CBPQ: High Performance Lock-Free Priority Queue*

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

[1] Yahoo! Labs Haifa anastas@yahoo-inc.com
[2] Technion - Israel Institute of Technology {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, artificial intelligence, 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 [10,12] 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.

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 [2] 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 performant 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 [19,9], which resolves the contention of concurrent inverse operations: the insert of a small key and a deleteMin. In our PQ, all small keys, being currently inserted, are placed in a special buffer, which a concurrent deleteMin operation can easily access to find a minimum. Such elimination bears no additional overhead for the insert and little overhead for the deleteMin operation.

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

The deleteMin operation is a bottleneck for concurrent PQ implementations, because it creates a single point of contention, and concurrent deleteMin invocations cause a performance degradation [14]. This is exactly the setting for which a $F\&I$ instruction is highly effective, improving over the use of a CAS instruction, as noted in [18,6].

In order to build a priority queue that can use $F\&I$, we adopt the *chunked linked-list* from [2], where each node (denoted *chunk*) in the list is an array of M elements. A main idea in the design is to distinguish the first chunk, which is hit by all the deleteMin operations from the other chunks, which handle most of the insert operations. The first chunk maintains sorted keys, allowing a fast deleteMin using the $F\&I$ instruction to delete the minimum entry, whereas the other chunks are not sorted allowing a quick insert with a fast $F\&I$ instruction that provides access to the next available cell in the chunk.

Following [2], the chunks are concurrently split or merged with other chunks to maintain predetermined size boundaries. This is done using a *freezing* mechanism, which notifies threads that old parts of the data structure (which they may currently be accessing) have been replaced by new ones. We denote the obtained PQ design a *Chunk-Based Priority Queue (CBPQ)*.

Various constructions for the concurrent PQ exist in the literature. Hunt et. al. [13] 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) [5]. A quiescently consistent skip-list based priority queue was first proposed by Lotan and Shavit [16] using fine-grained locking, and was later made non-blocking [7]. Another skip-list based priority queue was proposed by Sundell and Tsigas [20]. Liu and Spear [15] 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 performant among concurrent implementations of the PQ. Linden and Jonsson [14] 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 nodes marked as deleted exceed a given threshold. Their construction outperforms previous algorithms by $30-80\%$. Recently, Calciu et al. [3] 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 [19,9] 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 [3], 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 [14], the lock-free and lock-based implementations of the Mounds PQ [15], and the adaptive PQ (APQ) of Calciu et al. [3]. We evaluated the performance of our design 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 algorithm 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

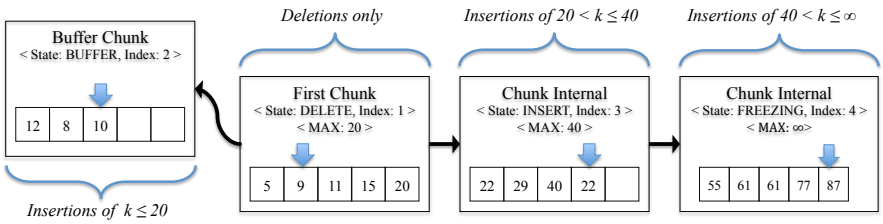


Figure 1: Overview of the CBPQ data structure

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 therefore supports deleteMin operations. We do not allow inserts into the first chunk. Instead an insertion of a key that is in the range of the first chunk goes through special handling, as discussed below. The remaining chunks (all but the first) 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 [2]. The CBPQ scheme is 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.

We then add several optimizations. First, we add a skip-list to enable fast access of the insert operation to the relevant chunk. Next, we add elimination, i.e. deleting threads randomly access a single entry in the buffer in hope to find a minimum that is being concurrently inserted. Such a bounded, random, opportunistic elimination enjoys a small cost on deleteMin operations with no overhead on insert operations. It improves scalability by increasing concurrency.

Finally, we add a throttling mechanism that delays inserts to the first chunk in order to allow elimination and the accumulation of such inserts before a costly freezing mechanism handles a replacement of the first chunk. The outcome of this design plus optimizations provides the desirable scalability and performance.

3 The Full CBPQ Design

In this section we describe the core CBPQ algorithm. In Sections 4 we describe the skip-list on top of the list and the elimination.

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.

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 [14], we use Keir Fraser's epoch based reclamation (EBR) [7] as a (simplified) solution for memory reclamation. In short, EBR lets each thread signal when it starts or completes an operation, by incrementing a per thread counter and setting a flag. When a frozen chunk can no longer be reached from the CBPQ, the maximal counter among all threads is recorded on it. When all threads' counters pass the recorded value, this chunk can be deallocated and reused.

The disadvantage of the EBR scheme is that it is not non-blocking. A stuck thread, blocked in the middle of an operation can block all further chunk reclamation. Since this event is rare, it is sometimes used to provide an initial memory reclamation technique, and this is exactly the scheme that was used in previous work to which we compare [14]. The other algorithms that we compare against [15,3] did not implement memory reclamation at all. More advanced lock-free memory reclamation solutions appear in [17,11,1,4]. Similarly to previous work, an implementation of such algorithms for the proposed algorithm is outside the scope of this paper.

3.3 Operations Implementation

Insert: The insert pseudo-code is presented in the insert() method in Listing 3. 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 5). If an insert must be performed to the first chunk, the insert_to_buffer() method is invoked (Line 7), 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 11). 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 14-19). 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 C is not freezing (Line 17), the insert is completed. Else if C is freezing, but our key was already marked as frozen (Line 18), 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 20, 21), 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 insert_to_buffer() method in Listing 3. It starts by allocating a new buffer holding the relevant key, if needed (Line 29). The create_buffer() 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. The create_buffer() method's pseudo-code can be found at the bottom of Listing 3. 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 insert_to_buffer() 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 39), 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 43-45) are explained later 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 3.

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

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

Listing 1: Status and Chunk records

```
1  int freezeKeys(Chunk c) {
2      // go over entries which are held by one freeze word
3      for(int k=0; k<M/VALS_PER_WORD; ++k){
4          uint64_t freezeWord = 1<<64; // 0s with MSB set to
               1
5          uint64_t mask = 1;
6          // prepare a mask for one frozen word
7          // VALS_PER_WORD is 63
8          for(int i=0; i<VALS_PER_WORD; ++i, mask<<=1)
               {
9              // read the entry
10             int entry = c->vals[i+k*VALS_PER_WORD];
11             if( entry != EMPTY_ENTRY )
12                 // the value exists, mark the relevant entry
13                 freezeWord |= mask;
14         } // end of preparing mask for a single frozen word
15         // try to update the frozen array
16         // after this CAS, surely MSB of the frozen word is set
17         atomicCAS(&c->meta.frozen[k], 0, freezeWord);
18     }
19 }
20
21 void freezeChunk(Chunk* c) {
22     int idx, frozenIdx = 0; Status localS; // locally copied status
23     while(1){ // PHASE I : set the chunk status if needed
24         // read the current status to get its state and index
25         localS = c->status; idx = localS.getIdx();
26         switch (localS.getState()){
27             case BUFFER: // in insert or buffer chunks
28             case INSERT: // frozenIdx was and remained 0
29                 c->status.aOr(MASK_FREEZING_STATE);
30                 break;
31             case DELETE:
32                 if (idx>M) frozenIdx=M; else frozenIdx=idx;
33                 Status newS; // set: state, index, frozen index
34                 newS.set(FREEZING, idx, frozenIdx);
35                 // New state CAS can be prevented by a delete
36                 // updating the index
37                 if ( c->status.CAS(localS, newS) ) break;
38                 else continue;
39             case FREEZING: // in process of being freezed
40                 frozenIdx = localS.frozenIdx;
41                 break;
42             case FROZEN:  // c was frozen by someone else
43                 // mark the chunk out-pointers as deleted
44                 c->markPtrs();
45                 return;
46         }
47         break;  // continue only if CAS from DELETE state
               failed
48     }
49     if (c != head) freezeKeys(c);  // PHASE II: freeze the
           entries
50     // move from FREEZING to FROZEN state using atomic
           OR
51     c->status.aOr(MASK_FROZEN_STATE);
52     c->markPtrs();  // set the chunk pointers as deleted
53 }
```

Listing 2: Freezing the keys and the entire chunk

```
1  void insert(int key) {
2      Chunk* cur = NULL, *prev = NULL;
3      while(1) {
4          // set the current and previous chunk pointers
5          getChunk(&cur, &prev, key);
6          if ( cur==head ) { // first chunk
7              if ( insert_to_buffer(key, cur, head) ) return;
8              else continue;
9          }
10         // atomically increase the index in the status
11         Status s = cur->status.aIncIdx();
12         int idx = getIdx(s);
13         // insert into a non-full and non-frozen chunk
14         if ( idx<M && !s.isInFreeze() ) {
15             cur->entries[idx] = key;
16             memory_fence;
17             if (!cur->status.isInFreeze()) return;
18             if (cur->entryFrozen(idx)) return; // key got
                   copied
19         }
20         freezeChunk(cur); // restructure the CBQP, then retry
21         freezeRecovery(cur, prev);
22     }
23 }
24
25 bool insert_to_buffer(int key, Chunk* cur, Chunk* curhead) {
26     Chunk *curbuf = cur->buffer; bool result = false;
27     // PHASE I: key insertion into the buffer
28     if( curbuf==NULL ) // the buffer is not yet allocated
29         if ( create_buffer(key,cur,&curbuf) )
30             goto phaseII;  // key added during buffer creation
31
32     // atomically increase the index in the status
33     Status s = curbuf->status.aIncIdx();
34     int idx = getIdx(s);
35     if ( idx<M && !s.isInFreeze() ) {
36         curbuf->entries[idx] = key;
37         memory_fence;
38         if (!curbuf->status.isInFreeze()) result = true;
39         if (curbuf->entryFrozen(idx)) return true;
40     }
41     // PHASE II: first chunk merges with buffer before insert
           ends
42 phaseII:
43     usleep(0); // yield, give other threads a chance
44     freezeChunk(cur);
45     freezeRecovery(cur, NULL);
46     return result;
47 }
48
49 int deleteMin() {
50     Chunk* cur, next;
51     while(1){
52         cur = head;
53         // atomically increase the index in the status
54         Status s = cur->status.aIncIdx();
55         int idx = getIdx(s);
56         // delete from not full and non-frozen chunk
57         if ( idx<M && !s.isInFreeze() )
58             return curr->entries[idx];
59         // First freeze, then remove frozen chunk from CBPQ
60         freezeChunk(cur);
61         freezeRecovery(cur, NULL); // retry after restructure
62     }
63 }
64
65 bool create_buffer(int key, Chunk chunk, Chunk* curbuf) {
66     Chunk *buf = alloc();
67     buf->vals[0] = key; // buffer is created with the key
68     bool result = CAS(&chunk->buffer,NULL,buf);
69     *curbuf = buf; // update buffer ptr (ours or someone's else)
70     return result;
71 }
```

Listing 3: Common code path: insertion of a key and deletion of the minimum

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 not new and we adopt the mechanism developed in [2] with minor modifications for this procedure. 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 [2], 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 (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 should be able 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 we note that 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 to the freezing process that an insert has not completed. In this case, it does not copy this entry 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. This processing appears in the freezeKeys() routine (Listing 2). 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 significantly.

Freezing the chunk: Following the pseudo-code of method freezeChunk() in Listing 2, 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 25-47). 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 29). 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 32). 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 37 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 44 or Line 52) as deleted (using the common Harris delete-bit technique [8]). 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 49) as explained above. 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 51) 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 [2] 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 (Lines 4-5). 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

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

the second chunk into the new first chunk. So if cur is the second chunk and the first chunk is currently freezing, then we know 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 7-17). This may cause recursive recovery calls until the head of the chunk list, Line 11. 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 14. 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 19,20). 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 the from the frozen first chunk, the t buffer and from the second chunk. This is done using the mergeFirstChunk() method. If there are too many frozen keys, a new 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 can return. Otherwise, the recovery is tried again, 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

A Skip-List Optimization: The CBPQ algorithm achieves very fast deletions. Previous work required complexity O(log$n$), whereas most CBPQ deletions can be executed in $O(1)$ complexity, which makes CBPQ deletions very efficient. However, as described in Section 3, insertions use linear search on the chunk list, which is slow compared to the logarithmic randomized search used in both the Mound PQ and the skip-list-based PQ. Initial measurements showed that most of the insertion time is spent on reaching the relevant chunk. In order to improve the search time for the relevant chunk, we added a simple lock-free skip-list from [12] to index the chunks. We have made only two changes to the skip-list of [12]. We first turned it into a dictionary, that holds a value in addition to the key. In our setting the key is the max value of a chunk and the value is a pointer to the chunk. Next, we modified it to return the value (associated with the key) as a result of a search for a key k. If k is in the set, then its associated value is returned. Otherwise, the value associated with (i.e., a pointer to the chunk whose max value is) the largest key smaller than k is returned.

Deletions are not affected by the skip-list. It is only used for insert operations. When a key needs to be inserted, the relevant chunk is found using the skip-list. As the chunk list is modified by splits and merges, we modify the skip-list to reflect the changes. Updates to the skip-list are executed as follows. After finding the appropriate chunk, the insert operation is executed as specified in the previous section. When a chunk restructuring is needed, we start by executing a split or a merge as described in Section 3.4. Next, we let the (unique) thread that succeeded to insert its local new chunk into the chunk list and make it the replacement chunk (Lines 23,26 or 27 in the freezeRecovery() method) make the necessary updates to the skip-list. It uses CAS to

[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 23, 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()) )
5              toSplit = false;
6          // PHASE II: in split, if prev is frozen, finish its recovery first
7          if ( toSplit && prev->status.isInFreeze() ){
8              freezeChunk(prev); // ensure prev freeze is done
9              if ( getChunk(&prev, &p) ){ // search the previous to prev
10                 // the frozen prev found in the list, p precedes prev
11                 freezeRecovery(prev, p); // invoke recursive recovery
12             }
13             // prev is already not in the list; re-search the current chunk
14             if ( !getChunk(&cur, &p) ) { // and find its new predecessor
15                 return; // the frozen cur is not in the list
16             } else {prev = p; continue;}
17         }
18         // PHASE III: apply the decision locally
19         if (toSplit) local = split(cur);
20         else local = mergeFirstChunk(cur);
21         // PHASE IV: change the PQ accordingly to the previous decision
22         if (toSplit) {
23             if ( CAS(&prev->next, cur, local) ) return;
24         } else { // when modifying the head, check if cur second or first
25             if(prev==NULL)
26                 if( CAS(&head, cur, local) ) return
27             else if( CAS(&head, prev, local) ) return;
28         }
29         if ( !getChunk(&cur,&p) ) // before retry check for new location
30             return; // the frozen cur is not in the list
31         else prev = p;
32     }
33 }
```

Listing 4: CBPQ recovery from a frozen chunk

replace the frozen, obsolete chunk pointer in the skip-list with the pointer to the new chunk, then it adds the second, new chunk.

Since the skip-list is not tightly coupled with the insertions and deletions of chunks from the chunk list, it is possible that the skip-list will lead to a frozen chunk, or that it will not find a desired chunk. If, during execution of an insert operation, a search of the skip-list leads to a frozen chunk, a recovery of that chunk is invoked, the chunk is removed from the skip-list, and the search can be restarted. In contrast, if we can not find the chunk we need through the skip-list search, we choose a chunk that is closest to it and that precedes it in the skip-list. Next, we simply proceed by walking linearly on the actual chunk list to find the chunk we need.

A Processor Yielding Optimization: When an insert operation needs to insert a key to the first chunk, it initiates a freeze of the first chunk. Frequent freezings of the first chunk reduce performance. We found out that it is worth letting other threads run for a while before executing the freeze. This allows several insert operations to place their items in the buffer and complete, before the freezing begins. When the freezing begins, the buffer is blocked for further insertions and all current items in the buffer are processed. We implemented this by yielding the processor to another thread (using usleep(0)), after the key is successfully placed in the buffer chunk (Line 43, Listing 3). Attempting to sleep for 5 microseconds, instead of just yielding, turned out to decrease the performance.

A Lightweight Elimination Optimization: Elimination suits the CBPQ design well. According to the CBPQ algorithm, an insert of a key k into the first chunk is done by inserting k into the buffer and waiting until the first chunk is rebuilt from itself and the buffer. During this blocking 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. The two eliminated operations may return as soon as the elimination is complete.

We let the insert put its key into the buffer as usual. When a deleteMin starts, it checks that the buffer exists and is not in the process of freezing. If this is the case, it randomly picks a random key k from the buffer[4]. If k is smaller than the current minimum and not yet paired, it attempts to pair with it. It does so by attempting to atomically set the most significant bit (MSB) of the key using a CAS instruction. If the CAS succeeds, then the deleteMin operation returns k as the current minimum. If the elimination attempt fails, the elimination attempt can be repeated X times. In our experiments we try three times. If all attempts fail, the deletion proceeds as usual with no elimination. We empirically found that such a random choice of the an elimination pair is more efficient than a linear exhaustive search for an elimination pair.

It remains to explain how elimination interacts with freezing. This interaction only slightly complicates the algorithm and it works as follows. The freezing procedure ignores keys that are marked as paired for elimination. These keys of the buffer chunk are not frozen and are not copied to the new chunks.

The elimination procedure acts as follows. It starts by checking that the current buffer is not marked as frozen (i.e., that its state is not freeze). If it is freezing, elimination cannot proceed and it returns failure. Otherwise, suppose a key is found and marked as paired. Then, the elimination procedure checks again whether the buffer is marked as frozen. If no freeze is detected, the paired key is returned for deleteMin and elimination is complete. Otherwise, the elimination proceeds carefully by executing part of the concurrent freezing of the buffer. It looks at the word of freeze bits that is relevant for the current entry and it attempts to write it for the freeze procedure, leaving the relevant entry (for the elimination) unfrozen. Whether it succeeds or fails, this freeze word must have been written at this point, and recall that the freeze word can only be written once. The elimination procedure uses the freeze word to determine whether the relevant entry is set as frozen or not. If it is not frozen, then elimination has succeeded. Otherwise, it failed.

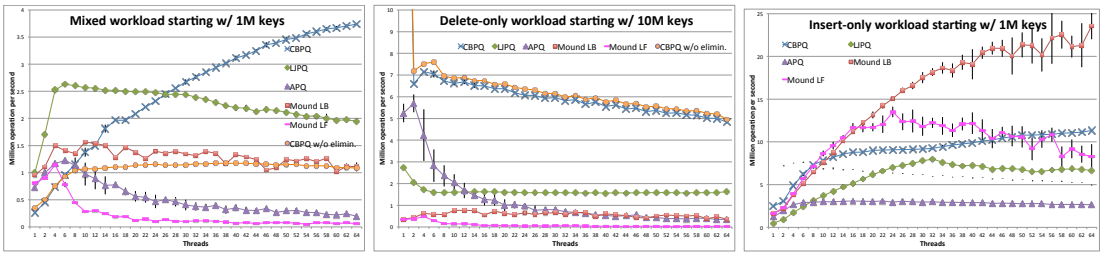
The proposed elimination mechanism has very low overhead. It requires no scanning of any additional elimination structure, and there is no additional waiting time for matching. The thread that inserts a small key only needs to yield the processor after the key is placed in the buffer and before checking whether the key was paired with a deletion.

5 Performance Evaluation

We implemented the CBPQ and compared it to the Linden's and Jonsson's PQ [14] (LJPQ), to the lock-free and lock-based implementation of Mounds, and to the adaptive priority queue (APQ) [3]. We chose these implementations, because they are the best performing priority queues in the literature and they were compared to other PQ implementations in [14,15,3]. We thank the authors of [15], [3] and [14] 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). Running measurements without using memory management provides better performance by at most 4%. Therefore, comparing

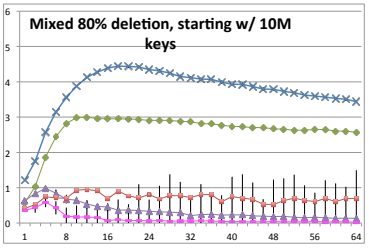
[4] The last T keys in the buffer have higher chances to not be already paired, where T is number of threads. Therefore, as an additional optimization, we prefer to choose one of the last T inserted keys.



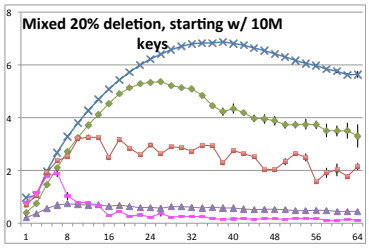
(a) Mixed throughput for a PQ with 1M keys

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

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



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



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

Figure 2: Throughput in different workloads.

to schemes that do not use memory reclamation is fair. 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. Following tradition, we evaluate the new algorithm using a stress test micro-benchmark, where each operation is invoked with the same probability. Note that such a workload is not favorable for the CBPQ design. A workload of equally probable insert or deleteMin operations drastically increases the probability that an insert will hit the first chunk. The reason for that is that smaller values are repeatedly deleted, and so the first chunk holds higher and higher values. In contrast, the inserted values remain uniformly chosen in the entire range, and so they hit values that are within the range of the first chunk more frequently. Measurements show that when only (random) insertions are performed, then the probability of hitting the first chunk is 0.15% (about once every 600 inserts). However, when we run a mix of inserts and deleteMins, the probability of insertion to the first chunk increases dramatically into 15%. Hitting the first chunk often is not good for 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. The reported measurements indeed show good scalability and high performance under contention. We expect CBPQ to do even better for different workloads, e.g., for a workload that is built of alternating batches of inserts and batches of deletes.

In Figure 2a we report the throughput of the mix-benchmark during one second on a PQ that is initiated with 1M entries before the measurement starts. It turns out that 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. We demonstrate the importance of elimination by also including performance results when no elimination is used. 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 Linearization Points

Let us specify the CBPQ linearization points. As a rule, the CBPQ's linearization points happen only during modifications of non-frozen chunks.When an operation accesses a frozen chunk C, it recovers C, but a linearization point will not happen during the recovery. We first describe the linearization points when no elimination occurs.

The linearization point of deleteMin is straightforward. It is set to the atomic increase of the index in a non-frozen first chunk. In the deleteMin() method the linearization point is set to Line 54 (Listing 3), conditioned on the event that later in Line 57 the chunk is not detected as frozen. If the chunk is frozen, or the index is too high, then the operation will restart after the recovery and Line 54 will be executed again.

We divide the discussion for the insert operation's linearization point into two different cases. The first case is that the entry is inserted into the first chunk and the second case is that the entry is inserted into a non-first chunk. We start with the latter and select the linearization point of an insert to a non-first chunk to be at the write of a key-value pair into the appropriate array entry of the non-frozen chunk. In the insert() method this linearization point happens in Line 15, conditioned on the event that later in Line 17 the chunk is not detected as frozen.

An insertion into the first chunk is more complicated. First, the entry is put in the buffer chunk, then, the first chunk freezes, and then the buffer freezes, and during the recovery of the first chunk, the entry is added to it. We let the linearization point for this insertion be the point in which the buffer was frozen, given that the entry was successfully added to the buffer before the buffer froze. Formally, we condition on the entry being inserted into the buffer in Line 36 of the insert_to_buffer() method and later in Line 38 it turns out that this entry has been properly

admitted into the buffer before the buffer itself was frozen. Otherwise, if this entry was not admitted, its insertion will be retried after the first chunk (and the buffer) are recovered. Given the above condition, the linearization point is set to the time in which the buffer's state becomes FROZEN. This happens with a state frozen bit being firstly set in Line 51 of the freezeChunk() method (Listing 2).

When elimination occurs (the pair-marking and the following freeze check are successful), both the deleteMin() and the insert() methods' linearization points are set to the successful marking of the key in the buffer as paired, conditioned on the event that later the entry is not found frozen. If multiple keys are eliminated at the same time, the linearization points follow the keys order.

7 Conclusions

We presented a novel concurrent, linearizable, and lock-free design of the priority queue data structure, called CBPQ. CBPQ cleverly combines the chunked linked-list, elimination 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) [14], adaptive PQ (APQ) [3] and the Mounds [15] (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., Kogan, A., Petrank, E.: Drop the anchor: lightweight memory management for non-blocking data structures. SPAA (2013)
2. Braginsky, A., Petrank, E.: Locality-conscious lock-free linked lists. In: Proc. ICDCN (2011)
3. Calciu, I., Mendes, H., Herlihy, M.: The adaptive priority queue with elimination and combining. In: DISC (2014)
4. Cohen, N., Petrank, E.: Automatic memory reclamation for lock-free data structures. OOPSLA '15 (2015)
5. Dragicevic, K., Bauer, D.: Optimization techniques for concurrent stm-based implementations: A concurrent binary heap as a case study. In: IPDPS (2009)
6. Fatourou, P., Kallimanis, N.D.: A highly-efficient wait-free universal construction. SPAA (2011)
7. Fraser, K.: Practical lock-freedom. In: PhD dissertation, University of Cambridge (2004)
8. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. pp. 300–314 (2001)
9. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. Journal of Parallel and Distributed Computing (2010)
10. Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. 13(1), 124–149 (1991)
11. Herlihy, M., Luchangco, V., Martin, P., Moir, M.: Nonblocking memory management support for dynamic-sized data structures. ACM Trans. Comput. Syst. (2005), http://doi.acm.org/10.1145/1062247.1062249
12. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Pub. Inc. (2008)
13. Hunt, G., Michael, M., Parthasarathy, S., Scott, M.: An efficient algorithm for concurrent priority queue heaps. In: Information Processing Letters (1996)
14. Linden, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. OPODIS'13
15. Liu, Y., Spear, M.: Mounds: Array-based concurrent priority queues. In: Proc. ICPP (2012)
16. Lotan, I., Shavit, N.: Skiplist-based concurrent priority queues. In: Proc. IPDPS (2000)
17. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Transactions on Parallel and Distributed Systems 15, 491–504 (2004)
18. Morrison, A., Afek, Y.: Fast concurrent queues for x86 processors. In: Proc. PPOPP (2013)
19. Shavit, N., Touitou, D.: Elimination trees and the construction of pools and stacks. Theory of Computing Systems (1997)
20. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. In: Journal of Parallel and Distributed Computing (2005)