CBPQ: High Performance Lock-Free Priority Queue*

Anastasia Braginsky[1], Nachshon Cohen[2], and Erez Petrank[2]

[1] Yahoo Research, Haifa, Israel anastas@yahoo-inc.com
[2] Technion - Israel Institute of Technology, Haifa, Israel
{ncohen,erez}@cs.technion.ac.il

Abstract Priority queues are an important algorithmic component and are ubiquitous in systems and software. With the rapid deployment of parallel platforms, concurrent versions of priority queues are becoming increasingly important. In this paper, we present a novel concurrent lock-free linearizable algorithm for priority queues that scales significantly better than all known (lock-based or lock-free) priority queues. Our design employs several techniques to obtain its advantages including lock-free chunks, the use of the efficient fetch-and-increment atomic instruction, and elimination. Measurements under high contention demonstrate performance improvement by up to a factor of 1.8 over existing approaches.

Keywords: non-blocking, priority queue, lock-free, performance, freezing

1 Introduction

Priority queues serve as an important basic tool in algorithmic design. They are widely used in a variety of applications and systems, such as simulation systems, job scheduling (in computer systems), networking (e.g., routing and realtime bandwidth management), file compression, numerical computations, and more. With the proliferation of modern parallel platforms, the need for a high-performance concurrent implementation of the priority queue has become acute.

A priority queue (PQ) supports two operations: insert and deleteMin. The abstract definition of a PQ provides a set of key-value pairs, where the key represents a priority. The insert() method inserts a new key-value pair into the set (the keys don't have to be unique), and the deleteMin() method removes and returns the value of the key-value pair with the lowest key (i.e., highest priority) in the set.

Lock-free (or non-blocking) algorithms [12,13] guarantee eventual progress of at least one operation under any possible concurrent scheduling. Thus, lock-free implementations avoid deadlocks, live-locks, and priority inversions. Typically, they also demonstrate high scalability, even in the presence of high contention.

* This research was supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 274/14).

In this paper we present the design of a high performance lock-free linearizable PQ. The design builds on a combination of three ideas. First, we use a chunked linked-list [3] as the underlying data structure. This replaces the standard use of heaps, skip-lists, linked-lists, or combinations thereof. Second, we use the fetch-and-increment ($F\&I$) instruction for an efficient implementation of deleteMin and insert. This replaces the stronger, but less efficient compare-and-swap (CAS) atomic primitive (used in all other lock-free PQ studies). Third, the resulting design is a great platform for applying an easy variant of elimination [18,11], which resolves the contention of concurrent inverse operations: the insert of a small key and a deleteMin.

Various constructions for the concurrent PQ exist in the literature. Hunt et. al. [14] used a fine-grained lock-based implementation of a concurrent heap. Dragicevic and Bauer presented a linearizable heap-based priority queue that used lock-free software transactional memory (STM) [8]. A quiescently consistent skip-list based priority queue was first proposed by Lotan and Shavit [17] using fine-grained locking, and was later made non-blocking [9]. Another skip-list based priority queue was proposed by Sundell and Tsigas [19]. Liu and Spear [16] introduced two concurrent versions of data structure called *mounds* (one is lock-based and the other is lock-free). The mounds data structure aims at very fast $O(\log(\log(N)))$ insert operations. It is built of a rooted tree of sorted lists that relies on randomization for balance. The deleteMin operations have a slower $O(\log(N))$ complexity. Mounds' insert operation is currently the most productive among concurrent implementations of the PQ. Linden and Jonsson [15] presented a skip-list based PQ. Deleted elements are first marked as deleted in the deleteMin operation. Later, they are actually disconnected from the PQ in batches when the number of such nodes exceed a given threshold. Their construction outperforms previous algorithms by $30 - 80\%$. Recently, Calciu et al. [5] introduced a new lock-based, skip-list-based adaptive PQ that uses elimination and flat combining techniques to achieve high performance at high thread counts.

Elimination [18,11] provides a method to match concurrent inverse operations so that they exchange values without synchronizing on a centralized data structure. Elimination for PQ was presented in [5], where threads that insert small keys and threads that delete minimum keys post their operations in an elimination array and wait for their request to be processed. Our elimination variant requires no additional similar waiting time and it bears minimal overhead.

We implemented CBPQ in C++ and compared its performance to the currently best performing PQs: the Linden and Jonsson's PQ [15], the lock-free and lock-based implementations of the Mounds PQ [16], and the adaptive PQ (APQ) of Calciu et al. [5]. We evaluated the performance of CBPQ using targeted micro-benchmarks: one that runs a mix of insert and deleteMin operations, where each occurs with equal probability, a second one that runs only insert operations, and a third with only deleteMin operations.

The results demonstrate that our CBPQ design performs excellently under high contention, and it scales best among known algorithms, providing the best performance with a large number of threads. Under low contention, our algo-

rithm is not a winner, and it turns out that the LJPQ design performs best. In particular, under high contention and for a mix of deleteMin and insert operations, CBPQ outperforms all other algorithms by up to 80%. When only deleteMin operations run, and with high contention, CBPQ performs up to 5 times faster than deletions of any other algorithm we compared to. As expected, Mounds perform best with insert only operations, outperforming CBPQ (which comes second) by a factor of up to 2.

2 A Bird's Eye Overview

The CBPQ data structure is composed of a list of chunks. Each chunk has a range of keys associated with it, and all CBPQ entries with keys in this range are located in this chunk. The ranges do not intersect and the chunks are sorted by the ranges' values. To improve the search of a specific range, an additional skip-list is used as a directory that allows navigating into the chunks, so inserting a key to the CBPQ is done by finding the relevant chunk using a skip-list search, and then inserting the new entry into the relevant chunk.

The first chunk is built differently from the rest of the chunks since it holds the smallest keys and supports deleteMin operations. We forbid inserts into the first chunk. Instead, a key in the range of the first chunk is inserted through special handling, as discussed below. The remaining chunks are used for insertions only.

The first chunk consists of an immutable sorted array of elements. To delete the minimum, a thread simply needs to atomically fetch and increment a shared index to this array. All other chunks consist of unsorted arrays with keys in the appropriate range. To insert a key to a chunk other than the first, the insert operation simply finds the adequate chunk using the skip-list directory, and then adds the new element to the first empty slot in the array, again, simply by fetching and incrementing the index of the first available empty slot in the array.

When an insert operation needs to insert a key to the first chunk, it registers this key in a special buffer and requests the first chunk rebuild. Subsequently, a new first chunk with a new sorted array is created from the remaining keys in the first chunk, all keys registered in the buffer, and if needed, more keys from the second chunk. The thread that attempts to insert a (small) key into the buffer yields the processor and allows progress of other threads currently accessing the first chunk, before making everybody cooperate on creating a new first chunk. During this limited delay, elimination can be executed and provide even more progress. By the time a rebuild of the first chunk actually happens, either much progress has occurred, or the buffer has been filled with keys, making the amortized cost of a new first chunk construction smaller. The creation of a new first chunk is also triggered when there are no more elements to delete in it. The creation of a new first chunk is made lock-free by allowing all relevant threads to take part in the construction, never making a thread wait for others to complete a task.

When an internal chunk is filled due to insertions, it is split into two half-full chunks using the lock-free freezing mechanism of [3]. The CBPQ scheme is

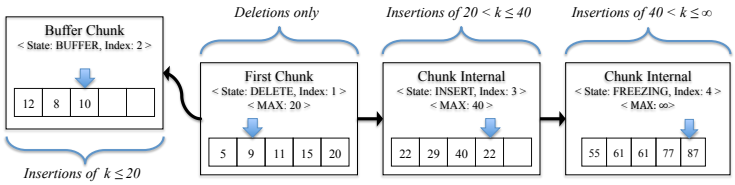


Figure 1: Overview of the CBPQ data structure

illustrated in Figure 1 (with some more algorithmic details shown in text in the angle brackets, to be discussed later). The full description of the algorithm is provided in Section 3.

Key design ideas: The key design ideas in the proposed priority queue are as follows. First, we aim at using $F\&I$ instructions for the contention bottlenecks. For that, we employ the chunked linked-list as the underlying data structure. This allows most insert and deleteMin operations to be executed with an (atomic) increment of a chunk index. To make the above work, we distinguish the design of the first chunk (that mostly handles deleteMin operations) from the design of the remaining chunks (which handle only insert operations). An extra buffer chunk supports insertions to the first chunk. Finally, we add elimination, which fits this algorithm like a glove with negligible overhead and significant performance benefits.

3 The Full CBPQ Design

In this section we describe the core CBPQ algorithm. In Section 4 we describe the skip-list on top of the list and the elimination. The CBPQ linearization points presentation can be found in the full version of this paper [1].

3.1 Underlying Data Structures

Each chunk in the CBPQ has a *status* word, which combines together an array index (shortly denoted *index*), a *frozen index*, and the chunk *state*. The status is always atomically updated. The state of a newly created chunk is INSERT, DELETE, or BUFFER, indicating that it is created for further insertions, for deletions, or for serving as a buffer of keys to be inserted to the first chunk, respectively. In addition, when a chunk has to be replaced by a new chunk, the old chunk enters the FREEZING state, indicating that it is in the process of being frozen. The FROZEN state indicates that the chunk is frozen and thus, obsolete.

The frozen index is used only for freezing the first chunk, as will be explained in Section 3.4. In Listing 1, we list the Status and Chunk classes that we use. The relevant state, index, and frozenIdx fields are all held in a single machine word that is called the Status.

Listing 1: Status and Chunk records

```
1  class Status{
2    uint29_t frozenIdx;
3    uint3_t state;
4    uint32_t index;
5  } // 64 bits, machine word
6
7
8
9  class Chunk{
10   Status status;
11   uint64_t entries[M];
12   uint32_t max;
13   uint64_t frozen[M/63+1];
14   Chunk *next, *buffer;
15 }
```

In addition to the status, each chunk consists of an array (entries in the Chunk class), which holds the key-value pairs contained in the chunk. The entries are machine words, whose bits are used to represent the key and value. For simplicity, in what follows we will refer to the keys only. Our keys can take any value that can be represented in 31 bits, except 0, which is reserved for initialization. Each chunk has an immutable maximal value of a key that can appear on it, defined when the chunk is created (max in Chunk). A chunk holds keys less than or equal to its max value and greater than the max value of the previous chunk (if it exists). This max field is not relevant for the buffer chunk. Any chunk (except the buffer) uses a pointer to the next chunk (the next field). Finally, only the first chunk uses a pointer to a buffer chunk (the buffer field). The meaning of the frozen array that appears in the Chunk specification of Listing 1 is related to the freeze action and will be explained in Section 3.4. Finally, the CBPQ is a global pointer head to the first chunk. In Figure 1 the chunk's fields are depicted.

3.2 Memory management

Similar to previous work [15], we use Keir Fraser's epoch based reclamation (EBR) [9] as a (simplified) solution for memory reclamation. The EBR scheme was used in one previous work to which we compare [15]. The other algorithms that we compare against [16,5] did not implement memory reclamation at all. Measurements show that avoiding reclamation buys a performance advantage of up to 4% (but does not provide a full solution). More advanced lock-free memory reclamation solutions appear in [2,6,7,4].

3.3 Operations Implementation

Insert: The insert pseudo-code is presented in the insert() method in Listing 2. In order to insert a key into the CBPQ, we first need to find the relevant chunk C. Because chunks are ordered by their ranges, a simple search can be used, skipping the chunks with smaller maximums (Line 4). If an insert must be performed to the first chunk, the insertToBuffer() method is invoked (Line 6), as explained in the next paragraph. Otherwise, C is not first. After C is found, its array index is atomically incremented to make room for the new entry (Line 9).

```
1  void insert(int key) {
2    Chunk* cur = NULL, *prev = NULL;
3    while(1) {
4      getChunk(&cur, &prev, key);            // set the current and previous chunk pointers
5      if ( cur==head ) {                     // first chunk
6        if ( insertToBuffer(key, cur, head) ) return;
7        else continue;
8      }
9      Status s = cur->status.aIncIdx();      // atomically increase the index in the status
10     int idx = getIdx(s);
11     if ( idx<M && !s.isInFreeze() ) {      // insert into a non-full and non-frozen chunk
12       cur->entries[idx] = key; memory_fence;
13       if (!cur->status.isInFreeze()) return;
14       if (cur->entryFrozen(idx)) return;  // key got copied
15     }
16     freezeChunk(cur);                      // restructure the CBQP, then retry
17     freezeRecovery(cur, prev);
18   }
19 }
20 bool insertToBuffer(int key, Chunk* cur, Chunk* curhead) {
21   Chunk *curbuf = cur->buffer;  bool result = false;   // PHASE I: key insertion into the buffer
22   if( curbuf==NULL )                                   // the buffer is not yet allocated
23     if ( createBuffer(key,cur,&curbuf) ) goto phaseII; // key added during buffer creation
24   Status s = curbuf->status.aIncIdx();                 // atomically increase the index in the status
25   int idx = getIdx(s);
26   if ( idx<M && !s.isInFreeze() ) {
27     curbuf->entries[idx] = key; memory_fence;
28     if (!curbuf->status.isInFreeze()) result = true;
29     if (curbuf->entryFrozen(idx)) return true;
30   }
31 phaseII:          // PHASE II: first chunk merges with buffer before insert ends
32   usleep(0);      // yield, give other threads a chance
33   freezeChunk(cur); freezeRecovery(cur, NULL);
34   return result;
35 }
36 int deleteMin() {
37   Chunk* cur, next;
38   while(1){
39     cur = head;
40     Status s = cur->status.aIncIdx();  // atomically increase the index in the status
41     int idx = getIdx(s);
42     if ( idx<M && !s.isInFreeze() )    // delete from not full and non-frozen chunk
43       return curr->entries[idx];
44     freezeChunk(cur); freezeRecovery(cur, NULL);  // Freeze, then restructure the CBPQ and retry
45   }
46 }
```

Listing 2: Common code path: insertion of a key and deletion of the minimum

The aIncIdx() method wraps an $F\&I$ instruction and returns the status with the new value of the chunk index. The index is incremented first and only later we check whether it surpassed the end of the array. However, the number of bits required to represent the chunk size (M) is much smaller than the number of bits in the index, and so even if each thread increments it once after the chunk is full, an overflow would not occur.[1] If C is not frozen and the incremented index does not point beyond the chunk's capacity, we simply write the relevant key to the array entry (Lines 11-15). The write is followed by a memory fence in order to ensure it will be visible to any other thread that may freeze C concurrently. If

[1] The size of the array plus the number of operating threads limits the index value. In our implementation the array size is less then 2^{10} plus the number of threads (less then 2^6), so 11 bits suffice.

C is not freezing (Line 13), the insert is completed. Else if C is freezing, but our key was already marked as frozen (Line 14), the insert is also finished. Otherwise (if the index has increased too much or a freeze has been detected), then the freeze is completed and C is split (Lines 16, 17), as will be explained later. After the chunks restructure, the insert is restarted.

Insert to the first chunk: The lowest range keys are inserted into the buffer pointed to from the first chunk. The pseudo-code of an insertion to the buffer chunk is presented in the insertToBuffer() method in Listing 2. It starts by allocating a new buffer holding the relevant key, if needed (Line 23). The createBuffer() method returns *true* if the new buffer was successfully connected to the first chunk, or *false* if another thread had connected another buffer. In the latter case, a new pointer to the buffer is inserted into curbuf.

Keys are inserted to the buffer in a manner similar to their insertion to other (non-first) chunk: the index is increased and the key is placed. If this cannot be done because the buffer is full or frozen, the insertToBuffer() returns *false* (after the first chunk's freeze and recovery) to signal that the insert operation has to be retried. The insert to buffer operation cannot end until the new key is included in the first chunk and considered for deletion. So after a key is successfully inserted into a buffer, the freeze and merge of the first chunk is invoked. However, if this key is already frozen, the insert to the first chunk can safely return (Line 29), because no deletion can now happen until the new key is taken into the new first chunk. After the first chunk is replaced, the insertion is considered done. The yielding, freeze and merge (Lines 32-33) are explained in Sections 3.4 and 4.

Delete minimum: The deletion is very simple and usually very fast. It goes directly to the first chunk, which has an ordered array of minimal keys. The first chunk's index is atomically increased. Unless the need to freeze the first chunk is detected, we can just return the relevant key. The pseudo-code for the deletion operation is presented in the deleteMin() method in Listing 2.

3.4 Split and Merge Algorithms

It remains to specify the case where a freeze is needed for splitting a non-first chunk or merging the first chunk with the buffer and possibly also with the second chunk. This mechanism is developed in [3] and we adopt it with minor modifications. For completeness, we explain this mechanism below.

For splitting or merging chunks, a freeze is first applied on the chunks, indicating that new chunks are replacing the frozen ones. A frozen chunk is logically immutable. Then, a recovery process copies the relevant entries into new chunks that become active in the data structure. Threads that wake up after being out of the CPU for a while may discover that they are accessing a frozen chunk and they then need to take actions to move into working on the new chunks that replace the frozen ones. In [3], the freezing process of a chunk was applied by atomically setting a dedicated freeze bit in each machine word (using a CAS loop), signifying that the word is obsolete. Freezing was achieved after all words were marked in this manner. Applying a CAS instruction on each obsolete word

```
1  void freezeChunk(Chunk* c) {
2    int idx, frozenIdx = 0; Status localS;   // locally copied status
3    while(1){                                // PHASE I: set the chunk status if needed
4      localS = c->status; idx = localS.getIdx();   // read the current status to get its state and index
5      switch (localS.getState()){
6        case BUFFER: // in insert or buffer chunks frozenIdx was and remained 0
7        case INSERT: c->status.aOr(MASK_FREEZING_STATE); break;
8        case DELETE:
9          if (idx>M) frozenIdx=M; else frozenIdx=idx;
10         Status newS; newS.set(FREEZING, idx, frozenIdx); // set: state, index, frozen index
11         if (c->status.CAS(localS,newS)) break;  // can fail due to delete updating the index
12         else continue;
13       case FREEZING: break; // in process of being freezed
14       case FROZEN:  // c was frozen by someone else
15         c->markPtrs(); return;   // mark the chunk out-pointers as deleted
16     }
17     break;  // continue only if CAS from DELETE state failed
18   }
19   if (c != head) freezeKeys(c);   // PHASE II: freeze the entries
20   c->status.aOr(MASK_FROZEN_STATE);  // from FREEZING to FROZEN using atomic OR
21   c->markPtrs();  // set the chunk pointers as deleted
22 }
```

Listing 3: Freezing the keys and the entire chunk

(sometimes repeatedly) may be slow and it turns out that in the context of CBPQ we can freeze a chunk more efficiently.

The freezing mechanism coordinates chunk replacements with concurrent operations. What may come up in the CBPQ is a race between insertion and freezing. An insert operation increments the array index reserving a location for the insert. But such an operation may then be delayed for a long while before actually inserting the item to the array. This operation may later wake up to find that the chunk has been frozen and entries have already been copied to a newer chunk. Since the item's content was not installed into the array, the freezing process could not include it in the new chunk and insertion should be retried. The inserting thread needs to determine whether its item was inserted or not using a freeze bit that is associated with his entry of the frozen chunk. This motivates a freeze bit for each entry, but these bits do not need to reside on the entry.

In CBPQ, all freeze bits are located separately from the entries, with a simple mapping from them to their freeze bits. In the CBPQ Chunk class (Listing 1), the data words (storing the keys and the values) are located in the entries array. All freeze bits are compacted in the frozen array. Each frozen bit signifies whether the key in the associated entry has been copied into the chunks that replace the current frozen chunk. Assuming a 64-bit architecture, we group each 63 entries together and assign a *freeze word* of 64 bits to signify the freeze state of all 63 entries. We use one bit for each of the 63 entries and reserve the most significant bit (MSB) of the freeze word to make sure that it is written only once (modifying this bit from 0 to 1 when written).

The freezing process reads the actual entries of the 63 entries. Since a value of zero is not a valid key, having a zeroed key word indicates an uncompleted insert. In this case, the entry is not copied into the new chunk. After determining

which entries should be copied, the freezing process attempts to set the freeze word accordingly (1 for an existing entry and 0 for an invalid entry) using an atomic CAS of this word in the frozen array. Atomicity is guaranteed, because each freeze word is updated only once, due to the MSB being set only once. The compaction of the freeze bits allows setting them with a single CAS instead of 63 CAS operations, reducing the synchronization overhead for the freeze processing.

Freezing the chunk: Following the pseudo-code of method freezeChunk() in Listing 3, here is how we execute the freeze for a chunk C. In the first phase of the operation, we change C's status, according to the current status (Lines 4-17). Recall that the status consists of the state, the index and the frozen index. If C is not in the process of freezing or already frozen, then it should be in a BUFFER, an INSERT or a DELETE state, with a zeroed frozen index and an index indicating the current array location of activity. For insert or buffer chunks, we need only change the state to FREEZING; this is done by setting the bits using an atomic OR instruction (Line 7). The frozen index is only used for the first chunk, in order to mark the index of the last entry that was deleted before the freeze. Upon freezing, the status of the first chunk is modified to contain FREEZING as a state, the same index, and a frozen index that equals the index if the first chunk is not exhausted, or the maximum capacity if the first chunk is exhausted, i.e., all entries have been deleted before the freeze (Line 9). Let us explain the meaning of the frozen index.

As the deletion operation uses a $F\&I$ instruction, it is possible that concurrent deletions will go on incrementing the index of the first array in spite of its status showing a frozen state. However, if a thread attempts to delete an entry from the first chunk and the status shows that this chunk has been frozen, then it will not use the obtained index. Instead, it will help the freezing process and then try again to delete the minimum entry after the freezing completes. Therefore, the frozen index indicates the last index that has been properly deleted. All keys residing in locations higher than the frozen index must be copied into the newly created first chunk during the recovery of the freezing process. If all keys in the frozen first chunk have been deleted, then no key needs to be copied and we simply let the frozen index contain the maximum capacity M, indicating that all keys have been deleted from the first chunk.

In Line 11 the status is updated using a CAS to ensure that concurrent updates to the index due to concurrent deletions are not lost. If C is already in the FREEZING state because another thread has initiated the freeze, we can move directly to phase II. If C is in the FROZEN state, then the chunk is in an advanced freezing state and there is little left to do. It remains to mark the chunk pointers buffer and next so that they will not be modified after the chunk has been disconnected from CBPQ. These pointers are marked (in Line 15 or Line 21) as deleted (using the common Harris delete-bit technique [10]). At this point we can be sure that sleeping threads will not wake up and add a link to a new buffer chunk or a next chunk to C and we may return.

The second phase of the freeze assumes the frozen index and state have been properly set and it executes the setting of the words in the frozen array in method

freezeKeys() (Line 19). However, in the first chunk no freeze bits are set at all. In the first chunk it is enough to insert the chunk into the FREEZING state. This is so, because no one ever checks the frozen bits on the first chunk. Once we get the index and find that the state is DELETE the relevant minimum is just returned. With the second phase done, it remains to change the state from FREEZING to FROZEN (using the atomic OR instruction in Line 20) and to mark the chunk's pointers deleted as discussed above. The atomic OR instruction is available on the x86 platform and works efficiently. However, this is not an efficiency-critical part of the execution as freezing happens infrequently, so using a simple CAS loop to set the state would be fine.

CBPQ recovery from a frozen chunk: Once the chunk is frozen, we proceed like [3] and replace the frozen chunk with one or more new chunks that hold the relevant entries of the frozen chunk. This is done in the freezeRecovery() method, presented in Listing 4. The input parameters are: cur – the frozen chunk that requires recovery, and prev – the chunk that precedes cur in the chunk list or NULL if cur is the first chunk.[2] The first phase determines whether we need to split or merge the frozen chunk (Line 4). If cur is the first chunk (which serves the deleteMin operation), a merge has to be executed; as the first chunk gets frozen when there is need to create a new first chunk with other keys. If it is not the first chunk, then another chunk (which serves the insert operation) must have been frozen because it got full and we need to split it into two chunks. There is also a corner case in which a merge of the first chunk happens concurrently with a split of the second chunk. This requires coordination that simply merges relevant values of the second chunk into the new first. So if cur is the second chunk and the first chunk is currently freezing, then we should work on a merge.[3]

In order to execute the entire recovery we will need to place the new chunks in the list of chunks following the previous chunk. In the split case, we therefore proceed by checking if prev is in the process of freezing and if it is, we help it finish the freezing process and recover. Namely, we freeze prev, we look for prev's predecessor and then invoke the freeze recovery for prev (Lines 5-12). This may cause recursive recovery calls until the head of the chunk list, Line 8. During this time, there is a possibility that some other thread has helped recovering our own chunk and we therefore search for it in the list, Line 10. If we can't find it, we know that we are done and can return.

In the third phase we locally create new chunks to replace the frozen one (Lines 13,14). In the case of a split, two new half-full chunks are created from a single full frozen chunk, using the split() method. The first chunk, with the lower-valued part of the keys, points to the second chunk, with the higher-valued part. In the case of a merge, a new first chunk is created with M ordered keys taken from the frozen first chunk, the buffer and from the second chunk. This is done using the mergeFirstChunk() method. If there are too many frozen keys, a new

[2] The freezeRecovery() method is never called with a cur chunk being the buffer chunk.

[3] It is possible that we miss the freezing of the first chunk and start working on a split of the second chunk. In this case a later CAS instruction, in Line 16, will fail and we will repeat the recovery process with the adequate choice of a merge.

```
1  void freezeRecovery(Chunk* cur, Chunk* prev) {
2   bool toSplit = true; Chunk *local=NULL, *p=NULL;
3   while(1) { // PHASE I: decide whether to split or to merge
4       if (cur==head||(prev==head && prev->status.isInFreeze())) toSplit = false;
5       if (toSplit && prev->status.isInFreeze()){ //PHASE II: in split, if prev is frozen, recover it first
6         freezeChunk(prev);    // ensure prev freeze is done
7         if ( getChunk(&prev, &p) )      // search the previous to prev
8           freezeRecovery(prev, p);      // the frozen prev found, p precedes prev; recursive recovery
9         // prev is already not in the list; re-search the current chunk and find its new predecessor
10        if ( !getChunk(&cur, &p) ) return; // the frozen cur is not in the list
11        else {prev = p; continue;}
12      }
13      if (toSplit) local = split(cur);   // PHASE III: apply the decision locally
14      else     local = mergeFirstChunk(cur);
15      if (toSplit) { // PHASE IV: change the PQ accordingly to the previous decision
16        if ( CAS(&prev->next, cur, local) ) return;
17      } else { // when modifying the head, check if cur second or first
18        if (prev==NULL)
19          if( CAS(&head, cur, local) ) return
20          else if( CAS(&head, prev, local) ) return;
21      }
22      if (!getChunk(&cur,&p)) return; // look for new location; finish if the frozen cur is not found
23      else prev = p;
24  }
25 }
```

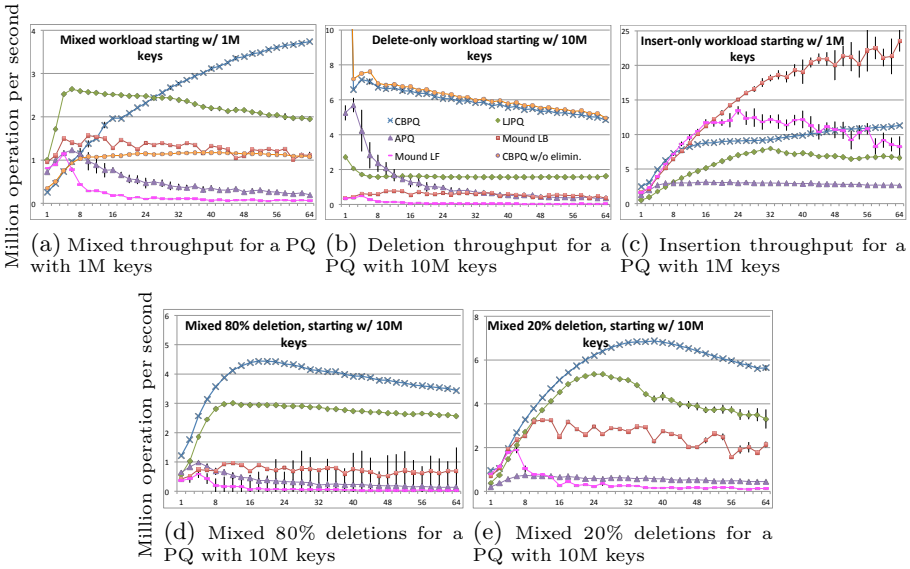
Listing 4: CBPQ recovery from a frozen chunk

first chunk and new second chunk can be created. The new first chunk is created without pointing to a buffer, it will be allocated when needed for insertion.

In phase IV, the thread attempts to attach its local list of new chunks to the chunk list. Upon success the thread returns. Otherwise, the recovery is retried, but before that, cur is searched for in the chunk list. If it is not there, then other threads have completed the recovery and we can safely return. Otherwise, a predecessor has been found for cur in the search and the recovery is re-executed.

4 Optimizations

First, in order to improve the search time for the relevant chunk, we use a simple lock-free skip-list from [13] to index the chunks. Updates to the skip-list are executed during splits and merges of chunks. Second optimization is exploiting the context switch waiting time. When an insert operation needs to insert a key to the buffer and it initiates a freeze of the first chunk. It is worth letting other threads run for a while before executing the freeze. We implemented this by yielding the processor to another thread (using usleep(0)). Third optimization is the elimination. According to the CBPQ algorithm, an insert of a small key k is done by inserting k into the buffer and waiting until the first chunk is rebuilt. During this time, if k becomes smaller than the current minimum key, then the insert operation can be eliminated by a deleteMin operation that consumes its key k. This can be viewed as if the insert and the deleteMin happened instantaneously one after the other just at the moment that k was smaller than the minimum key. Due to lack of space, the further details of the optimizations are omitted here and can be found in the full version of this paper [1].



(a) Mixed throughput for a PQ with 1M keys

(b) Deletion throughput for a PQ with 10M keys

(c) Insertion throughput for a PQ with 1M keys

(d) Mixed 80% deletions for a PQ with 10M keys

(e) Mixed 20% deletions for a PQ with 10M keys

Figure 2: Throughput in different workloads with an increasing threads number

5 Performance Evaluation

We implemented the CBPQ and compared it to the Linden's and Jonsson's PQ [15] (LJPQ), to the lock-free and lock-based implementation of Mounds, and to the adaptive priority queue (APQ) [5]. We chose these implementations, because they are the best performing priority queues in the literature and they were compared to other PQ implementations in [15,16,5]. We thank the authors of [16], [5] and [15] for making their code available to us. All implementations were coded in C++ and compiled with a -O3 optimization level.

We ran our experiments on a machine with 4 AMD Opteron(TM) 6272 16-core processors, overall 64 threads. The machine was operated by Linux OS (Ubuntu 14.04) and the number of threads was varied between 1 and 64. The chunk capacity (M) was chosen to be 928, so one chunk occupies a virtual page of size 4KB. The CBPQ implementation included the skip-list optimization of Section 4, we report results with and without elimination, and the results include the EBR memory management (as described in Section 3.2). The performance was evaluated using targeted micro-benchmarks: insertion-only or deletion-only workloads and mixed workloads where deletions and insertions appear with equal probability. The keys for insertions were uniformly chosen at random among all 30-bit sized keys. We ran each test 10 times and report average results. Error bars on the graphs show 95% confidence level.

A Mixed workload. We considered different percentages of insert and deleteMin operations. First, we evaluate the new algorithm using a stress test micro-benchmark, where each operation is invoked with the same probability. Figure 2a

shows the throughput of the 50-50-benchmark during one second on a PQ that is initiated with 1M keys before the measurement starts. The CBPQ (with elimination) is not a winner for a small number of threads, but outperforms LJPQ (the best among all competitors) by up to 80% when contention is high. Also, the CBPQ is the only implementation that scales for a large number of threads. The balanced workload is not favorable for the CBPQ design, because it drastically increases the probability that an insert will hit the first chunk. This happens because smaller values are repeatedly deleted and the first chunk holds higher and higher values. In contrast, the inserted values remain uniformly chosen in the entire range and hit the range of the first chunk more frequently. Hitting the first chunk often slows the CBPQ because inserts to the first chunk are the most costly. However, elimination excellently ameliorate this problem, especially for an increasing number of threads. Figures 2d and 2e show the CBPQ with 80% and 20% of deleteMin respectively, after the PQ was initiated with 10M keys. In both cases, the CBPQ surpasses the competitors for almost every thread count.

Deletion-only workload. Next, we measure the performance of the PQs when only deleteMin is executed. In order to make the deletion measurement relevant with deletion-only workload, we ensured that there are enough keys in the PQ initially so that deletions actually delete a key and never operate on an empty PQ. This requires initiating the PQ with 10M entries for the CBPQ. Elimination is not beneficiary in this case because there exist no pairs to match. Nevertheless, we show the results with and without elimination to highlight the negligible overhead of elimination for the CBPQ. In a deletion-only workloads we see a drastic performance improvement for the CBPQ. Results for the deletion-only workload are reported in Figure 2b. For a substantial number of threads, the CBPQ deletion throughput is up to 5 times higher than LJPQ throughput, and up to 8 times higher than the rest of the competitors.

Insertion-only workload. Similarly to the mixed workload, we start with a PQ that initially contains 1M random keys in it. During the test, we let a varying number of concurrent threads run simultaneously for 1 second, and we measure the throughput. Figure 2c shows the results. Mounds are designed for best performance with inserts of complexity $O(\log(\log(N)))$ and this indeed shows in our measurements. The CBPQ throughput is about 2 times worse than that of lock-based Mound, for a large number of threads. Note that for a smaller amount of threads, the advantage of Mounds is reduced significantly. More over, in spite of the advantage of Mounds with inserts, CBPQ significantly outperforms Mounds on a mixed set of operations. The CBPQ implementation outperforms LJPQ for inserts-only workloads. The performance of the insert operation with CBPQ is not affected by elimination, therefore the performance of CBPQ on inserts only operations does not change when using or not using elimination.

6 Conclusions

We presented a novel concurrent, linearizable, and lock-free design of the priority queue, called CBPQ. CBPQ cleverly combines the chunked linked-list, elimina-

tion technique, and the performance advantage of the $F\&I$ atomic instruction. We implemented CBPQ and measured its performance against Linden's and Jonsson's PQ (LJPQ) [15], adaptive PQ (APQ) [5] and the Mounds [16] (lock-free and lock-based), which are the best performing priority queues available. Measurements with a mixed set of insert and delete operations show that under high contention CBPQ outperforms all competitors by up to 80%.

References

1. Braginsky, A., Cohen, N., Petrank, E.: CBPQ: High performance lock-free priority queue (full version), http://www.cs.technion.ac.il/~erez/papers.html
2. Braginsky, A., Kogan, A., Petrank, E.: Drop the anchor: lightweight memory management for non-blocking data structures. SPAA (2013)
3. Braginsky, A., Petrank, E.: Locality-conscious lock-free linked lists. In: Proc. ICDCN (2011)
4. Brown, T.A.: Reclaiming memory for lock-free data structures: There has to be a better way. In: PODC (2015)
5. Calciu, I., Mendes, H., Herlihy, M.: The adaptive priority queue with elimination and combining. In: DISC (2014)
6. Cohen, N., Petrank, E.: Automatic memory reclamation for lock-free data structures. OOPSLA '15 (2015)
7. Cohen, N., Petrank, E.: Efficient memory management for lock-free data structures with optimistic access. pp. 254–263. SPAA '15, ACM (2015), http://doi.acm.org/10.1145/2755573.2755579
8. Dragicevic, K., Bauer, D.: Optimization techniques for concurrent stm-based implementations: A concurrent binary heap as a case study. In: IPDPS (2009)
9. Fraser, K.: Practical lock-freedom. In: PhD dissertation, University of Cambridge (2004)
10. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. pp. 300–314 (2001)
11. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. Journal of Parallel and Distributed Computing (2010)
12. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
13. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Pub. Inc. (2008)
14. Hunt, G., Michael, M., Parthasarathy, S., Scott, M.: An efficient algorithm for concurrent priority queue heaps. In: Information Processing Letters (1996)
15. Linden, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. OPODIS'13
16. Liu, Y., Spear, M.: Mounds: Array-based concurrent priority queues. In: Proc. ICPP (2012)
17. Lotan, I., Shavit, N.: Skiplist-based concurrent priority queues. In: Proc. IPDPS (2000)
18. Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks. Theory of Computing Systems (1997)
19. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. In: Journal of Parallel and Distributed Computing (2005)