Proc. of the 11th Intl Symposium on High-Level Parallel Programming and Applications (HLPP)
July 12-13 2018, Orléans, France

Fast Concurrent Skip Lists with HTM

Marios Kardaras · Dimitrios Siakavaras · Konstantinos Nikas · Georgios Goumas · Nectarios Koziris

Received: May 3rd, 2018 / Accepted: June 4th, 2018

Abstract In this paper we explore the efficacy of HTM for implementing concurrent skip lists, a widely used indexing data structure. We first implement the most naive version of an HTM-based skip list where we enclose each update operation in a single HTM transaction. This implementation showcases the simplicity of using HTM, however our experimental results show that it performs poorly, due to its large transactions.

The novelty of our work lies in the way we use HTM to implement a fine-grained HTM-based version of a concurrent skip list with minimal transaction sizes. Our analysis shows that exploiting HTM in such a fine-grained fashion results in very fast concurrent skip lists.

In our experimental evaluation we use our skip list implementations as the basis for implementing ordered sets and priority queues, two of the most common abstract data types. We compare our HTM-based versions with state-of-the-art lock-based and lock-free implementations. Our evaluation reveals advantages and disadvantages of HTM as well as cases where the fine-grained HTM implementation can outperform the other solutions.

Keywords Hardware Transactional Memory · Concurrent Data Structures · Skip Lists

1 Introduction

Multi-core architectures are nowadays ubiquitous and we expect more and more cores to be packed in a single processor chip in the future. However, to

Marios Kardaras · Dimitrios Siakavaras · Konstantinos Nikas · Georgios Goumas · Nectarios Koziris
National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
E-mail: {mkardaras,jimsiak,knikas,goumas,nkoziris}@cslab.ece.ntua.gr

fully exploit the increasing compute capabilities of multi-core hardware, the software needs to be redesigned.

Concurrent data structures are one of the most crucial components of many multi-threaded applications and, as such, it makes sense to focus on making them as fast as possible. A concurrent data structure apart from being thread-safe, i.e., multiple threads can perform operations on it without returning erroneous results or leaving the data structure in an incorrect state, needs to be efficient as well. However, the design and implementation of efficient concurrent data structures typically requires extensive programming effort.

When multiple threads access a data structure some kind of synchronization is necessary to guarantee that the data structure remains in a consistent state. The most widely used synchronization mechanism are locks. With locking, threads acquire exclusive access to the parts of the data structure they modify. Locks can be used through a simple API (i.e., acquire/release locks), but they have drawbacks such as deadlocks, lack of robustness and priority inversion.

Lock-free concurrent data structures eliminate the fallbacks of lock-based implementations. They avoid locking parts of the data structure by exploiting the atomic operations provided by modern processors, such as CAS and LL/SC. Although lock-free implementations can be devised relatively easy for simple data structures such as linked lists, they are challenging for more complex ones such as skip lists [19] and binary search trees [1,10]. The fact that modifications of such complex data structures typically include several memory locations makes it difficult to implement concurrent versions by using the single memory atomic primitives of modern processors.

Transactional Memory (TM) [13] is a synchronization mechanism motivated by such scenarios and provides a simple programming interface that guarantees the atomic modification of multiple memory locations. TM can be used for the implementation of complex data structures, such as skip lists, by enclosing each data structure operation in a transaction. The underlying TM system is then responsible for the correct execution.

In this work we use Intel's HTM, namely Transactional Synchronization eXtensions (TSX) [30] and implement fast concurrent skip lists. We first implement the most naive HTM-based skip list by enclosing each operation (except for contains) in a single HTM transaction to illustrate the simplicity of TM. However, as expected, our experimental evaluation shows that this naive implementation has several performance issues.

The novelty of our work lies in the way we exploit HTM using a fine-grained approach to minimize the size of the executed transactions and avoid the majority of conflict and capacity transactional aborts of the coarse-grained HTM version. Our experimental evaluation reveals the performance benefits of fine-grained HTM when compared against the coarse-grained HTM version as well as other state-of-the-art lock-based and lock-free concurrent skip lists. More specifically, we use concurrent skip lists to implement ordered sets and priority queues and we find out that our fine-grained HTM solution manages to outperform other alternatives in most cases.

2 Backround

2.1 Intel's Transactional Synchronization Extensions

Intel was the first to provide HTM support in commercial processors with the introduction of Transactional Synchronization Extensions (TSX) [1] in Haswell processors and their successors. TSX is a set of assembly instructions that are used by the programmer to enclose critical sections of code which need to be executed atomically. These critical sections are executed as *transactions* whose memory reads and writes are being tracked by the underlying HTM system. The read memory locations are kept in the transaction's *read-set* and the written ones in the *write-set*. TSX can be used in two modes, namely HLE and RTM. We only use RTM because it provides the flexibility to choose what actions are taken upon a transaction's abort. RTM provides the following instructions:

- *xbegin*: Starts a transaction.
- *xend*: Commits a transaction.
- *xabort*: Explicitly aborts a transaction. A representative code is passed to the abort instruction to enable the distinction among different abort reasons.
- *xtest*: Checks whether a transaction is in progress or not.

An HTM transaction will either *commit*, in which case all its memory modifications become visible to other threads, or *abort* and none of its memory writes become visible to other threads. A transaction may abort for the following reasons:

- *Data conflict*: When another thread executing in transactional or non-transactional mode writes to a memory location that has been added to the transaction's read or write set or reads a memory location that has been added to the transaction's write set.
- *Capacity abort*: When the transaction's memory footprint has exceeded the size of the provided hardware transactional buffers.
- *Explicit abort*: When the programmer explicitly aborts the transaction using the *xabort* instruction.
- *Other*: A transaction may abort due to other reasons including interrupts, unsupported instructions, system calls, etc.

TSX is a *best-effort* HTM implementation and provides no guarantees that any transaction will eventually commit; persistent aborts may lead to livelock. It is thus the programmer's responsibility, when using RTM, to provide an alternative path of execution that uses no transactions, i.e., a *non-transactional fallback path*. The most common practice is to retry a transaction for a given amount of times and, if it fails to commit, fallback to the acquisition of a lock that allows only a single thread to enter the critical section. This is also known as *transactional lock elision (TLE)* [8].

[1] `https://en.wikipedia.org/wiki/Transactional_Synchronization_Extensions`

2.2 Skip Lists

Skip Lists [19] are an alternative to binary search trees with applications mainly in main-memory databases, like MemSQL [2]. They provide expected logarithmic time search and update operations without the need to rebalance; a major source of contention in balanced binary search trees.

A skip list is composed of levels, each of which is a singly linked list. The bottom level linked list contains all elements ordered by their key. Each higher-level list is a sublist of the lower-level lists. Each new node is assigned a top-level (or height) value so that it belongs to all lists up to that level. Top-levels are chosen so that the expected number of nodes in each level's list decreases exponentially. Every skip list is initialized with two sentinel nodes, head and tail, with the maximum allowed height and minimum and maximum keys, respectivelly. Each node keeps an array of pointers to successors, one for each list to which it belongs. The search and update algorithms use these pointers to navigate through the structure. Figure 1 shows an example of a skip list with integer keys.

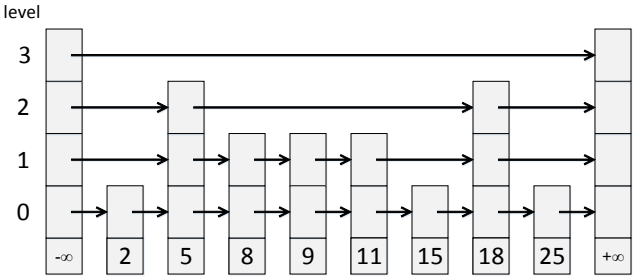

Fig. 1: A skip list with 4 levels. The number below each node is the node's key, with $-\infty$ and $+\infty$ as the keys of head and tail sentinel nodes respectively.

3 HTM-based Skip Lists

3.1 Coarse-Grained HTM Skip List

To illustrate the simplicity of using HTM for concurrent skip lists we implemented a coarse-grained version using TLE. More specifically, we enclose each operation of the skip list (except contains) inside a single HTM transaction. In our non-transactional fallback path, which ensures forward progress, a single global lock is acquired which prevents the execution of concurrent operations. We refer to this naive implementation as *cg-htm*.

[2] `https://docs.memsql.com/concepts/v6.0/indexes/`

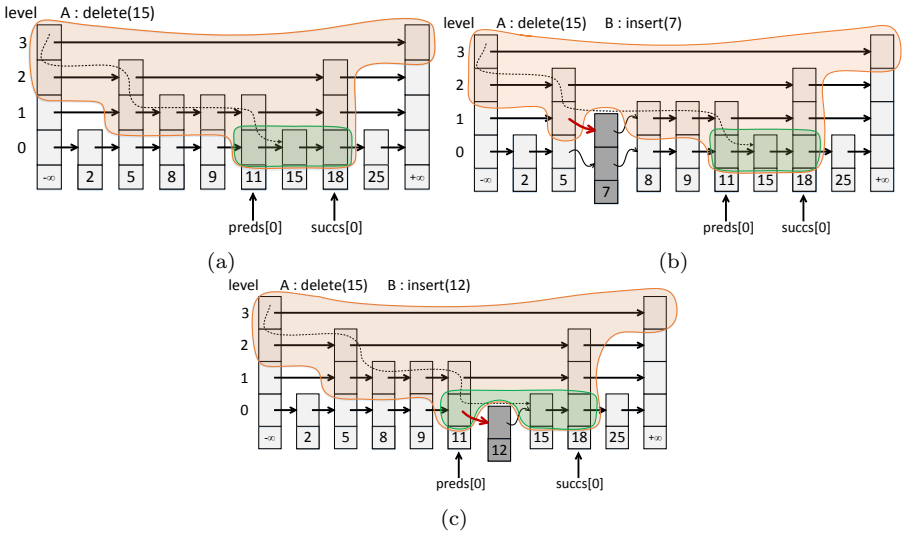

Fig. 2: The memory footprint of skip list operations.

3.2 Fine-grained HTM Skip List

On the one hand, cg-htm is an implementation that highlights the simplicity of using HTM and performs well in certain execution scenarios. On the other hand, enclosing the whole skip list operation in a single HTM transaction results in large transactional footprints that set the ground for excessive amounts of aborts that, as we show in our experimental evaluation, are drastically reduced when smaller transactions are used. In fact, transactions in cg-htm include unnecessary data in their read-set; correct execution can be guaranteed without including all these memory locations in the read-set.

Figure 2a depicts the memory footprint of a delete operation of a skip list. With orange color we mark all the nodes that are accessed and are included in the transaction of cg-htm. Green color denotes the nodes that are modified. These are actually the only nodes that need to be tracked by the HTM transaction to guarantee the consistency of the operation. We refer to this set of nodes as the *critical nodes* of the operation. Figure 2b shows an example of two operations executing in parallel. The two operations modify disjoint parts of the data structure and should be allowed to execute concurrently. However, cg-htm does not allow concurrency in this case since node with key 5 is in the read-set of the delete operation and a conflict abort is triggered. In contrast, Figure 2c shows a scenario where two operations have a real data conflict and should not execute concurrently. In this case, even if only the critical nodes are included in the transaction's read/write sets a conflict abort is triggered and correct execution is guaranteed.

To minimize the transactional footprint and enable concurrent execution of non-conflicting operations, such as those shown in Figure 2b we sacrifice

some of the simplicity of cg-htm and implement a fine-grained HTM-based skip list which we refer to as *fg-htm*. The key idea to achieve that is to avoid transactions during traversals. Extra attention has to be paid though to ensure that data collected during traversal is still valid when performing the actual update. We attained those objectives by employing consistency oblivious programming (COP) [2] technique. In COP-based implementations there are two phases: an oblivious read-only phase that executes with no synchronization followed by an atomic phase that verifies that the oblivious phase operated correctly and modifies the data structure appropriately.

Insertion: The insert operation, presented in algorithm 1, begins by traversing the skip list using pred and curr references, starting at the head, at the top level, in order to find the right position to insert the new element with the given key (lines 12-19). During this traversal, it locates the predecessors and successors of the new element. If during the traversal a node was found with a matching key, it returns 0 (line 25), otherwise it proceeds by creating a new node with the matching key and a random height (lines 26,27).

We then call `tx_start` (line 28) to begin a transaction which consists of two phases:

1) A validation phase (lines 29–33) that checks two things: 1) if the predecessors and successors are still in the set and not deleted, and, 2) if the predecessors still point to the successors.
1) An update phase (lines 34–38) that performs the necessary modifications in the skip list structure. More specifically, we change the pointers of the new node to point to the successors of the node and then the next pointers of all its predecessors to point to that node.

Since the validation and the update phases are performed inside a single HTM transaction everything that has been read during the validation phase is maintained inside the read set of the transaction. The HTM system guarantees that if any other thread alters anything that was checked earlier, then a data conflict will arise and the transaction will abort.

In case the validation is unsuccessful it means that some other thread made changes to the skip list and the predecessors and/or successors arrays that were composed during traversal are now obsolete. In this case the operation needs to restart. Before restarting we delete the new node and then terminate the transaction (lines 31-33). When the number of operation retries exceeds a predefined number (`MAX_OPERATION_RETRIES`) the global lock is acquired and the sequential version of the insert operation is executed (lines 7–11).

In case a non explicit abort is encountered we do not restart the whole operation again but instead we transactionaly retry the validation and update phases only. If the total number of transactional retries exceeds a predifined number of times (`MAX_TX_RETRIES`) we retry once more but this time with the global lock acquired.

Deletion: Similarly to insertion, the *delete* operation, presented in Algorithm 2, traverses the skip list keeping track of only the predecessors this time (lines

Algorithm 1: Fine-Grained HTM - Insert operation

```
1  int fg_htm_insert(int key)
2      int h, ret, status, nretries = -1;
3      Node preds[MAX_LEVEL], succs[MAX_LEVEL];
4      Node curr, pred;
5      START_FROM_SCRATCH_INS:
6      nretries++;
7      if (nretries ⩾ MAX_OPERATION_RETRIES) then
8          acquire_lock(global_lock);
9          ret = sequential_insert(key);
10         release_lock(global_lock);
11         return ret;
12     pred = head;
13     for (h = MAX_LEVEL-1; h ⩾ 0; h–) do
14         curr = pred.next[h];
15         while (key >curr.key) do
16             pred = curr;
17             curr = pred.next[h];
18         preds[h] = pred;
19         succs[h] = curr;
20     if (key == curr.key) then
21         while (curr.state == INITIAL) do
22             ;
23         if (curr.state == DELETED) then
24             goto START_FROM_SCRATCH_INS;
25         return 0;
26     int nodeHeight = get_rand_level();
27     Node newNode = createNewNode(key, nodeHeight);
       /* begin transaction                                          */
28     status = tx_start(MAX_TX_RETRIES, global_lock);
       /* check consistency                                          */
29     for (h = 0; h <nodeHeight; h++) do
30         if (preds[h].next[h] != succs[h] || preds[h].state == DELETED ||
           succs[h].state == DELETED) then
31             deleteNode(newNode);
32             tx_end(global_lock);
33             goto START_FROM_SCRATCH_INS;
       /* update fields                                              */
34     for (h = 0; h <nodeHeight; h++ ) do
35         newNode.next[h] = succs[h];
36     for (h = 0; h <nodeHeight; h++ ) do
37         preds[h].next[h] = newNode;
38     newNode.state = INSERTED;
39     tx_end(global_lock);
40     return 1;
```

Algorithm 2: Fine-Grained HTM - Delete operation

```
1  int fg_htm_delete(int key)
2      int h, nodeHeight, status, ret, nretries = -1;
3      Node preds[LEVEL_MAX];
4      Node curr, pred;
5      START_FROM_SCRATCH_DEL:
6      nretries++;
7      if (nretries ⩾ MAX_OPERATION_RETRIES) then
8          acquire_lock (global_lock);
9          ret = sequential_remove(key);
10         release_lock(global_lock);
11         return ret;
12     pred = head;
13     for ( h = LEVEL_MAX-1; h ⩾ 0; h–) do
14         curr = pred.next[h];
15         while (key >curr.key) do
16             pred=curr;
17             curr = pred.next[h];
18         preds[h] = pred;
19     if (curr.key != key) then
20         return 0;
       /* begin transaction                                          */
21     status=tx_start(MAX_TX_RETRIES, global_lock);
22     nodeHeight=curr.height;
       /* check consistency                                          */
23     for (h = 0; h <nodeHeight; h++) do
24         if (preds[h].next[h] != curr || preds[h].state == DELETED ) then
25             tx_end(global_lock);
26             goto START_FROM_SCRATCH_DEL;
       /* update fields                                              */
27     curr.state = DELETED;
28     for (h = nodeHeight-1; h ⩽0; h - -) do
29         preds[h].next[h] = curr.next[h];
30     tx_end(global_lock);
31     return 1;
```

12–18). If the key is not found in the skip list the operations returns 0 (lines 19,20). Otherwise, we start a transaction (line 21) in which we check 1) that the predecessors still point to the same node, and, 2) that they have not been removed from the skip list. If the validation fails, the same procedure as the one we described for insertion is followed. If the validation succeeds, we first mark the node as deleted (line 27) and then update the appropriate pointers, i.e., the predecessors' next fields. If at any point the transaction aborts, it behaves exactly like we described for insertion.

Contains: The contains operation, presented in algorithm 3 traverses the skip list in order to find a node with a matching key, similarly to insertion and

Algorithm 3: Contains

```
1  boolean contains(int key)
2      boolean result=false;
3      Node curr, pred;
4      pred= head;
5      for ( int h = LEVEL_MAX-1; h ⩾ 0; h - -) do
6          curr = pred.next[h];
7          while (key >curr.key) do
8              pred=curr;
9              curr = pred.next[h];
10     if (key == curr.key) then
11         while (curr.state == INITIAL) do
12             ;
13         if (curr.state != DELETED) then
14             result=true;
15     return result;
```

deletion. If a node with a matching key is found, we first wait for the element's status code to be set to INSERTED and then, if the element is still part of the skip list, it returns true. In all other cases it returns false.

3.3 Correctness

In this section we provide an informal proof of correctness of our fg-htm implementation by providing the linearization points of the three operations, insert, delete and contains.

Linearization point of successful insertions and deletions: The linearization point of a successful insert and delete operation in transactional mode is the step where the thread commits its transaction after completing the operation (line 39 for insert / line 30 for delete). If the thread executes the operation while holding the global lock, the linearization point for deletion is the step where the state of the node is changed from INSERTED to DELETED in line 27 and for insertion is the step where the state of the node is changed from INITIAL to INSERTED in line 38.

Linearization point of unsuccessful insertions and deletions: In order to properly define the linearization point of an unsuccessful insertion we have to be certain that the point in which the algorithm claims that an element with a matching key exists in the set (line 25) happens after the linearization point of that element's successful insertion. To guarantee that, we use the same method as in the contains algorithm and spin (line 21) waiting for the element's status to become INSERTED. There is always a point between checking if the node is deleted (line 23) and returning the value 0 in line 25 that the node was part of the set and we define that point to be the linearization point of an unsuccessful insertion.

Regarding deletions, if the traversal did not find an element with a matching key, there is always a point between finishing traversal and checking if `curr.key!=key` (lines 18–19) where there is no element with a matching key in the skip list, thus this point is the linearization point of an unsuccessful deletion.

Linearization point of contains: The linearization point of a successful contains, occurs when the element with the matching key is traversed, having been observed to be unmarked and fully linked, similarly to an unsuccessful insert operation. The linearization point of an unsuccessful contains occurs when the method does not find a node with a matching key or finds one that is marked.

4 Evaluation

For our experimental evaluation we used a dual socket server equipped with two Intel Haswell-EP E5-2697 v3 processors. Each processor is clocked at 2.6GHz and provides 14 physical cores and 28 hardware threads when hyperthreading is enabled. Each core has 32KB L1 cache and 256KB L2 and all 14 cores of each processor share a 35MB L3 cache. The server has 64GB of RAM and runs Debian 8.1 with linux kernel version 4.0.4. For the compilation of our executables we used GCC 4.9.2 with -O3 optimizations enabled.

Below are the details of our experiments:

- Each run lasts 1 second during which each thread performs randomly chosen operations. All reported results are the average of 6 independent executions with negligible variance.
- Each software thread is pinned to a hardware thread.
- Skip list nodes are padded and aligned to occupy exactly three cache lines.
- No implementation uses memory reclamation.
- The key range is set to 10^9 and the skip list is initialized by one thread before the start of the measurements with 10^6, 10^5 or 10^3 elements.
- Update operations are always 50% insertions, 50% deletions, thereby preserving the size of the underlying data structure. Insertions choose keys randomly from the key range and deletions attempt to delete the key last inserted by the same thread. Contains operations search for the last key inserted by the same thread, if the last operation was insertion, or a random key if the last operation was deletion.
- The number of transactional retries (`MAX_TX_RETRIES`) is set to 30 and the number of operation retries (`MAX_OPERATION_RETRIES`) to 15.

4.1 Coarse-grained vs Fine-grained HTM

As already mentioned, the coarse-grained HTM-based skip list showcases how HTM facilitates the implementation of complex concurrent data structures. However, as shown in figure 2 enclosing whole skip list operations in single

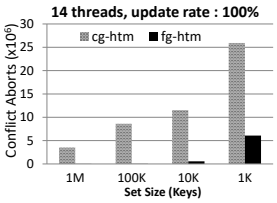

Fig. 3: Number of conflict aborts for varying skip list sizes.

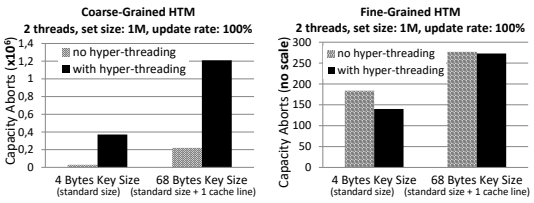

Fig. 4: Number of capacity aborts for varying key sizes, with and without hyperthreading.

HTM transactions results in footprints that contain far more elements than those needed to guarantee consistency. As a result, a large number of unnecessary conflict aborts appear. Furthermore, the larger duration of the transactions lengthens the window of contention and increases the likelihood of conflict aborts. Finally, conflict aborts are also encountered when threads execute the non transactional fallback path. When the fallback path's lock is acquired all the running transactions encounter a conflict abort on the lock's shared variable. Regarding capacity aborts, the large memory footprint of skip list operations might, in many cases, exceed the size of the available TSX buffers resulting in a large number of capacity aborts.

The above limitations of cg-htm has led us to sacrifice some of HTM's simplicity and explore how we can, with more programming effort, implement fg-htm. fg-htm minimizes the amount of data read and/or written inside transactions, effectively minimizing the size of these transactions.

In Figures 3 and 4 we compare cg-htm with fg-htm in terms of transactional aborts. Figure 3 presents the number of conflict aborts encountered in each of the two implementations for different skip list sizes, i.e., different contention levels. The number of threads is fixed to 14 (i.e., we occupy one socket of the server without hyperthreading enabled) and the update rate is set to 100% meaning that all operations modify the skip list. As is evident, cg-htm suffers far more conflict aborts than fg-htm with a maximum difference of 20 million aborts for skip lists with 1K keys.

In Figure 4 we present the capacity aborts of our two HTM-based skip lists for 1M keys where we vary the size of the keys stored in the set. We present results with and without hyperthreading to gain a better insight on how the sharing of the TSX hardware resources among two hyperthreads affects each of the two implementations. The coarse-grained HTM implementation is negatively affected by increasing the key size and when hyperthreading is enabled. More specifically, with 68 bytes keys we observe 10 times more capacity aborts and when we enable hyperthreading 5 times more.

On the other hand, fg-htm is much less sensitive to the key size and hyperthreading. This is due to its already minimal transaction size which even in the case of the 68 bytes key is far less than the TSX buffers' size. It is also worth noting the difference in the number of capacity aborts between the two

implementations. fg-htm suffers much less capacity aborts (about 185) than cg-htm (about 20000).

The above results reveal the disadvantages of HTM algorithms with large transactional footprints but also the advantages of more complex fine-grained HTM solutions. The fact that the majority of memory accesses involving keys take place in the traversal phase and that the critical sections require small transactional footprints, make the fg-HTM implementation insensitive to key sizes and hyper-threading.

4.2 Skip list-based ordered sets

One of the most common uses of skip lists is for implementing ordered sets. We compare our algorithms against the state-of-the-art implementations that use lock-based and lock-free synchronizations mechanisms. Our comparison includes the following five skip-list-based set implementations:

- **seq:** A sequential skip list. This implementation uses no synchronization and does not produce valid results. However, we use it for throughput comparison.
- **lb:** The lock-based skip list by Herlihy et al. [12].
- **lf:** The lock-free skip list by Frazer with Herlihy's optimization [21].
- **fg-htm:** Our fine-grained HTM skip list.
- **cg-htm:** Our coarse-grained HTM skip list.

All the above implementations are written in C and our source code is freely available on our github repository [3]. The non-HTM ones are derived from the ASCYLIB library [4].

Figure 5 presents the results we obtained by measuring the throughput and the transactional statistics [5] for three different workloads:

- **Low contention:** 1M keys, 5% update rate.
- **Medium contention:** 100K keys, 50% update rate.
- **High contention:** 1K keys, 100% update rate.

First, we observe that in low contention executions (1M keys, 5% update rate, <42 threads) cg-htm performs equally to the rest implementations. As the contention grows, though, the increasing amount of aborts results in performance degradation. On the other hand, fg-htm manages to provide high throughput for all contention levels, thanks to its low abort ratio.

If we compare our fine-grained htm-based implementation with the lock-free skip list we observe a relatively constant interval between their throughput for medium and high contention levels. This is caused by the overhead of the lock-free version having to check and clean marked nodes during traversals. On the contrary, the non-blocking properties of the lock-free implementation

[3] `https://github.com/mkardaras/HTM_Skip-Lists.git`

[4] `https://github.com/LPD-EPFL/ASCYLIB`

[5] Terminating a transaction inside the validation phase counts as an abort.

enables it to scale better in high contention workloads (1K keys, 100% update rate, >28 threads). The lock-based implementation has similar performance with the fine-grained HTM version in the first two workloads and a bit worse in the third.

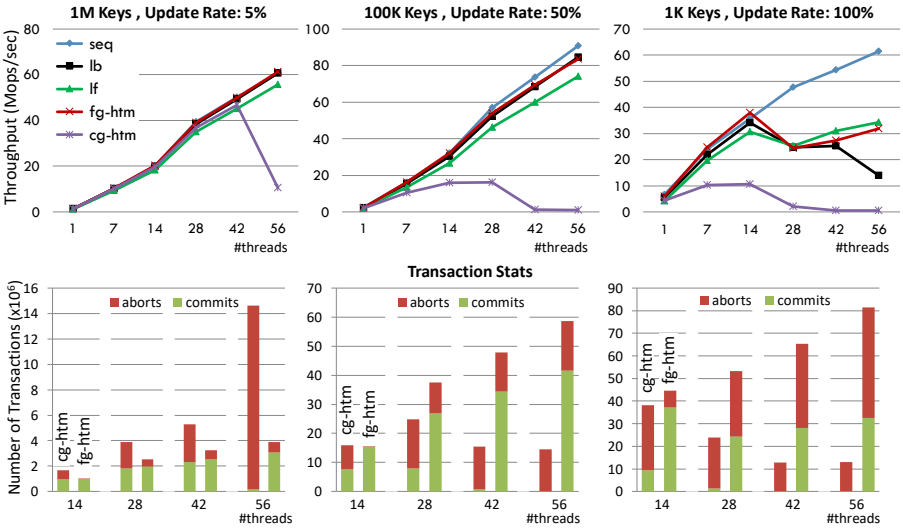

Fig. 5: Throughput and transactional statistics of skip-list based ordered set implementations.

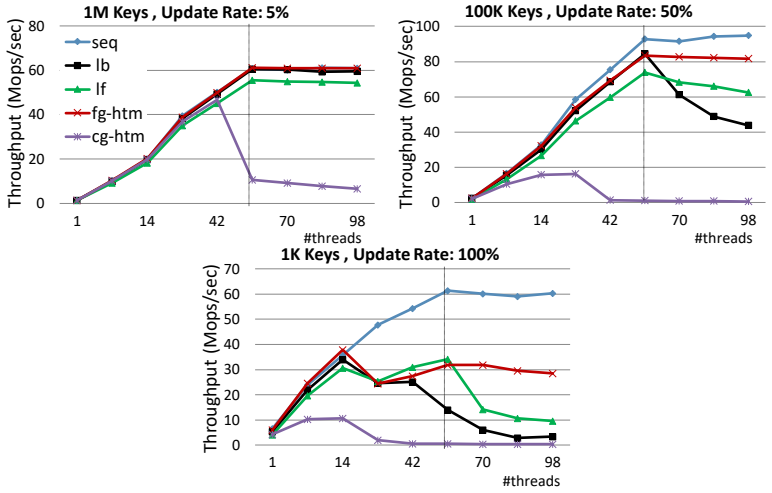

Fig. 6: Throughput of skip list-based ordered set implementations within over-subscription scenarios. Software threads are pinned to 56 hardware threads.

The big advantage of fg-htm over the lock-based arises when the threads start to delay for some reason. Theoretically, both algorithms acquire locks, therefore they are both blocking. The difference is that with HTM the probability of acquiring the fallback lock, even under high contention, is very small, thus the HTM algorithm is practically non-blocking most of the time. The big problem with blocking algorithms comes up when some threads that hold locks need to stop their execution for a period of time while other threads are waiting for the locks to be released. This is a very common scenario in practice, as threads can be descheduled, wait for data to arrive from the disk or from some input, etc.

To test the implementations in a scenario where the threads have to delay their execution for a period of time, we ran experiments were we pinned up to 98 software threads to 56 hardware threads. This results to oversubscription and certain threads being scheduled out regularly. In Figure 6 we can see the results of our experiments.

It is evident that the lock-based implementation is affected the most as it uses a large number of locks per update operation, thus increasing the potential of a thread being scheduled out while holding a lock. On the contrary, fg-htm addresses the issue and maintains a stable throughput in all oversubscription scenarios.

4.3 Skip list-based Priority Queues

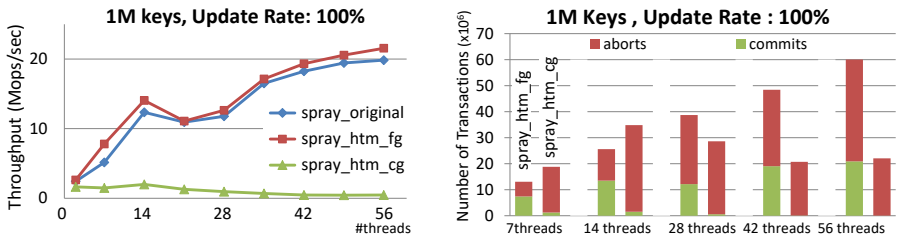

Fig. 7: Measuring throughput and transactional statistics for SprayLists.

Another very common use of skip lists is for implementing priority queues. Many well known algorithms and applications like task managing in operating systems (e.g. load balancing, interrupt handling), graph algorithms (e.g. Dijkstra, Prim), compression algorithms(e.g. Huffman coding), need to extract data according to a specific order. They achieve that by using priority queues. Priority queues support two operations: 1) *insert*, which inserts an element with a given priority, and, 2) *deleteMin*, which returns the element with the minimum key (i.e., highest priority).

In the standard skip list-based concurrent priority queues [6, 25, 14, 22, 16] the threads compete with each other over the first element or over the p first

elements, where p is the number of running threads. It is pointless to use HTM to implement skip list-based concurrent priority queues as repetitious conflict aborts become inevitable due to the threads acting in neighbouring nodes.

The introduction, however, of relaxed priority queues gives a footing for HTM ideas. A relaxed priority queue doesn't necessarily delete the minimum element but one of the first x elements, where x is a relatively small number. Contention-wise, this is similar to performing DeleteKey operations in a data set with x elements. As we saw in the previous subsection, the fine-grained HTM implementation performed well in small data sets, something that gives us a hint for implementing relaxed priority queues with HTM. Recently, there has been intensive work on relaxed priority queues [3, 29, 20] and a surge of interest both from academia [11] and practice [17].

In this paper we focus on **SprayList** [3], a relaxed priority queue whose DeleteMin operation returns an element among the first $O(p \log^3 p)$ in the list, with high probability, where p is the number of threads. In order to do so, it initially calls the operation *spray* that performs a carefully designed random walk that finishes in one of the first $O(p \log^3 p)$ elements. Afterwards, the standard *DeleteKey* operation is called which removes the element selected by the spray operation.

We implemented a coarse-grained and a fine-grained HTM version of SprayList, where the *spray* operations are the same as the one in the original lock-free implementation but without the cleaning and the *DeleteKey* operations are identical to the ones we described on section 3.

We measured throughput of each implementation and the results are shown in Figure 7. At first, as we would expect, with abort percentages exceeding 90%, coarse-grained HTM is completely unsuitable for implementing a SprayList. On the contrary, fine-grained HTM takes advantage of the distribution caused by spray and limits the aborts per update operation ratio to numbers less than 2,2. This is imprinted in throughput results, with the fg HTM version achieving slightly better performance than the original lock-free implementation.

5 Related Work

TM has been extensively used for the implementation of concurrent data structure such as linked lists [26], hash tables [15] and search trees [4, 28, 23, 24]. We mostly focus on related work that involves concurrent TM-based Skip Lists.

Zhaoguo Wang et al. [27] have applied Intel's RTM in order to implement concurrent Skip Lists. In their algorithms they do not consider separate delete operations, instead they treat a delete as a special insertion operation which inserts a key with an empty value. They test coarse-grained and fine-grained HTM versions as well. In the fg version the insertion starts by traversing the set without transactions, locating the predecessors. Afterwards, they begin a transaction and instead of doing validations they keep traversing at each level in order to find the correct preds and succs pairs and finish by executing the

update and commit the transaction. The contains operation is executed inside a transaction.

Pirkelbauer et al. [18] offer an interpretation of the epoch-based memory reclamation technique for HTM systems. In order to test memory managers, they implemented an HTM-based Skip List where traversals both in update and find operations are executed inside dynamic sized transactions. The execution of the critical sections is similar to the fine-grained locking but instead of locking they use single transactions. Both Zhaoguo Wang et. al and Pirkelbauer et. al provide fallback paths where a global fallback lock is acquired after multiple transactional retries.

David and Guerraoui [7] also study the problem of oversubscription in their paper. They do propose HTM as a solution to the problem as well and among other data structures they also tested HTM Skip Lists. What they did was to add Intel's TSX instructions to the acquire and release methods of the locks in the standard lock-based implementations. The fallback path in this case uses the actual locks and no global lock is required. Their evaluation revealed that the use of TSX increased the performance of Skip Lists up to 53.28 times for workloads with high contention.

Software Transactional Memory has also been used to parallelize Skip Lists. Dragojević and Harris [9] present a specialized STM system (SpecTM) that allows the programmer to use special short transactions for small updates like inserting or deleting a node with level one and traditional transactions for bigger updates.

Avni et al. [5] introduced a new concurrent data-structure called Leap-List. Leap-List is a Skip List where each node holds up-to K immutable key-value pairs. For the implementation they use Software Transactional Memory and a variation of COP with Locking Transactions (LT). In this variation the read-only part is checking for locks, and retries. Then the transaction atomically verifies validity and locks the written addresses. After the transaction commits, a postfix of the operation writes the data to the locked locations and releases them.

6 Conclusions and Future Work

In this work we explored the efficacy of HTM for implementing concurrent skip lists. In our first approach we used HTM in the most straightforward way, i.e., we enclosed each operation (except contains) in a single HTM transaction. Although simple, this naive implementation failed to provide high performance due to its large transactions. To minimize the transaction size and unlock performance we devised a fine-grained HTM based skip list which surpasses the coarse-grained HTM's limitations and manages to provide performance competitive or even better than state-of-the-art lock-based and lock-free concurrent skip lists.

As part of our future work we plan to extend our analysis on more data structures and explore how HTM can be used to facilitate the implementation of more complex concurrent data structures. Moreover, as in this work we did not deal with memory reclamation of the removed nodes, we intend to do

so in our next steps. More specifically, we would like to explore how memory reclamation can be combined with HTM-based algorithms and its performance impact.

References

1. AdelsonVelskii, M., Landis, E.M.: An algorithm for the organization of information. Tech. rep., DTIC Document (1963)
2. Afek, Y., Avni, H., Shavit, N.: Towards consistency oblivious programming. In: Principles of Distributed Systems - 15th International Conference, OPODIS 2011, Toulouse, France, December 13-16, 2011. Proceedings, pp. 65–79 (2011). DOI 10.1007/978-3-642-25873-2_6. URL https://doi.org/10.1007/978-3-642-25873-2_6
3. Alistarh, D., Kopinsky, J., Li, J., Shavit, N.: The spraylist: A scalable relaxed priority queue. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, pp. 11–20. ACM, New York, NY, USA (2015). DOI 10.1145/2688500.2688523. URL http://doi.acm.org/10.1145/2688500.2688523
4. Avni, H., Kuszmaul, B.C.: Improving htm scaling with consistency-oblivious programming. TRANSACT **6**, 5 (2014)
5. Avni, H., Shavit, N., Suissa, A.: Leaplist: Lessons learned in designing tm-supported range queries. In: Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing, PODC '13, pp. 299–308. ACM, New York, NY, USA (2013). DOI 10.1145/2484239.2484254. URL http://doi.acm.org/10.1145/2484239.2484254
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press (2009)
7. David, T., Guerraoui, R.: Concurrent search data structures can be blocking and practically wait-free. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '16, pp. 337–348. ACM, New York, NY, USA (2016). DOI 10.1145/2935764.2935774. URL http://doi.acm.org/10.1145/2935764.2935774
8. Dice, D., Lev, Y., Moir, M., Nussbaum, D., Olszewski, M.: Early experience with a commercial hardware transactional memory implementation. Tech. rep., Mountain View, CA, USA (2009)
9. Dragojević, A., Harris, T.: Stm in the small: Trading generality for performance in software transactional memory. In: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, pp. 1–14. ACM, New York, NY, USA (2012). DOI 10.1145/2168836.2168838. URL http://doi.acm.org/10.1145/2168836.2168838
10. Guibas, L.J., Sedgewick, R.: A dichromatic framework for balanced trees. In: Foundations of Computer Science, 1978., 19th Annual Symposium on, pp. 8–21 (1978). DOI 10.1109/SFCS.1978.3
11. Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, pp. 317–328. ACM, New York, NY, USA (2013). DOI 10.1145/2429069.2429109. URL http://doi.acm.org/10.1145/2429069.2429109
12. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: Proceedings of the 14th International Conference on Structural Information and Communication Complexity, SIROCCO'07, pp. 124–138. Springer-Verlag, Berlin, Heidelberg (2007). URL http://dl.acm.org/citation.cfm?id=1760631.1760646
13. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, ISCA '93, pp. 289–300. ACM, New York, NY, USA (1993). DOI 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164
14. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming, Revised Reprint, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2012)
15. Li, X., Andersen, D.G., Kaminsky, M., Freedman, M.J.: Algorithmic improvements for fast concurrent cuckoo hashing. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, pp. 27:1–27:14. ACM, New York, NY, USA (2014). DOI 10.1145/2592798.2592820. URL http://doi.acm.org/10.1145/2592798.2592820

16. Lindén, J., Jonsson, B.: A skiplist-based concurrent priority queue with minimal memory contention. In: Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304, OPODIS 2013, pp. 206–220. Springer-Verlag New York, Inc., New York, NY, USA (2013). DOI 10.1007/978-3-319-03850-6_15. URL http://dx.doi.org/10.1007/978-3-319-03850-6_15
17. Nguyen, D., Lenharth, A., Pingali, K.: A lightweight infrastructure for graph analytics. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pp. 456–471. ACM, New York, NY, USA (2013). DOI 10.1145/2517349.2522739. URL http://doi.acm.org/10.1145/2517349.2522739
18. Pirkelbauer, P., Wilson, A., Ahmed, H.: Memory management for concurrent data structures on hardware transactional memory. In: 12th ACM SIGPLAN Workshop on Transactional Memory (TRANSACT), TRANSACT 2017 (2017)
19. Pugh, W.: Skip lists: A probabilistic alternative to balanced trees. Commun. ACM **33**(6), 668–676 (1990). DOI 10.1145/78973.78977. URL http://doi.acm.org/10.1145/78973.78977
20. Rihani, H., Sanders, P., Dementiev, R.: Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. CoRR **abs/1411.1209** (2014). URL http://arxiv.org/abs/1411.1209
21. Shavit, N., Lev, Y., Herlihy, M.: Concurrent lock-free skiplist with wait-free contains operator (2011). URL https://www.google.gr/patents/US7937378. US Patent 7,937,378
22. Shavit, N., Lotan, I.: Skiplist-based concurrent priority queues. In: Proceedings of the 14th International Parallel & Distributed Processing Symposium (IPDPS'00), Cancun, Mexico, May 1-5, 2000, pp. 263–268 (2000). DOI 10.1109/IPDPS.2000.845994. URL https://doi.org/10.1109/IPDPS.2000.845994
23. Siakavaras, D., Nikas, K., Goumas, G., Koziris, N.: Performance analysis of concurrent red-black trees on htm platforms. TRANSACT (2015)
24. Siakavaras, D., Nikas, K., Goumas, G., Koziris, N.: Rcu-htm: Combining rcu with htm to implement highly efficient concurrent binary search trees. In: 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 1–13 (2017). DOI 10.1109/PACT.2017.17
25. Sundell, H., Tsigas, P.: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. **65**(5), 609–627 (2005). DOI 10.1016/j.jpdc.2004.12.005. URL http://dx.doi.org/10.1016/j.jpdc.2004.12.005
26. Timnat, S., Herlihy, M., Petrank, E.: A practical transactional memory interface. In: Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings, pp. 387–401 (2015). DOI 10.1007/978-3-662-48096-0_30. URL https://doi.org/10.1007/978-3-662-48096-0_30
27. Wang, Z., Qian, H., Chen, H., Li, J.: Opportunities and pitfalls of multi-core scaling using hardware transaction memory. In: Proceedings of the 4th Asia-Pacific Workshop on Systems, APSys '13, pp. 3:1–3:7. ACM, New York, NY, USA (2013). DOI 10.1145/2500727.2500745. URL http://doi.acm.org/10.1145/2500727.2500745
28. Wang, Z., Qian, H., Li, J., Chen, H.: Using restricted transactional memory to build a scalable in-memory database. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14, pp. 26:1–26:15. ACM, New York, NY, USA (2014). DOI 10.1145/2592798.2592815. URL http://doi.acm.org/10.1145/2592798.2592815
29. Wimmer, M., Versaci, F., Träff, J.L., Cederman, D., Tsigas, P.: Data structures for task-based priority scheduling. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, pp. 379–380. ACM, New York, NY, USA (2014). DOI 10.1145/2555243.2555278. URL http://doi.acm.org/10.1145/2555243.2555278
30. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance evaluation of intel transactional synchronization extensions for high-performance computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pp. 19:1–19:11. ACM, New York, NY, USA (2013). DOI 10.1145/2503210.2503232. URL http://doi.acm.org/10.1145/2503210.2503232