Project Report
Multicore Programming
05/09/2011

Concurrent Queues

George Poulos
gmp289@cs.nyu.edu

Simon Tso
st1450@nyu.edu

Introduction

The need for software concurrency has grown rapidly over the last decade. As raw processor speed has steadied over the last few years, the number of cores per processor has increased. This has further emphasized the importance of concurrent software. However, introducing concurrency into most programming models has not been effortless. There is a great deal of complexity involved in properly harnessing the power of multicore systems. One aspect of this complexity can be seen in concurrent data structures. Specifically, in this report will describe our design and report on our implementations of concurrent queues.

Queues in General

Queues are a elementary data structure. It is a container with first-in-first-out semantics. A Queue interface consists of two basic operations for adding and removing items into the queue. Here is the interface we used for our queues.

```
interface IQueue<T> {
  void Enqueue(T value);
  bool Dequeue(T *value);
}
```

Enqueue will add the value into the queue. Dequeue will attempt to remove a value from the queue. If the queue is empty, it will return false. Otherwise, it will fill in the supplied pointer with the dequeued value and return true.

A simple implementation of a queue uses a list of nodes and has a pointer to the head and the tail. When an enqueue is performed, the item is added to the back of the list and tail pointer is adjusted. Similarly, when a dequeue is performed the first item is removed from the list and the head pointer is adjusted. This is where the complexity in implementing concurrent queues begins. There is a need to perform two actions at once; to link/unlink the item and to adjust the head/tail pointers. We explored two implementations that allow for this behavior to occur concurrently.

As a note, a key factor that allows both our queue implementations to work properly is the idea of using a sentinel node inside the queue. Initially, the queue will start with a dummy node that allows each operation to blindly follow Next pointers without checking for null. Additionally, this allows each operation to only need either the head or tail pointer and not both.

Here is pseudo code for our simple queue implementation.

```
SimpleQueue<T>() {
  Node<T> Node = new Node<T>();
  node->Next = NULL;
  head = tail = node;
}
```

```
void Enqueue(T value) {
  Node<T>* node = new Node<T>();
  node->Value = value;
  node->Next = NULL;
  tail->Next = node;
  tail = node;
}
```

```
bool Dequeue(T* value) {
  Node<T>* node = head;
  Node<T>* next = node->Next;
  if(!next) return false;
  *value = next->Value;
  head = next;
  delete node;
  return true;
}
```

Locking Queues

Using a lock has been the most straight-forward and popular implementation in the past. Locks are somewhat simple compared to other implementations. A locking queue allows enqueues and dequeues to occur concurrently and safely. The downside to the lock is lock contention. Since a lock only grants access of each operation to one thread at a time, a large performance hit can occur when there are many threads trying to acquire the lock. Lock acquisition actually serializes each thread that is trying to perform an operation that another thread is also trying to perform.

Here is an example of adding a lock to the Enqueue operation for a locking queue.

```
void Enqueue(T value) {
  Node<T>* node = new Node<T>();
  node->Value = value;
  node->Next = NULL;
  lock(EnqueueMutex){
    tail->Next = node;
    tail = node;
  }
}
```

Lockless Queues

One way to avoid lock acquisition is not to use locks at all. An alternative to locks can be atomic operations. Using atomic operations requires a somewhat more complicated thought process. One powerful atomic operation is compare and swap (CAS). Using CAS, a lockless queue can be implemented. However, CAS is not a silver bullet. CAS has a few disadvantages up front. First, heavy use of CAS between threads will invalidate CPU caches. This can cause coherency traffic between the cores and tremendously decrease performance. Secondly, the data structure may fall into intermediate states during operations. It is therefore required to handle all intermediate states during the course of normal operation. Despite these shortcomings, it is possible to comply with these caveats and implement a concurrent queue using CAS.

In general, CAS operations work by taking snapshots of memory. Values in memory that are required to perform the operation will be copied locally. Then the execution will enter a loop and prepare its local structure to be swapped into the queue structure. A CAS will be performed to check if the snapshot has changed and attempt to swap in the prepared value. If the snapshot is unchanged, the swap will succeed and the execution will break out of the loop. If the snapshot has changed, the execution will loop and try again by taking a new snapshot and continuing from there.

Here is an implementation of enqueue using CAS instead of locks.

```
void Enqueue(T value) {
  Node<T>* node = new Node<T>();
  node->Value = value;
  node->Next = NULL;

  Node<T>* t;
  Node<T>* next;
  while(true){
    t = Tail;
    if(Tail != t) continue;
    next = t->Next;
    if(Tail != t) continue;
    if(next){ CAS(&Tail, t, next); continue; }
    if(CAS(&t->Next, NULL, node)) break;
  }
  CAS(&Tail, t, node);
}
```

Further Problems with CAS

With proper consideration, avoiding to use CAS recklessly and handling intermediate states is fairly easy. However, there are two addition problems that arise when using CAS in a queue. These problems can be subtle at first glance and while they are related, they are indeed separate problems.

The first problem is the memory reclamation problem. This problem is caused by the inability to detect when a node is not use by any threads. Usually, in a dequeue operation, the calling thread deletes the node that was removed the queue. There is a possibility though that other threads are accessing this node in a CAS operation. Therefore, it is unsafe to delete the node at the time of a dequeue. Determining when it is safe to free memory of the nodes in the queue is the memory reclamation problem.

The second problem is the ABA problem. The ABA problem occurs when after a thread takes a snapshot for a CAS operation, another thread deletes a node in the snapshot, frees it and allocates a new node that just happens to have the same memory address of the original. This will allow the snapshot to appear unchanged even though the node is really a new one. Avoiding this scenario will solve the ABA problem.

Hazard Pointers

Since both problems are related, they can be solve collectively through a mechanism know as Hazard Pointers. Hazard Pointers provide a way for threads to broadcast which nodes are currently being used in their CAS snapshots. This is referred to as declaring pointers as hazards. Upon completion of a dequeue, the freeing of the dequeued node will be deferred to a later time instead of being freed immediately. This is known as retiring the node. Once some preset amount of nodes have been retired, a scan is done and any nodes not declared hazards by any threads can be safely freed.

To accomplish this, a structure called a Hazard Pointer Record (HPRec) is used. The queue maintains an internal chain of HPRecs. When a thread requires access to the queue, it is given an HPRec.

```
struct HPRec {
  Node* HP[K];
  HPRec* Next;
  bool Active;
  list<Node*> RetireList;
}
```

Inside the HPRec is a an array of Hazard Pointers (HP[]). When a thread takes a snapshot for a CAS and needs to declare some pointers as hazards, it inserts them into this array. Each thread can only modify the HP[] array inside its HPRec but can read all others in the queue. The Next pointer is used by the queue to maintain the chain of HPRecs. The Active boolean field is used to deactivate a HPRec when a thread finishes and save it for reuse when a new thread asks for access to the queue. When a dequeue occurs, the node is retired into the RetireList of the calling thread.

Since an HPRec is given to each thread, each thread has its own RetireList. Once this list grows large enough, a scan is performed. A scan is similar to a garbage collection where it will determine which nodes can be freed based on whether they are in use by any threads. A scan consists of two parts.

```
void scan(){
  map<Node*,bool> hazards;
  foreach(HPRec rec in HPRecChain)
    foreach(Node* hp in rec.HP)
      hazards.insert(hp);

  foreach(Node* node in RetireList)
    if(!hazards.lookup(node)){
      RetireList.remove(node);
      delete node;
    }
}
```

The first part of the scan will traverse the queue's HPRec chain and store any hazards into a local map. This map will be used in part 2 of the scan to quickly lookup pointers and see if they are hazards. The map is used to avoid the need to traverse the entire HPRec chain to do a lookup.

The second part of the scan will then iterate through the local RetireList of the calling thread. For each node in the RetireList, it will perform a lookup in the map created in part 1. If the node is found in the map, it has been declared a hazard by another thread and is unsafe to delete. As a result, it will simply be left in the RetireList. On the other hand, if the node is not found in the map, it is not in use by any other threads and safely freed. It is then removed from the RetireList and deleted.

The next part of using Hazard Pointers is determining what the hazards are. Most data structures will only require one or two hazards per thread at a time. A hazard analysis must be done to determine exactly how many are needed. Hazard pointers are by definition the pointers that are unsafe to delete at a given time. The way pointers become unsafe is when they are referenced by one or more threads. The way this usually happens is during a CAS snapshot. Looking at the points that are used during these snapshots will determine the number of hazard pointers are needed for the entire structure.

In the case of the queue, since an enqueue and a dequeue from the same thread cannot be going on at the same time, the number of hazards needed per thread is equal to the max of the ones used during an enqueue or a dequeue. During an enqueue operation, a snapshot is only utilizing the tail pointer. A dequeue operation however takes a snapshot using both the head and the head->next pointers. Therefore, the size of the HP[] inside a HPRec is 2. Once this is analysis is done, it is simple just a matter of storing the pointers into the HP[] at the right time as seen here.

```
void Enqueue(T value) {
  Node<T>* node = new Node<T>();
  node->Value = value;
  node->Next = NULL;

  Node<T>* t;
  Node<T>* next;
  while(true){
    t = Tail;
    hprec->HP[0] = t;
    if(Tail != t) continue;
    next = t->Next;
    if(Tail != t) continue;
    if(next){ CAS(&Tail, t, next); continue; }
    if(CAS(&t->Next, NULL, node)) break;
  }
  CAS(&Tail, t, node);
}
```

Using the LocklessQueue Implementation

As described earlier, we created an IQueue<T> interface that is implemented directly by our SimpleQueue and LockingQueue. However, the LocklessQueue is unable to do this because each thread is required to have an HPRec and be able to access it somehow. There are a few ways to go about accomplishing this. Thread Local Storage could be used to save a reference to an HPRec and than that reference can be passed to an `Enqueue(HPRec* rec, T value)` method. However, this breaks the abstraction of using an IQueue<T> in first place. The way we implemented a solution was to use something we call a ThreadAccessor.

A ThreadAccessor is a private class of LocklessQueue that implements the IQueue<T> interface. When a thread needs access to a queue, it calls the `CreateAccessor()` method of an instance of LocklessQueue. This will return an IQueue<T> reference that the thread can use to access the queue. The IQueue<T> reference in actuality as a instance of ThreadAccessor that encapsulates the HPRec for that particular thread. A slight subtlety to make this work correctly was declaring ThreadAccessor a friend of LocklessQueue. Friend classes can break encapsulation but since ThreadAccessor is a private class of LocklessQueue it seemed like a legitimate solution. As an example, here is how the LocklessQueue can be used.

```
LocklessQueue<int> lockless;
IQueue<int>* a = lockless.CreateAccessor();
int x = 0;
a->Enqueue(1);
a->Enqueue(2);
a->Dequeue(&x);  // returns true, x = 1
delete a;
```

Performance

We have performed a few performance tests using both the locking and lockless queues. The system we used for testing was energon (8 core 1.8GHz Intel Xeon). Our tests have evolved over the course of the project. In this report, we will present the tests based on the bench program found in bench.cpp.

This program runs two types of tests. The first one runs either a enqueue or dequeue randomly. The second runs a series of enqueues followed by a series of dequeues. The series test runs twice, once for a bias towards and enqueues and then with a bias towards dequeues.

In addition to the concurrent tests, a sequential test is also done. A simple queue implementation is added which is identical to the locking queue without the locks. This is to compare the single-threaded performance of a naive queue versus the concurrent queue implementations.

To simulate a workload, the benchmark can optionally run a sieve function in between queue operations. A sieve function calculates the primes up to an given upper bound. Increasing the upper bound will cause the sieve to run longer causing the queue operations to be more spread out over time

All the tests are run using integer typed queues. The tests also run a correctness check, which accumulates the sum of enqueued items. Another sum is used for dequeued items. These sums are compared and adjusted (when the queue does not start or finish empty). If they are equal at the end of a test, it can be assumed that the queue has not failed in correctness.

Compiling

To compile, we used the following command:

```
g++ IQueue.h SimpleQueue.h LockingQueue.h LocklessQueue.h bench.cpp
-Wall -lrt -lpthread -o bench
```

-lrt is used to take timestamps and only required for the bench.cpp test. If just compiling the queues, the only library you need to include is pthread.

Test Results

Time shown is displayed in the last column in milliseconds.

```
Iterations 1000000
Sieve Bound 0
Threads 2
Series Bias 2

Random Tests
Sequential Simple   PASS   62634
Sequential Locking  PASS   125316
Sequential Lockless PASS   465140
Concurrent Locking  PASS   680483
Concurrent Lockless PASS   687245

Enqueue Bias Series Tests
Sequential Simple   PASS   120932
Sequential Locking  PASS   200324
Sequential Lockless PASS   469473
Concurrent Locking  PASS   994875
Concurrent Lockless PASS   571673

Dequeue Bias Series Tests
Sequential Simple   PASS   147297
Sequential Locking  PASS   236097
Sequential Lockless PASS   823506
Concurrent Locking  PASS   1359808
Concurrent Lockless PASS   896773
```

```
Iterations 1000000
Sieve Bound 0
Threads 4
Series Bias 2

Random Tests
Sequential Simple   PASS   61954
Sequential Locking  PASS   125904
Sequential Lockless PASS   465275
Concurrent Locking  PASS   1369490
Concurrent Lockless PASS   488774

Enqueue Bias Series Tests
Sequential Simple   PASS   122865
Sequential Locking  PASS   199504
Sequential Lockless PASS   469541
Concurrent Locking  PASS   1559039
Concurrent Lockless PASS   434948

Dequeue Bias Series Tests
Sequential Simple   PASS   137535
Sequential Locking  PASS   243589
Sequential Lockless PASS   816693
Concurrent Locking  PASS   1815145
Concurrent Lockless PASS  611822
```

```
Iterations 1000000
Sieve Bound 0
Threads 8
Series Bias 2

Random Tests
Sequential Simple   PASS   62666
Sequential Locking  PASS   124256
Sequential Lockless PASS   467526
Concurrent Locking  PASS   1020029
Concurrent Lockless PASS   425888

Enqueue Bias Series Tests
Sequential Simple   PASS   129104
Sequential Locking  PASS   202686
Sequential Lockless PASS   471317
Concurrent Locking  PASS   863716
Concurrent Lockless PASS   529741

Dequeue Bias Series Tests
Sequential Simple   PASS   133589
Sequential Locking  PASS   237955
Sequential Lockless PASS   838871
Concurrent Locking  PASS   913842
Concurrent Lockless PASS  496769
```

```
Iterations 1000000
Sieve Bound 0
Threads 16
Series Bias 2

Random Tests
Sequential Simple   PASS   61470
Sequential Locking  PASS   124619
Sequential Lockless PASS   466233
Concurrent Locking  PASS   738649
Concurrent Lockless PASS   499883

Enqueue Bias Series Tests
Sequential Simple   PASS   126330
Sequential Locking  PASS   200271
Sequential Lockless PASS   441234
Concurrent Locking  PASS   885405
Concurrent Lockless PASS   500252

Dequeue Bias Series Tests
Sequential Simple   PASS   135313
Sequential Locking  PASS   238540
Sequential Lockless PASS   840656
Concurrent Locking  PASS   980647
Concurrent Lockless PASS  669595
```

```
Iterations 1000000
Sieve Bound 100
Threads 2
Series Bias 2

Random Tests
Sequential Simple   PASS   1684332
Sequential Locking  PASS   1743544
Sequential Lockless PASS   2158061
Concurrent Locking  PASS   1676784
Concurrent Lockless PASS   1631442

Enqueue Bias Series Tests
Sequential Simple   PASS   1838472
Sequential Locking  PASS   1917211
Sequential Lockless PASS   2206580
Concurrent Locking  PASS   1362624
Concurrent Lockless PASS   1620232

Dequeue Bias Series Tests
Sequential Simple   PASS   1954931
Sequential Locking  PASS   2061635
Sequential Lockless PASS   3005915
Concurrent Locking  PASS   1563148
Concurrent Lockless PASS   1891935
```

```
Iterations 1000000
Sieve Bound 100
Threads 4
Series Bias 2

Random Tests
Sequential Simple   PASS   1685974
Sequential Locking  PASS   1738497
Sequential Lockless PASS   2133310
Concurrent Locking  PASS   1805870
Concurrent Lockless PASS   918474

Enqueue Bias Series Tests
Sequential Simple   PASS   1840802
Sequential Locking  PASS   1913413
Sequential Lockless PASS   2207364
Concurrent Locking  PASS   1541419
Concurrent Lockless PASS   836019

Dequeue Bias Series Tests
Sequential Simple   PASS   2286186
Sequential Locking  PASS   2392008
Sequential Lockless PASS   2723637
Concurrent Locking  PASS   1357968
Concurrent Lockless PASS   1072946
```

```
Iterations 1000000
Sieve Bound 100
Threads 8
Series Bias 2

Random Tests
Sequential Simple   PASS   1681155
Sequential Locking  PASS   1733117
Sequential Lockless PASS   2124352
Concurrent Locking  PASS   1118121
Concurrent Lockless PASS   624164

Enqueue Bias Series Tests
Sequential Simple   PASS   1828144
Sequential Locking  PASS   1918603
Sequential Lockless PASS   2202487
Concurrent Locking  PASS   1076796
Concurrent Lockless PASS   585136

Dequeue Bias Series Tests
Sequential Simple   PASS   1867307
Sequential Locking  PASS   2041704
Sequential Lockless PASS   2712062
Concurrent Locking  PASS   1026498
Concurrent Lockless PASS   706690
```

```
Iterations 1000000
Sieve Bound 100
Threads 16
Series Bias 2

Random Tests
Sequential Simple   PASS   1687690
Sequential Locking  PASS   1735187
Sequential Lockless PASS   2196187
Concurrent Locking  PASS   832173
Concurrent Lockless PASS   711443

Enqueue Bias Series Tests
Sequential Simple   PASS   1819161
Sequential Locking  PASS   1914556
Sequential Lockless PASS   2205970
Concurrent Locking  PASS   1133394
Concurrent Lockless PASS   625436

Dequeue Bias Series Tests
Sequential Simple   PASS   1858490
Sequential Locking  PASS   2049763
Sequential Lockless PASS   2728057
Concurrent Locking  PASS   1216327
Concurrent Lockless PASS   829559
```

```
Iterations 100000
Sieve Bound 1000
Threads 16
Series Bias 2

Random Tests
Sequential Simple    PASS   1710704
Sequential Locking   PASS   1714754
Sequential Lockless  PASS   1750059
Concurrent Locking   PASS   270345
Concurrent Lockless  PASS   297413

Enqueue Bias Series Tests
Sequential Simple    PASS   1722922
Sequential Locking   PASS   1721790
Sequential Lockless  PASS   1753386
Concurrent Locking   PASS   284296
Concurrent Lockless  PASS   302928

Dequeue Bias Series Tests
Sequential Simple    PASS   1724600
Sequential Locking   PASS   1733508
Sequential Lockless  PASS   1776346
Concurrent Locking   PASS   269701
Concurrent Lockless PASS   315333
```

```
Iterations 100000
Sieve Bound 1000
Threads 96
Series Bias 2

Random Tests
Sequential Simple    PASS   1711124
Sequential Locking   PASS   1716284
Sequential Lockless  PASS   1744635
Concurrent Locking   PASS   250760
Concurrent Lockless  PASS   323766

Enqueue Bias Series Tests
Sequential Simple    PASS   1726977
Sequential Locking   PASS   1730159
Sequential Lockless  PASS   1754679
Concurrent Locking   PASS   243276
Concurrent Lockless  PASS   261952

Dequeue Bias Series Tests
Sequential Simple    PASS   1729141
Sequential Locking   PASS   1732679
Sequential Lockless  PASS   1776111
Concurrent Locking   PASS   250506
Concurrent Lockless PASS   261257
```

```
Iterations 100000
Sieve Bound 100
Threads 96
Series Bias 2

Random Tests
Sequential Simple    PASS   169286
Sequential Locking   PASS   173489
Sequential Lockless  PASS   212566
Concurrent Locking   PASS   93006
Concurrent Lockless  PASS   129832

Enqueue Bias Series Tests
Sequential Simple    PASS   181989
Sequential Locking   PASS   186203
Sequential Lockless  PASS   215674
Concurrent Locking   PASS   113348
Concurrent Lockless  PASS   65567

Dequeue Bias Series Tests
Sequential Simple    PASS   178142
Sequential Locking   PASS   184810
Sequential Lockless  PASS   243683
Concurrent Locking   PASS   116422
Concurrent Lockless PASS   99960
```

```
Iterations 100000
Sieve Bound 0
Threads 96
Series Bias 2

Random Tests
Sequential Simple    PASS   6021
Sequential Locking   PASS   12309
Sequential Lockless  PASS   46734
Concurrent Locking   PASS   89177
Concurrent Lockless  PASS   84244

Enqueue Bias Series Tests
Sequential Simple    PASS   12562
Sequential Locking   PASS   17438
Sequential Lockless  PASS   42747
Concurrent Locking   PASS   91847
Concurrent Lockless  PASS   56740

Dequeue Bias Series Tests
Sequential Simple    PASS   10005
Sequential Locking   PASS   16251
Sequential Lockless  PASS   66385
Concurrent Locking   PASS   73489
Concurrent Lockless PASS   59347
```

The test results are promising.   First off, in a sequential scenario, the concurrent locks get destroyed in performance. The overhead introduced to handle the concurrency is a real burden when it is not being used. The first four results show raw queue access with no sieve being performed. There is literally no motivation here to use a concurrent queue. This is not a real-world scenario though so comparing the concurrent queues to sequential queues is of little value. It is interesting to note though that with very high concurrent access, the lockless queue is much faster than the locking queue.

Moving onto the next set of four, these show a small sieve and are differentiated by number of threads. In the two-thread case, the overhead of the concurrency is immediately diminished and all queues are about the same except for the lockless which is far behind. As the number of threads increases, the concurrent queues pull farther ahead. Also, as the lock contention starts to grow, the lockless threads sees some even further gains.

The next test result shows a large sieve with 16 threads. The concurrent queues performance is magnitudes ahead of the sequential here as the execution is CPU bound. Here we can see that the lockless queue has lost its glory and has fallen behind the locking queue slightly. Since the majority of time is spent in computation, the contention for the lock has become a smaller part of the execution.

The last three results show a small sieve with a large, 96, amount of threads. The results follow the trend of the more lock contention there is, the more benefit of the lockless queue.

Notes and Overall Remarks

There are few things that should be noted in regard to the overall project. First, there are few things that could tuned to increase performance. The frequency of the scans can be adjusted to increase performance at the cost of using more memory. Also, the list and maps used during the scan can most likely be optimized.   In addition, pooling nodes can drastically reduce the times for enqueues. If memory is pre-allocated, the intuition is that there enqueuing will be much faster.

Testing also turned out to more difficult than first conceived. Our testing tools have evolved over the course of the project and even now we are not sure they completely reliable. A lot of performance gains by using Hazard Pointers can be machine and usage dependent. It is a very hard thing to test and feel completely confident about. A more purposeful test would be to swap in the lockless queue into a system that is already using a concurrent queue, tune it correctly and then observe the results compared to the original.

As an additional note, there is a slightly subtlety in the HPRec chain mechanism. The queue is unable to actually delete any nodes from the HPRec chain. Trying to do so using CAS would be a catch 22 since safely reclaiming memory is the problem the HPRec chain is used to solve. The algorithm simply deactivates nodes instead of deleting them. Any deactivated nodes are reused when possible instead of allocating new ones. This avoids the problem completely.

Conclusion

Using a lockless queue is not an automatic solution to any concurrency woes. It should be observed as a tool to help fine-tune performance where it can be used appropriately. The simple implementation and usage of the locking queue can sometimes outweigh the lockless queue in both complexities and performance. It is important to note though that both the locking and lockless queues can be embrace the power of multicore systems. It is up to the programmer to correctly identify and use the proper techniques for solving every problem.

References

Our implementation is primarily based on the first paper listed by Maged Michael.

Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects
Maged M. Michael
http://www.research.ibm.com/people/m/michael/ieeetpds-2004.pdf

Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms
Maged M. Michael; Michael L. Scott
http://www.cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf

The Art of Multiprocessor Programming
Herlihy, Maurice, and Nir Shavit. Morgan Kaufmann, 2008. Print.

CAS-Based Lock-Free Algorithm for Shared Deques
Maged M. Michael
http://www.research.ibm.com/people/m/michael/europar-2003.pdf

Lock-free Dynamically Resizable Arrays
Damian Dechev; Peter Pirkelbauer; Bjarne Stroustrup
http://www2.research.att.com/~bs/lock-free-vector.pdf