volatile node_t *tmp = (lock->clh).pred;
  ((lock->clh).me)->locked = 0;
  (lock->clh).me = tmp;
}

int clh_try(volatile lock_t *lock) {
  if ((*((lock->clh).tail))->locked) {
    return 1;
  }
  clh_lock(lock);
  return 0;
}
```

The advantage of this greater abstraction is that it allows me to pass function pointers to the worker thread functions, thus having a single worker function for each program rather than one for each type. Thread arguments contain a lock type which is a union of all the lock types. All lock arguments are initialized in a programs respective dispatch thread, via a switch statement for different types. Apart

from these changes to the infrastructure, the design of the counter programs is largely unchanged from the original design. The thread argument struct for counter programs is given below:

```
typedef struct thr_data_t{
  volatile long *counter;
  volatile lock_t *locks;
  volatile int my_count;
  void (*lock_f) (volatile lock_t *);
  void (*unlock_f) (volatile lock_t *);
} thr_data_t;
```

1.2 Packets

My packets programs are slightly more complex. Built from the firewall project 2, rather than a single thread dequeue function (calling the single dequeue function), parallel_packets.c has one different thread function for each queue selection strategy. Like the counter programs, threads take a thread argument struct that contains function points of the appropriate type. The dispatch thread is responsible for calling the spawn function, holding all thread arguments in static memory, enqueing to all queues and spawning a timekeeper thread whose function is to sleep for a specified amount of time, then set a global flag to 0 which will halt all threads. A sample worker function:

```
\\Home queue strategy
void *homeq(void *args)
{
  pack_data_t *data = (pack_data_t *)args;
  SerialList_t **q = data->queue;
  long int *fp = data->fingerprint;
  volatile lock_t *locks = data->locks;
  void (*lockf)(volatile lock_t *) = data->lock_f;
  void (*unlockf)(volatile lock_t *) = data->unlock_f;
  int i = data->id;

  while(go) {
    (*lockf)(locks+i);
    (data->my_count) += dequeue(q[i], fp+i);
    (*unlockf)(locks+i);
  }
  pthread_exit(NULL);
}
```

1.3 AWESOME lock

I noticed that Last-Queue consistently outperformed other queue strategies, particularly where packet size was large or in the case of exponential load. Random, on the other hand, was significantly worse than other methods, since it partially eliminates benefits such as backoff, or cache-conflict avoidance. In order to leverage off the benefits of Last Queue, that is, the savings that come from moving on to another lock rather than waiting (logically, this is ideal for our program since it is impossible for one thread to be unoccupied while all queues are locked since there is only one thread per queue. Trying several locks is likely to acquire a queue faster than waiting on a particular one.), AWESOME is based off of Last Queue. However, rather than using a random id each time, awesome begins at its person id (as an i value), tries the lock, and if it is occupied, proceeds sequentially. Based on the performance of random queue, this will be an improvement over Last Queue.

2 Experiments

1. **Idle Lock Overhead 1**
   Hypothesis : Since the serial program has strictly less overhead, the speedup of the parallel is bound to be less than one. I predicted that TAS would perform the best, followed by BACKOFF, and that ALOCK and CLH would have even greater overhead, with CLH performing the worst.

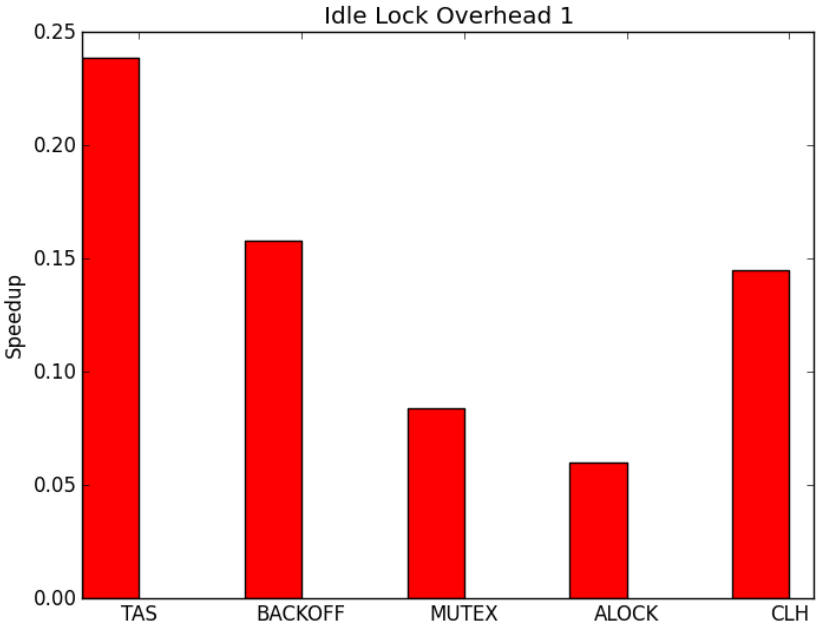

Figure 1:

Results : TAS did indeed have the least overhead cost, with Backoff as second best. This makes sense given that both locks operated using the same logic and necessary resources (a state flag), but Backoff has a few extra features, notable an if statement in the body of the lock function. If statements are costly, and even the few extra operations (also negating the fetched value) incurred some overhead. Mutex is a much more robust lock designed to operate successully in a variety of contexts. As a more complicated lock, it performs more operations in a call to lock-/unlock than the simple TAS locks. As predicted, ALOCK performed amoung the worst, which is unsurprising given its initialization requirements: the array and filling with flags, etc. It also requires the calucation of an index with every grab of the lock, using a handful of operations there. The suprising entry was defintiely the CLH lock. While I initially expected CLH to incurr the most cost, it outperformed ALOCK by an order of magnitude. Since the primary difference betwen ALOCK and CLH are their locations in memory, I surmise that the better performance is due to avoidance of cache conflicts (despite the fact that ALOCK was inplemented with a buffer of 4*int size), as suggested in the textbook. Apart from having centralized/decentralized flags, their algorithms are essentially the same. Distributing the lock flags throughout memory avoids having two flags in the same cache line, which would mean that the cache would be voided whenever either of those flags were updated.

2. **Idle Lock Overhead 2**
   Hypothesis : I predicted that results for this experiments would be very similar to the previous

experiment, but with the role of time and counter value reversed. Since this is just a different method of measuring the same overhead, I expected all the lock performance relationships to be the same as describes above.

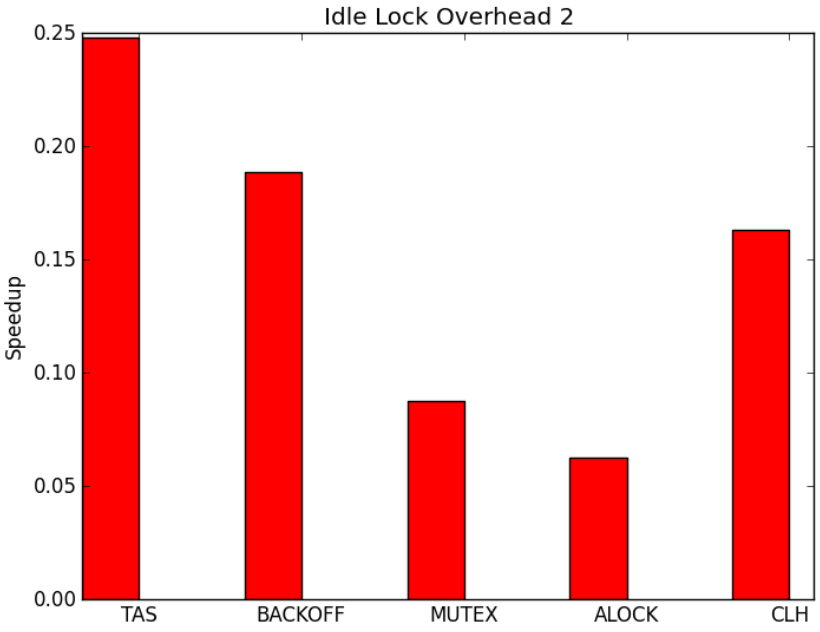

Figure 2:

Results: As predicted, the overhead relationships calculated using work rather than time are virtually identical.

3. **Lock Scaling 1**
Hypothesis : This experiment in essence measures how often threads compete for the same lock and the delay between successful lock grabs. By optimizing backoff time, we are reducing the probability that a thread will try to grab a locked lock. I predicted Backoff to scale better than TAS, and that ALOCK and CLH would scale the worst in this case.

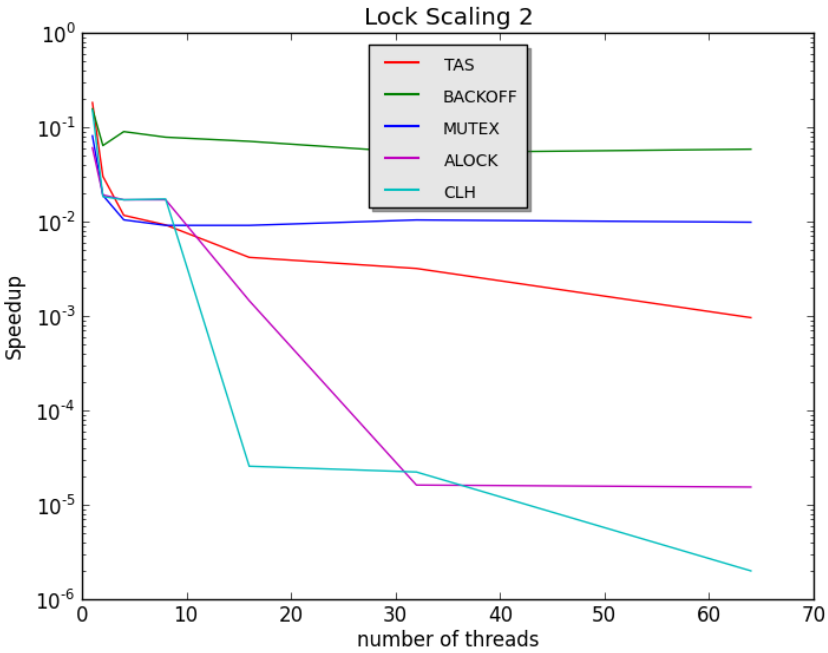

Figure 3:

Results: As predicted, ALOCK and CLH scaled significantly worse than the other locks, although the fact that they were worse by several orders of magnitude is very suprising. In this specific experiment, since all threads are competing for the same single lock and no work can be parallelized, it's only logical that the locks with more memory overhead scale the worst. As there are more threads, there is not only more contention for the lock, but for CLH and ALOCk, greater memory requirements: more storage needed and more memory locations being accessed at one, leading to very poor cache usage. Note also that the steepest dropoff in speedup comes at n = 16 (a total of 17 threads), since that means that not only must the processor schedule multiple cores, but multiple processes for each core. That amount of context switching is extremely expensive.

The rest of the locks scale better because they require no additional memory. No matter how many threads there are, TAS locks and Mutex require only a single lock structure, which explains the relatively flat line. Backoff outperforms the other locks because the more contention there is for a single lock, the more likely a large portion of threads will sleep before trying the lock again. In this case, having threads sleep is beneficial so that the holder of the lock may proceed uninterrupted. Reducing the number of active threads conpensates for contention potential as n grows, which is why Backoff's plot is especially flat.

4. **Lock Scaling 2**
   Hypothesis: As with experiments 1 and 2, experiments 3 and 4 measure the same aspects of performance using only slightly different metrics. Thus, I predicted that the lock performance relationships should be the same as in the previous experiment.

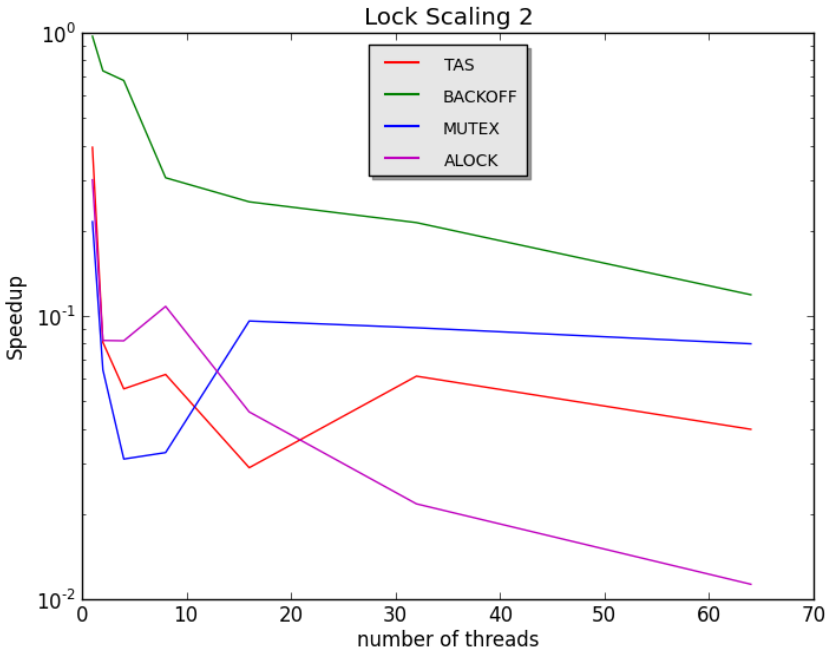

Figure 4:

Results : The lock scaling depicted here is very different from the previous experiment. Despite multiple runs of both experiments, the two lock scaling experiments produce significantly different graphs. Notice in this case, ALOCK still scales poorly, but, to a much less extreme degree than in the previous experiment. As before, ALOCK is the most constant speedup. I am still unsure about why time and work in this case yield different lock speedups. If this cannot be attributed to noisy data, I suspect it may be because different lock strategies may not operate at a steady pace. for instance, the first few uses of an anderson lock are likely to be cache misses, but cache hits thereafter. This "warm-up" period may cause very different relationships for certain time vs work values.

5. **Fairness**
   Hypothesis: I predicted that ALOCK and CLH would be extremely fair, while TAS and Backoff would be very unfair, with TAS as the most unfair.

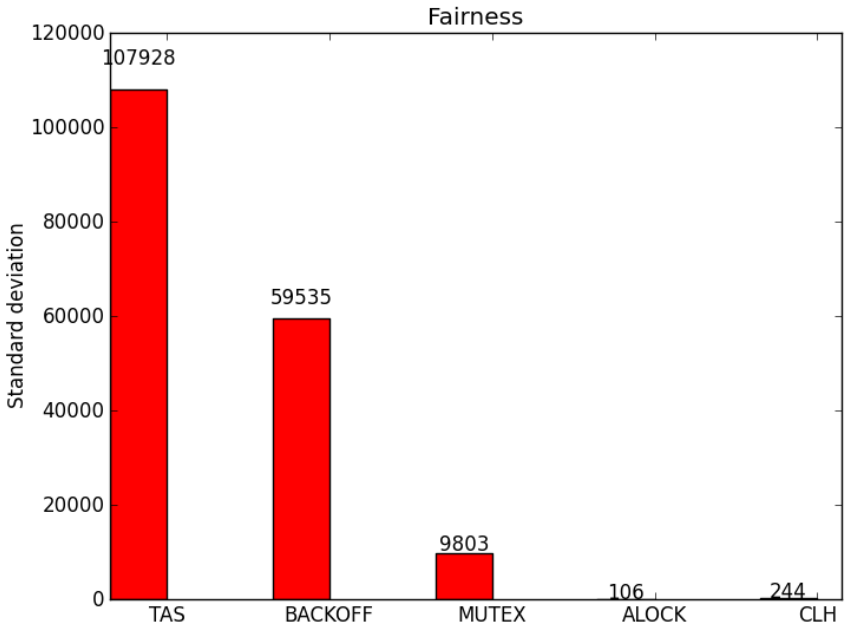

Figure 5:

Results: As predicted, the TAS locks are extremely unfair, having standard deviations of many orders of mmagnitude greater than either CLH or ALOCK. TAS has a SD 100x that of ALOCK! This general ordering makes sense as the TAS locks make abosolutely no guarentees of fairness, while CLH and ALOCK maintain a first-come-first-serve ordering. Backoff is more fair than TAS because the backoff reduces contention for the lock, and gives each thread the opportunity (at random) to acquire the lock relatively uncontested.

2.1 Packets

1. **Idle Lock Overhead**
   Hypothesis : HomeQueue will perform worse than LockFree since the programs will be identical save for lock overhead. The overhead observed should be virtually identical to that observed in the Lock Overhead experiments run previously; the ordering of lock overhead will be the same.

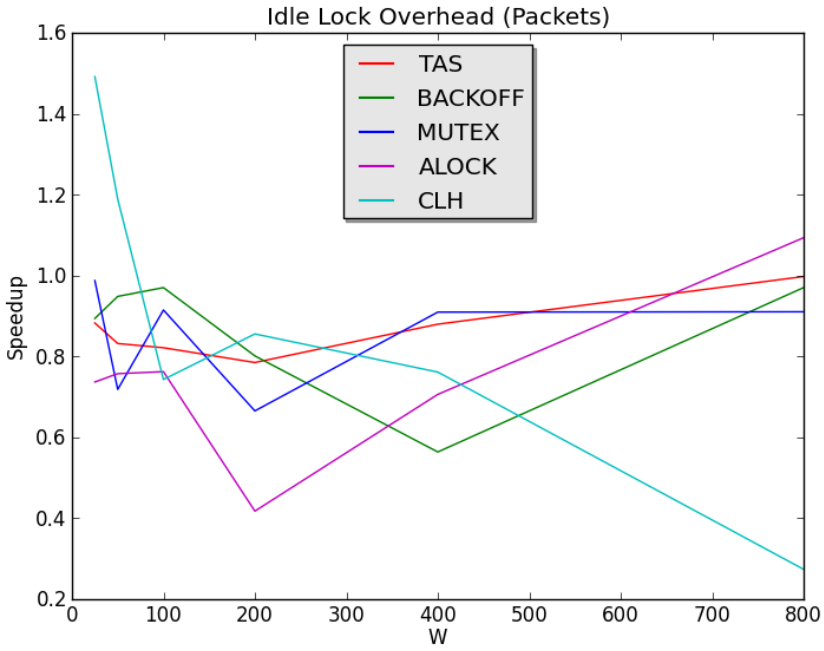

Figure 6:

2. **Speedup with Uniform Load**
   Hypothesis : I predicted that the lock free version to outperform the lock versions since the load inbalance will be small. The load balancing benefits of the other strategies will not compensate for the added overhead of using a lock. RandomQueue, I expected to perform better than LastQueue for small mean work per packet, since contention for locks will be frequent if each thread attempts to acquire locks in a set pattern. TAS may outperform Backoff, since the queue selected is already random. Inserting a delay will provide no benefits. Finally, LastQueue will significantly reduce the number of fruitless lock grabs, as it will prevent a thread from trying the same lock consecutively. Performance will increase with mean packet work, since larger W values means that a lock is more likely to be held for a longer period and thus consecutive checks are unlikely to help. This stratgey will render Backoff lock unhelpful. As in project two, I predicted that performance will peak at $n = 8$ and $W = 8000$.

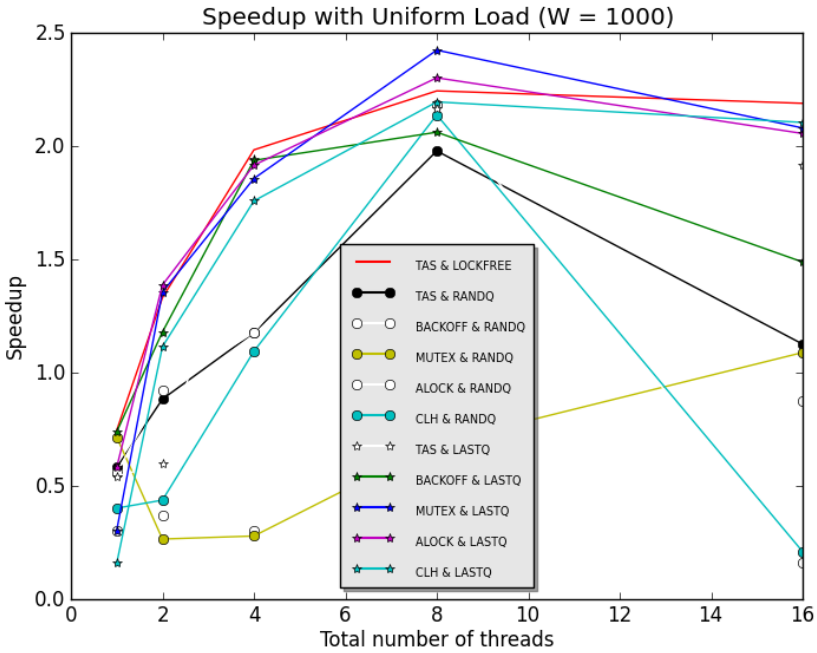

Figure 7:

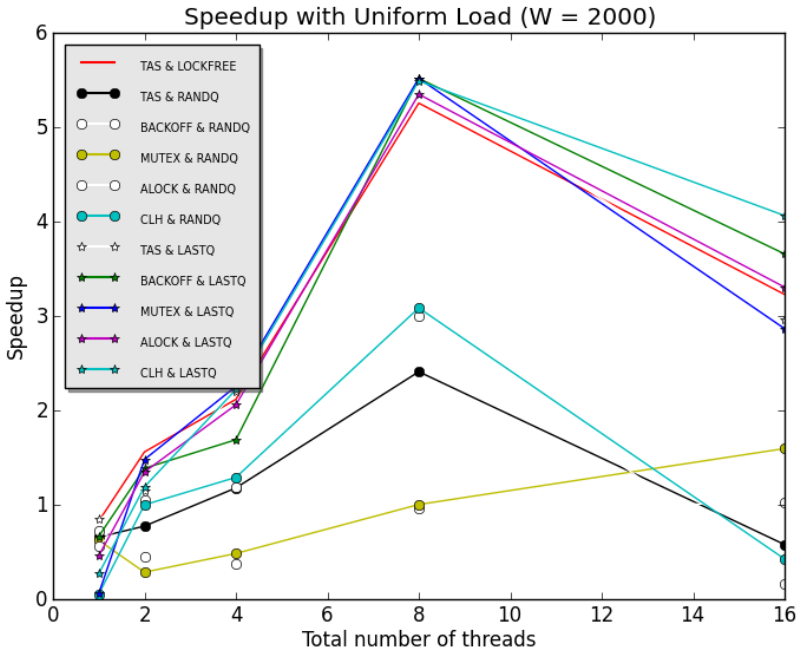

Figure 8:

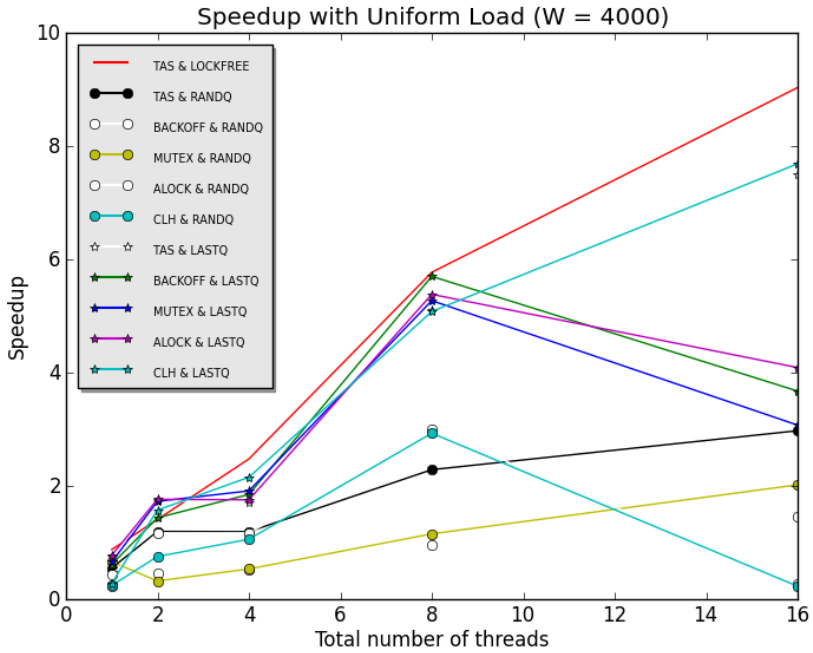

Figure 9:

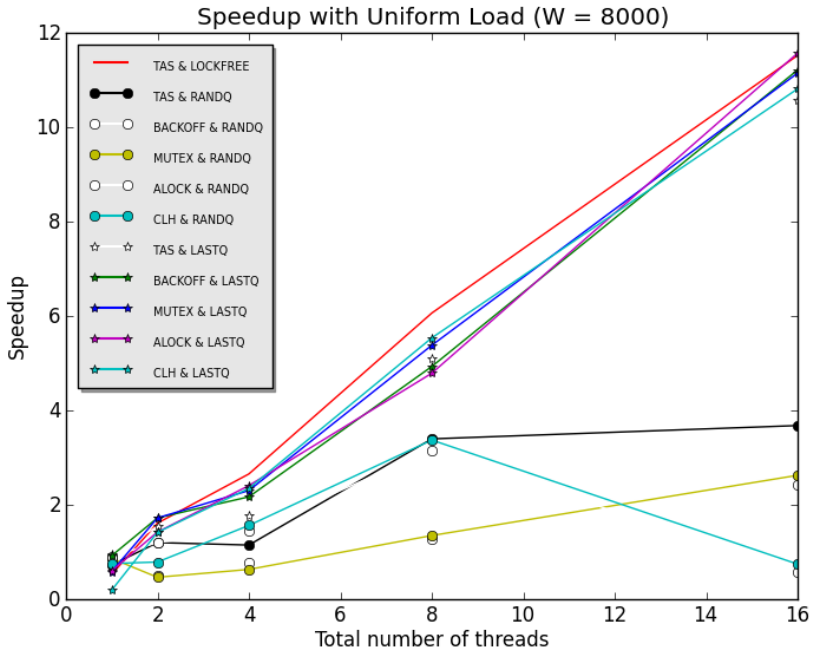

Figure 10:

Results: W does indeed have a large effect on speedup. Larger packets benefit more from parallelization, as the cost of potential contention given more threads is minimized since a thread i more likely to be doing work at any given time rather than grabbing for a lock. Maximum

speedup was about 12 at $W = 8000$ and $n = 16$. It's interesting that all smaller W values peak at 8 (note that this is a 16 core machine), suggesting that the overhead cost, particularly for the CLH and ALOCK locks, are significant and require a large load in order for their load balancing effects to become visible. CLH and ALOCK's perform better as W increases. Note however that the relative advantages of the lock types become difficult to observe.

As for queue selection, The lock free method consistently performed among the best, which is unsurprising given its lack of overhead cost and inherently balanced load. Random performed very poorly, although beyond that, trends are difficult to distinguish. Last Queue consistently performed very well, scaling with W. This makes sense since for larger W sizes, a lock is likely tobe held for a significnat period, making it unfruitful to wait rather than trying other locks. CLH with Last Queue had the best overall performance, while Backoff with Random Queue was the worst. Backoff loses all of its advantage when it jumps to an entirely different lock after waiting, becoming a slightly less efficient TAS lock (although in this case, the implementations are similar enough that more difference should be attributed to the random queue selection).

3. **Speedup with Exponential Load**
   Hypothesis Results should be similar to the previous experiment, except that load balancing is far more important in this scheme, so locked strategies will outperform the unlocked. In the exponetial scheme, the load will not be balanced by default. The gap between the locked and unlocked strategies should increases as W increases, since load imbalance with large mean load will lead to an even more unbalanced load.

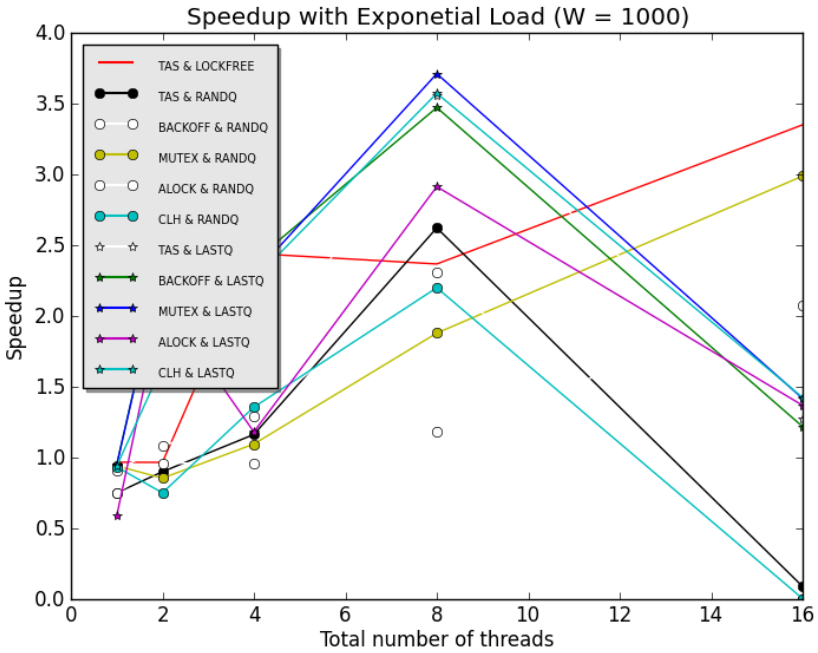

Figure 11:

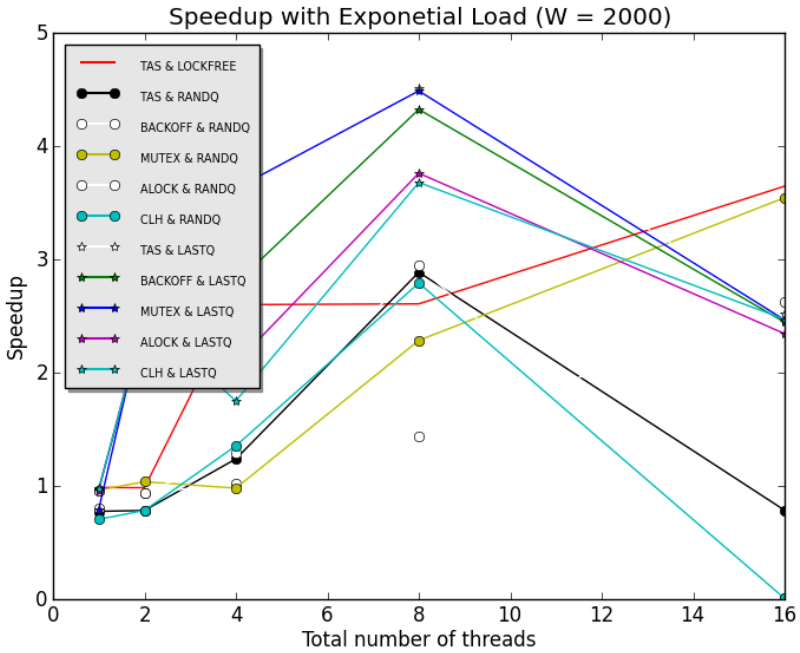

Figure 12:

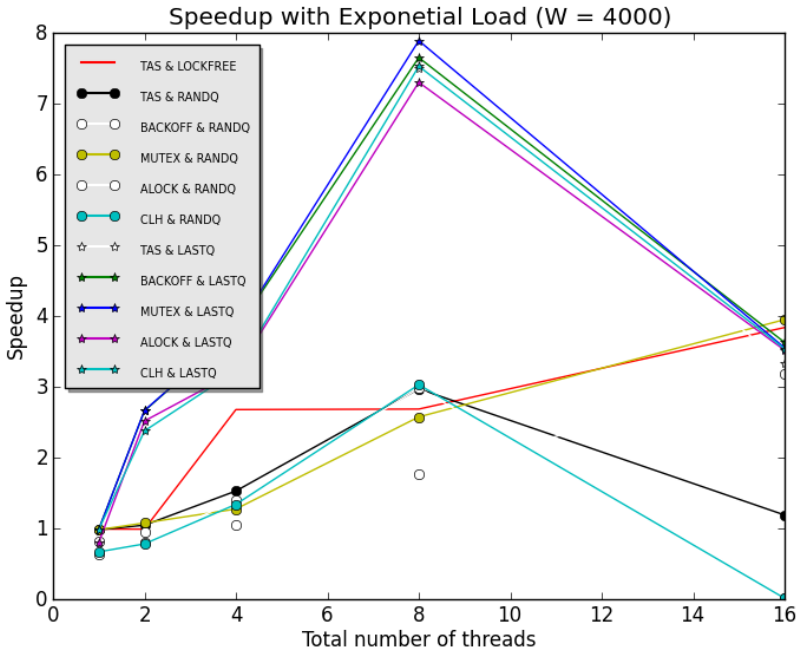

Figure 13:

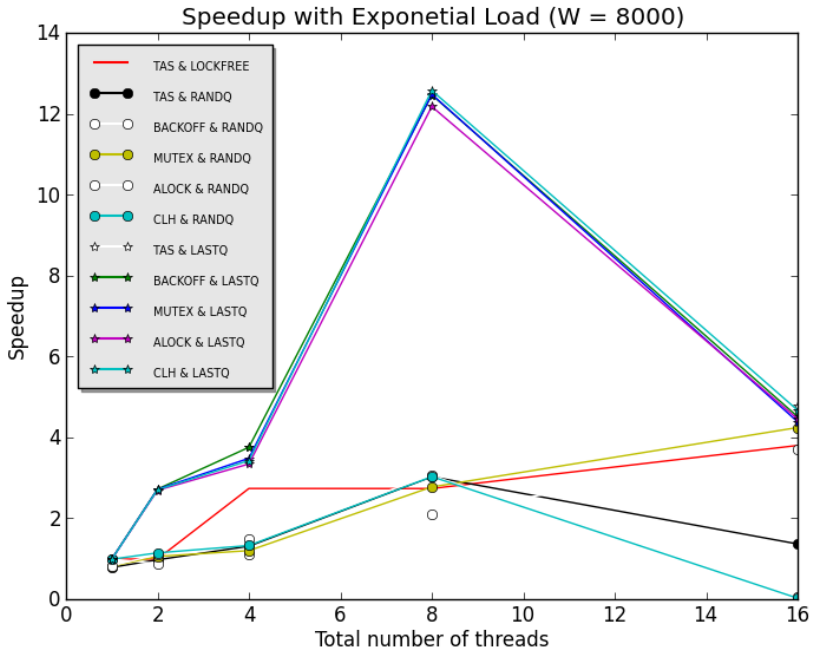

Figure 14:

Results: As predicted, better balanced loads performed better than the unfair by a signifiant margin that increased with packect size. For a packet size of $W = 1000$, CLH, ALOCK and MUTEX have the best performance with Last Queue. This holds up to $W = 8000$, where all locks except for TAS with Last Queue outperform the Random Queue locks by a factor of 6. We observe that queue-selection strategy can outweigh lock type: despite being very fair, CHL with the random strategy is among the worst performing. [IMPORTANT NOTE: Due with issues with the job queue, this experiment (and only this one) was performed on a cs-MacLab machine with 8 cores, rather than 16. This explains the sharp dropoff in performance for the high-performing strategies, since parallelization past a machines maximum cores tends to only incurr overhead cost. However, this does not explain the performance of TAS and LOCKFREE, increasing steadily beyond the maximum cores. I'm afraid this may be in part attributed to noise (see end note).

4. **Speedup with Awesome**
   Hypothesis : As discussed in the design section, I predict that having each thread start in a fixed location and try locks sequentially will afford a small speedup.

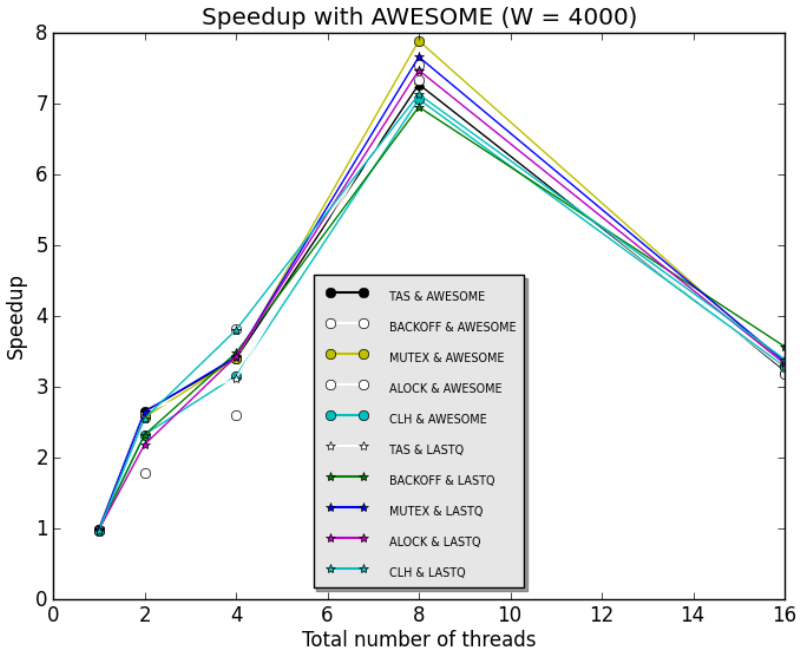

Figure 15:

Results : AWESOME does in fact provide a small speedup! The difference is only a small percentage of the total speedup, but at the peak performance, $n = 8$, AWESOME strategy outperforms Last Queue for every lock type. Trying locks sequentially may not necessarily guarentee finding a free lock in less steps than the random method in ever scenario, but given the case where only a few queues are free, because AWESOME tryies sequentially, we can be guarenteed not to miss one. Additionally, this method is far more likely to use the cahce optially, since locks may be stored in contiguous blocks of memory. Consider an array of TAS variables, or possible a variant of an anderson lock. Moving through memory in a systematic fashion can take advantage of memory properties, and may be much easier for the compiler to optimize, since a next step is always predetermined.