Implementation of an Extensible Lock-Free Hash table with Memory Management

Aaron Cherian
atc8@rice.edu

John Mellor-Crummey
johnmc@rice.edu

Abstract. *Parallelism provides speedup when incorporated into data structures and algorithms. The primary motivation of this project was to put some of the learning from Comp 522 into practice. With this in mind, the goal set for this project was to do a small exploration of the parallel data structure literature and to develop and incorporate one such data structure to an existing application. The application selected was HPCToolkit, a framework for application profiling. HPCToolkit uses lock-based splay-trees, and we want to see what could be used as a more optimal substitute. Through the literature survey, lock-free hash tables seemed the most promising option to pursue. The lock-free hash table implementation chosen was Split-Ordered-Lists. We discuss the implementation details for Split-Ordered-Lists and compare its performance to splay-trees. Although there is still scope for optimizations, the performance analysis shows promising results. The code for the implementation of Split-Ordered-Lists developed as part of this project can be found at https://github.com/aarontcopal2/Split-Ordered-Lists*

1. Introduction

Concurrent data-structures are a necessity in a parallel application. Almost all of the data-structures that we are familiar with; arrays, binary trees, linked-lists, hash tables, queues, skip-lists can be tweaked concurrent usage. Based on the performance needs of one's application, the choice of the data-structure can be made.

The motivation of my work is to identify a concurrent data-structure to replace the existing lock-based splay-tree implementation inside HPCToolkit.[1] Lock-free data structures have no dead-locks, no delays due to slow threads and give equal or better performance than lock-based implementations for large parallel workloads. Ideally, the replacement should be lock-free and obviously, better than the current solution.

Factors for choosing a concurrent data-structure are performance, synchronization techniques employed, ease of implementation, programming language, etc. Gramoli[2] does a comprehensive analysis of 31 concurrent data-structures and compares their performance in his Synchrobench paper. I have taken help of this work to select the data-structure for our use case inside HPCToolkit. The performance results from Synchrobench show that hash tables support the highest operations/second. This is intuitive, since hash tables are the only data structure that promise O(1) time complexity.

The hash table choices presented in the paper are j.u.c.ConcurrentHashMap(Java), Micheal's hash table(C/C++), Cliff Click's hash map(Java), Contention-friendly hash table(Java), Resizable hash table(Java) and Elastic hash table(C/C++). Since HPCToolkit

uses C/C++, I narrowed down the choices to Micheal's hash table and Elastic hash table (Java data structures can be considered, but after going through the efforts of porting the code to C++, there is no guarantee of getting a similar performance). Micheal's hash table outperforms Elastic hash table by a big margin. Another advantage of Micheal's hash table is that it uses the Read-Modify-Write or Compare and Swap(CAS) synchronization technique. As per Gramoli's paper, CAS is the fastest synchronization technique for multi-core architectures (other alternatives are locks, transactions, Copy-on-Write). Thus we decided to explore Micheal's hash table implementation in detail.

Roadmap. In Section 2, we will look at Micheal's hash table implementation and its succcessors. Section 3 and 4 will explore the implementation details and performance comparison of a variant of Micheal's hash table called Split-Ordered-Lists. Section 5 describes the code analysis for the same.

2. Background

We will look at the implementation details of Micheal's hash table (and variants) and the memory reclamation techniques that go along with it.

2.1. Algorithms

First we look at the original implementation of Micheal's hash table, then at 2 variants: the implementation used by Gramoli[2] and an extensible hash table implementation.

2.1.1. Micheal's Hash table

Micheal's hash table[3] implements the buckets in a linked list fashion. It provides lock-free synchronization and has other desirable properties such as space-efficiency, dynamic and linearizable. For memory management, there are multiple feasible techniques(IBM freelist, Safe Memory Reclamation (SMR),etc). Since the algorithm mentioned in the paper includes the logic for SMR, we will explore SMR in detail in section 2.2.1.

2.1.2. Micheal's Hash-table with Harris Linked-List

This is the implementation that Gramoli uses in his Synchrobench paper.[2] It reuses the lock-free linked list by Harris.[4] The reason for the difference in implementation in Gramoli's work is because Micheal published the source code long after the original work was published. When Gramoli started with his work, he did not have the source code that we refer to in section 2.1.1

A key difference in Gramoli's work is the memory management strategy used. Unlike Micheal's hash table, there are no simple memory reclamation strategies for Harris Linked-List. Harris's Linked-List allows a thread traversing the list to access the contents of a node or a sequence of nodes after they have been removed from the list. If removed nodes are allowed to be reused immediately, the traversing thread may reach an incorrect result or corrupt the list. This precludes Harris implementation from using simple and efficient memory management techniques like SMR and freelist.

Gramoli's Synchrobench incorporated memory reclamation using a library called ThreadScan.[5] We will explore ThreadScan in more detail in section 2.2.2.

2.1.3. Split-Ordered Lists: Lock-Free Extensible Hash Tables

Split-ordered lists[6] extends over the ordered list implementation of Micheal's[3] data structure. This data-structure tackles the problem of increasing the bucket count in hash tables without using locks or sub-optimal methods i.e this variant of hash table is extensible. Optimally changing the bucket size is done using the concept of recursive split-ordered lists. The basic idea of the data structure is this: rather than transferring objects between hash-buckets, data is stored in a single linked list and bucket pointers(maintained in a separate data-structure) are updated during a resize operation. The advantages of this data structure are: lock-free, relatively simpler implementation, extensible, good performance in both multi-core and non multi-core environments.

2.2. Memory Management Methods

In lock-based data structures, holding a lock on an object ensures that delete operation(which is responsible for freeing memory) by a thread can be performed without worrying that some other thread may be accessing that object. Lock-free data structures, on the other hand, need complicated strategies to ensure safe deletion.

In this subsection, we will see 2 memory management techniques that work with Micheal's hash table: SMR and ThreadScan.

2.2.1. Safe memory reclamation

Safe Memory Reclamation[7] is a memory management technique developed by Micheal. Internally, it uses hazard-pointers methodology which ensures constant amortized time for each delete operation. Other advantages of hazard-pointers are: wait-free, does not need shared memory, lock-free solution for ABA problem.

The core idea of SMR is to associate a small number of single-writer multi-reader shared pointers, called hazard pointers, with each thread. The hazard pointers hold references to nodes inside Micheal's Linked-List. If a hazard pointer points to a node, it means that node is currently in use by a thread and shouldn't be deleted. This thread can access the node without worrying whether the node reference is valid. Each hazard pointer can be written only by its owner thread, but can be read by other threads.

Whenever a thread wants to delete a node, it stashes the node reference in a private list. When it is determined that its time to free up nodes in the list, the thread scans the hazard pointers of other threads to check if any thread holds references to the nodes inside its private list. If a node(in the private list) is not matched by any of the hazard pointers, then it is safe for this node to be reclaimed. Otherwise, the thread keeps the node inside its list for its next scan. This prevents the freeing of any node marked for deletion if it was pointed to by one or more hazard pointers from a point prior to its removal.

2.2.2. ThreadScan

ThreadScan[5] is a garbage collection tool for C/C++. If the memory node to be reclaimed cannot be reached by any application thread, then the node is safe for deletion. The

way ThreadScan determines this is by using OS signalling techniques to find if the to-be-deleted block is safely reclaimable i.e. no other thread is accessing it. ThreadScan provides 2× performance improvement over Hazard Pointers, which is used internally inside SMR.

A shared buffer for deletion purpose is maintained internally by ThreadScan. When a memory block is marked for deletion, it should be passed to the ThreadScan free function. The free function will add this memory block to the delete buffer. When the delete buffer becomes full, the last thread which called free becomes the memory reclaiming thread. This thread will send signals to all threads accessing the shared data-structure (in our case, a hash table) and check within its call stacks and registers to see if the memory blocks are being accessed by them. If yes, then the block is marked. Once all application threads have responded to the signal, the reclaiming thread safely frees all unmarked blocks.

Modern operating systems give higher priority to signals over threads and have measures to avoid thread starvation. Thus ThreadScan is lightweight and avoids the memory reclaiming thread to be blocking. Another advantage is, since its a garbage collector, the only thing the programmer needs to do is call ThreadScan API for memory reclamation. The memory management code is eliminated altogether.

2.3. Our implementation choice

Split-Ordered Lists is the most recent of Micheal's hash table variants. Apart from giving all the advantages of Micheal's hash table, it ensures that the hash table buckets increase as more nodes get added. This is important, else for large hash tables, we will experience sub-linear time performance.

For memory management, we opted to go with SMR. Although ThreadScan is promising, claims to be better than SMR and removes the burden of managing memory, the end goal is for it to work inside HPCToolkit. And it is possible that ThreadScan's signalling might disrupt the existing behaviour of HPCToolkit for O(threads) time. Further evaluation is required to check for possible side-effects of ThreadScan.

3. Implementation

We will look at the implementation of Split-Ordered Lists along with memory management and other helper functions in detail in this section.

3.1. Data Types and variables

Source code 1 contains the node structure used to store the key-value pair inside the data structure. There are 2 type of nodes used inside Split-Ordered Lists, regular nodes and dummy nodes. For user operations, regular nodes are created and used. For maintaining stable entry-points to buckets inside the hash table, dummy nodes are used. The key that the data structure uses is so_key. For regular nodes, so_key is calculated by setting most significant bit of original key to 1 and reversing the bits. For dummy nodes, so_key is generated by just reversing the bits of the original key. so_key helps maintain the sort order for the nodes. key and isDummy variables inside the struct is maintained just for debugging purposes. The value is represented as a void pointer, thus any custom

memory structure can be retrieved by the hash table. Since we want the link updates to be atomically made, the next pointer is marked by _Atomic keyword.

```
typedef unsigned int uint;
typedef uint so_key_t;
typedef uint t_key;
typedef void* val_t;
typedef struct __node NodeType;


// MarkPtrType will be used as a tagged pointer
// the last bit will be used to mark the node for deletion
typedef NodeType* MarkPtrType;


//Node: contains key and next pointer
struct __node {
    so_key_t so_key;
    t_key key;
    val_t val;
    bool isDummy;
    _Atomic(MarkPtrType) next;
};

// variables for buckets
// segment_t is an array of MarkType pointers
typedef MarkPtrType *segment_t;


// ST is the bucket data structure
// (2D array of MarkPtrType pointers)
struct hashtable {
    _Atomic(segment_t*) ST;
    ...
}
```

Source Code 1. The core data-types used by Split-Ordered-Lists

There is a type-definition for NodeType pointer named as MarkPtrType. This is used as a tagged pointer. When compilers perform memory alignment, the least significant bit of the address is always 0. If a pointer is tagged, it means the last bit is used to store some additional data. In the case of MarkPtrType, the last bit is used to mark a node for deletion. Each node has a next MarkPtrType pointer which will contain the address to the next node in the chain and the last bit set to 1 if the node is marked for deletion.

3.2. API functions for the hash table

The 3 core API functions that the user will use to work with the hash table are map_search, map_insert, map_delete. Along with hashtable_initialize and hashtable_destroy func-

tions, these are the only functions exposed to the user. The code snippets for map_search, map_insert and map_delete is given in source code 2, 3, 4.

map_search, map_insert and map_delete will first check if a resize operation is in progress concurrently by some other thread. The current thread will proceed only after determining that resize operation is not in progress. Due to the nature of resize operation (which we will discuss in the upcoming subsection), the number of buckets keep increasing as more elements are inserted. The next step in map_search, map_insert and map_delete is to find the bucket for the node/key that we are concerned with and initialize it if necessary. To understand the rest of the code, it is important to understand how the hash table is structured. Unlike a conventional hash table that has buckets and a linked-list of nodes sprouting from each bucket, Split-Ordered-Lists has a single linked-list that stores all the data. The bucket variable ST is responsible for providing fast entry-points into this linked-list. Each API function then calls its corresponding internal linked-list function that will take care of the insertion, search or deletion.

```
val_t map_search(hashtable *htab, t_key key) {
    // if a resize is in progress(initiated by another thread),
    // block operations and make current thread a resize helper
    while (pthread_rwlock_tryrdlock(&htab->resize_rwl) != 0) {
        resize_replica(htab);
    }
    uint bucket = key % atomic_load(&htab->size);

    // ensure that bucket is initialized
    try_again: ;
    MarkPtrType bucket_ptr = get_bucket(htab, bucket);
    if (bucket_ptr == NULL) {
        bucket_ptr = initialize_bucket(htab, bucket);
        if (bucket_ptr == NULL) {
            goto try_again;
        }
    }

    // find node in linked-list with corresponding key
    MarkPtrType result = list_search(htab, &bucket_ptr,
            so_regular_key(key));
    pthread_rwlock_unlock(&htab->resize_rwl);
    if (result && result->val) {
        return result->val;
    }
    return NULL;
}
```

Source Code 2. map search

Inside map_insert, we check if its necessary to resize the hash table, i.e resize the bucket variable ST. And when I say resize, I mean only expansion. We don't shrink the buckets since there is no benefit for the same. Inside map_insert, new nodes to be inserted are malloc'ed. An improvement for this would be to reuse nodes that were previously deleted. We can save some time spent in memory allocation and freeing. This improvement also fits well with the memory reuse strategy employed inside HPCToolkit.

```
bool map_insert(hashtable *htab, t_key key, val_t val) {
    // if a resize is in progress(initiated by another thread),
    // block operations and make current thread a resize helper
    while (pthread_rwlock_tryrdlock(&htab->resize_rwl) != 0) {
        resize_replica(htab);
    }

    uint bucket = key % atomic_load(&htab->size);
    MarkPtrType *seg = atomic_load(&htab->ST)[bucket / SEGMENT_SIZE];

    // intialize bucket if not already done
    try_again: ;
    MarkPtrType bucket_ptr = get_bucket(htab, bucket);
    if (!bucket_ptr) {
        bucket_ptr = initialize_bucket(htab, bucket);
        if (bucket_ptr == NULL) {
            goto try_again;
        }
    }

    // inside the node, key is stored in split-ordered form
    NodeType *node = malloc(sizeof(NodeType));
    node->so_key = so_regular_key(key);
    node->key = key;
    node->val = val;
    node->isDummy = false;
    atomic_init(&node->next, NULL);

    // list_insert will fail if the key already exists
    if (!list_insert(htab, &bucket_ptr, node)) {
        retire_node(htab, node);
        pthread_rwlock_unlock(&htab->resize_rwl);
        return false;
    }

    pthread_rwlock_unlock(&htab->resize_rwl);
    // if insertion is succesful, increment the count of nodes.

    // If the load factor of the hashtable > MAX_LOAD, resize the hash table
    size_t count = fetch_and_increment_count(htab);
    size_t size = atomic_load(&htab->size);
    size_t load =  count / (size * SEGMENT_SIZE);
    if (load > MAX_LOAD) {
        resize_hashtable(htab);
    }
    return true;
}
```

Source Code 3. map insert

```
bool map_delete(hashtable *htab, t_key key) {
    // if a resize is in progress(initiated by another thread),
    // block operations and make current thread a resize helper
    while (pthread_rwlock_tryrdlock(&htab->resize_rwl) != 0) {
        resize_replica(htab);
    }
    uint bucket = key % atomic_load(&htab->size);

    // ensure that bucket is initialized
    try_again: ;
    MarkPtrType bucket_ptr = get_bucket(htab, bucket);
    if (bucket_ptr == NULL) {
        bucket_ptr = initialize_bucket(htab, bucket);
        if (bucket_ptr == NULL) {
            goto try_again;
        }
    }

    // delete node in linked-list with corresponding key
    if (!list_delete(htab, &bucket_ptr, so_regular_key(key))) {
        // node doesnt exist in the list,
        // release lock and return false
        pthread_rwlock_unlock(&htab->resize_rwl);
        return false;
    }

    // if deletion is succesful, decrement the count of nodes.
    fetch_and_decrement_count(htab);
    // release lock and return true
    pthread_rwlock_unlock(&htab->resize_rwl);
    return true;
}
```

Source Code 4. map delete

3.3. Internal functions for Linked-List

As mentioned in the previous subsection, the entire data is maintained in one linked-list. Here we will look at the helper functions for searching, inserting and deleting elements in the list(source code 5, 6, 7). One important thing to notice is that list_search, list_insert, list_delete all take a head pointer to the list as an argument. This is not the real head of the linked list, but a pseudo head. This pseudo head node (aka dummy nodes) are stored in the ST variable. Thus the buckets act as faster entry points to the linked-list.

Note that list_find is called inside each of these functions. list_find (function omitted from the paper for brevity) is responsible for finding the right position inside the list to do the search, insert and delete operation. Since list_insert calls list_find, we are inserting the element in a sorted order in the list. And the so_key based arrangement ensures that even after the resizing of the data structure, there is no need to reshuffle the elements inside the linked-list to maintain the sorted order.

```
MarkPtrType list_search(hashtable *htab, MarkPtrType *head, so_key_t key) {
    MarkPtrType *prev;
    MarkPtrType node = list_find(htab, head, key, &prev);
    clear_hazard_pointers(htab);

    if (node && node->so_key != key) {
        // if key is not present, we return NULL
        return NULL;
    }
    return node;
}
```

Source Code 5. list search

```
bool list_insert(hashtable *htab, MarkPtrType *head, NodeType *node) {
    bool result;
    MarkPtrType cur, *prev;
    so_key_t so_key = node->so_key;

    while (true) {
        // find the insertion position for the new element in the list
        cur = list_find(htab, head, so_key, &prev);
        if (cur && cur->so_key == so_key) {
            // if a key is already present, we return
            result = false;
            break;
        }

        // creating a link from node->cur and prev->node
        // Thus link prev->cur is removed
        atomic_store(&node->next, create_mark_pointer(get_node(cur), 0));

        MarkPtrType expected = create_mark_pointer(get_node(cur), 0);
        MarkPtrType desired = create_mark_pointer(node, 0);
        if (atomic_compare_and_swap(prev, expected, desired)) {
            result = true;
            break;
        }
    }
    clear_hazard_pointers(htab);
    return result;
}
```

Source Code 6. list insert

```
bool list_delete(hashtable *htab, MarkPtrType *head, so_key_t key) {
    bool result;
    MarkPtrType *prev;
    NodeType *cur, *next;

    while (true) {
        // find the position of the element in the list
        cur = list_find(htab, head, key, &prev);
        if (!cur || cur->so_key != key) {
            // exit if element not found
            result = false;
            break;
        }
        next = get_hazard_pointer(htab, 0);

        // cur->next will contain pointer to next node. Its last bit
        // will denote if cur is marked for deletion.
        // bit=0/1 -> not-deleted/deleted

        // expected: cur points to next and cur is not marked for deletion
        MarkPtrType expected = create_mark_pointer(next, 0);
        MarkPtrType desired = create_mark_pointer(next, 1);

        // if expected matches, set cur marked for deletion
        if (!atomic_compare_exchange_strong(&cur->next,
                &expected, desired)) {
            continue;
        }

        // expected: prev points to cur and prev is not marked for deletion
        expected = create_mark_pointer(cur, 0);
        desired = create_mark_pointer(next, 0);

        // if expected matches, make prev point to next
        if (atomic_compare_and_swap(prev, expected, desired)) {
            // free the deleted node
            retire_node(htab, cur);
        } else {
            // some other thread has changed prev
            // (either marked it for deletion or inserted another element)
            list_find(htab, head, key, &prev);
        }
        result = true;
        break;
    }
    clear_hazard_pointers(htab);
    return result;
}
```

Source Code 7. list delete

3.4. Memory management

Any node that needs to be deleted is passed to the retire_node function. The reason for not freeing the node immediately is because we need to ensure that no other thread is using the node while another thread tries to delete it. Every retired node is pushed to a thread private variable called retired_list. Once the count of nodes inside the retired_list crosses a certain threshold, we scan the list for nodes that are ready to be deleted. This work is done by the function local_scan_for_reclaimable_nodes (function omitted from the paper for brevity)

The way we check if a node is being used is by using the hazard pointers data structure. Each active thread keeps the nodes that its currently using inside its hazard pointer array(each thread maintains a hazard pointer array). Inside the function local_scan_for_reclaimable_nodes, the list of nodes present inside hazard pointers across all threads is created. For reference, lets call this list hp_list. Any node present both inside retired_list and hp_list is not ready to be freed yet. All other nodes inside retired_list are freed. The snippet for retire_node can be found in source code 8.

```
// variables for SMR
__thread NodeType *local_retired_list_head;
__thread NodeType *local_retired_list_tail;
__thread uint local_retired_node_count = 0;

// variables for hazard pointers
__thread hazard_ptr_node *local_hp_head;

void retire_node(hashtable *htab, NodeType *node) {
    /* RETIRE_THRESHOLD should be small so that unneeded nodes are
    * removed on regular basis.
    * large threshold values will cause problems in case of idle threads */
    uint RETIRE_THRESHOLD = atomic_load_explicit(
        &htab->hazard_pointers_count, memory_order_acquire) + 10;
    atomic_store(&node->next, NULL);

    if (local_retired_list_head) {
        atomic_store(&local_retired_list_tail->next, node);
        local_retired_list_tail = atomic_load(&local_retired_list_tail->next);
    } else {
        local_retired_list_head = node;
        local_retired_list_tail = node;
        retired_list_node *rln = malloc(sizeof(retired_list_node));
        rln->thread_retired_list_head = local_retired_list_head;
        atomic_store(&rln->next, NULL);
        update_global_retired_list(htab, rln);
    }
    local_retired_node_count++;

    if (local_retired_node_count >= RETIRE_THRESHOLD) {
        local_scan_for_reclaimable_nodes(htab, atomic_load(&htab->hp_head));
    }
}
```

Source Code 8. retire_node() function that contains the skeleton for the memory reuse strategy

3.5. Data structure resizing

Resizing requires malloc'ing a new memory region, copying old values to this new region and swapping the data structure pointers. This operation needs to be atomic, else we may loose some insertions happening between memcpy and swapping of old table with new.

Calling function resize_hashtable(source code 9) starts the resize. There are 3 helper functions for atomic resizing: resize_primary, resize_replica and resize_task. resize_primary(source code 10) is called by the first thread that wants to resize the hash table. This thread will be the primary thread. Threads trying to do an operation (search, insert, delete or resize) while a resize is in progress will become secondary resizing threads and help with the resizing by calling the function resize_replica(source code 11). Thus, although the resize operation is blocking, other blocked threads can speedup the operation.

Both resize_primary and resize_replica delegate the actual resizing work to resize_task. For the sake of brevity, the code snippet for that function is not included in the report. resize_task does 2 things, initialize the newly allocated ST variable and move contents from old_ST to ST. Both ST and old_ST are divided into blocks that can be atomically operated on by different threads.

The entire logic of atomic resizing used in this project is referenced from Jonathon Anderson and Srđan Milaković's Comp 522 project work from last year.

```
struct hashtable {
    _Atomic(segment_t*) ST;
    _Atomic(segment_t*) old_ST;
    atomic_size_t size;
    atomic_size_t old_size;

    atomic_size_t resizing_state;
    atomic_size_t next_init_block;
    atomic_size_t num_initialized_blocks;
    atomic_size_t next_move_block;
    atomic_size_t num_moved_blocks;
    pthread_rwlock_t resize_rwl;
    ...
}

static void resize_hashtable(hashtable *htab) {
   size_t resizing_state = atomic_load(&htab->resizing_state);
    if (resizing_state == 0 &&
        atomic_compare_exchange_strong(&htab->resizing_state,
            &resizing_state, ALLOCATING_MEMORY)) {
        // Primary thread
        pthread_rwlock_wrlock(&htab->resize_rwl);
        resize_primary(htab);
        pthread_rwlock_unlock(&htab->resize_rwl);
    } else {
        // Replica(Secondary) thread
        resize_replica(htab);
    }
}
```

Source Code 9. resize_hashtable and related variables

```c
static void resize_primary(hashtable *htab) {
    // initialize values
    size_t old_size = atomic_load(&htab->size);
    size_t size = old_size * 2;
    segment_t *old_ST = atomic_load(&htab->ST);
    atomic_store(&htab->old_size, old_size);
    atomic_store(&htab->size, size);
    atomic_store(&htab->old_ST, old_ST);

    // malloc new table.
    segment_t *ST = malloc(sizeof(segment_t) * size);
    for (int i = 0; i < size; i++) {
        ST[i] = (MarkPtrType*)malloc(SEGMENT_SIZE * sizeof(MarkPtrType*));
    }
    assert(ST);
    atomic_store(&htab->ST, ST);

    // change state from allocation to moving data
    size_t resize_state = atomic_fetch_xor(&htab->resizing_state,
            ALLOCATING_MEMORY ^ MOVING_DATA);

    // resize_task with blocking=1
    resize_task(htab, BLOCKING);

    // change state from moving data to cleaning
    resize_state = atomic_fetch_xor(&htab->resizing_state,
            MOVING_DATA ^ CLEANING);

    // wait for active replica threads to be zero
    while (GET_ACTIVE_REPLICAS(resize_state) != 0) {
        resize_state = atomic_load(&htab->resizing_state);
    }

    // no more active replicas
    // reinitialize values for future resize
    atomic_store(&htab->next_init_block, 0);
    atomic_store(&htab->num_initialized_blocks, 0);
    atomic_store(&htab->next_move_block, 0);
    atomic_store(&htab->num_moved_blocks, 0);

    // freeing old table
    // free child segments first
    for (int i = 0; i < old_size; i++) {
        free(old_ST[i]);
    }
    free(old_ST);

    // change state from cleaning to no_resizing
    resize_state = atomic_fetch_xor(&htab->resizing_state,
            CLEANING ^ NO_RESIZING);
}
```


Source Code 10. resize primary

```c
static void resize_replica(hashtable *htab) {
    // get resize state for hashtable
    size_t resize_state = atomic_load(&htab->resizing_state);

    // if state = (cleaning | no_resizing), resize has finished
    if (IS_NO_RESIZE_OR_CLEANING(resize_state)) {
        return;
    }

    // register as replica and check again if resize has finished
    resize_state = atomic_fetch_add(&htab->resizing_state, STATE_INCREMENT);
    if (IS_NO_RESIZE_OR_CLEANING(resize_state)) {
        atomic_fetch_sub(&htab->resizing_state, STATE_INCREMENT);
        return;
    }

    // wait for new table allocation to complete
    while (GET_STATE(resize_state) == ALLOCATING_MEMORY) {
        resize_state = atomic_load(&htab->resizing_state);
    }

    // check if resize is done
    assert(GET_STATE(resize_state) != NO_RESIZING);
    if (GET_STATE(resize_state == CLEANING)) {
        atomic_fetch_sub(&htab->resizing_state, STATE_INCREMENT);
        return;
    }

    // resize_task with blocking=0
    resize_task(htab, NON_BLOCKING);

    // deregister replica
    atomic_fetch_sub(&htab->resizing_state, STATE_INCREMENT);
}
```

Source Code 11. resize replica

4. Performance Comparison

First we describe the system on which we ran the performance test. The tests were run on gpu.cs.rice.edu which has 2 sockets, 18 cores/socket and 2 threads/core i.e 72 hardware threads. To do the performance comparison, 5 benchmarks are used. Each benchmark runs $> 1M$ operations and is run 6 times for different thread values(T=1, 2, 4, 8, 16, 32). The benchmarks consist of an intermix of insert, search and delete operations and the split of operations for each benchmark is listed below:

1. Benchmark 1: 50% inserts, 0% finds, 50% deletes
2. Benchmark 2: 33% inserts, 33% finds, 33% deletes
3. Benchmark 3: 25% inserts, 50% finds, 25% deletes
4. Benchmark 4: 15% inserts, 70% finds, 15% deletes

5. Benchmark 5: 5% inserts, 90% finds, 5% deletes

Figure 1 shows the graphical comparison of Splay trees with Split-Ordered-Lists. For the 1st 4 benchmarks, single thread execution runs slow for Split-Ordered-Lists. For multi-thread executions, Split-Ordered-Lists surpass splay trees eventually for all benchmarks. I believe that with some refactorings, our implementation can outperform splay trees in all scenarios. Running HPCToolkit to profile the benchmarks showed that a lot of time is spent in the hash table initialization and destruction calls. We will discuss how to work on these in section 6.2.

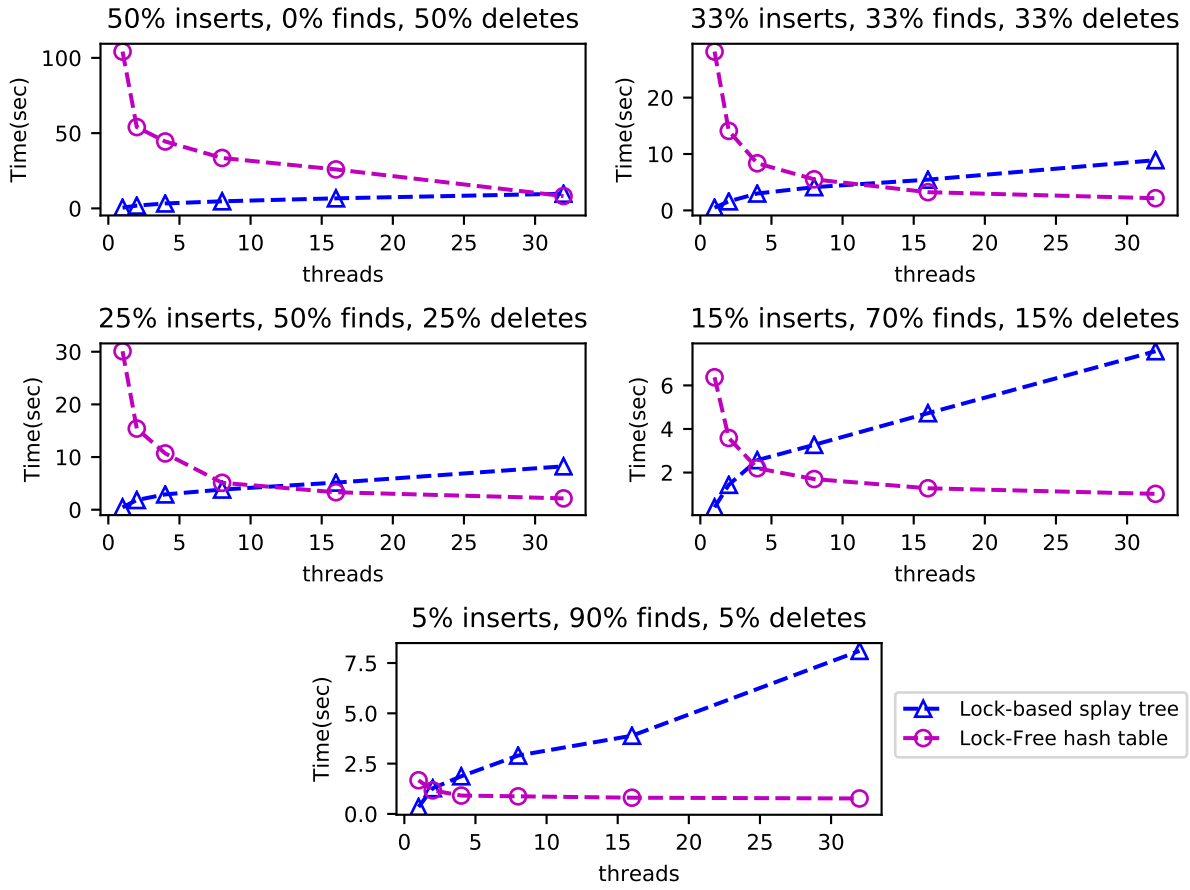

Figure 1. Comparison of Lock-based splay-trees and Lock-free hash table (1M operations)

On some occasions, when operation count $> 2M$, Split-Ordered-Lists takes up a lot of time. This is an issue that needs further investigation.

5. Code Analysis using Valgrind

Valgrind is an instrumentation framework that can automatically detect memory management and threading bugs. We see how this was done in this section.

5.1. Data Race Detection

For detecting data races in the code-base, Helgrind was used. Helgrind is a tool inside the Valgrind framework. It detects synchronization errors based on Happens-Before relationships. This means that unlike tools like Non-Determinator (data race detector for Cilk), which check the code syntax for inherent race conditions, Helgrind checks at runtime if data races occur. This means that Helgrind wont capture all data races lurking in the code, it will only detect the data races that happen during runtime.

Thus Helgrind race reports are dependent on the inter-leavings of threads and how rigorously one strains the parallelism of the code. Thus it is important to stress the concurrency of the code base in the microbenchmark. With this in mind, benchmarks were run by upto 32 threads for 4096 operations. Some real data races were reported by Helgrind, which were fixed duly. Helgrind also reports false race. To fix this, we inserted annotations in the code (like ANNOTATE_HAPPENS_BEFORE, ANNOTATE_HAPPENS_AFTER, etc). Helgrind currently reports no data races for the code-base.

5.2. Memory Leaks

For detecting memory leaks in the program, Valgrind's Memcheck tool was used. It helped in confirming that all the allocations had been freed. Memcheck was especially helpful in finding out some non-trivial memory leaks. E.g. each thread maintained a private list of nodes that needed to be freed back to the global memory. The freeing process was triggered only when this list reached a certain threshold size. Thus a small count of nodes were always left to be de-allocated at the end of execution. Memcheck brought this issue to light and thus this memory leak was taken care of. Current memcheck report states that no memory leaks are present in the data structure.

6. Future Work

6.1. Integration with HPCToolkit

The integration of Split-Ordered-Lists data structure with HPCToolkit is still a work in progress. For memory allocation/deallocation, we are currently using malloc/free calls. The existing memory reuse strategy inside HPCToolkit is to use wait-free channels that facilitate the reuse of nodes. Once the malloc/free calls are replaced by channels, we should be able to integrate Split-Ordered-Lists data structure into HPCToolkit. An advantage of this conversion is performance speed-up, since we are reducing the count of actual reallocations and deallocations.

6.2. Memory Ordering and Optimizations

The primary effort was to get a working prototype of Split-Ordered-Lists data structure. There is scope for optimizing the performance of operations.

1. Atomic operations are using sequential memory ordering in many places. Changing it to weaker memory orders wherever possible will make the parallel execution more efficient.
2. list_find is called by all data structure API's. Optimizations in this function will speedup each operation of Split-Ordered-Lists.

3. hashtable_destroy function is responsible for freeing hash table memory. It does all of its work sequentially. Adding parallelism in this function will be beneficial.
4. Tweaking the buckets parameters(MAX_LOAD and SEGMENT_SIZE) to find the right balance between constant time operations and resizing cost is an avenue that needs to be explored.

6.3. ThreadScan

ThreadScan abstracts out the memory management code for the parallel data structure. That's one less thing to maintain. Since it claims to be faster than SMR, it would be worth the effort to see if it can be made to work with HPCToolkit without any side-effects.

7. Acknowledgements

Firstmost, I would like to thank Prof. John-Mellor Crummy for his help throughout this work. He has provided the needed guidance; from the project-topic selection to the required literature references pointers. His help throughout this project is greatly appreciated.

I thank Rodrigo Kumpera (https://github.com/kumpera/Lock-free-hastable) and Mpastyl(github username, real name unknown) (https://github.com/mpastyl/split-ordered-list). I took the help of their implementations of Split-Ordered-Lists to fill in some of the implementation gaps from the Split-Ordered-Lists paper.[6] Also, their existing work was of great help when I was stuck in implementation details.

I referred Jonathon Anderson and Srđan Milaković's work for resizing concurrent data structures from their Comp 522 project last year. Their work on resizing hash tables in a concurrent environment using a lock-free approach was based on Cliff's wait-free concurrent hash table in Java.[8] Also, Jonathon's help with Helgrind for data race detection is appreciated.

8. Conclusion

We performed a literature survey to identify a good substitute for the spin-lock based Splay-tree data structure inside HPCToolkit. The inference from this survey was that Lock-free hash table given by Split-Ordered-Lists was a promising solution because of 3 attractive properties: lock-free, CAS synchronization operation and O(1) time complexity. Then we discussed the implementation of the data structure with SMR strategy as the choice for memory reclamation. Finally, the performance comparison showed that Lock-free hash tables are promising and are a fit choice for replacing splay tress with some optimizations.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[2] V. Gramoli, "More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), p. 1–10, 2015.

[3] M. M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, (New York, NY, USA), p. 73–82, 2002.

[4] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, (Berlin, Heidelberg), p. 300–314, Springer-Verlag, 2001.

[5] D. Alistarh, W. Leiserson, A. Matveev, and N. Shavit, "Threadscan: Automatic and scalable memory reclamation," *ACM Trans. Parallel Comput.*, vol. 4, May 2018.

[6] O. Shalev and N. Shavit, "Split-ordered lists: Lock-free extensible hash tables," *J. ACM*, vol. 53, p. 379–405, May 2006.

[7] M. M. Michael, "Hazard pointers: safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 6, pp. 491–504, 2004.

[8] C. Click, "A lock-free wait-free hash table," *work presented as invited speaker at Stanford*, 2008.