Locality-Conscious Lock-Free Linked Lists*

Anastasia Braginsky and Erez Petrank

Dept. of Computer Science, Technion - Israel Institute of Technology
{anastas,erez}@cs.technion.ac.il

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 (a.k.a. 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]. Lock-free algorithms 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. 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 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

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

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 searching 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 an 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 a single word CAS commands, were the keys are unique and ordered. The algorithm assumes no lock-free garbage collector. The list design is locality conscious. The design poses a restriction on the keys and data length. For 64-bit 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. More detailed explanations and pseudo-code can be found in full version of this article [2].

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 of the linked list as representing a set of keys,

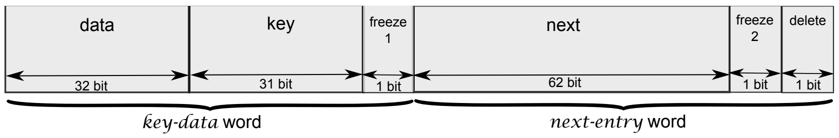

Fig. 1. The entry structure

each associated with a data part. Following previous work [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 be derived, by 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

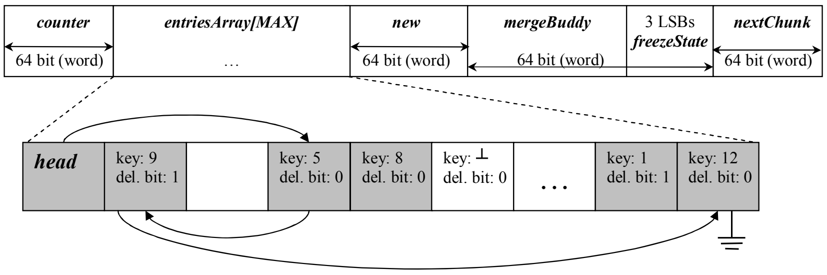

Fig. 2. The chunk structure

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 MAX $>$ 2·MIN+1, since splitting a large chunk must create two well-formed new chunks. In practice MAX will be substantially larger than 2·MIN to avoid 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.

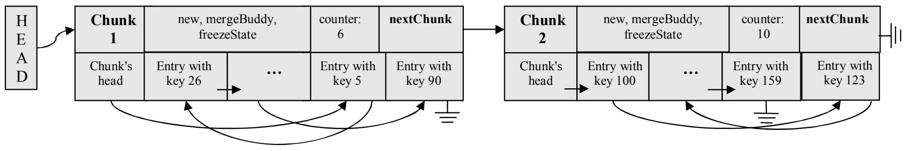

Fig. 3. The list structure

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 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. Each of this operations starts by invoking *FindChunk* method to find the relevant chunk. Then they call *SearchInChunk*, or *InsertToChunk*, or *DeleteInChunk*according to the desired operation, and finally, the hazard pointers *hp2*, *hp3*, *hp4*, and *hp5* are nullified, to release the hazard pointers set by the *FindChunk* method and 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 appear in [2].

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. For lack of space they are not specified in this short paper. The details appear in [2]. 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 1. 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 to the list, then the chunk counter is incremented only by

Algorithm 1. 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;      // Increase the chunk's counter
   case SUCCESS_OTHER:                         // Entry was inserted by other thread
      result = TRUE; break;                                 // due to help in freeze
   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
}
```

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 [2]). 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.

Algorithm 2. 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 (prev, cur)
 4:     if ( Find(chunk, key) )                               // This key existed in the list
 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. First attempt setting next field
10:     if ( !CAS(&(entry→next), savedNext, cur) ) continue;
11:     if ( !CAS(prev, cur, entry) ) continue;                        // Attempt linking
12:     return SUCCESS_THIS;                              // both CASes were successful
13: }
}
```

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 2. The input parameter *entry* is a 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. Finally, two CASs are used to link the entry to the list. Whenever a CAS fails, the insertion starts from scratch.

Algorithm 3. The main freeze method

```
chunk* Freeze(chunk* chunk, key, data, triggerType tgr, Bool* res)
{
 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 marked EXTERNAL_FREEZE before Line 1 executed.
 7:    master = chunk→mergeBuddy;                    // Get the master chunk
 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, tgr, res);
13: }
14: decision = FreezeDecision(chunk);        // The freeze state is INTERNAL_FREEZE
15: if ( decision == MERGE ) mergePartner = FindMergeSlave(chunk);
16: return FreezeRecovery(chunk, key, data, decision, mergePartner, trigger, res);
}
```

5 The Freeze Procedure

We now provide more details about the freeze procedure. The full description is presented in [2]. 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 3. 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*. To finish this freeze, we make sure that the master 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. Finally, the *FreezeRecovery* method is called to complete the freeze process. Next, we explain each of the stages. The full details including pseudo-code appear in [2].

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. 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. 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. Full Version of Locality-Conscious Lock-Free Linked Lists, `http://www.cs.technion.ac.il/~erez/Papers/lf-linked-list-full.pdf`
3. Fomitchev, M., Rupert, E.: Lock-free linked lists and skip lists. In: Proc. PODC (2004)
4. Fraser, K.: Practical lock-freedom, Technical Report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory (February 2004)
5. Frias, L., Petit, J., Roura, S.: Lists revisited: Cache-conscious STL lists. J. Exp. Algorithmics 14, 3.5–3.27 (2009)
6. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Proc. PODC (2001)
7. Herlihy, M.: Wait-free synchronization. TOPLAS (1991)
8. Michael, M.M.: High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In: Proc. SPAA (2002)
9. Michael, M.M.: Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In: Proc. PODC (2002)
10. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. TPDS (June 2004)
11. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann, San Francisco (2008)
12. Valois, J.D.: Lock-free linked lists using compare-and-swap. In: Proc. PODC (1995)
13. Treiber, R.K.: Systems programming: Coping with parallelism, Research report RJ 5118, IBM Almaden Research Center (1986)