Locality-Conscious Lock-Free Linked Lists [*]

Anastasia Braginsky[†] Erez Petrank[‡]

December 17, 2010

Abstract

We extend state-of-the-art lock-free linked lists by building linked lists with special care for locality of traversals. These linked lists are built of sequences of entries that reside on consecutive chunks of memory. When traversing such lists, subsequent entries typically reside on the same chunk and are thus close to each other, e.g., in same cache line or on the same virtual memory page. Such cache-conscious implementations of linked lists are frequently used in practice, but making them lock-free requires care. The basic component of this construction is a chunk of entries in the list that maintains a minimum and a maximum number of entries. This basic chunk component is an interesting tool on its own and may be used to build other lock-free data structures as well.

1 Introduction

Lock-free (also known as non-blocking) data structures provide a progress guarantee. If several threads attempt to concurrently apply an operation on the structure, it is guaranteed that one of the threads will make progress in finite time [7]. Many lock-free data structures have been developed since the original notion was presented [11]. Concurrent algorithms in general, and lock-free algorithms in particular, are error-prone and modifying existing algorithms requires care. In this paper we study lock-free linked lists and propose a design for a cache-conscious linked list.

The first design of lock-free linked lists was presented by Valois [12]. He maintained auxiliary nodes in between the list's normal nodes, in order to resolve concurrent operations' interference problems. Also, each node in his list had a backlink pointer, which pointed to its predecessor when the node was deleted. These backlinks were then used to backtrack through the list when there was interference from a concurrent operation. A different lock-free implementation of linked lists was given by Harris [6]. His main idea was to mark a node before deleting it in order to prevent concurrent operations from changing its next-entry pointer. Harris' algorithm is simpler than Valois's algorithm and his experimental results generally also perform better. Michael [8, 10] proposed an extension to Harris' algorithm that did not assume a garbage collection but reclaimed entries of the list explicitly. To this end, he developed an underlying mechanism of *hazard pointers* that was later used for explicit reclamation in other data structures as well. An improvement in

[*]Supported by THE ISRAEL SCIENCE FOUNDATION (grant No. 845/06).

[†]Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel `anastas@cs.technion.ac.il`

[‡]Dept. of Computer Science, Technion - Israel Institute of Technology, Haifa 32000, Israel `erez@cs.technion.ac.il`

complexity was achieved by Fomitchev and Rupert [3]. They use a smart retreat upon CAS failure, rather than the standard restart from scratch.

In this paper we further extend Michael's design to allow cache-conscious linked lists. Our implementation partitions the linked list into sub-lists that reside on consecutive areas in the memory, denoted *chunks*. Each chunk contains several consecutive list entries. For example, setting each chunk to be one virtual page, causes list traversals to form a page-oriented memory access pattern. This partition of the list into sub-lists, each residing on a small chunk of memory is often used in practice (e.g., [1, 5]), but there is no lock-free implementation for such a list. Breaking the list into chunks can be trivial if there is no restriction on the chunk size. In particular, if the size of each chunk can decrease to a single element, then clearly, each chunk can trivially reside in a single memory block, Michael's implementation will do, but no locality improvement will be obtained for list traversals. The sub-list's chunk that our design provides maintains upper and lower bounds on the number of elements it has. The upper bound simply follows from the size of the memory block on which the chunk is located, and a lower bound is provided by the user. If a chunk grows too much and cannot be held in a memory block, then it is *split* (in a lock-free manner) creating two chunks, each residing at a separate location. Conversely, if a chunk shrinks below the lower bound, then it is *merged* (in a lock-free manner) with the previous chunk in the list. In order for the split to create acceptable chunks, it is required that the lower bound (on the number of objects in a chunk) does not exceed half of the maximum number of entries in the chunk. Otherwise, a split would create two chunks that violate the lower bound.

A natural optimization of search for such a list is to quickly jump to the next chunk (without traversing all its entries), if the desired key is not within the key-range of this chunk. This gives us additional performance improvement since the search progress is done in skips, where the size of each skip is at least the chunk's minimal boundary. Furthermore the retreat upon CAS failure, in the majority of the cases is done by returning to beginning of the chunk, rather than the standard restart from the beginning of the list.

To summarize, the contribution of this paper is the presentation of a lock-free linked list, based on single word CAS commands, were the keys are unique and ordered. The algorithm does not assume a assume no lock-free garbage collector. The list design is locality conscious. The design poses a restriction on the keys and data length. For 64bit architecture the key is limited to 31 bit, and the data is limited to 32 bit.

Organization. In Section 2 we specify the underlying structure we use to implement the chunked linked list. In Section 3 we introduce the freeze mechanism that will serve the split and join operations. In Section 4 we provide the implementation of the linked list functions. A closer look at the freezing mechanism details appear in Section 5 and we conclude in Section 6. The full details of the freezing mechanism are presented in Appendix A. Appendices B, C, and D provide the full implementations of the operations on the list that did not fit into the main body of the paper. In Appendix E we specify the linearization points for the linked list operations, and in Appendix F we explain the design considerations and provide some intuition about the correctness of the algorithm.

2 Preliminaries and Data Structure

A linked list is a data structure that consists of a sequence of data records. Each data record contains a key by which the linked list is ordered. We denote each data record *an entry.* We think

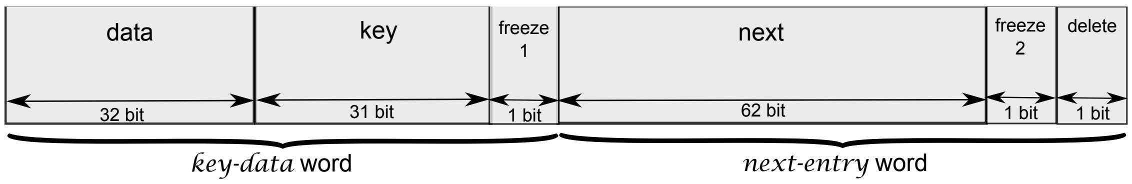

Figure 1: The entry structure.

of the linked list as representing a set of keys, each associated with a data part. Following previous work [2, 4, 6], a key cannot appear twice in the list. Thus, an attempt to insert a key that exists in the list fails. Each entry holds the key and data associated with it. Generally, this data is a pointer, or a mapping from the key to a larger piece of data associated with it. Next, we present the underlying data structure employed in the construction. We assume a 64-bit platform in this description. A 32-bit implementation can easily be derived, by either cutting each field in half, or by keeping the same structure, but using a wide compare-and-swap, which writes atomically to two consecutive words.

The structure of an entry A list entry consists of a *key* and a *data* fields, and the *next* pointer (pointing to next entry). These fields are arranged in two words, where the *key* and *data* reside in the first word and the *next* pointer in the second. Three more bits are embedded in these two words. First, we embed the *delete* bit in the least bit of the *next* pointer, following Harris [6]. The *delete* bit is set to mark the logical deletion of the entry. The *freeze* bits are new in this design. They take a bit from each of the entry's words and their purpose is to indicate that the entire chunk holding the entry is about to be retired. These three flags consume one bit of the key and two bits from the *next* pointer. Notice that the three LSBs of a pointer do not really hold information on a 64-bit architecture. The entry structure is depicted in Figure 1. In what follows, we refer to the first word as the *keyData* word, and the second word as the *nextEntry* word.

We further reserve one key value, denoted by $\perp$ to signify that the entry is currently not allocated. This value is not allowed as a key in the data structure. As will be discussed in Section 4, an entry is available for allocation if its key is $\perp$ and its other fields are zeroed.

The structure of a chunk The main support for locality stems from the fact that consecutive entries are kept on a *chunk*, so that traversals of the list demonstrate better locality. In order to keep a substantial number of entries on each chunk, the linked list makes sure that the number of entries in a chunk is always between the parameters MIN and MAX. The main part of a chunk is an array that holds the entries in a chunk and may hold up to MAX entries of the linked list. In addition, the chunk holds some fields that help manage the chunk. First, we keep one special entry that serves as a dummy header entry, whose *next* pointer points to the first entry in this chunk. The dummy header is not a must, but it simplifies the algorithm's code. To identify chunks that are too sparse, each chunk has a counter of the number of entries currently allocated in it. In the presence of concurrent mutations, this counter will not always be accurate, but it will always hold a lower bound on the number of allocated entries in the chunk. When an attempt is made to insert too many entries into a chunk, the chunk is split. When it becomes too small due to deletions, it is merged with a neighboring chunk. We require $\text{MAX} > 2{\cdot}\text{MIN}{+}1$, since splitting a large chunk must create two well-formed new chunks. In practice MAX will be substantially larger than $2{\cdot}\text{MIN}$ to avoid

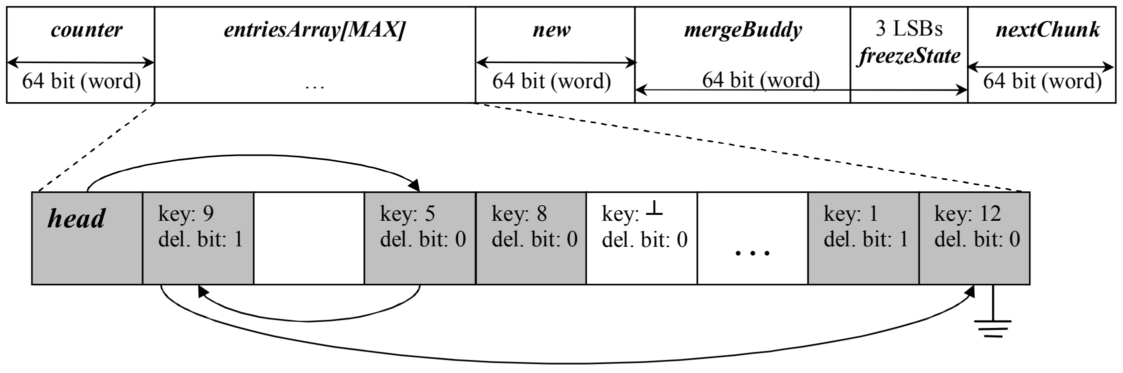

Figure 2: The chunk structure.

frequent splits and merges. Additional fields (*new*, *mergeBuddy* and *freezeState*) are needed for running the splits and the merges and are discussed in Section 5. The chunk structure is depicted in Figure 2.

The structure of entire list The entire list consists of a list of chunks. Initially we have a HEAD pointer pointing to an empty first chunk. We let the first chunk's MIN boundary be set to 0, to allow small lists. The list grows and shrinks due to the splitting and merging of the chunks. Every chunk has a pointer *nextChunk* to the next chunk, or to NULL if it is the last chunk of the list. The keys of the entries in the chunks never overlap, i.e., each chunk contains a consecutive subset of keys in the set, and a pointer to the next chunk, containing the next subset (with strictly higher keys) in the set. The entire list structure is depicted in Figure 3. We set the first key in a chunk as its lowest possible key. Any smaller key is inserted in the previous chunk (except for the first chunk that can also get keys smaller than its first one.)

Hazard pointers Whole chunks and entries inside a chunk are reclaimed manually. Note that garbage collectors do not typically reclaim entries inside an array. To allow safe (and lock-free) reclamation of entries manually, we employ Michael's hazard pointers methodology [8, 10]. While a thread is processing an entry - and a concurrent reclamation of this entry can foil its actions - the thread registers the location of this entry in a special pointer called a *hazard pointer*. Reclamation of entries that have hazard pointers referencing them is avoided. Following Michael's list implementation [10], each thread has two hazard pointers, denoted *hp0* and *hp1* that aid the processing of entries in a chunk. We further add four more hazard pointers *hp2*, *hp3*, *hp4*, and *hp5*, to handle the operations of the chunk list. Each thread only updates its own hazard pointers, though it can read the other threads' hazard pointers.

3 Using a Freeze to Retire a Chunk

In order to maintain the minimum and maximum number of entries in a chunk, we devised a mechanism for *splitting* dense chunks, and for *merging* a sparse chunk with its predecessor. The main idea in the design of the *split* and *merge* lock-free mechanisms is the *freezing* of chunks. When a chunk needs to be split or merged, it is first frozen. No insertions or deletions can be executed on

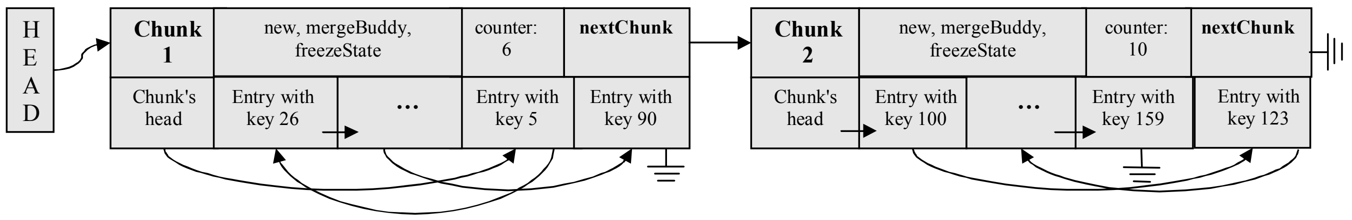

Figure 3: The list structure.

a frozen chunk. To split a frozen chunk, two new chunks are created and the entries of the frozen chunk are copied into them. To merge a frozen chunk with a neighbor, the neighbor is first frozen, and then one or two new chunks are allocated and the relevant entries from the two merging chunks are copied into them. Details of the freezing mechanism appear in Section 5. We now review this mechanism in order to allow the presentation of the list operations.

The freezing of a chunk comprises three phases:

Initiate Freeze: When a thread decides a chunk should be frozen, it starts setting the freeze bits in all its entries one by one. During the time it takes to set all these bits, other threads may still modify the entries not yet marked as frozen. During this phase, only part of the chunk is marked as frozen, but this freezing procedure cannot be reversed, and frozen entries cannot be reused.

Stabilizing: Once all entries in a chunk are frozen, allocations and deletions can no longer be executed. At this point, we link the non-deleted entries into a list. This includes entries that were allocated, but not yet connected to the list. All entries that are marked as deleted are disconnected from the list.

Recovery: The number of entries in the stabilized list is counted and a decision is made whether to split this chunk or merge it with a neighbor. Sometimes, due to changes that happen during the first phase, the frozen chunk becomes a good one that does not require a split or a join. Nevertheless, the retired chunk is never resurrected. We always allocate a new chunk to replace it and copy the appropriate values to the new chunk. Whatever action is decided upon (*split*, *join*, or *copy chunk*) must be carried through.

Any thread that fails to insert or delete a key due to the progress of a freeze, joins in helping the freezing of the chunk. However, threads that perform a search, continue to search in frozen chunks with no interference.

4 The List Operations: Search, Insert and Delete

We now turn to describe the basic linked list operations. The high-level code for an insertion, deletion, or search of a key is very simple and is presented in the Algorithm 1. We need to find the appropriate chunk associated with the appropriate range of keys and then invoke the relevant method on the returned chunk. Finally, we need to release the hazard pointers set by the *FindChunk* method to allow future reclamation. The main challenge is in the work inside the chunk and the handling of the freeze process, on which we elaborate below. More details on handling the higher level chunks list appears in Appendix C.

Turning to the operations inside the chunks, the delete and search methods are close to the previous design [10], except for the special treatment of the chunk bounds and the freeze status.

Algorithm 1 Search, Insert, and Delete – High Level Methods.

```
(a) BOOL Search (key, *data) {
 1: chunk* chunk = FindChunk(key);
 2: result = SearchInChunk(chunk, key, data);
 3: hp5 = hp4 = hp3 = hp2 = NULL; return result;
}
(b) BOOL Insert (key, data) {
 1: chunk* chunk = FindChunk(key);
 2: result = InsertToChunk(chunk, key, data);
 3: hp5 = hp4 = hp3 = hp2 = NULL; return result;
}
(c) BOOL Delete (key, data) {
 1: chunk* chunk = FindChunk(key);
 2: result = DeleteInChunk(chunk, key);
 3: hp5 = hp4 = hp3 = hp2 = NULL; return result;
}
```

However, the insert method is quite different, because it must allocate an entry in a shared memory (on the chunk), whereas previously, it was assumed that the insert allocates a local space for a new entry and privately prepares it for insertion in the list.

For the purpose of handling the entries list in the chunk, we maintain five variables that are global and appear in all the code below. These variables are global for each thread's code, but are not shared between threads, and all of them follow Michael's design [10]. The first three per-thread shared variables are (*entry** prev*), (*entry* cur*), and (*entry* next*). The other two are the two pointers (*entry** hp0*) and (*entry** hp1*) that point to the two hazard pointers of the thread. All other variables are local to the method that mentioned them.

4.1 The insert operation

The *InsertToChunk* method inserts a key with its associated data into a chunk. It first attempts to find an available entry and allocate it with the given key. If no available entry exists, a split is executed and the operation is retried. If an entry is obtained, the *InsertEntry* method is invoked to insert the entry into the list. The insertion will fail if the key already exists in the chunk. In this case *InsertToChunk* clears the entry to free it for future allocations.

The *InsertToChunk* code is presented in Algorithm 2. It starts by an attempt to find an available entry for allocation. A failure occurs when all entries are in use and in this case a freeze is initiated. The *Freeze* method gets the key and data as an input, and also an input indicating that it is invoked by an insertion operation. This allows the *Freeze* method to try to insert the key to the newly created chunk. When successful, it returns a NULL pointer to indicate the completion of the insertion. It also sets a local variable *result* to indicate whether the completed insertion actually inserted the key or it completed by finding that the key already existed in the list (which is also a legitimate completion of the insertion operation). If the insertion is not completed by the *Freeze* method, then it returns a pointer to the chunk on which the insertion should be retried.

Connecting the entry to the list is done by *InsertEntry*. If the entry gets allocated and linked

Algorithm 2 Insert a key and its associated data into a chunk

```
Bool InsertToChunk (chunk* chunk, key, data) {
current = AllocateEntry(chunk, key, data);                    // Find an available entry
while ( current == NULL ) {                          // No available entry. Freeze and try again
    chunk = Freeze(chunk, key, data, INSERT, &result);
    if ( chunk == NULL ) return result;                    // Freeze completed the insertion.
    current = AllocateEntry(chunk, key, data);                // Otherwise, retry allocation
}
returnCode = InsertEntry(chunk, current, key);
switch ( returnCode ) {
    case SUCCESS_THIS:
        IncCount(chunk); result = TRUE; break;  // Increments the entries' counter in the chunk
    case SUCCESS_OTHER:           // Entry was inserted by other thread due to help in freeze
        result = TRUE; break;
    case EXISTED:                                  // This key exists in the list. Reclaim entry
        if ( ClearEntry(chunk, current) )                           // Attempt to clear the entry
            result = FALSE;
        else                           // Failure to clear the entry implies that a freeze thread
            result = TRUE;                                         // eventually inserts the entry
        break;
} // end of switch
*hp0 = *hp1 = NULL  return result;                       // Clear all hazard pointers and return
}
```

to the list, then the chunk counter is incremented only by the thread that linked the entry itself. If the key already existed in the list, then *ClearEntry* attempts to clear the entry for future reuse. However, a rare scenario may foil clearing of the entry. This happens when the other occurrence of the key (which existed previously in the list) gets deleted before our entry gets cleared. Furthermore, a freeze occurs, in which the semi-allocated entry gets linked by other threads into the new chunk's list. At this point, clearing this entry is avoided, and *ClearEntry* returns FALSE. In such a scenario, clearing the entry fails and the insert operation succeeds.

At the end of *InsertToChunk*, all hazard pointers are cleared and we return with a code specifying if the insert was successful, or the key previously existed in the list.

The allocation of an available entry is executed using the *AllocateEntry* method, depicted in Algorithm 3. An available entry contains $\perp$ as a key and zeros otherwise. An available entry is allocated by assigning the key and data values in the *keyData* word in a single atomic compare-and-swap (CAS) that assumes this word has the $\perp$ symbol and zeros in it. An entry whose *keyData* has the freeze bit set cannot be allocated as it is not properly zeroed. Note also that once an entry is allocated, all the information required for linking it to the list is available to all threads. Thus, if a freeze starts, then all threads may create a stabilized list of the allocated entries in a chunk. The *AllocateEntry* method searches for an available entry. If no free entry can be found, NULL is returned.

Next, comes the *InsertEntry* method, which takes an allocated entry and attempts to link it to the linked list. The *InsertEntry* code is presented in Algorithm 4. The input parameter *entry* is a

Algorithm 3 Entry allocation

```
entry* AllocateEntry(chunk* chunk, key, data) {
 1: keyData = combine(key, data);            // Combine into the structure of a keyData word
 2: expecEnt = combine(⊥, 0);
 3: foreach entry e                                                  // Traverse entries in chunk
 4:    if ( e→keyData == expecEnt )
 5:       if ( CAS(&(e→keyData), expecEnt, keyData) ) return e;   // Atomically try to allocate
 6: return NULL;                                                  // No free entry was found
}
```

Algorithm 4 Connecting an allocated entry into the list

```
returnCode InsertEntry (chunk* chunk, entry* entry, key) {
 1: while ( TRUE) {
 2:    savedNext = entry→next;
 3:    // Find insert location and pointers to previous and current entries
 4:    if ( Find(chunk, key) )        // This key existed in the list, cur is global initiated by Find
 5:       if ( entry == cur ) return SUCCESS_OTHER; else return EXISTED;
 6:    // If neighborhood is frozen, keep it frozen
 7:    if ( isFrozen(savedNext) ) markFrozen(cur);                    // cur will replace savedNext
 8:    if ( isFrozen(cur) ) markFrozen(entry);                            // entry will replace cur
 9:    // Attempt linking into the list
10:    if (!CAS(&(entry→next), savedNext, cur) ) continue;          // Attempt setting next field
11:    if ( !CAS(prev, cur, entry) ) continue;  // Attempt linking, prev is global initiated by Find
12:    return SUCCESS_THIS;                                          // both CASes were successful
13: }
}
```

pointer to an entry that should be inserted. It is already allocated and initiated with key and data.

Before searching for the location to which to connect this entry, we memorize this entry's next pointer. Normally, this should be a NULL, but in the presence of concurrent executions of *InsertEntry* (which may happen during a freeze), we must make sure later that the entry's next pointer was not modified before we atomically wrote it in Line 10. After saving the current next pointer, we search for the entry's location via the *Find* method. If the key already exists in the list, *InsertEntry* checks whether the returned entry is the same as the one it is trying to insert (by address comparison). The result determines the return code: either the key existed and we failed, or the key was inserted, but not by the current thread. (This can happen during a freeze when all threads attempt to stabilize the frozen list.) Otherwise, the key does not exist, and *Find* sets the global variable *cur* with a pointer to the entry that should follow our entry in the list, and the global variable *prev* with the pointer that should reference our entry. The *Find* method protects the entries referenced by *prev* and *cur* with the hazard pointers *hp1* and *hp0*, respectively. There is no need to protect the newly allocated entry because it cannot be reclaimed by a different thread.

If any to-be-modified pointer is marked as frozen, we make sure that its replacement is marked as frozen well. An allocation of an entry can never occur on a frozen entry. However, once the allocation is successful, the new entry may freeze and still *InsertEntry* should connect it to the list.

Algorithm 5 The main freeze method.

```
chunk* Freeze(chunk* chunk, key, data, triggerType trigger, Bool* result) {
1:  CAS(&(chunk→freezeState), NO_FREEZE, INTERNAL_FREEZE);
2:  // At this point, the freeze state is either INTERNAL_FREEZE or EXTERNAL_FREEZE
3:  MarkChunkFrozen(chunk);
4:  StabilizeChunk(chunk);
5:  if ( chunk→freezeState == EXTERNAL_FREEZE ) {
6:      // This chunk was in EXTERNAL_FREEZE before Line 1 executed. Find the master chunk.
7:      master = chunk→mergeBuddy;
8:      // Fix the buddy's mergeBuddy pointer.
9:      masterOldBuddy = combine(NULL, INTERNAL_FREEZE);
10:     masterNewBuddy = combine(chunk, INTERNAL_FREEZE);
11:     CAS(&(master→mergeBuddy), masterOldBuddy, masterNewBuddy);
12:     return FreezeRecovery(chunk→mergeBuddy, key, data, MERGE, chunk, trigger, result);
13: }
14: decision = FreezeDecision(chunk);            // The freeze state is INTERNAL_FREEZE
15: if ( decision == MERGE ) chunkMergePartner = FindMergeSlave(chunk);
16: return FreezeRecovery(chunk, key, data, decision, chunkMergePartner, trigger, result);
}
```

Finally, two CASs are used to link the entry to the list. Whenever a CAS fails, the insertion starts from scratch on same chunk.

4.2 Delete and Search

The other two list operations, *delete* and *search*, are closer to previous work, but care is required for these operations to work well with the freezing mechanism and the merge procedure. For lack of space their description is relegated to Appendix B. The *Find* method, invoked by *insertEntry*, is also specified there.

5 The Freeze Procedure

We now provide more details about the freeze procedure. The full description is presented in Appendix A. The freezing process occurs when the number of entries in a chunk exceeds its boundaries. At this point, splitting or merging happens by copying the relevant keys (and data) into a newly allocated chunk (or chunks). This process comprises three phases: *initiation*, *stabilization* and *recovery*.

The code for the *Freeze* method is presented in Algorithm 5. The input parameters are the chunk that needs to be frozen, the key, the data, and the event that triggered the freeze: INSERT, DELETE, ENSLAVE (if the freeze was called to prepare the chunk for merge with a neighboring chunk), or NONE (if the freeze is called while clearing an entry). The freeze will attempt to execute the insertion, deletion, or enslaving and will return a NULL pointer when successful. It will also set an input boolean flag to indicate the return code of the relevant operation. When unsuccessful, it will return a pointer to the new chunk on which the operation should be retried.

The *Freeze* method starts with an attempt to atomically change the freeze state from NO_FREEZE to INTERNAL_FREEZE. This freeze state of the chunk is normally NO_FREEZE and is switched to INTERNAL_FREEZE when a freeze process of this chunk begins. But it can also be EXTERNAL_FREEZE when a neighbor requested a freeze on this chunk to allow a merge between the two. Thus, an external freeze can start even when no size violation is detected in this chunk.

Whether or not the modification succeeds, we know that the freeze state can no longer be NO_FREEZE. It can be either INTERNAL_FREEZE or EXTERNAL_FREEZE. The *Freeze* method then calls *MarkChunkFrozen* to mark each entry in the chunk as frozen and *StabilizeChunk* to finish stabilizing the entries list in the chunk. At this point, the entries in the chunk cannot be modified anymore. *Freeze* then checks if the freeze is external or internal.

An external freeze can occur when a freeze is concurrently executed on the next chunk, and it has already enslaved the current chunk as its merge buddy. In this case, we cooperate with the joint freeze and joint recovery. When the state of the freeze is external, then the current chunk must have its *mergeBuddy* pointer already pointing to the chunk that initiated the merge, denoted the *master* chunk. To finish this freeze, we make sure that the master chunk has its merge buddy properly pointing back at the current chunk. The master chunk's *mergeBuddy* pointer must be either NULL or already pointing to the buddy we found. Thus it is enough to use one CAS command to verify that it is not NULL. Finally, we execute the recovery phase on the master chunk and return its output. We do not need to check the decision about the freeze of the buddy. It must be a merge.

If the freeze is internal, then we invoke *FreezeDecision* to see what should be done next (Line 14). If the decision is to merge, then we find the previous chunk and "enslave" it for a joint merge using the *FindMergeSlave* method (specified in Appendix A). Finally, the *FreezeRecovery* method is called to complete the freeze process. Next, we explain each of the stages. The full details including the pseudo-code appear in Appendix A.

Marking the chunk as frozen. The *MarkChunkFrozen* method simply goes over the entries one by one and marks each one as frozen. The setting of the freeze flags is atomic and it is retried repeatedly until successful. By the end of this process all entries (including the free ones) are marked as frozen.

Stabilizing the chunk. After all the entries in the chunk are marked as frozen, new entries cannot be allocated and existing entries cannot be marked as deleted. However, the frozen chunk may contain allocated entries that were not yet linked, and entries that were marked as deleted, but which have not yet been disconnected and reclaimed. The *StabilizeChunk* method disconnects all deleted entries and links all allocated ones. It uses the *Find* method to disconnect all entries that are marked as deleted. Such entries do not need to be reclaimed (when marked as frozen), but they should not be copied to the new chunk. Next, *StabilizeChunk* attempts to connect entries. It goes over all entries and searches for ones that are disconnected, but neither reclaimed nor deleted. Each such entry is linked to the list by invoking *InsertEntry*, which will only fail if the key already exists in a different entry in the chunk's list. In this case, this entry should indeed not be connected to the stabilized list.

Reaching a decision. After stabilizing the chunk, everything is frozen, the list is completely connected, and nothing changes in the chunk anymore. At this point, we need to decide whether or not splitting or merging is required. To that end, a count is performed and a decision is made by comparison to MIN and MAX. It may happen that the resulting count is higher than MIN and lower than MAX, and then no operation is required. Nevertheless, the frozen chunk is never resurrected. Instead, we copy the chunk to a new chunk in the (upcoming) recovery stage.

Making the recovery. Once a decision is reached, a recovery starts. The recovery procedure allocates a chunk (or two) and copies the relevant information into the new chunk (or chunks). If a merge is involved, the previous chunk in the list is first frozen (under an external freeze) and both chunks bring entries for the merge. Several threads may perform the freeze procedure concurrently, but all of them will make the same recovery decision about the freeze, as the frozen stabilized chunk looks the same to all threads. A thread that performs the recovery creates a local chunk (or chunks) into which it copies the relevant entries. At this point all threads create the same new chunk (or chunks). But now, each thread performs the operation with which it initiated the freeze on the new chunks. It can be an insert, delete, or enslave. Performing the operation is easy because the new chunks are local to this thread and no race can occur. (Enslaving a chunk is simply done by modifying its freeze state from NO_FREEZE to EXTERNAL_FREEZE and registration of the merge buddy.) But the success of making the local operation visible in the data structure is determined by whether the thread succeeds in creating a link to its new chunks in the frozen chunk, as explained next.

After creating the new chunks locally and executing the original operation on them, there is an attempt to atomically insert the address of its local chunk into a dedicated pointer in the frozen chunk (*new*). When two chunks are created, the second one is locally linked to the first one by the *nextChunk* field. The *nextChunk* field of the last chunk points to NULL' (different from NULL). If the insertion is successful, then this thread has also completed the the operation it was performing (insert, delete, or enslave). If the insertion is unsuccessful, then this means that a different thread has already completed the installation of new chunks and this thread's local new chunks will not be used (i.e., can be reclaimed). In this case, the thread must try its operation again from scratch.

According to the number of (live) entries on the frozen chunk there are three ways to recover from the freeze.

Case I: MIN$<$ *count* $<$ MAX**.** In this case, the required action is to allocate a new chunk and copy all of the entries from the frozen chunk to the new chunk. Next we perform the insert, delete, or enslave operation on the local new chunk and attempt to link it to the frozen one.

Case II: *count* $==$ MIN**.** In this case we need to merge the frozen chunk with its previous chunk. We assume that the previous chunk has already been frozen by an external freeze before the recovery is executed, and that the freeze states in both chunks are properly set so that no thread can interfere with the freeze process.

We start by checking the overall number of entries in these two chunks, to decide if the merged entries will fit into one or two chunks. We then allocate a second new chunk, if needed, and perform the (local) copy to the new chunk or chunks. When copying into two new chunks, we split the entries evenly, and return the smallest key in the second chunk as the *separating key.* As before, we perform the original operation that started the freeze and try to create a link from the old chunk to the new chunk or chunks.

Case III: *count* $==$ MAX**.** In this case we need to split the old chunk into two new chunks. The basic operations of this case resemble those of the previous cases. We allocate two new chunks, perform the split locally, perform the original operation, and attempt to link the new chunks to the old one.

6 Conclusion

We have presented a chunking and freezing mechanisms that build a cache-conscious lock-free linked list. Our list consists of chunks, each containing consecutive list entries. Thus, a traversal of the list stays mostly within a chunk's boundary (a virtual page or a cache line), and therefore, the traversal enjoys a reduced number of page faults (or cache misses) compared to a traversal of randomly allocated nodes, each containing a single entry. Maintaining a linked list in chunks is often used in practice (e.g., [1, 5]) but a lock-free implementation of a cache-conscious linked list has not been available heretofore. We believe that the building blocks of this list, i.e., the chunks and the freeze operation, can be used for building additional data structures, such as lock-free hash functions, and others.

References

[1] *Unrolled Linked Lists*, http://blogs.msdn.com/devdev/archive/2005/08/22/454887.aspx

[2] D. Dechev., P. Pirkelbauer., B. Stroustrup. *A Lock-Free Dynamically Resizable Array*, OPODIS, 2006

[3] M. Fomitchev, and E. Rupert. *Lock-free linked lists and skip lists*, In Proc. PODC, 2004.

[4] K. Fraser., *Practical lock-freedom*, Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, Feb. 2004.

[5] Frias, L., Petit, J., and Roura, S. 2009. *Lists revisited: Cache-conscious STL lists.* J. Exp. Algorithmics 14 (Dec. 2009), 3.5-3.27.

[6] T.L. Harris., *A pragmatic implementation of non-blocking linked-lists*, In Proc. PODC, 2001.

[7] M. Herlihy., *Wait-free synchronization.* TOPLAS, 1991.

[8] M.M. Michael., *High Performance Dynamic Lock-Free Hash Tables and List-Based Sets*, In Proc. SPAA, 2002.

[9] M. M. Michael., *Safe memory reclamation for dynamic lock-free objects using atomic reads and writes*, In Proc. PODC, 2002.

[10] M. M. Michael., *Hazard pointers: Safe memory reclamation for lock-free objects*, TPDS, June 2004.

[11] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming*, Morgan Kaufman 2008

[12] J. D. Valois., *Lock-free linked lists using compare-and-swap*, In Proc. PODC, 1995.

[13] R. K. Treiber., *Systems programming: Coping with parallelism.*, Research report RJ 5118, IBM Almaden Research Center, 1986.

Algorithm 6 Freezing all entries in a chunk

```
void MarkChunkFrozen(chunk* chunk) {
    foreach entry e {
        savedWord = e→next;
        while ( !isFrozen(savedWord) ) {            // Loop till the next pointer is frozen
            CAS(&(e→next), savedWord, markFrozen(savedWord));
            savedWord = e→next;                      // Reread from shared memory
        }
        savedWord = e→keyData;
        while ( !isFrozen(savedWord) ) {            // Loop till the keyData word is frozen
            CAS(&(e→keyData), savedWord, markFrozen(savedWord));
            savedWord = e→keyData;                   // Reread from shared memory
        }
    } // end of foreach
    return;
}
```

A The Freeze Procedure

In this appendix we provide the entire freeze procedure. To save the reader from going back and forth between the main body of the paper and this appendix, we include everything here, repeating the explanations of Section 5 and adding all the pseudo-code and details.

The freezing process occurs when the number of entries in a chunk exceeds its boundaries. At this point, splitting or merging happens by copying the relevant keys (and data) into a newly allocated chunk (or chunks). This process comprises three phases: *initiation*, *stabilization* and *recovery*.

A.1 The initiation of a freeze

A thread will initiate a freeze when the MIN or MAX are exceeded or when there is an external request to freeze the current chunk in order for it to serve in a merge procedure with a neighbor. Once a freeze process has started, other threads may join and help the freeze since they need the resulting chunks to proceed in their activities. We distinguish between an external freeze (imposed upon a chunk by a sparse neighbor) and a regular freeze, using the freeze state field in the chunk. The main goal is to avoid the possibility that one thread plans to use this chunk for merging with a small neighbor, while another thread is splitting it because it is full. We let one type of freeze terminate before executing a different one.

A freeze starts by modifying the freeze state of a chunk from NO_FREEZE to INTERNAL_FREEZE or EXTERNAL_FREEZE. The location of the freeze state field is in the three LSBs of the merge buddy pointer (see Figure 2). In the discussion below we assume we are dealing with an internal freeze. An external freeze is discussed separately in Section A.4.

After changing the chunk's state, the initiation invokes *MarkChunkFrozen*, which goes through the chunk's entries one by one and marks them as frozen by setting the freeze bit first on the *nextEntry* word and then on the *keyData* word. The setting of these flags is atomic and it is retried repeatedly until successful. By the end of this process all entries (including the free ones)

Algorithm 7 Freeze stabilization.

```
void StabilizeChunk(chunk* chunk) {
 1: maxKey = $\infty$;
 2: Find(chunk, maxKey);                                  // Implicitly remove deleted entries
 3: foreach entry e {
 4:    key = e$\rightarrow$key; eNext = e$\rightarrow$next;
 5:    if ( (key != $\perp$) && (!isDeleted(eNext)) )          // This entry is allocated and not deleted
 6:        if ( !Find(chunk, key) ) InsertEntry(chunk, e, key);     // This key is not yet in the list
 7: } // end of foreach
 8: return;
}
```

are marked as frozen. The freeze bit of the *head* entry is set last and at this point in time we consider the initiation phase to be completed. The pseudo code of *MarkChunkFrozen* is presented in Algorithm 6.

A.2 The stabilization phase

After all the entries in the chunk are marked as frozen, new entries cannot be allocated and existing entries cannot be marked as deleted. However, the frozen chunk may contain allocated entries that were not yet linked, and entries that were marked as deleted, but which have not yet been disconnected and reclaimed. The stabilization operation disconnects all deleted entries and links all allocated ones. The pseudo-code of the *StabilizeChunk* method appears in Algorithm 7. It starts by running *Find* on the maximal possible key value. This is done because the *Find* method (described in Appendix B.1) always disconnects all entries that are marked as deleted (even when frozen). Such entries do not need to be reclaimed (when marked as frozen), but they should not be copied to the new chunk. Next, *StabilizeChunk* attempts to connect entries. It goes over all entries and searches for ones that are disconnected, but neither reclaimed nor deleted. Each such entry is linked to the list by invoking *InsertEntry*, which will only fail if the key already exists in a different entry in the chunk's list. In this case, this entry should indeed not be connected to the stabilized list.

A.3 The decision and the recovery

After stabilizing the chunk, everything is frozen, the list is completely connected, and nothing changes in the chunk anymore. At this point, we need to decide whether or not splitting or merging is required. Recall that the decision to freeze is initiated in the presence of many concurrent updates. It is possible that one thread could not find an entry to allocate and initiated a freeze for the purpose of splitting the chunk, but many other threads deleted entries concurrently and when the chunk actually stabilized, there was no need to split. There may even be a need for a merge. Thus, we make the decision on which operation to execute only after the chunk has stabilized and cannot change anymore.

At this point, we count the number of entries in the frozen chunk, and decide if a split or a merge is required according to the count. If the resulting count equals MIN we run a merge, and if it equals MAX, we run a split (in the recovery phase). The resulting count can never exceed

Algorithm 8 Determining the freeze action.

```
recovType FreezeDecision (chunk* chunk) {
 1: entry* e = chunk→head→next;    int cnt = 0;
 2: while ( clearFrozen(e) != NULL ) { cnt++; e = e→next; }     // Going over the chunk's list
 3: if ( cnt == MIN) return MERGE; if ( cnt == MAX) return SPLIT; return COPY;
}
```

the bounds, because there is no space to allocate more than MAX entries in the chunk, and since the chunk counter that is maintained during the run holds a lower bound on the actual number of entries and can never reach a value below MIN. If the resulting count is higher than MIN and lower than MAX, then no operation is required. Nevertheless, the frozen chunk is never resurrected. Otherwise, correctness cannot be guaranteed when a long-sleeping thread wakes to find a chunk that was resurrected. Instead, we copy the chunk to a new chunk in the (upcoming) recovery stage.

The *FreezeDecision* method is presented in Algorithm 8. It computes the number of entries and returns the recovery code: SPLIT, MERGE, or COPY.

The recovery procedure allocates a chunk (or two) and copies the relevant information into the new chunk (or chunks). If a merge is involved, the previous chunk in the list is first frozen (externally) and both chunks bring entries for the merge. Several threads may perform the freeze procedure concurrently, but all of them will make the same recovery decision about the freeze, as the frozen stabilized chunk looks the same to all threads. A thread that performs the recovery creates a local chunk into which it copies the relevant entries. At this point all threads create the same new chunk (or chunks). But now, each thread performs the operation with which it initiated the freeze on the new chunks. It can be an insert, delete, or enslave. Performing the operation is easy because the new chunks are local to this thread and no race can occur. (Enslaving a chunk is simply done by modifying its freeze state from NO_FREEZE to EXTERNAL_FREEZE and registration of the merge buddy.) But the success of making the local operation visible in the data structure is determined by whether the thread succeeds in creating a link to its new chunks in the frozen chunk, as explained next.

After creating the new chunks locally and executing the original operation on them, there is an attempt to atomically insert the address of its local chunk into a dedicated pointer in the frozen chunk (*new*). When two chunks are created, the second one is locally linked to the first one by the *nextChunk* field. The *nextChunk* field of the last chunk points to NULL' (different from NULL). If the insertion is successful, then this thread has also completed the the operation it was performing (insert, delete, or enslave). If the insertion is unsuccessful, then this means that a different thread has already completed the installation of new chunks and this thread's local new chunks will not be used (i.e., can be reclaimed). In this case, the thread must try its operation again from scratch.

The code for the recovery is presented in Algorithm 9. If a merge occurs, the merging chunk is supplied as a parameter. According to the number of (live) entries on the frozen chunk there are three ways to recover from the freeze.

Case I: MIN$<$ *count* $<$ MAX**.** In this case, the required action is to allocate a new chunk and copy all of the entries that reside on the frozen chunk's list to the new chunk (which is only locally visible and requires no synchronization). We do not specify the copying routine (in this case, as well as in the other cases) since the copy is from a frozen chunk that does not change, to a local chunk. This means that no concurrency is involved and the implementation is simple. The new chunk becomes the replacement of the old chunk when the pointer *new* in the old chunk points to it. An upper-level routine that handles the chunked list *ListUpdate* is then invoked to replace the frozen chunk with the chunk that is referenced by *new*. The new chunk that holds the input key (after the freeze is completed) is then returned.

Case II: *count* == MIN**.** In this case we need to merge the old chunk with its previous chunk supplied through *mergeChunk*. We assume that the supplied chunk has already been frozen by an external freeze before the recovery is executed. Finally, we assume that the freeze states are properly set to internal on the old chunk and external on the previous chunk (so that no thread can interfere with the freeze process), and the *mergeBuddy* pointers on these two chunks point to each other.

Algorithm 9 The freeze recovery.

chunk* FreezeRecovery(chunk* oldChunk, key, input, recovType, chunk* mergeChunk, triggerType trigger, Bool* result) {

```
retChunk = NULL; newChunk2 = NULL; newChunk1 = Allocate();     // Allocate a new chunk
newChunk1→nextChunk = NULL';
switch ( recovType ) {
   case COPY:
      copyToOneChunk(oldChunk, newChunk1); break;
   case MERGE:
      if ( (getEntrNum(oldChunk)+getEntrNum(mergeChunk))≥MAX ) {
         // The two neighboring old chunks will be merged into two new chunks
         newChunk2 = Allocate();                          // Allocate a second new chunk
         newChunk1→nextChunk = newChunk2;                 // Connect two chunks together
         newChunk2→nextChunk = NULL';
         separatKey=mergeToTwoChunks(oldChunk,mergeChunk,newChunk1,newChunk2);
      } else mergeToOneChunk(oldChunk,mergeChunk,newChunk1); // Merge to single chunk
      break;
   case SPLIT:
      newChunk2 = Allocate();                             // Allocate a second new chunk
      newChunk1→nextChunk = newChunk2;                    // Connect two chunks together
      newChunk2→nextChunk = NULL';
      separatKey = splitIntoTwoChunks(oldChunk, newChunk1, newChunk2); break;
} // end of switch
// Perform the operation with which the freeze was initiated
HelpInFreezeRecovery(newChunk1, newChunk2, key, separatKey, input, trigger);
// Try to create a link to the first new chunk in the old chunk.
if ( !CAS(&(oldChunk→new), NULL, newChunk1) ) {
   RetireChunk(newChunk1); if (newChunk2) RetireChunk(newChunk2);
   // Determine in which of the new chunks the key is located.
   if ( key<separatKey ) retChunk=oldChunk→new; else retChunk=FindChunk(key);
} else { retChunk = NULL; }
ListUpdate(recovType, key, oldChunk);                    // User defined function
return retChunk;
}
```

Algorithm 10 Perform the operation with which the freeze was initiated.

chunk* HelpInFreezeRecovery(chunk* newChunk1, chunk* newChunk2, key, separatKey, input, triggerType trigger) {

```
switch ( trigger ) {
   case DELETE:                    // If key will be found, decrement counter has to succeed
      *result = DeleteInChunk(newChunk1, key);
      if ( newChunk2 != NULL ) *result = *result || DeleteInChunk(newChunk2, key);
      break;
   case INSERT:                    // input should be interpreted as data to insert with the key
      if ( (newChunk2!=NULL) && (key<separatKey) )
         result = InsertToChunk(newChunk2, key, input);
      else *result = InsertToChunk(newChunk1, key, input);
      break;
   case ENSLAVE:                   // input should be interpreted as pointer to master trying to enslave
      if ( newChunk2 != NULL )
         newChunk2→mergeBuddy = combine(input, EXTERNAL_FREEZE);
      else newChunk1→mergeBuddy = combine(input, EXTERNAL_FREEZE);
} // end of switch
}
```

We start by checking the overall number of entries in these two chunks, to decide if the merged entries will fit into one of two chunks. We then allocate a second new chunk, if needed, and perform the (local) copy to the new chunk or chunks. When copying into two new chunks, we split the entries evenly, and return the smallest key in the second chunk as the *separating key.* As before, we try to create a link from the old chunk to the new chunk or chunks. Next, the new chunk that holds the input key is determined according to the *separating key*, and finally, the *ListUpdate* method is called to replace the frozen chunk in the list with the two new chunks. This completes the recovery for the merge case.

Case III: *count* == MAX**.** In this case we need to split the old chunk into two new chunks. The basic operations of this case resemble those of the previous cases. We allocate a new chunk, perform the split locally, attempt to link the new chunks to the old one, update the list, and return the chunk holding the key.

At Algorithm 10 we present *HelpInFreezeRecovery* method that is invoke in order to try and accomplish the activity that initiated the freeze as part of the freeze. This is needed in order to avoid loop of freezes of chunk where no threads are making progress.

A.4 Managing the freeze activities

The code for the *Freeze* method is presented in Algorithm 5 and here in Algorithm 11. The input parameters are the chunk that needs to be frozen, the key, the data, and the event that triggered the freeze: INSERT, DELETE, ENSLAVE (if the freeze was called to prepare the chunk for merge with a neighboring chunk), or NONE (if the freeze is called while clearing an entry). The freeze will attempt to execute the insertion, deletion, or enslaving and will return a NULL pointer when successful. It will also set a input boolean flag to indicate the return code of the relevant operation. When unsuccessful, it will return a pointer to the new chunk on which the operation

Algorithm 11 The main freeze method.

```
chunk* Freeze(chunk* chunk, key, data, triggerType trigger, Bool* result)
{
 1: CAS(&(chunk→freezeState), NO_FREEZE, INTERNAL_FREEZE);
 2: // At this point, the freeze state is either INTERNAL_FREEZE or EXTERNAL_FREEZE
 3: MarkChunkFrozen(chunk);
 4: StabilizeChunk(chunk);
 5: if ( chunk→freezeState == EXTERNAL_FREEZE ) {
 6:     // This chunk was in EXTERNAL_FREEZE before Line 1 executed. Find the master chunk.
 7:     master = chunk→mergeBuddy;
 8:     // Fix the buddy's mergeBuddy pointer.
 9:     masterExpected = combine(NULL, INTERNAL_FREEZE);
10:     masterNew = combine(chunk, INTERNAL_FREEZE);
11:     CAS(&(master→mergeBuddy), masterExpected, masterNew);
12:     return FreezeRecovery(chunk→mergeBuddy, key, data, MERGE, chunk, trigger, result);
13: }
14: decision = FreezeDecision(chunk);                    // The freeze state is INTERNAL_FREEZE
15: if ( decision == MERGE ) chunkMergePartner = FindMergeSlave(chunk);
16: return FreezeRecovery(chunk, key, data, decision, chunkMergePartner, trigger, result);
}
```

should be retried.

The *Freeze* method starts with an attempt to atomically change the freeze state from NO_FREEZE to INTERNAL_FREEZE. Whether or not the modification succeeds, we know that the freeze state can no longer be NO_FREEZE. It can be either INTERNAL_FREEZE or EXTERNAL_FREEZE. The *Freeze* method then calls *MarkChunkFrozen* to mark each entry in the chunk as frozen and *StabilizeChunk* to finish stabilizing the entries list in the chunk. At this point, the entries in the chunk cannot be modified anymore. *Freeze* then checks if the freeze is external or internal.

An external freeze can occur when a freeze is concurrently executed on the next chunk, and it has already enslaved the current chunk as its merge buddy. In this case, we cooperate with the joint freeze and joint recovery. When the state of the freeze is external, then the current chunk must have its *mergeBuddy* pointer already pointing to the chunk that initiated the merge, denoted the *master* chunk. To finish this freeze, we make sure that the master chunk has its merge buddy properly pointing back at the current chunk. The master chunk's *mergeBuddy* pointer must be either NULL or already pointing to the buddy we found. Thus it is enough to use one CAS command to verify that it is not NULL. Finally, we execute the recovery phase on the master chunk and return its output. We do not need to check the decision about the freeze of the buddy. It must be a merge.

If the freeze is internal, then we invoke *FreezeDecision* to see what should be done next (Line 14). If the decision is to merge, then we find the previous chunk and "enslave" it for a joint merge using the *FindMergeSlave* method (explained below). Finally, the *FreezeRecovery* method is called to complete the freeze process.

Let us now explain the *FindMergeSlave* method, which is presented in Algorithm 12. This method finds the previous chunk, sets its freeze state and *mergeBuddy* pointer, initiates its freeze, stabilizes it, and sets the current *mergeBuddy* to point at the obtained chunk. This method starts

Algorithm 12 Setting a chunk partner for a merge.

```
chunk* FindMergeSlave(chunk* master) {
1:  while ( TRUE ) {                    // Find a slave and set its freeze state & mergeBuddy pointer
2:     slave = listFindPrevious(master);           // upper-level funciton returning previous chunk.
3:     // Set slave's mergeBuddy pointer and freeze state (both reside on the same word).
4:     expected = combine(NULL, NO_FREEZE);
5:     new = combine(master, EXTERNAL_FREEZE);
6:     if ( !CAS(&(slave→mergeBuddy), expected, new) ) {
7:         if ( slave→mergeBuddy == new ) break;                  // Someone else has set it right.
8:         Freeze(chunk, 0, master, ENSLAVE, &result);       // This slave is under a different freeze
    activity, help.
9:     } else break;
10: }                                                                            // end of while
11: MarkChunkFrozen(slave);
12: StabilizeChunk(slave);
13: // slave is externally frozen - make sure the master's mergeBuddy points to the slave.
14: expected = combine(NULL, INTERNAL_FREEZE);                // Combine two values in one word
15: new = combine(slave, INTERNAL_FREEZE);
16: CAS(&(master→mergeBuddy), expected, new);
17: return slave;
}
```

by invoking the (upper-level) *listFindPrevious* method in order to find the chunk that precedes the current chunk. Sometimes, because of concurrent activity, *listFindPrevious* does not find its input chunk in the list (since it was already frozen and disconnected from the list of chunks). In this case, it cannot identify the previous chunk, and instead, it just returns the *mergBuddy* pointer, which properly points to its slave for the merge (that was already completed in a concurrent manner).

We denote the previous chunk *a slave* as it joins the merge initiated by the input chunk, which is the *master*. After identifying the slave, we attempt to atomically modify its freeze state and merge buddy to indicate an external freeze joint with the master chunk. Once the slave is marked with an external freeze, the two chunks are destined for a joint freeze and no chunk can come between them. (New chunks are only added as a result of a split.) If the change in the slave state fails, a search for a new slave is attempted, after making sure that the current one is out of the way, by participating in completing its current freeze. Next, we ensure that the master's chunkBuddy pointer points to the slave and then a pointer to the slave is returned to the caller.

B Additional Chunk-level Methods

Here we present operations that did not fit into the limited-space main body of the paper.

B.1 The search operation

The search operation, implemented in the *searchInChunk* method of Algorithm 13, uses the *Find* method, described hereafter. The *searchInChunk* method starts by call for *Find*, which protects its

Algorithm 13 Searching for data associated with the key

```
Bool SearchInChunk (chunk* chunk, key, *data) {
1: if ( Find(chunk, key) ) { data = cur→data; result = TRUE; } else result = FALSE;
2: *hp0 = *hp1 = NULL; return result;
}
```

output with hazard pointers. The *SearchInChunk* method finishes by clearing the hazard pointers and returning.

Finding the location in the chunk's list: the *Find* method. We now present the *Find* method, invoked by several other methods. The pseudo-code for *Find* appears in Algorithm 14. This method finds the location of a given key in the list. It returns FALSE if the key does not exist in the list, or TRUEotherwise. It also sets in a global (indirect) pointer ***prev* to the entry that contains the highest key value between all keys smaller than the input key, and in a global pointer **cur*, the entry with the minimal key value that is larger or equal to the input key. Finally, if the key is found, the entry that follows *cur* is returned in a global pointer **next*. *Find* is very similar to the Find method presented in [10] up to changes needed for dealing with the freeze bit.

The *Find* method protects the entries that it uses and returns using hazard pointers so they are not being concurrently reclaimed. This holds upon return from *Find* so the calling method may assume that the referenced entries could not be reclaimed and re-allocated, until the calling method clears the thread's hazard pointers.

If the traversal of *Find* finds an entry that is marked for deletion (i.e., the delete bit is set on its *next* pointer), then it disconnects the entry from the list and attempts to recycle it. Recycling is executed via *RetireEntry*, which is explained later in Section B.2. Disconnecting and recycling a deleted entry is a service of *Find* to the structure of the list that will be assumed in the rest of this paper. The key of a deleted entry is not checked, and cannot influence the search for the input key.

Any failing CAS causes a restart of the search. Also, in general, whenever we replace a pointer by another, e.g., in Line 8, we first make sure that if the old pointer was marked as frozen, then the replacement pointer is marked as frozen as well. This way the freeze bits of an entry are preserved everywhere.

B.2 The delete operation

The deletion algorithm (inside a chunk) is similar to the well-known one for lock-free linked lists [6, 10]. The deletion operation is partitioned into a logical deletion, which marks the entry as deleted by setting the delete bit (LSB) in the entry's *next* pointer. Next, the physical deletion disconnects the entry from the list and reclaims its space. The difference between our deletion method and the standard one is the need to check if the chunk's counter has reached the lower threshold MIN and call *Freeze* when it does. Additionally, we do not let the delete bit be set on a frozen entry. A delete can only occur before an entry gets frozen. Notice that *Freeze* can also help this deletion and we check if help happened any time *Freeze* is invoked. Finally, we need to maintain the counter of entries allocated in the chunk. In order to make sure that the counter holds a lower bound on the number of entries in the presence of concurrent updates, we decrement the counter before we delete the entry. If the delete fails, we increment the counter to account for the failure. A failure to decrement the counter can only happen when the lower bound has been

Algorithm 14 Find the location of an entry in the chunk's list

```
Bool Find (chunk* chunk, key) {
try_again: prev = &(chunk→head);                                  // Restart point
cur = *prev;
while ( clearFrozen(cur) != NULL ) {          // Ignore freeze bit when comparing to NULL
    *hp0 = cur;                                        // Progress to an unprotected entry
    if ( *prev != cur ) goto try_again;               // Validate progress after protecting
    next = cur→next;
    if ( isDeleted(next) ) {                              // Current entry is marked deleted
        if ( isFrozen(cur) ) markFrozen(next);        // next replaces cur; save freeze bit
        // Disconnect current: prev gets the value of next with the delete bit cleared
        if ( !CAS(prev, cur, clearDeleted(next)) ) goto try_again;
        RetireEntry(cur);                                   // CAS succeeded - try to reclaim
        cur = clearDeleted(next);
    } else {
        ckey = cur→key;
        if ( *prev != cur ) goto try_again; // Check new insertion between them or new deletion
        if (ckey $\geq$ key) return (ckey == key);
        prev = &(cur→next);
        tmp = hp0; hp0 = hp1; hp1 = tmp;      // All private. hp0, hp1 are ptrs to hazard ptrs
        cur = next;
    }
}
return FALSE;
}
```

reached. In this case, we initiate a freeze, which returns with a new chunk (containing the range of values that includes our input key). The decrement attempt is then repeated and this loop repeats until the decrement succeeds on the current chunk.

The deletion algorithm (inside a chunk) is presented in Algorithm 15. It starts by decrementing the counter, *Find* is invoked to find the entry holding the key. If the key does not exist in the list, then the counter is incremented, hazard pointers zeroed and FALSE is returned to the caller. Otherwise, we attempt a CAS to mark the entry as deleted. The CAS assumes that the freeze bit and the delete bit are not set at the deletion time (for proper counter measurement, we should know exactly who sets the delete bit). If the CAS fails due to a freeze bit, then a freeze action must be executed, then either freeze succeeded to promote this deletion or the delete should restart on the newly obtained chunk. Otherwise, the CAS failed due to some other thread deleting the entry, or a pointer modification. In this case, we should search for the entry again before deleting it. The *Find* method will not return this entry again if it has already been deleted. Furthermore, it will disconnect it from the list and reclaim it before returning.

After marking the entry as deleted, we attempt to disconnect it from the list. If the freeze bit is set, we keep it set. If the disconnect succeeds, we reclaim the entry via *RetireEntry*. Otherwise, we call *Find*, which repeatedly attempts to disconnect an entry that is marked deleted, until the disconnection is achieved. Finally, we clear the hazard pointers that are set by the *Find* method,

Algorithm 15 The pseudo-code of deletion of an entry in a chunk

```
Bool DeleteInChunk (chunk* chunk, key) {
try_again:
while ( !DecCount(chunk) ) {                          //If too few entries in chunk; call freeze
   chunk = Freeze(chunk, key, 0, DELETE, &result);
   if ( chunk == NULL ) return result; // If Freeze succeeded to proceed with deletion, return
} // end of decrement counter while
while ( TRUE ) {
   if ( !Find(chunk, key) ) {
      IncCount(chunk); *hp0 = *hp1 = NULL; return FALSE;       // No such entry was found
   }
   // Mark entry as deleted, assume entry is not deleted or frozen
   clearedNext = clearFrozen(clearDeleted(next));
   if ( !CAS(&(cur→next), clearedNext, markDeleted(clearedNext)) ) {
      if ( isFrozen(cur→next) ) {                                  // CAS failed due to freeze
         IncCount(chunk); chunk = Freeze(chunk, key, 0, DELETE, &result);
         if ( chunk == NULL ) return result;  // If Freeze succeeded to proceed with deletion, return
         goto try_again;
      } else continue;
   }
   // Remove entry
   if ( isFrozen(cur) ) markFrozen(next);                    // next replaces cur; retain freeze bit
   if ( CAS(prev, cur, next) ) RetireEntry(addr); else Find(chunk, key);
   *hp0 = *hp1 = NULL; return TRUE;
}
}
```

to allow future reclamation of the involved entries.

Entry reclamation Special care is required for reclaiming an entry in the presence of hazard pointers. First, it must be clear that the reclamation is not being executed on an entry that has a hazard pointer, and second, if an entry cannot be reclaimed right now, it will be properly scheduled for future reclamation (in a non-blocking manner). We follow the scheme presented by Michael in [10]. In this scheme, an entry can be reclaimed only by the very same thread that disconnects it from the list. There can only be one such thread, as the disconnection is executed with a CAS.

Each thread has its own list of to-be-retired entries. After successfully disconnecting an entry, the thread invokes *RetireEntry* (depicted in Algorithm 16), which pushes the given entry into the list of entries waiting to be reclaimed, and then attempts to reclaim all entries in the list via the *HandleReclamationBuffer* method. The *HandleReclamationBuffer* method compares the entries in the to-be-retired list with the ones in the hazard pointers array (HPA) and reclaims the entries that do not appear in the HPA. Our adaptation to this scheme does not reclaim entries marked as frozen even when no hazard pointer points to them. The *HandleReclamationBuffer* method is invoked on every *RetireEntry* call, in order to make sure that an entry is reclaimed as soon as possible, when

Algorithm 16 The reclamation code employs Michael's reclamation scheme

```
void RetireEntry (entry* entry) {
 1: addToRetList(entry);              // Add the entry to the (local) list of to-be-retired entries
 2: HandleReclamationBuffer();                 // Scan the list and reclaim the entries if possible
}
void HandleReclamationBuffer() {
 1: plist = initializeList();                 // Local list for recording current hazard pointers
 2: hprec = getHPhead();                        // Obtain head of hazard pointers array (HPA)
 3: // Stage 1: Save current hazard pointers in plist (locally)
 4: while ( hprec != NULL ) {
 5:    for ( i=0; i<2; ++i ) {                                     // 2 hazard pointers per thread
 6:        hptr = hprec→HP[i];
 7:        if ( hptr != NULL ) insertList(plist, hptr);
 8:    }
 9:    hprec = getNextHPrecord(hprec);
10: }
11: // Stage 2: Reclaim to-be-retired entries that are not protected by a hazard pointer
12: tmplist = popAllRetList();             // Copy all local to-be-retired entires and clear RetList
13: entry = popList(tmplist);
14: while ( entry != NULL ) {
15:    if ( lookUp(plist, entry) ) pushRetList(entry);     // Entry protected, push back to RetList
16:    else { if ( !isFrozen(entry) ) ClearEntry(entry); }       // Reclaim unprotected (non-frozen)
    entry
17:    entry = popList(tmplist);
18: }
19: freeList(plist);
}
```

no more hazard pointers point to it. Michael's reclamation scheme, slightly modified to support our notations, is depicted in Algorithm 16. For further discussion on the reclamation scheme we refer the reader to the *RetireNode* method and the *Scan* method in [10].

The actual clearing of an entry in our list means zeroing the entry and assigning $\perp$ as key's value. This is executed in the *ClearEntry* method depicted in Algorithm 17. This method is invoked either in case of a trial to insert a key that already existed in the chunk's list (*InsertEntry*, Line 14) or by *HandleReclamationBuffer* when a deleted and disconnected entry is found to be safe for reuse (in Line 16 there). We do not reclaim an entry when it is found frozen, because this reclamation is not needed anymore, and it complicates the code to reclaim it.

The entry clearance is executed by two CAS operations. When the *keyData* word is cleared, the entry might immediately be re-allocated. Therefore, we first zero the *nextEntry* word, and only then put $\perp$ on the *keyData* word. An entry that is marked frozen is not reclaimed and this is ensured by the atomic CAS. We claim that a CAS can only fail when the entry's freeze bit is marked. If the chunk is not being frozen, a cleared entry is handled only by the current thread. The reason is that the entry is already disconnected from the list and no other thread has a hazard pointer to it, neither can it find the entry at this point. Furthermore, only one thread holds it in

Algorithm 17 The pseudo-code for clearing an entry and reclaiming it's space/

```
Bool ClearEntry (chunk* chunk, entry* entry) {
 1: savedKeyData = clearFrozen(entry→keyData);
 2: savedNext = clearFrozen(entry→next);
 3: newKeyData = combine(⊥, 0);
 4: if ( CAS(&(entry→next), savedNext, 0) )
 5:    if (CAS(&(entry→keyData), savedKeyData, newKeyData) )
 6:       return TRUE;                                    // Both CASes were successful
 7: Freeze(chunk, 0, 0, NONE, &result);     // A CAS failure indicates a freeze. Help freeze before
    proceeding.
 8: if ( Find(chunk, entry→key) )     // Check whether the entry to be reclaimed was linked back
    by the freeze
 9:    if ( entry == cur ) return FALSE;                  // cur is global initiated by Find
10: return TRUE;
}
```

his to-be-retired list. Therefore, the clearing can only fail when a freeze process is executing.

When *ClearEntry* is called from *HandleReclamationBuffer* the freeze cannot resurrect it. The entry is deleted, and after executing the freeze procedure to make sure that it is completed, we know that the entry cannot exist in the newly created chunk anymore. However, when *ClearEntry* is called by *InsertEntry*, the entry is not deleted and a freeze process may resurrect it (as discussed in the description of *InsertEntry*). In this case, *ClearEntry* discovers the resurrection and returns FALSE.

B.3 Counter Functionalities

Here we present the lock-free counter functionalities we use in the *InsertInChunk* and *DeleteInChunk* methods.

It may happen that a thread fails or stops just before or after updating the counter, thus an accurate count for the number of entries in the chunk cannot be expected (in a lock-free execution). Our counter only ensures that the counter value is always *less or equal* to the real number of entries in the chunk's list, which is what we actually need for keeping the number of entries in the chunk between MIN and MAX. Recall that MAX is the number of entries in a chunk, and so even if the counter did not exist, no more than MAX entries could be allocated on a chunk. In order to make sure that the number of entries does not go below MIN, we maintain the counter as a lower bound on the actual number of entries. If the counter drops below MIN, we try to merge the chunk with a neighboring chunk. Since the counter is a lower bound on the actual entry number, we may find that no merging is really needed after the freeze.

To ensure that the counter is a lower bound on the number of entries, we apply a couple of rules. First, the counter is only incremented *after* an entry is successfully allocated. This means that the counter does not supersede the number of entries in the chunk (MAX). Second, we decrement the counter *before* we delete an entry. So that if the executing thread halts between the counter decrement and the deletion, we know that the counter is smaller than the actual number of entries. It is never larger than it.

The code for handling the counter appears in Algorithm 18. The increment is straightforward

Algorithm 18 The increment and the decrement of the chunk's counter

```
void IncCount (chunk* chunk) {
1: while ( TRUE ) {
2:     counter = chunk→counter;
3:     if ( CAS(&(chunk→counter),counter,counter+1) ) return;
4: }
}
Bool DecCount (chunk* chunk) {
1: while ( TRUE ) {
2:     counter = chunk→counter;
3:     if ( counter == MIN ) return FALSE;          // comparison with minimal, MIN-1 illegal
4:     if ( CAS(&(chunk→counter),counter,counter-1) ) return TRUE;
5: }
}
```

and it always succeeds. The decrement method returns a failure if an attempt is made to reduce the counter below the MIN value.

C The Upper-Level List Operations

Let us now specify the upper-level list handling. When operations, such as split or merge, are executed on a chunk, they may sometimes cause a split or a merge of chunks. In this case, the list of chunks needs to be updated. Note that the list of chunks need not handle inserts or deletes. It only handles splits and merges that follow inserts and deletes of entries inside the chunks.

We start by presenting the *ListUpdate*() method. This method is called after a chunk has finished the freeze and recovery phases, at Line 29 of *FreezeRecovery*() method. The *ListUpdate*() gets as input the recovery type (SPLIT, MERGE, or COPY), an (arbitrary) key located or should be located on the frozen chunk, and a pointer to the frozen chunk. At this point, the list on the frozen chunk is stabilized and cannot be changed anymore. The *new* field in the frozen chunk points to a new chunk that contains some of the entries copied from the frozen chunk, and, when a second new chunk is required (for SPLIT or a MERGE that ended up with two new chunks), then the *nextChunk* field of first new chunk points to the new chunk with the higher key values, copied from the frozen chunk. We also make the new chunks sequence to be finished with a special NULLpointer - NULL', in order to distinguish between NULLthat comes at the end of the upper-level list. This is needed in order to synchronize the concurrent insertions of the chunks into upper-level list.

In addition, the least-significant bit of the *nextChunk* pointer holds a *swapped* bit, which is very similar to the *delete* bit that marks the logical delete of an entry. The *swapped* bit, when set, signifies that the chunk is about to be swapped with new chunks, and its *nextChunk* pointer cannot be modified anymore.

The code for *ListUpdate*() is presented in Algorithm 19. First, we memorize the value of *nextChunk* field of last new chunk, to be used later. After, we search for the frozen chunk by invoking the *FindChunk*() method on the input key (in the range of the old chunk). If *FindChunk*() returns a chunk different from the frozen chunk, then it means that some other thread has already removed the frozen chunk from the list of chunks, and we can just return. The search for a chunk

Algorithm 19 Update chunk list at the end of a freeze.

```
void ListUpdate(recovType, key, chunk* chunk) {
while ( TRUE ) {
    if ( chunk→new→nextChunk == NULL' ) // // There is only one new chunk (ref'd by new)
        expected = chunk→new→nextChunk; // Memorize new last chunk's next for the further update
    else expected = chunk→new→nextChunk→nextChunk;        // There are two new chunks
    if ( FindChunk(key) != chunk ) return; // Find the frozen chunk (and set NEXT and PREV).
    // Mark the next pointer of the frozen chunk as swapped.
    if ( !CAS(&(chunk→nextChunk), NEXT, markSwapped(NEXT)) ) continue;
    if ( !HelpSwap(expected) ) continue;
    return;
} // end of while
}
Bool HelpSwap(chunk* expected) {
if ( CUR→new→nextChunk == NULL' ) {        // There is only one new chunk (ref'd by new)
    addr = &(CUR→new→nextChunk);                       // address to insert pointer to next.
} else {                      // There are two new chunks (last ref'd by nextChunk of new)
    addr = &(CUR→new→nextChunk→nextChunk);
}
if ( !CAS( addr, expected, NEXT) ) return FALSE;
if ( CUR→mergeBuddy == NULL) {               // For COPY and SPLIT, there is one old chunk
    if ( !CAS(PREV, CUR, CUR→new) ) return FALSE; else RetireChunk(chunk);
} else {                                         // For MERGE, there are two old chunks
    if (!CAS(PRE_PREV,CUR→mergeBuddy,CUR→new)) return FALSE; else RetireChunk(chunk);
} // end of if there is one old chunk
return TRUE;
}
```

always succeeds since each chunk has a range of keys and one of these ranges contains the input key. The *FindChunk*() method is presented later in this section (in Algorithm 20). This method also sets the global variables PRE_PREV, PREV, CUR and NEXT to point at the previous to the previous chunk (or NULL), previous chunk (or HEAD), currently found chunk that is also returned; and the next chunk (or NULL, if none exists) in the list. One important property of *FindChunk*() is that it takes care of any encountered chunk that is marked ”swapped” by replacing it with the new chunks that should replace it.

Next, we attempt to mark the frozen chunk as swapped, by setting the least-significant bit on the *nextChunk* pointer. On failure, we start from scratch. After the *nextChunk* pointer is marked, it can not be modified anymore. Now, we attempt to link the new chunks into the list instead of the frozen chunk (and possibly a merge buddy in case of a merge). It is done in a supporting *HelpSwap*() method, also presented at Algorithm 19. In *HelpSwap*() we start by making the new chunk point to the next chunk in the chunk list. If a *nextChunk* field of *new* chunk is not NULL, then we have two chunks to insert and we make its *nextChunk* pointer point to the next chunk.

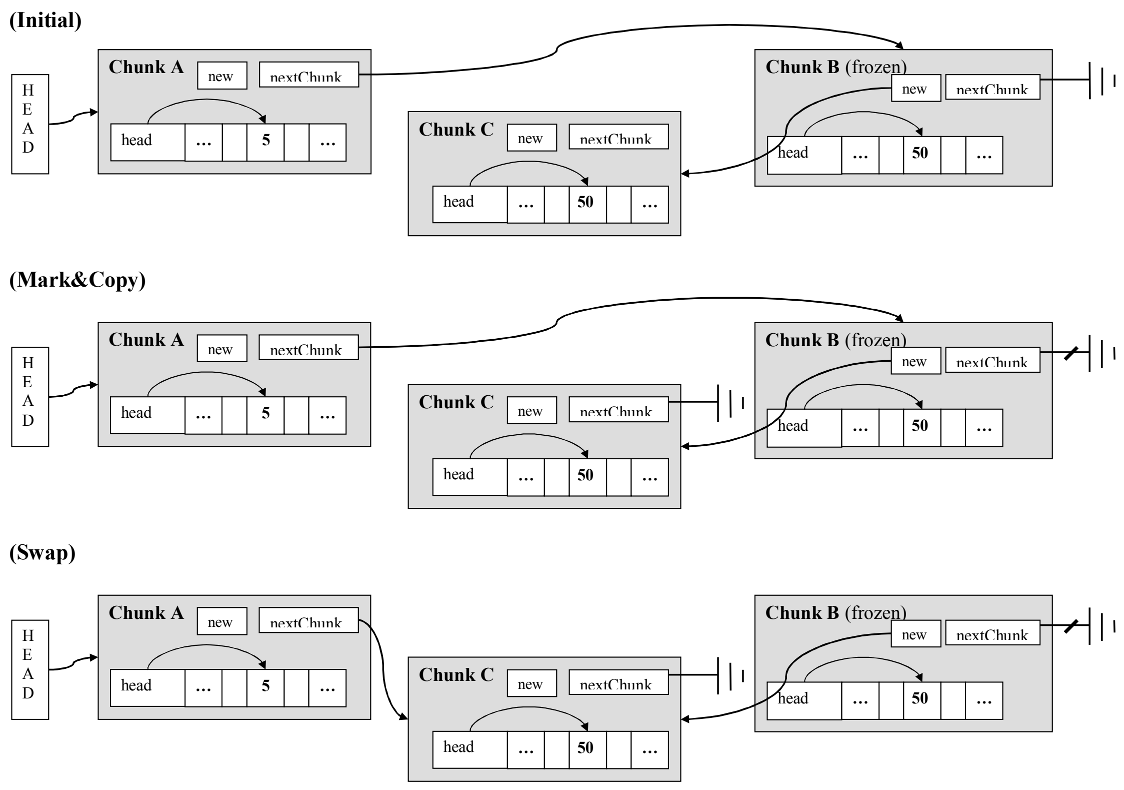

Figure 4: The COPY recovery in list of chunks.

Otherwise, we just have a single chunk to insert, which is pointed by *new*. In this case, we make its *nextChunk* pointer point at the next chunk in the list. The expected value is the one read before the *FindChunk*. If this setting of the *nextChunk* pointer fails, then we retry. Once the pointer to the next chunk is properly installed in the new chunks, we continue into linking it (or them) to the chunk list. In no merge is involved, we attempt to modify the previous chunk's pointer to point into the chunk referenced by *new*. If a merge is involved, then both the frozen chunk and its merge buddy (which is the chunk preceding the frozen chunk) need to be replace by the new chunks. *HelpSwap*() method make use of per thread global variables PRE_PREV, PREV, CUR and NEXT and assumes the CUR is marked as need to be swapped out.

We assume that the chunk list starts with a dummy record pointed by the global variable HEAD, and which also has a *nextChunk* field that can never be marked as swapped. We depict these steps for the COPY case of in Figure 4. The only difference for the SPLIT case is that we have two new chunks to insert, instead of one. The MERGE case is depicted in Figure 5.

The *FindChunk*() method. We now proceed with describing the *FindChunk*() method, which is similar to the *Find*() method. The code for this method is presented in Algorithm 20. We use four global variables per thread. The CUR variable points to the chunk that is currently being inspected. The PREV variable points to the chunk that precedes the one pointed by CUR. (It may point to HEAD.) The PRE_PREV variable points to the chunk that precedes the chunk pointed by PREV, if one exists. Finally, the NEXT pointer points to the chunk that comes after the currently inspected one. Another hazard pointers are also here to provide the correct reclamation of the chunks. We

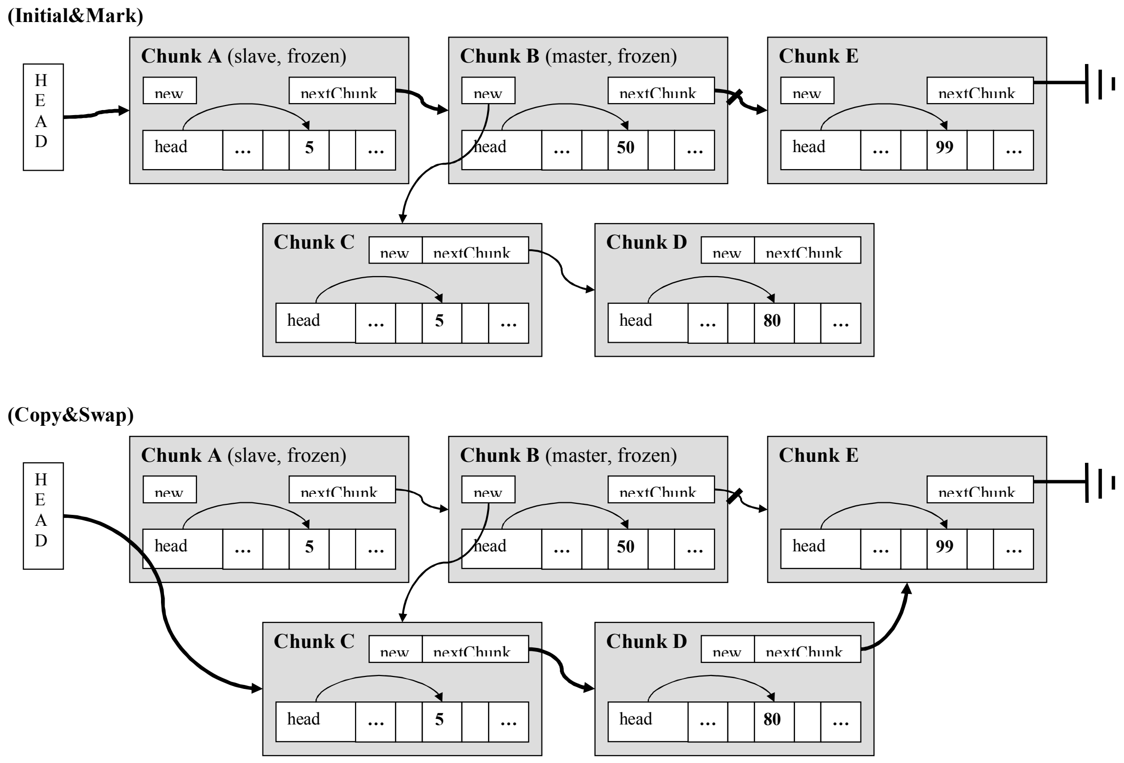

Figure 5: The MERGE recovery in list of chunks.

assume another array of hazard pointers separate from one used for entries reclamations. Chunk's hazard pointers are *hp2, hp3, hp4, hp5* we need them to protect NEXT, CUR, PREV, PREV_PREV respectively. After initiation of the global variables and some hazard pointers we continue going over the list till the NULL pointer is encountered at the end of the list. For each inspected chunk we check whether it is marked as swapped. If it is, we replace it with the new chunks, similarly to the code of *ListUpdate*. If we help in merge that involves swapping out current chunk and the previous chunk we restart from the beginning of the list. This is done for simplicity of the presented code, since we can recover from the same place as well. We then check the next chunk and repeat swapping until we reach an unmarked chunk. When we have a current chunk that is not marked as swapped, we check whether we found the chunk holding a range of keys that contains the given key. If we are at the last chunk of the list, then it is the right one, as it is associated with all larger keys. Otherwise, we check the smallest key in the next chunk. If the input key is smaller, then the current chunk is returned. Otherwise, we continue to check the next chunk. Implicitly, this means that a chunk is associated with the range of keys that start in its smallest key (of its first entry) and end in the smallest key of the following chunk. The last chunk in associated with a range whose highest value is ∞.

One last method to be specified that handles the list of chunks is the *listFindPrevious* method, predented at Algorithm 21 that finds the previous chunk to the input one, for use of the merge procedure. If it doesn't find its input chunk in the list, then this input chunk must have already been frozen and disconnected from the list of chunks. In this case, a previous chunk is not well defined, and this routine just returns the *mergBuddy* pointer of the input chunk.

Algorithm 20 Find the chunk whose associated range of keys contains the input key.

```
chunk* FindChunk (key) {
 1: try_again:
 2: PRE_PREV = NULL; PREV = &(HEAD→nextChunk); CUR = *PREV;
 3: hp3* = CUR; if (*PREV != CUR) goto try_again;
 4: while ( clearSwapped(CUR) != NULL ) {
 5:    NEXT = CUR→nextChunk;
 6:    hp2* = NEXT; if ((*PREV != CUR) || (*NEXT != CUR→nextChunk)) goto try_again;
 7:    if ( isSwapped(NEXT) ) {                    // Perform swap for a logically-marked swap.
 8:       if ( !HelpSwap() ) goto try_again;
 9:       if ( CUR→mergeBuddy != NULL ) goto try_again;       // PREV & CUR were swapped out
10:       CUR = CUR→new;
11:       hp3* = CUR; if (*PREV != CUR) goto try_again;
12:    } else { // current chunk does not need to be swapped out
13:       if ( NEXT == NULL ) return CUR;
14:       nextKey = NEXT→head→next→key;
15:       if ((*PREV != CUR) || (*NEXT != CUR→nextChunk)) goto try_again;
16:       if ( nextKey > key ) return CUR;                 // Next chunk's key range is too high.
17:       PRE_PREV = PREV;                                         // Continue to next chunk
18:       PREV = &(CUR→nextChunk);
19:       tmp = hp5; hp5 = hp4; hp4 = hp3; hp3 = hp2; hp2=tmp;   // promote hazard
    pointers
20:       CUR = NEXT;
21:    }
22: } // end of while
}
```

D Supporting functionalities

Several trivial low-level methods were not specified. For completeness, we provide a short specification for them in Figure 6. These functions are all local, and involve no concurrency (or contention) issues.

E Linearization Points

When designing a concurrent data structure, it is important to spell out the linearization points for the different operations. This is done in this section. In particular, we specify the linearization points of the insertion, deletion and search operations.

The linearization point of insertion. We partition the insertion linearization point determination into two cases. If the insertion operation is successful, i.e., no other entries with same key are found, then the linearization point is the successful execution of the CAS instruction at Line 11 of *InsertEntry*, where we actually modify the previous entry to point at the newly inserted entry. This modification creates the linearization point, whether it is executed by the thread executing the insert or by a different thread that is helping it (during a freeze). However, when this modification

Algorithm 21 Find previous chunk- High Level Method.

```
chunk* listFindPrevious (chunk* chunk) {
1: if ( FindChunk(chunk→head→next→key) != chunk )
2:     return chunk→mergeBuddy;
3: else return PREV;
}
```

is executed on a new local chunk that a thread is preparing to replace a frozen chunk, then the modification of the local chunk is not considered a linearization point. Instead, the linearization point of the insert becomes the point in which the new chunk is inserted into the list of chunks. For that this chunk need to be successfully linked to the the frozen chunk (successful Line 24 of the *FreezeRecovery*() method) and then linked into list (Lines 8 or 10 of the *HelpSwap*() method).

If the insertion is not successful, i.e., an entry with the same key is found in the list, then the linearization point is the linearization point of the successful *Find* that is invoked at Line 4 of *InsertEntry*. The linearization point of the *Find* method is specified below. And again, a special case is the one in which the freeze recovery is the one to find the key and decide on a failure. In this case, the finding of the key happens on a new local chunk that is copy of the frozen chunk and its linearization point is the point in which the new chunk is inserted into the list of chunks. The lines are explained above.

The linearization point of deletion. Again, we start by considering the successful case, in which the entry is found in the list, then the linearization point is the successful mark of the entry with the deletion bit. This happens at the successful execution of the CAS instruction at Line 12 of *DeleteInChunk*. Note that sometimes we need to wait until a chunk is frozen and only then can we attempt a deletion on a new chunk; however, the actual delete only happens when we manage to set the delete bit on an unfrozen entry containing the key. When this modification of the delete bit is executed on a new local chunk that a thread is preparing to replace a frozen chunk, then the modification of the local chunk is not considered a linearization point. Instead, the linearization point of the insert becomes the point in which the new chunk is inserted into the list of chunks. For that this chunk need to be successfully linked to the the frozen chunk (successful Line 24 of the *FreezeRecovery*() method) and then linked into list (Lines 8 or 10 of the *HelpSwap*() method).

When the deletion operation is not successful, i.e., an entry with the input key is not found, then the linearization point is the linearization point of the unsuccessful *Find* that is called on Line 7 of *DeleteInChunk*. And again, a special case is the one in which the freeze recovery is the one to not find the key and decide on a failure. In this case, the unsuccessful search of the key is executed on a new local chunk and the linearization point is the point in which the new chunk is inserted into the list of chunks. The lines are explained above.

The linearization point of search is the linearization point of the *Find* method invoked at Line 1 of the *SearchInChunk* method.

The linearization point of the *Find* method is the a delicate one. The *Find* method may traverse a chunk while it is being frozen. At the same time, the freeze may terminate concurrently, and *inserts* and *deletes* may occur on a new chunk that is not accessed by the find. Therefore, the find may fail to find a key that is inserted before it terminates. We, therefore, set the linearization point of *Find* to be the minimum between its standard linearization point and the time in which a frozen chunk is taken out of the list by insertion the new chunk, that is supposed to replace it.

Thus, the linearization point of a *Find* may happen earlier than the actual time when the find locates (or fails to locate) the input key. The point at which the frozen chunk is replaced satisfies that a new chunk can not modify keys before [1]. This discussion is formalized below.

Consider a *Find* operation on a chunk C, we define the *exchange point* of the find operation on C, denoted $EP(C)$ to be ∞ if the *freezeState* of C is NO_FREEZE at the time that the *Find* method returns. Otherwise, $EP(C)$ is defined to be the time in which the first thread succeeds to replace C with the new chunk pointed by C's *new* pointer. Namely, among all threads executing the *HelpSwap*() method on chunk C, $EP(C)$ is determined to be the minimum time in which one of them started executing Line Lines 8 or 10. Now that we have defined $EP(C)$, we consider the normal operation of *Find*, set linearization points to it, and then select the minimum between them and $EP(C)$.

Again, we separate for successful and unsuccessful cases. When the *Find* is successful, i.e., it returns a non-NULL *cur* pointer, the linearization point of *Find* happens when the *cur* pointer successful passes the validation check in Line 15 in the *Find* method. (Note that the validation is successful when the condition in Line 15 is evaluated to FALSE.) And as explained earlier, if $EP(C)$ happens earlier, then $EP(C)$ is the linearization point.

The unsuccessful case is more involved. Consider an execution of *Find* with input key *k*. There are two failure possibilities.

1. The first possibility is that an entry with k existed in the list but was marked as deleted. In this case the execution of *Find* disconnects it and the linearization point is the successful removal of the entry from the list, i.e., the successful CAS in Line 10 of *Find*.

2. The second possibility is that the entry with k did not exist in the list when *Find* searched for it (even not with a deletion mark). In this case, we set the linearization point of the failing *Find* to be when the pointer to the entry with the smallest key higher than k was loaded into the local variable *cur* in Lines 2, 12 or 19 of the *Find* method.

Again, the above two linearization points are set only if they happen before $EP(C)$. Otherwise, $EP(C)$ is the linearization point.

F Some words on design considerations and correctness

In this section we explain the main idea behind the algorithm, which form the intuition for a correctness proof of this paper. Various parts of the algorithm are not new. The use of hazard pointer is similar to previous work, and the synchronization operations are used in a standard manner. The main deviation from previous work is the use of the freeze process to avoid many of the concurrency problems that naturally arise without it. The main problem is that when many concurrent operations are run on a chunk, it is not easy to determine how many entries reside on it, and whether it requires a split or a merge or none. An attempt to decide on a split and then reverse the decision may run into serious synchronization difficulties. We therefore choose the freeze method to stabilize it and make all threads work in harmony on it afterwards.

When a thread fails to find space for allocation, or when the size of the the chunk appears to be too low for a delete, a freeze is initiated. the freeze process is not atomic. While entries

[1]Except those inserted/deleted in the same point of time by freeze help. They are considered to be modified ϵ after.

are marked as frozen, more inserts and deletes may happen and the need for a split or a merge may change during the freeze process. However, the freeze process is irreversible. The thread that started it will go on marking entries as frozen whenever it gains CPU access, and other threads that fail to insert or delete will join and help freezing the chunk. When all entries in the chunk are frozen, no more updates can occur on this chunk the continuation of the recovery for this chunk is completely determined from that point on. Thus, even if many threads attempt to build new chunks to replace the frozen one, they will all build exactly the same replacements and it does not matter which thread will do the final action of swapping the old chunk out of the list replacing it with the newly prepared chunks. It doesn't even matter if some of the work is done by one thread and some by others, they are all guaranteed to create the same structure. Only after finishing with the replacement of the old chunk, will the threads re-attempt the operation that failed. The only difference in the results of newly created chunks can be in result of promoting the insertion, deletion or enslaving during the freeze recovery.

To summarize, there are two main strategies. The first says that once an entry is marked frozen it will not be modified again. Furthermore, when all entries in a chunk are marked as frozen, all entries in the chunk will not be modified anymore, making the chunk data stable. This ensure that two threads cannot disagree on the "death" state of an entry. When a thread sees the entry marked as frozen, it knows that no other thread will see it not-frozen in the future of the execution. The second strategy says that once a chunk is stable, any thread can decide on what needs to be done with this chunk and any thread can actually do it. All threads must reach the same decisions exactly and they must all attempt to put exactly the same values in exactly the same format of new chunks. Therefore, it odes not matter which of these thread does what. The outcome is determined when the chunk gets stable, and all races become benign.

From these two design points many of the invariants follow. For example, a thread can get inactive for as long as it wishes. When it wakes up, the chunk it is accessing may be frozen, but hazard pointers ensure that the chunk has not been reclaimed, and any attempt of this thread to modify the chunk will reveal the fact that all entries are frozen. The thread will then try to take part in the freeze process and will quickly discover the chunks that replaced the frozen one and apply its modifications to them.

Function's Signature	**Explanations**
word combine (Xbits x, Ybits y);	Concatenates two strings of bits into one machine word, when x comes goes to the most-significant bits, and y to the least-significant bits.
bool isFrozen (entry* p);	Checks if the frozen bit (second LSB) is set in a given pointer p and returns TRUE or FALSE accordingly.
entry* markFrozen (entry* p);	Returns the value of a pointer p with the frozen bit set to one; it doesn't matter if in initial p this bit was set or not.
entry* clearFrozen (entry* p);	Returns the value of a pointer p with the frozen bit reset to zero; it doesn't matter if in initial p this bit was set or not.
bool isDeleted(entry* p);	Checks if deleted bit (LSB) is set in given pointer p.
entry* markDeleted (entry* p);	Returns the value of a pointer p with the deleted bit set to one; it doesn't matter if in initial p this bit was set or not.
entry* clearDeleted (entry* p);	Returns the value of a pointer p with the deleted bit reset to zero; it doesn't matter if in initial p this bit was set or not.
bool isSwapped (chunk* c);	Checks if swapped bit (LSB) is set in given pointer to a chunk c.
chunk* markSwapped (chunk* c);	Returns the value of a pointer c with the swapped bit set to one; it doesn't matter if in initial c this bit was set or not.
chunk* clearSwapped (chunk* c);	Returns the value of a pointer c with the swapped bit set to zero; it doesn't matter if in initial c this bit was set or not.
void copyToOneChunk (chunk* *old*, chunk* *new*);	Goes over all reachable entries in the *old* chunk linked list and copies them to the *new* chunk linked list. It is assumed no other thread is modifying the *new* chunk, and that the old chunk is frozen, so it cannot be modified as well.
key mergeToTwoChunks (chunk* $old1$, chunk* $old2$, chunk* $new1$, chunk* $new2$);	Goes over all reachable entries in the $old1$ and $old2$ chunks linked lists (which are sequential), finds the median key (which is returned) and copies the bellow-median-value keys to the $new1$ chunk linked list and the above-median-value keys to the $new2$ chunk linked list. In addition it sets the $new1$ chunk's pointer *nextChunk* to point to the $new2$ chunk. It is assumed that no other thread modifies the $new1$ and $new2$ chunks, and that the old chunks are frozen and thus cannot be modified as well.
void mergeToOneChunk (chunk* $old1$, chunk* $old2$, chunk* *new*);	Goes over all reachable entries on the $old1$ and $old2$ chunks linked lists (which are sequential and have enough entries to fill one chunk's linked list) and copies them to the *new* chunk linked list. It is assumed that no other thread modifies the *new* chunk and that the old chunks are frozen and thus don't change.
key splitIntoTwoChunks (chunk* *old*, chunk* $new1$, chunk* $new2$);	Goes over all reachable entries on the *old* chunk linked list, finds the median key (which is returned) and copies the bellow-median-value keys to the $new1$ chunk and the above-median-value keys to the $new2$ chunk. In addition it sets the $new1$ chunk's pointer *nextChunk* to point at the $new2$ chunk. It is assumed that no other thread is modifying the $new1$ and $new2$ chunks, and that the old chunk is frozen and cannot be modified.
int getEntrNum (chunk* c);	Goes over all reachable entries in Chunk c, counts them, and returns the number of entries. Chunk c is assumed to be frozen and thus cannot be modified.
void Allocate();	Allocates a new chunk as a zeroed memory chunk. The freeze state is set to NO_FREEZE.

Figure 6: The specification of (simple) supporting functions.