A Practical Transactional Memory Interface

Shahar Timnat[1](✉), Maurice Herlihy[2], and Erez Petrank[1]

[1] Computer Science Department, Technion, Haifa, Israel
stimnat@cs.technion.ac.il
[2] Computer Science Department, Brown University, Providence, USA

Abstract. Hardware transactional memory (HTM) is becoming widely available on modern platforms. However, software using HTM requires at least two carefully-coordinated code paths: one for transactions, and at least one for when transactions either fail, or are not supported at all. We present the MCMS interface that allows a simple design of fast concurrent data structures. MCMS-based code can execute fast when HTM support is provided, but it also executes well on platforms that do not support HTM, and it handles transaction failures as well. To demonstrate the advantage of such an abstraction, we designed MCMS-based linked-list and tree algorithms. The list algorithm outperforms all known lock-free linked-lists by a factor of up to X2.15. The tree algorithm builds on Ellen et al. [7] and outperforms it by a factor of up to X1.37. Both algorithms are considerably simpler than their lock-free counterparts.

1 Introduction

Transactional memory (TM) is becoming an increasingly central concept in parallel programming. Recently, Intel introduced the TSX extensions to the x86 architecture, which include RTM: an off-the-shelf hardware that supports hardware transactional memory. There are practical reasons for a developer to avoid using hardware transactional memory. First, HTM is only available for some of the computers in the market. Thus, a code that relies on HTM only suits a fraction of the available computers and must be accompanied with a different code base for the other platforms. Second, RTM transactions are "best effort" and are not guaranteed to succeed. Thus, to work with HTM, a *fall-back* path must also be provided and maintained, in case transactions repeatedly fail.

We propose a new programming discipline for highly-concurrent linearizable objects that takes advantage of HTM when it is available, and still performs reasonably (around X0.6) when it is not available. For this purpose, we suggest to encapsulate the HTM inside an intermediate level operation. The intermediate operation is compiled to an HTM implementation on platforms that support HTM, and to a non-transactional implementation otherwise. To a certain extent, our intermediate operation can even be implemented with an "out of the box" fall-back path for failing transactions. This fall-back path can be made lock-free, thus rendering our operation a valid alternative for designing lock-free operations.

© Springer-Verlag Berlin Heidelberg 2015
J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 387–401, 2015.
DOI: 10.1007/978-3-662-48096-0_30

The intermediate operation we find best suited for this purpose is a slight variation of the well-known MCAS (Multi-word Compare And Swap) operation. The MCAS operation executes atomically on several shared memory addresses. Each address is associated with an *expected-value* and a *new-value*. An execution of MCAS succeeds and returns true iff the content of each specified address equals its expected value. In this case, the data in each address is replaced with the new value. If any of the specified addresses contains data that is different from the expected value, then false is returned and the content of the shared memory remains unchanged.

We propose an extended interface of MCAS called MCMS (Multiple Compare Multiple Swap), in which we also allow addresses to be compared without being swapped. The extension is functionally redundant, because, in effect, comparing an address without swapping it is identical to an MCAS in which this address' expected value equals its new value. However, when implementing the MCMS using transactional memory, it is ill-advised to write a new (identical) value to replace an old one. Such a replacement may cause unnecessary transaction aborts.

In order to study the usability of the MCMS operation, we designed two algorithms that use it. One for the linked-list data structure, and one for the binary search tree. The MCMS tree is almost a straightforward MCMS-based version of the lock-free binary search tree by Ellen et al. [7]. But interestingly, attempting to design a linked-list that exploits the MCMS operation yielded a new algorithm that is highly efficient. The main idea is to mark a deleted node in a different and useful manner. Instead of using a mark on the reference (like Harris [9]), or using a mark on the reference and additionally a backlink (like Fomitchev and Ruppert [8]), or using a separate mark field (like the lazy linked-list [11]), we mark a node deleted by setting its pointer to be a back-link, referencing the previous node in the list. This approach works excellently with transactions.[1]

We present three simple fall-back alternatives to enable progress in case RTM executions of MCMS repeatedly fail. The simplest way is to use locks, in a similar manner to *lock-elision* [14]. The second approach is to use CAS-based MCMS [10] as a fall-back. The third alternative is a copying scheme, where a new copy of the data structure is created upon demand to guarantee progress. Both the linked-list and tree algorithm outperform their lock-free alternatives when using either a lock-based fall-back path or a copying fall-back path. The list algorithm performs up to X2.15 faster than Harris's linked-list, and the tree algorithm performs up to X1.37 faster than the tree of Ellen et al. A fall-back path (that does not use transactions) is at times a bit faster (up to X1.1) and at times a bit slower than the lock-free alternatives, depending on the specific benchmark and configuration.

Another important advantage of programming with MCMS is that the resulting algorithms are considerably simpler to design and debug compared to

[1] This approach can also be used with locks. In fact, a lock-based version of this new algorithm outperforms all known linked-list implementations. However, the design of effective lock-based linked-lists is beyond the scope of this paper.

standard lock-free algorithms that build on the CAS operation. The stronger MCMS operation allows lock-free algorithms to be designed without requiring complicated "helping" mechanisms that facilitate lock-freedom.

2 Related Work

The search of means for simplifying the design of highly concurrent data structures, and in particular lock-free ones, has been long and it led to several important techniques and concepts. Transactional memory [12,16] is arguably the most general of these; a transaction can pack any arbitrary operation to be executed atomically. But the high efficacy comes with a cost. State of the art software implementations of transactional memory incur a high performance cost, while hardware support only spans across few platforms, and usually only provides "best-effort" progress guarantee (e.g., the widely available Haswell RTM).

MCAS [13] is another tool for simplifying the design of concurrent data structures. It may be viewed as a special case of a transaction. Several CAS-based software implementations of MCAS exist [10,17] with reasonable performance. A similar, yet more restrictive primitive is the recent LLX/SCX [3]. These primitives enable to atomically read several words, but write only a single word. Atomically with the single write, it also allows to *finalize* other words, which has the effect of blocking their value from ever changing again. A CAS-based software implementation of these primitives is more efficient than any available implementation of MCAS, and these primitives have been shown to be particularly useful for designing trees [4]. Yet, allowing only a single word to be written atomically can be too restrictive: our MCMS linked-list algorithm, which atomically modifies two different pointers, cannot be easily implemented this way.

Dragojevic and Harris explored another form of restricted transactions in [6]. They showed that by moving much of the "book keeping" responsibility to the user, and keeping transactions very small, almost all of the overhead of software transactional memory can be avoided. Using their restricted transactions is more complicated than using MCAS, and they did not explore hardware transactional memory.

Speculative lock elision [14] is a technique to replace a mutual exclusion lock with speculative execution (i.e., transaction). This way several threads may execute the critical section concurrently. If a read/write or a write/write collision occurs, the speculative execution is aborted and a lock is taken. [1] studies the interaction between transactions and locks and identifies several pitfalls. Locks that are well suited to work with transactions are proposed in [15]. Intel's TSX extension also includes support of Hardware Lock Elision (HLE). Our MCMS interface lends itself to lock-elision, and also has the potential to use other fallback paths, which could be lock-free.

3 The MCMS Operation

In this section we specify the MCMS interface, its semantics and implementation. The semantics of the MCMS interface are depicted in Fig. 1(left). The MCMS

operation receives three parameters as input. The first parameter is an array of CAS descriptors to be executed atomically, where each CAS descriptor has an `address`, an `expected value`, and a `new value`. The second parameter, N, is the length of the array, and the last parameter C signifies the number of entries at the beginning of the array that should only be compared (but not swapped). We use a convention that the addresses that should only be compared and not swapped are placed at the beginning of the array. Their associated `new value` field is ignored.

3.1 Implementing MCMS with Hardware Transactional Memory

Intel Haswell Restricted Transactional Memory (RTM) introduces three new instructions: XBEGIN, XEND, XABORT. XBEGIN starts a transaction and receives a code location to which execution should branch in case of a transaction abort. XEND announces the end of a transaction, and XABORT forces an abort.

The implementation of MCMS, given in Fig. 1(right), is mostly straightforward. First, begin a transaction. Then check to see that all the addresses contain their expected value. If not, complete the transaction and return false. If all addresses hold the expected value, then write the new values, complete the transaction and return true. If the transaction aborts, restart from the beginning. However, before restarting, read all the addresses outside a transaction, and compare them to the expected value. If one of them has a value different than the expected value, return false.

This last phase of comparing after an abort is not mandatory, but has two advantages. The first is that in case the transaction failed because another thread wrote to one of the MCMS addresses, then it is possible for the MCMS to simply fail without requiring an additional transaction. The second advantage is that it handles a problem with page faults under RTM. A page fault causes a transaction to abort (without bringing the page). In such a case, simply retrying the transaction repeatedly can be futile, as the transaction will repeatedly fail without loading the page from the disk. Loading the addresses between transactions renders the possibility of repeated failures due to page faults virtually impossible.

3.2 Implementing MCMS Without TM Support

We also implemented the MCMS operation using the method of Harris et al. [10], including some optimizations suggested in that paper. As Harris's algorithm refers to MCAS, and not MCMS, we used identical expected value and new value for addresses that are only meant for comparison.

To execute an MCAS using Harris's algorithm, an object describing the MCAS operation is created. This descriptor holds an entry for each address that is to be CASed, and this entry holds the address, the expected value, and the desired new value. In addition, the MCAS descriptor holds a status field, which indicates one of three possible states: undecided, failed, and succeeded. After creating the descriptor, the target addresses are accessed one by one. For

The MCMS Semantics	HTM Implementation of the MCMS Operation

``Atomically execute:
1: bool MCMS (CASDesc* descriptors, int N, int C) {
2:   for i in 1 to N: {
3:     if (*(descriptors[i].address) !=
             descriptors[i].expected_val) {
4:           return false;
5:     }
6: }
7: for i in C+1 to N: {
8:    *(descriptors[i].address) =
           descriptors[i].new_val;
9: }
10: return true;
11:}```
	``1: bool MCMS(CASDesc* descriptors, int N ,int C) {
	2: while (true) {
	3: XBEGIN(retry); // an aborted transaction
	                     // jumps to the retry label
	4: for i in 1 to N: {
	5: if(*(descriptors[i].address) !=
	             descriptors[i].expected_val) {
	6: XEND();
	7: return false; } }
	8: for i in C+1 to N: {
	9: *(descriptors[i].address) =
	             descriptors[i].new_val; }
	10: XEND();
	11: return true;
	12: retry: // aborted transactions jump here
	13: for I in 1 to N: {
	14: if(*(descriptors[i].address) !=
	             descriptors[i].expected_val) {
	15: return false; } } } }```

Fig. 1. The MCMS semantics (left) and its HTM implementation (right)

each address, a CAS is used in an attempt to change the value from the expected value of the MCAS to a pointer that points to the MCAS descriptor. In fact, this is not done using a simple CAS, but a more evolved mechanism (named RDCSS in [10]) which also checks that the status field of the MCAS descriptor is still undecided. The implementation of RDCSS itself relies only on simple CAS operations, and is also described in [10].

If while executing the MCAS, an address that does not hold the expected value is found, then the status field is changed to failed, and any target address whose value was already changed from the expected value to a pointer to the MCAS descriptor is changed back to the old value using a simple CAS. If, on the other hand, all the addresses were successfully changed from the expected value to a pointer to the MCMS descriptor, then the status field is changed to succeeded, and all the target addresses are changed again, this time to hold the desired new value, using a simple CAS. The full details of [10] are considerably more complicated, and are not described here.

This MCAS algorithm burdens concurrent read executions. When a thread reads an address that is a part of an ongoing MCAS execution, it will see the pointer to the MCAS descriptor instead of the correct value (which is either the expected value or the new value) that should logically be stored in the address. Thus, every read execution must check that the read value is not a pointer to an MCAS descriptor, and if it is, it must first participate in completing the MCAS execution, and only afterwards return the (correct) value.

Our non-TM MCMS implementation is thus burdened with this complication. When the MCMS algorithm reads from an address that might be the target of an MCAS, it must be able to tell whether that memory holds regular data, or a special pointer to an MCAS descriptor. In our applications, we were able to

steal the two least significant bits from target fields. For the list algorithm, each target field holds a pointer to another node, and regular pointer values have zero in those two bits. For the tree algorithm, each target field holds either a pointer or a binary flag, and we shift the flag value to the left by two bits.

4 The Linked-List Algorithm

We consider a sorted-list-based set of integers, similar to [8,9,18], supporting the INSERT, DELETE, and CONTAINS operations. Without locks, the main challenge when designing a linked-list is to prevent a node's next pointer from changing concurrently with (or after) the node's deletion. A node is typically deleted by changing its predecessor to point to its successor. This can be done by an atomic CAS, but such a CAS cannot by itself prevent an update to the deleted node's next pointer. For details, see [9].

Harris [9] solved this problem by partitioning the deletion of a node into two phases. In the first phase, the node's next pointer is *marked*, by setting a reserved bit on this pointer. This locks this pointer from ever changing again, but still allows it to be used to traverse the list. In the second phase, the node is physically removed by setting its predecessor to point to its successor. Harris uses the pointer least significant bit as the *mark bit*. This bit is typically unused, because the next pointer points to an aligned address.

Harris's mark bit is an elegant solution to the deletion problem, but Harris's algorithm still has some drawbacks. First, when a mark bit is used, traversing the list requires an additional masking operation to be done whenever reading a pointer. This operation poses an overhead on list traversals. Second, a thread that fails a CAS (due to contention) often restarts the list traversal from the list head. Fomitchev and Ruppert [8] suggested a remedy for the second drawback by introducing back-links into the linked-list. The back-link is an additional field in each node and it is written during the node's deletion.

Fomitchev and Ruppert used three additional fields in each node in excess of the obligatory `key` and `next` pointer fields. Those fields are: the mark bit (similar to Harris), another *flag bit* (also adjoined to the next pointer), and a back-link pointer. To delete a node, a thread first flags its predecessor, then marks the node to be deleted, then writes the back-link from the node to the predecessor, and finally physically removes the node (the same CAS that removes the node also clears the flag of the predecessor.) Due to the overhead of additional CASes, this list typically performs slower in practice compared to the list of Harris.

To illustrate the simplicity of the MCMS operation we present a new linked-list algorithm. The MCMS list is simpler, faster (if HTM is available), and does not use any additional fields on top of the `key` and `next` pointer fields. Similarly to Fomitchev and Ruppert, the MCMS list never needs to start searching from the head on a contention failure.

The crux of our algorithm is that it uses the atomic MCMS to atomically modify the node's next pointer to be a back-link simultaneously with deleting it from the list (see Fig. 2(b)). Thus the `next` pointer points to the next node

while the node is in the list, and acts as a back-link once the node is deleted. Similar to [8,9,18] and others, we use a sentinel head node with a key of minus infinity, and a tail node with a key of infinity.

The algorithm is given in Fig. 2(a)(left), and is surprisingly simple. The SEARCH method receives three parameters, a key to search for, and pointers to pointers to the left and right nodes. When the search returns, the pointer fields serves as outputs. The left node is set to the last node with a key smaller than the given search key. The right node is set to the first node with a key equal to or greater than the search key. The left node parameter also serves as in input for the method, and indicates where to start the search from.

An invariant of the algorithm is that if a node A (which was already inserted to the list) points to node B, and B's key is greater than A's key, then both nodes are currently in the list. When node B is deleted, modifying its next pointer to point to A serves two purposes. First, it serves the purpose of the mark bit that ensures any concurrent operation that might try to modify B's next pointer will fail, which is vital to the correctness of the algorithm. Yet, without necessitating a masking operation before using the next pointer. Second, it establishes a back-link, which other threads might use to avoid the necessity of redoing the search from scratch. Yet, this back-link does not necessitate additional fields in the object, nor specific checks before following this back-link.

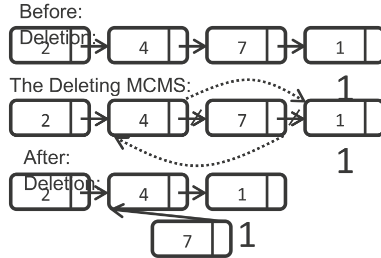
5 The Binary Search Tree Algorithm

We base our tree algorithm on the binary search tree of Ellen et al. [7] (this tree was shown in [5] to outperform both the lock-free skiplist Java implementation and the lock-based AVL tree of Bronson et al. [2]). Our tree is also a leaf oriented tree, meaning all the keys are stored in the leaves of the tree, and each internal node has exactly two children. However, in their original algorithm, each internal node stores a pointer to a designated *Info* object that stores all the information required to complete an operation. When a thread initiates an operation, it first searches the tree for appropriate location to apply it. Then it tests the internal node Info pointer to see whether there is already an ongoing operation, and helps such an operation if needed. Then it allocates an Info object describing the desired change, and attempts to atomically make the appropriate internal node points to this info object using a CAS. Then, it can proceed with the operation, being aware that it might get help from other threads in the process.

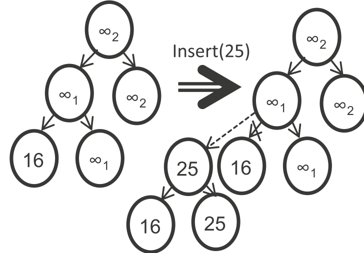
MCMS allows all changes to take place simultaneously. This saves the algorithm designer the need to maintain an Info object, and also boosts performance in the common case, in which an HTM successfully commits. Similarly to a list, a central challenge in a lock-free binary search tree is to ensure that pointers of an internal node will not be modified while (or after) the node is deleted (see [7] for details). For this purpose, in the MCMS tree algorithm, each internal node contains a mark bit (in addition to its key, and pointers to two children). The mark bit is in a separate field, not associated with any pointer. Leaf nodes contain only a key. Upon deleting an internal node, its mark bit is set.

```
MCMS List:
1. void search(int key, Node** left,Node** right) {
2.   *right = (**left).next;
3.   While ((**right).key < key) {
4.     *left = *right;
5.     *right = (**left).next;  } }
6.
7. bool insert(int key) {
8.   Node *left = head; // head is first node in list
9.   Node *right;
10. Node *newNode = new Node(key);
11. While (true) {
12.   search(key, &left, &right);
13.   if ((*right).key == key)
14.     return false;  // key already exists
15.   (*newNode).next = right;
16.   if (CAS(&((*left).next), right, newNode))
17.     return true;  // successfully inserted
18.   } }
19.
20. bool delete(int key) {
21.  Node* left = head;
22.  Node* right;
23.  While (true) {
24.   search(key, &left, &right)
25.   if ((*right).key != key)
26.     return false; // key doesn't exist
27.   Node* succ = (*right).next;
28.   if (MCMS(<&((*left).next), right, succ>,
29.       <&((*right).next), succ, left>))
30.     return true;  // successfully deleted
31.   } }
32.
33. bool contains(int key) {
34.  Node *left = head, *right;
35.  search(key, &left, &right);
36.  return (*right).key == key;
37. }
```

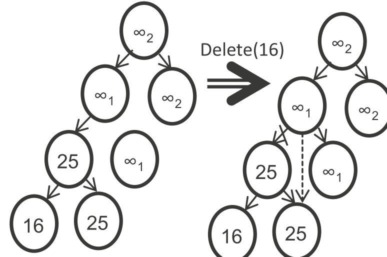
(a) The MCMS List Code



(b) The MCMS List Deletion



(c) The MCMS Tree Insertion



(d) The MCMS Tree Deletion

Fig. 2. The list and tree algorithms

Each MCMS operation that changes pointers of a node also reads the mark bit and compares it to zero. If the bit is set, the MCMS will return false without changing the shared memory, guaranteeing that a deleted node's pointers are never mistakenly altered.

In order to avoid corner cases, we initialize the tree with two infinity keys, ∞_1 and ∞_2, such that $\infty_2 > \infty_1 >$ any other value. The root always has the value ∞_2 its right child is always ∞_2 and its left child is always ∞_1. This idea is borrowed from the original algorithm [7]. Both the INSERT and DELETE operations begin by calling the search method. The search method traverses the tree looking for the desired key, and returns a leaf (which will holds the desired key if the desired key is in the tree), its parent, and its grandparent.

To insert a key, replace the leaf returned by the search method with a subtree containing an internal node with two leaf children, one with the new desired key, and one with the key of the leaf being replaced (See Fig. 2(c)). An MCMS operation atomically executes this exchange while guaranteeing the parent is unmarked (hence, not deleted).

To delete a key, the grandparent pointer to the parent is replaced by a pointer to the deleted node's brother (See Fig. 2(d)), atomically with setting the parent mark bit on, marking it as deleted, and guarding against concurrent (or later) changes to its child pointers. An MCMS instruction also ensures that the grandparent is unmarked, and that the parent's child pointers retain their expected value during the deletion.

6 Fall-Back Execution for Failed Transactions

Formally, transactions are never guaranteed to commit successfully, and spurious failures may occur infinitely without any concrete reason. Our experimental results show that such repeated failures are not observed in practice. Nevertheless, we implemented several fall-back avenues that general algorithms using MCMS may benefit from, and we briefly overview them here. Each transaction is attempted several times before switching to a fall-back execution path. The number of retries is a parameter that can be tuned, denoted *MAX_FAILURES*.

6.1 Using Locking for the Fall-Back Path

The idea of trying to execute a code snippet using a transaction, and take a lock if the transaction fails to commit, is known as *lock elision*. We add a single integer field, denoted *lock* to the data structure. In the HTM implementation of MCMS, before calling XEND the `lock` field is read, and compared to zero. If the lock is not zero, XABORT is called. This way, if any thread acquires the lock (by CASing it to one) all concurrent transactions will fail. If an MCMS operation fails to commit a transaction MAX_FAILURES times, the thread tries to obtain the `lock` by repeatedly trying to CAS it from 0 to 1 until successful. The MCMS is then executed safely. When complete, the thread sets the `lock` back to 0.

Our implementation of lock-elision is slightly different than that of traditional lock-elision. As described in Sect. 3.1, after each transaction abort we compare each address to its expected value, and thus in many cases we can return false after a failure without using any locking or transactions at all.

6.2 Non-Transactional MCMS Implementation as a Fall-Back Path

Another natural fall-back path alternative is to use the non-transactional MCMS implementation of Harris et al., described in Sect. 3.2. While this implementation was proposed for implementing the MCMS on a platform that does not support HTM, it may also be used as a fall-back when hardware transactions repeatedly fail. Several threads can execute this implementation of the MCMS operation

concurrently. However, as mentioned in Sect. 3.2, during the execution of the MCMS operations, the target addresses temporarily store a pointer to a special operation descriptors instead of their "real" data. This requires a careful test for any read of the data structure, which unfortunately comes with a significant overhead.

We experimented with several different mechanisms to guarantee that each read of the data structure is safe. The first mechanism is to always execute the same read procedure that is applied when MCMS is implemented without TM, as described in [10]. The second alternative is to use transactions for the reads as well. Instead of doing a simple read, we can put the read in a transaction, and before executing the transaction XEND, read a `lock` field and abort if it does not equal zero. Each thread that executes a non-transactional MCMS increments the `lock` before starting it, and decrements the `lock` once the MCMS is completed. The reads can be packed into a transaction in different granularity. One may place each read in a different transaction and add a read of the `lock` field; but one may also pack all the reads up to an MCMS into a single transaction and add a single read of the `lock`. We tried a few granularities and found out that packing five reads into a transaction was experimentally optimal.

6.3 A Copying-Based Fall-Back Path

A third avenue for implementing a fall-back for failing transactions is copying-based. Again, a `lock` field is added. Additionally, a single global pointer which points to the data structure is added. When accessing the data structure an indirection is added: the external pointer is read, and the operation is applied to the data structure pointed by it. As usual, each HTM based MCMS operation compares the `lock` to zero before committing, and aborts if the lock is not zero.

Unlike previous solutions, in the copying fall-back implementation the lock is permanent, and the *current copy* of the data structure becomes immutable. After setting the lock to one, the thread creates a complete copy of the data structure, and applies the desired operation on that copy. Other threads that observes the `lock` is set act similarly. The new copy is associated with a new lock that is initiated to zero. Then, a CAS attempts an atomic change of the global pointer to point to the newly created copy instead of the original copy of the data structure (from which it copied the data). Afterwards, execution will continue as usual on the new copy, until the next time a thread will fail to commit a transaction MAX_FAILURES times.

7 Performance

In this section we present the performance of the different algorithms and variants discussed in this paper. In Figs. 3 and 4 we present the throughput of the list and tree algorithms compared against their lock-free counterparts. Each line in each chart represent a different variant of an algorithm. In the micro-benchmarks tested each thread executes either 50 % INSERT and 50 % DELETE operations,

or 20 % INSERT, 10 % DELETE, and 70 % CONTAINS operations. The operation keys are integers that are chosen randomly and uniformly in a range of either 1–32, 1–1024, or 1–1048576. Before starting each test, a data structure is pre-filled to 50 % occupancy with randomly chosen keys from the appropriate range. Deleted nodes were not reclaimed. In addition to the reported results we also tested a work-load of a 100 % CONTAINS, and a work-load of 25 % INSERT, 25 % DELETE and 50 % CONTAINS We also tested a key range of 1–65536. The additional results are similar and are omitted from the figures for lack of space.

In all our experiments, we set the number of MAX_FAILURES to be 7. With this setting, we see MCMS operations that need to complete execution in the fallback path. Reducing this parameter to 6 causes a (slight) performance degradation in a few scenarios. We also tested the number of total MCMS transaction aborts, and the number of MCMS operations that were completed in the fall-back path, when valid. Higher MAX_FAILURES values yield similar performance, but with almost no executions in the fall-back path. This makes the measurements less informative, so 7 was chosen.

The measurements were taken on an Intel Haswell i7-4770, with 4 dual cores (overall 8 hardware threads) and 6 MB cache size, running Linux Suse. Haswell processors with more cores that support HTM are currently unavailable. The algorithms were written in C++ and compiled with GNU C++ compiler version 4.5.

In each chart we present nine algorithms. One for the original lock-free algorithm, which is either Harris's linked-list, or the binary search tree of Ellen et al. A line denoted *HTM MCMS* for the HTM based algorithm without any fall-back path. A line denoted *Software MCMS* for the algorithm in which MCMS is implemented without transactional memory, as described in Sect. 3.2. A line denoted *Locking* for the algorithm in which MCMS is implemented using HTM, and a locking fall-back path is used (Sect. 6.1). A line denoted *Software Read* for an HTM based implementation with a non-transactional based MCMS fall-back path (Sect. 6.2), where each read is executed as in Sect. 3.2. Three lines denoted *1-Read*, *5-Read*, *all-Read*, for HTM based implementations with a non-transactional based MCMS fall-back path (Sect. 6.2), where reads are executed inside transactions in different granularity. And a line denoted *Copying*, for an HTM based implementation with a copying based fall-back path (Sect. 6.3).

The fastest performing algorithm is always the HTM-based MCMS without any fall-back path. On a range of 1048576 available keys, this list algorithm outperforms Harris's by 30–60 %; on a range of 1024 available keys, it outperforms by 40–115 %, and on a range of 32 keys, it outperforms by 20–55 %. The tree algorithm outperforms the tree of Ellen et al. by 6–37 %. For both data structures the lock-based fall-back path adds very little overhead, and the corresponding algorithms trail behind the algorithms without the fall-back path by 1–5 %.

The copying fall-back path algorithm also performs excellently for the linked-list. On average, it performs the same as the lock-based algorithm, with a difference smaller than half a per cent. This makes the HTM MCMS algorithm with the copying fall-back path the fastest lock-free linked-list by a wide margin.

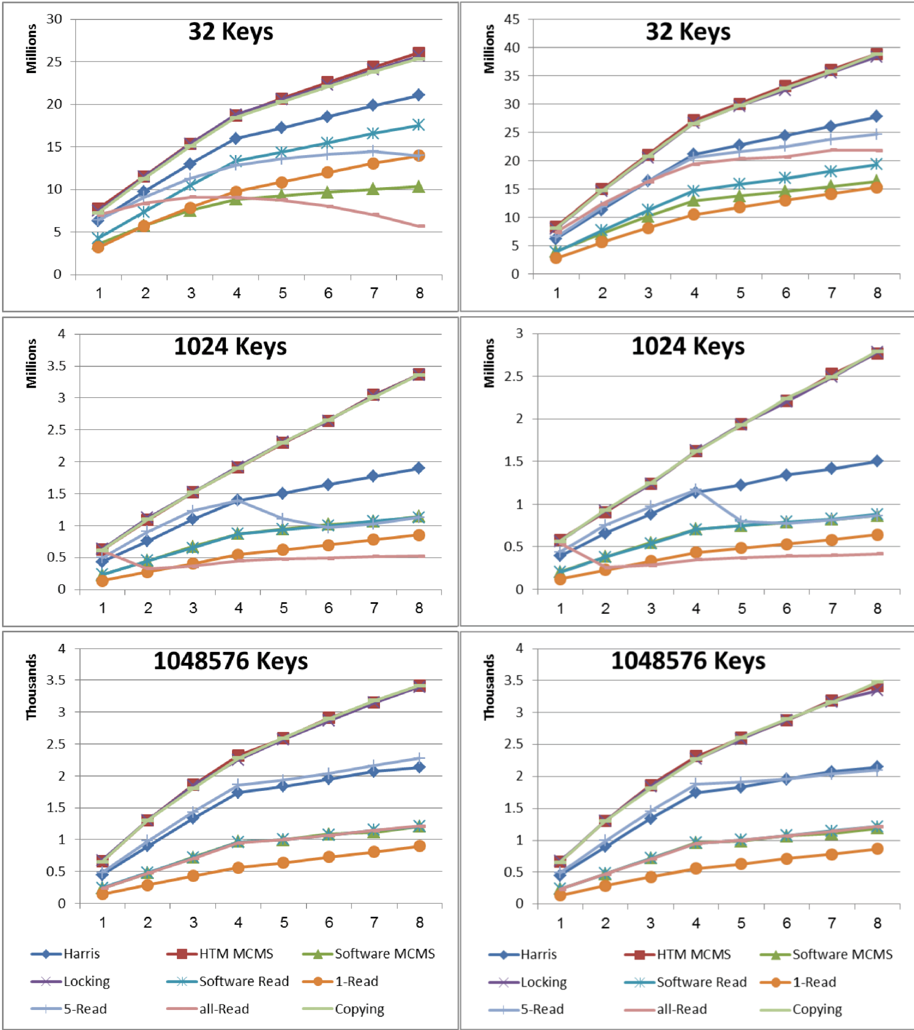


Fig. 3. MCMS-based lists vs. Harris's linked-list. The x-axis represents the number of threads. The y-axis represents the total number of operations executed per second (in millions for key ranges 32 and 1024, in thousands for key range 1048576.)

The copying tree algorithm is not as good, trailing behind the pure HTM algorithm by about 10 %. Yet this algorithm still beats the lock-free algorithm of Ellen et al. in all number of threads for all benchmarks, excluding, surprisingly, the benchmark of 100 % contains for 32 and 1024 available keys. This is surprising, because in this benchmark MCMS is not executed at all. We suspect that the reason is the fact that the search method of the copying based tree receives

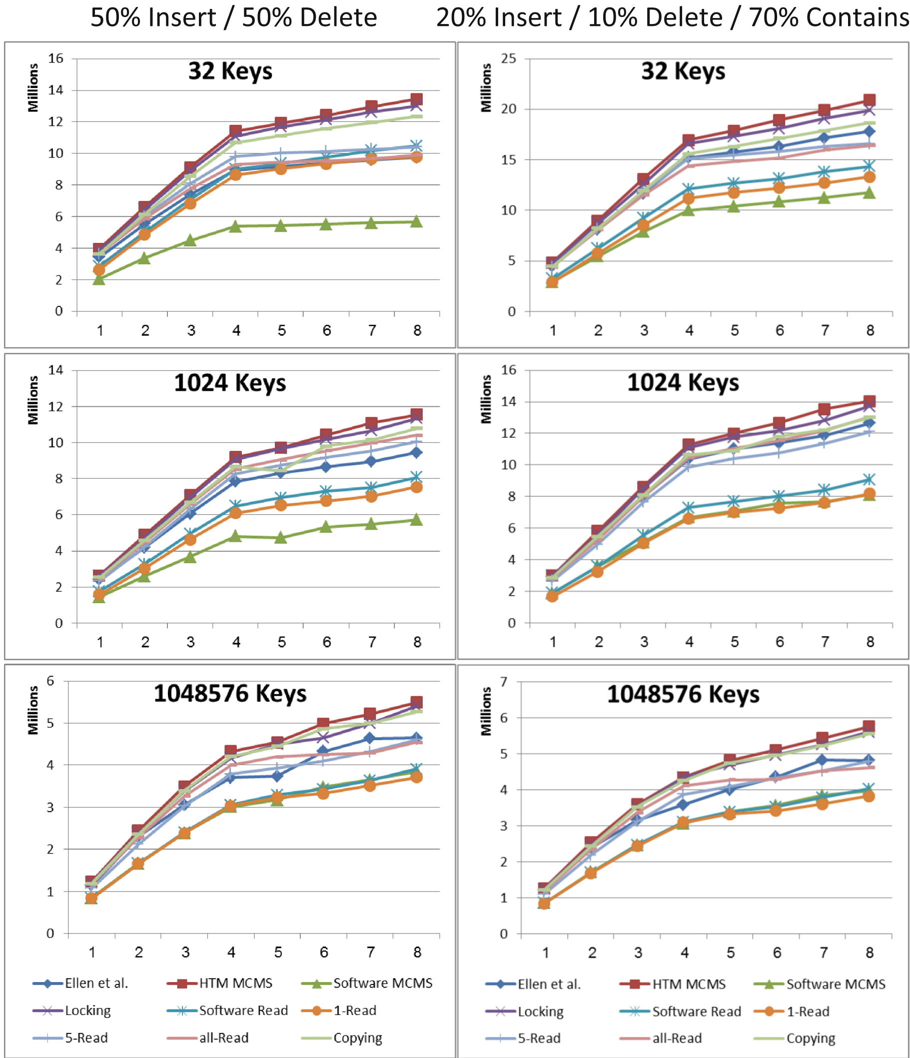


Fig. 4. MCMS-based trees vs. the BST of Ellen et al. The x-axis represents the number of threads. The y-axis represents millions of operations executed per second.

the root of the tree as an input parameter. In the pure HTM algorithm, the root is known at compile time to be final (never changed once it is allocated), which could allow the compiler to optimize its reading.

Using a CAS-based MCMS fall-back path does not work as well as the copying or the lock-based fall-back alternatives. For the list, packing five reads into a transaction yields reasonable performance, usually beating Harris's linked list for a lower number of threads and a larger range of keys (up 20 % faster), but trailing up to 40 % behind it for 8 threads in 32 or 1024 keys when the

micro-benchmark is 50 % INSERTS and 50 % DELETES. Packing all the reads into a single transaction works quite badly for the longer lists, were the large number of reads causes the vast majority of reading transactions to abort. It also works badly for a 32 keys range when the benchmark is 50 % INSERTS and 50 % DELETE. The high number of MCMS transactions combined with read transactions results in poor performance. For the tree, is at times better and at times worse than the tree of Ellen et al., and the difference is up to 10 %. This holds for the option of packing all the reads into a single transaction as well.

Aborts and Fall-back Executions. As expected from the performance results, the number of MCMS executions that are completed in the fall-back path is low. For instance, a copying of a list or a tree of 1048576 keys, which one would expect to be costly, never takes place. On the other end, In a list of 32 keys, for 8 threads, in the micro-benchmark of 50 % INSERTS and 50 % DELETES, copying is executed once every 5000 list operations. In a list of 1024, it is never executed. In a tree of 32 keys when executing with 8 threads, on the 50 % INSERTS and 50 % DELETES micro-benchmark, a copying occurs once every 1730 tree operations, and once every 54000 operations for a tree of 1024 keys running 8 threads. In general, note that once an MCMS is executed in the fall-back path, other MCMS's may abort as a result of the *lock* field being set.

8 Conclusions

In this paper we proposed to use MCMS, a variation of MCAS operation, as an intermediate interface that encapsulates HTM on platforms where HTM is available, and can also be executed in a non-transactional manner when HTM is not available. We established the effectiveness of the MCMS abstraction by presenting two MCMS-based algorithms, for a list and for a tree. When HTM is available, these algorithms outperform their lock-free counterparts. We have also briefly discussed possible "fall-back" avenues for when transactions repeatedly fail. We have implemented these alternatives, and explored their performance overhead.

References

1. Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: 34th International Symposium on Computer Architecture (ISCA 2007), 9–13 June 2007, San Diego, California, USA, pp. 81–91 (2007)
2. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: PPOPP, pp. 257–268 (2010)
3. Brown, T., Ellen, F., Ruppert, E.: Pragmatic primitives for non-blocking data structures. In: PODC, pp. 13–22 (2013)
4. Brown, T., Ellen, F., Ruppert, E.: A general technique for non-blocking trees. In: PPOPP, pp. 329–342 (2014)

5. Brown, T., Helga, J.: Non-blocking k-ary search trees. In: Fernàndez Anta, A., Lipari, G., Roy, M. (eds.) OPODIS 2011. LNCS, vol. 7109, pp. 207–221. Springer, Heidelberg (2011)
6. Dragojevic, A., Harris, T.L.: STM in the small: trading generality for performance in software transactional memory. In: European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys 2012, Bern, Switzerland, 10–13 April 2012, pp. 1–14 (2012)
7. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, 25–28 July 2010, pp. 131–140 (2010)
8. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: PODC 2004, pp. 50–59 (2004)
9. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)
10. Harris, T.L., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 265–279. Springer, Heidelberg (2002)
11. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)
12. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. In: ISCA, pp. 289–300 (1993)
13. Israeli, A., Rappoport, L.: Disjoint-access-parallel implementations of strong shared memory primitives. In: PODC, pp. 151–160 (1994)
14. Rajwar, R., Goodman, J.R.: Speculative lock elision: enabling highly concurrent multithreaded execution. In: MICRO, pp. 294–305 (2001)
15. Rossbach, C.J., Hofmann, O.S., Porter, D.E., Ramadan, H.E., Aditya, B., Witchel, E.: TxLinux: using and managing hardware transactional memory in an operating system. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, 14–17 October 2007, pp. 87–102 (2007)
16. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
17. Sundell, H.: Wait-free multi-word compare-and-swap using greedy helping and grabbing. Int. J. Parallel Prog. **39**(6), 694–716 (2011)
18. Valois, J.D.: Implementing lock-free queues. In: Proceedings of 7th International Conference on Parallel and Distributed Computing Systems, pp. 64–69 (1994)