Performance Analysis of Concurrent Red-Black Trees on HTM Platforms

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

Abstract

In this paper we analyze the performance of concurrent red-black trees using two HTM implementations, Intel's Transactional Synchronization Extensions (TSX) on Haswell processors and IBM's Power8 HTM. We parallelize bottom-up and top-down red-black trees using coarse-grained transactions and evaluate their performance. Our experimental results show that HTM can outperform lock-based implementations with minimal programming effort. Furthermore, they reveal that scalability depends on limits imposed by the underlying hardware, indicating that a programmer needs to take them into account in order to utilize HTM efficiently.

Keywords Hardware Transactional Memory, Concurrent Data Structures, Red-Black Trees

1. Introduction

With the prevalence of multi-core systems, concurrent programming has come to the forefront of software development. When developing concurrent applications, a programmer needs to explicitly control the synchronization and communication between multiple threads of execution. The most common approach to synchronization is locking. By controlling the granularity of locks, a programmer can trade-off between performance and programmability. Coarse-grained locking is simple to implement but can lead to serialization of accesses and thus loss of performance. On the other hand, fine-grained locking enables more parallelism and thus higher performance, but it is considerably harder to implement and much more error-prone. In general, locking approaches suffer from subtle problems like priority inversion, convoying, deadlock and lack of robustness. To avoid these pitfalls, non-blocking algorithms have been proposed that use hardware-supported atomic instructions (e.g. compare-and-swap). However, non-blocking approaches are at least as difficult to implement as fine-grained locking.

Transactional Memory (TM) [1] has emerged as an attractive alternative to lock-based and non-blocking approaches, with particular focus on the ease of programming. A programmer needs only to annotate regions of code that must be executed atomically, without worrying how the concurrent accesses to shared state should be synchronized. The underlying TM system is then responsible for guaranteeing correctness by aborting transactions that would lead to data inconsistencies.

Until recently research on TM had focused mainly on software solutions (STM) or simulated hardware. However, the TM landscape has recently changed, as IBM and Intel introduced Hardware TM (HTM) support in their processors [2–4], targeting HPC and commodity systems. In this paper we evaluate the performance of a classic data structure, the Red-Black tree (RBT), when HTM is used for synchronization. RBTs are an excellent example of a data structure that combines the following properties: *a*) It is widely applicable, as it is contained in several industrial quality software projects (Java, C++, Python, Linux kernel etc. [5]). *b*) It has non-trivial properties, that bring up the challenges of programmability both in the serial and, more importantly, its concurrent versions. *c*) It can accommodate substantial parallelism for a relevant analysis to be meaningful.

We implement and evaluate concurrent RBT versions in two HTM implementations: Intel's Transactional Synchronization Extensions (TSX) provided on Haswell processors and IBM's Power8 HTM. We need to note that, based on both our prior experimentation and Intel's announcement in Q3 2014, TSX is not guaranteed to work correctly. However, we include the results of our performance analysis here, as we anticipate that future systems with validated TSX results

TRANSACT '15, June 15–16, 2015, Portland, Oregon, USA.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

will exhibit similar performance. Based on our experimental results we draw several interesting conclusions:

1. A straightforward HTM RBT implementation with minimal programming effort can outperform lock-based implementations.
2. Alternative concurrent RBT implementations can lead to dramatically different scalability properties when HTM comes into play.
3. The size of the RBT has significant impact on the performance of HTM-based implementations.

The rest of this paper is organized as follows. Section 2 provides the necessary background information on RBTs. In Section 3 we analyze each of three operations supported by an RBT and the ways to synchronize them. Section 4 presents the characteristics of the two HTM implementations. Finally, the results of our experiments are presented in Section 5 and we summarize our conclusions and directions for future work in Section 6.

2. Background

RBTs [6] are a class of height-balanced binary search trees (BSTs). In addition to the properties of a BST, the following must be satisfied by an RBT:

1. Every node is either red or black.
2. The root is black.
3. All leaves are black.
4. Every red node must have two black child nodes.
5. Every path from a given node to any of its descendant leaves contains the same number of black nodes.

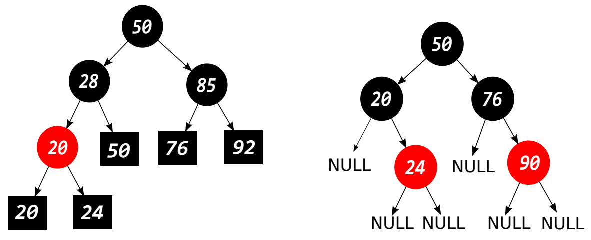
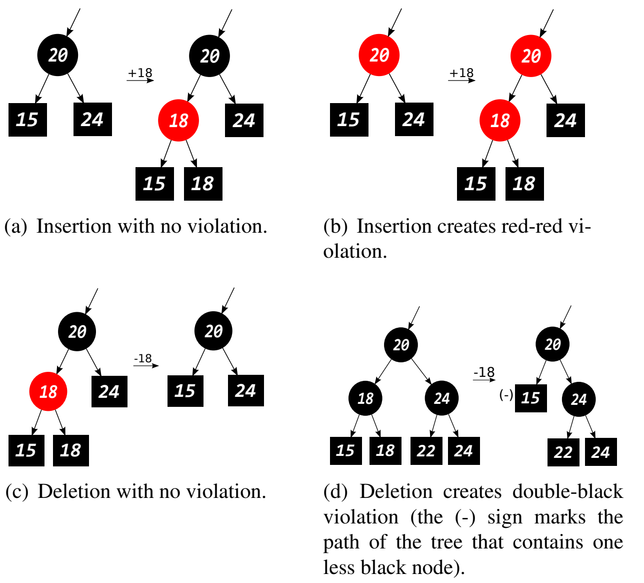


Figure 1. An example of an RBT in external and internal format.

When the above properties are satisfied, the tree is guaranteed to have $O(logn)$ height, where n is the total number of nodes currently in the tree. It can be shown that the maximum path length from the root to a leaf node in an RBT is at most twice as long as the minimum path length. Keeping the tree balanced has the advantage of bounded traversal time but introduces an additional overhead, both in terms of performance when rebalancing operations are required, and in terms of programming effort as one has to take care of many different cases in which the tree properties are violated. RBTs are widely used to store key-value pairs and



(a) Insertion with no violation.

(b) Insertion creates red-red violation.

(c) Deletion with no violation.

(d) Deletion creates double-black violation (the (-) sign marks the path of the tree that contains one less black node).

Figure 2. Examples of insertion and deletion in external RBT.

support the following operations: *a) lookup* to search for a specific key, *b) insert* and *c) delete* to add and remove a key-value pair from the tree respectively.

Depending on the way the key-value pairs are being stored in the tree structure, RBTs are categorized as *internal* and *external*. Internal trees store a key-value pair in every node of the tree. On the other hand external trees store the values only in the leaves while the internal nodes contain only keys and are used solely for routing purposes. An external and an internal RBT containing 5 keys are shown in Figure 1. We use square shapes to distinguish the leaves from the internal nodes in the external tree. To remove a node with two children from an internal RBT, we must first find its successor (the leaf node with the greater key that is less than the key of that node), swap their keys and delete the successor leaf node. In a concurrent configuration this operation requires that the deleting thread has exclusive access to every node between these two nodes. To avoid such complex situations we use external RBTs, where the node to be deleted is always a leaf.

When inserting or deleting a node, some of the aforementioned properties might be violated and actions must be taken to restore them and rebalance the tree. The two possible violations are: *a) red-red violation*, when a red node acquires a red child (violation of property 4) and *b) double-black violation*, when a path of the tree contains one less black node than the other paths (violation of property 5). Figure 2 illustrates possible scenarios when inserting or deleting a node in an external RBT. Figures 2(b) and 2(d) depict the cases when a violation is caused. To deal with these violations a number of node recolors and rotations are applied.

3. Concurrent RBTs

Three operations are supported by an RBT: lookup, insertion and deletion. In this section we provide a brief overview of each of the three operations and the different approaches for synchronizing them using either locks or HTM.

3.1 Lookup

The lookup for a key in an RBT is performed in the same way as in the conventional BST. Starting from the root of the tree, a path of nodes is traversed until a leaf is reached. If the key searched for is present in the tree, it will be contained in the reached leaf.

In a concurrent implementation, to synchronize the lookup operation with a fine-grained approach, the well known *hand-over-hand locking* technique [7] can be used. Locking is performed at the granularity of an RBT node and at each distinct step of the traversal, the lock of the next node to be visited is acquired before the lock of the current node is released.

On the other hand, when HTM is used, the lookup operation can be enclosed in a single transaction. It will be a read-only transaction whose read-set consists of all the nodes in the traversed path. Consequently, the size of the transaction's read-set is proportional to the length of the path and, for external RBTs, the height of the tree.

3.2 Insertion

The classic implementation of the insert operation of an RBT [8] consists of two phases. The first one traverses the tree in a top-down manner, i.e. from the root towards the leaves, and locates the place where the new key is going to be inserted. The second phase traverses the tree in a bottom-up manner, i.e. from the leaf towards the root, performing recolors and rotations to restore the RBT properties and rebalance the tree. Whereas the top-down phase always reaches a leaf, the bottom-up phase backtracks a number of times depending on the violation (only in the worst case it shall reach the root of the tree). This type of insertion is called *bottom-up*.

In concurrent implementations, parallel threads might traverse the tree in opposite directions, thus complicating the correct implementation using fine-grained locking. On the other hand, when HTM is used, the insertion is enclosed in a single transaction. Its read-set will contain all the nodes traversed during the first top-down phase as well as a number of additional nodes that are read during the bottom-up rebalancing phase. Its write set includes all the nodes that have been modified, either by recoloring or rotating, during the rebalancing phase.

To enable fine-grained synchronization, *top-down* approaches have been proposed [6, 9], where the insertion is performed in a single top-down phase. While traversing the path from the root to the appropriate leaf, the necessary recolors and rotations are being applied, assuring that when the new node is inserted no backtrack is necessary. Every step of the top-down insertion can be viewed as a window operating only on a small subtree. The pessimistic nature of this top-down approach results in generally more tree modifications compared to the bottom-up counterpart and consequently to worse performance in serial executions.

Fine-grained locking can be applied in the top-down approach using the hand-over-hand locking technique. At each step the nodes that consist the current window are locked and they are released only after the nodes of the next window are locked.

Unfortunately, HTM cannot be used in a similar manner. As in the bottom-up version, a single transaction needs to enclose the top-down insertion. Compared to its bottom-up counterpart, this transaction will have larger read-set as it will contain not only the nodes of the traversed path, but additional nodes that have been examined during the traversal. Furthermore, the write-set will contain nodes from more levels of the tree due to the pessimistic nature of the top-down operation.

3.3 Deletion

The delete operation of an RBT is basically similar to the insertion. It too can be implemented as bottom-up or top-down, allowing in the latter case the employment of fine-grained locking.

3.4 Implementations

For our analysis we have implemented and evaluated four concurrent RBTs. They are summarized in Table 1, categorized by the synchronization method used and the way insertions and deletions are implemented (bottom-up or top-down).

Table 1. RBT implementations.

	Bottom-up	**Top-down**
Coarse-grained locking	*bu_cg_lock*	-
Fine-grained locking	-	*td_fg_lock*
HTM	*bu_cg_htm*	*td_cg_htm*

Coarse-grained locking versions are trivial to implement, but they provide no parallelism as all the accesses on the RBT are serialized. In our analysis we include only the bottom-up, coarse-grained locking implementation as a reference point, as the top-down counterpart performs even worse due to its pessimistic nature. Using fine-grained locking we implement a top-down version because, as explained before, a bottom-up version is hard, if possible at all, to be implemented.

Finally, using HTM, we implement both a bottom-up and a top-down RBT. In general, bottom-up is expected to employ smaller transactions than top-down and exhibit less conflicts, as the pessimistic nature top-down approach perform more node updates (recolors or rotations).

4. Hardware Transactional Memory

IBM and Intel have recently shipped processors with HTM support, namely Power8 [3] and Haswell [2]. The basic TM characteristics of these two implementations are similar:

- *Data versioning:* Both implementations use lazy versioning and all memory writes that are being performed inside a transaction become visible to other threads only after the successful commit of the transaction.
- *Best-effort:* Both implementations are best-effort HTMs. As no forward progress is guaranteed using only transactional mode, a transaction may always fail to commit and therefore a non-transactional fallback path is necessary.
- *Conflict detection:* Both TSX and Power8 HTM detect conflicting operations in the granularity of a cache line. Moreover, they both provide *strong isolation*, meaning that a conflict is detected even if the conflicting access occurs in non-transactional code.

In general, a transaction may fail to commit for various reasons including:

- *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. While Power8 distinguishes between transactional and non-transactional conflicts, Haswell just reports both as conflicts.
- *Capacity:* When the transaction's footprint has exceeded a size limit imposed by the HTM implementation. Table 2 presents the hardware transactional buffers' sizes of each HTM. It is evident that Haswell can support larger transactions.
- *Explicit:* When the programmer explicitly aborts the transaction. In our implementations we explicitly abort transactions whenever the global lock that we use at the fallback handler is checked and found to be taken.

Table 2. The size of the transactional buffers of the HTMs.

	Haswell	**Power8**
Read set	4MB	8KB
Write set	22KB	8KB

5. Experimental Evaluation

5.1 Experimental Setup

The main characteristics of the two systems used in our study are shown in Table 3. In the Haswell machine we only employ the RTM mode of the HTM infrastructure. To demonstrate the difference of single thread performance of the two platforms, we execute the serial, bottom-up implementation for 3 different tree sizes. Table 4 presents the achieved throughput in terms of operations per usec. The Haswell core outperforms Power8 by 42% to 73% depending on the size of the tree.

Table 3. The hardware characteristics of our platforms.

Name	**Haswell**	**Power8**
Processors	1 x Intel Core i7-4771	2 x Power8
# Cores	4	2 x 10
# Threads	8	160
Core clock	3.5 GHz	3.7 GHz
L1 (Data)	8-way, 32 KB, 64B block size	8-way, 64 KB, 128B block size
L2	8-way, 256 KB, 64B block size	8-way, 512 KB,
L3	16-way, 8 MB, 64B block size (shared)	8-way, 80 MB, 128B block size (shared per die)
Memory	16 GB	256 GB

Table 4. Single thread throughput (ops/usec).

Tree Size (Nodes)	**Haswell**	**Power8**
1K	11.82	6.82
1M	1.38	0.97
10M	0.86	0.53

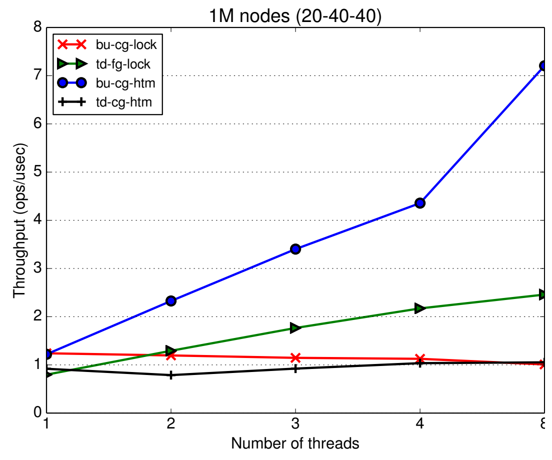
All RBT versions were implemented in C and compiled with GCC 4.9.1 with the -O3 optimization flag enabled. To avoid false conflicts we pad each tree node to fit in one cache line.

For the evaluation of our concurrent implementations we vary two parameters:

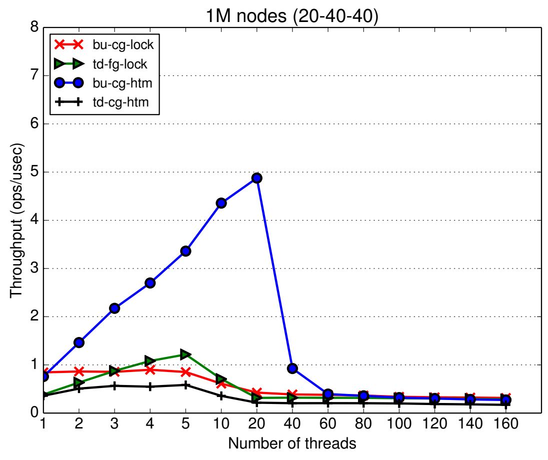
- *The size of the tree:* We use 5 different tree sizes that span from small (1K nodes, ≈250KB memory footprint) to large (10M nodes, ≈2.5GB memory footprint) trees. As the size of the tree increases, so does the length of the traversed paths in the tree and consequently the execution time per operation, as well as the size of the transactional read and write sets of our HTM implementations.
- *The mixture of operations:* We use 3 different configurations, where 80%, 50% and 20% of the operations respectively are lookups in the tree, i.e. read-only travesals, while the rest of the operations are equally divided between insertions and deletions.

For every run we execute 10 million operations equally divided between concurrent threads. A warm-up phase is being performed in order to pre-populate the tree with half the possible keys. This way we assure that the average execution time of each operation remains the same throughout the whole execution. We pin threads to logical cores in such a way that all the available physical cores are being employed before utilizing hyperthreads or SMT contexts.

As both HTMs are best-effort, in order to guarantee forward progress, our HTM implementations include a non-transactional fallback path, in which, after a number of failed

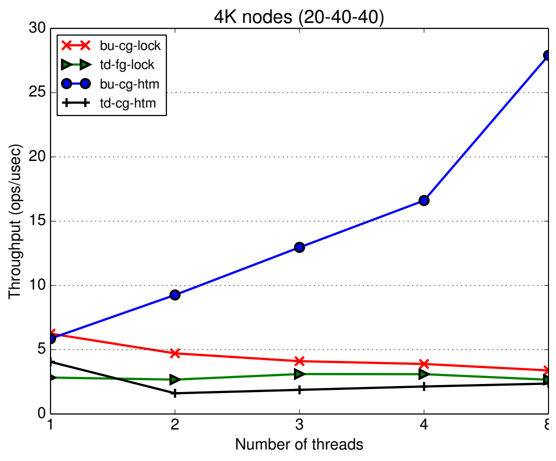


(a) Haswell

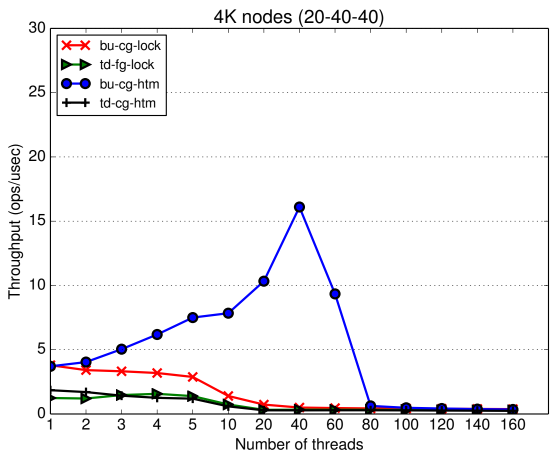


(b) Power8

Figure 3. Throughput of HTM-based and lock-based concurrent versions for an RBT with 1M nodes.



(a) Haswell



(b) Power8

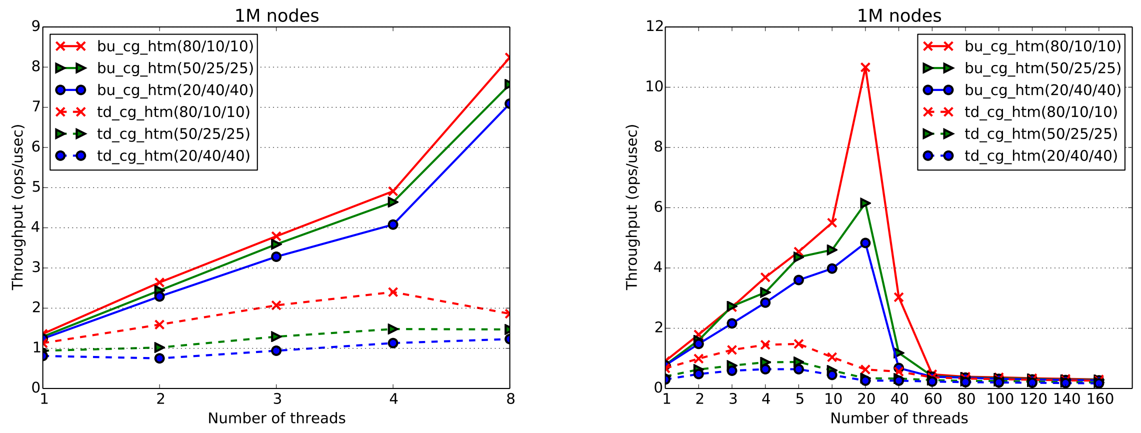
Figure 4. Throughput of HTM-based and lock-based concurrent versions for an RBT with 4K nodes.

transactions, a global lock is acquired, serializing all accesses on the RBT. When a transaction starts, the global lock is read and if it is not free, an explicit abort is performed. Adding the global lock in any transaction's read set is necessary to prohibit transactions from running concurrently with the non-transactional path. When the global lock is acquired, all running transactions fail due to the conflict on the global lock. In our executions we retry each transaction 10 times before resorting to the single global lock and serializing accesses.

5.2 HTM-based vs Lock-based

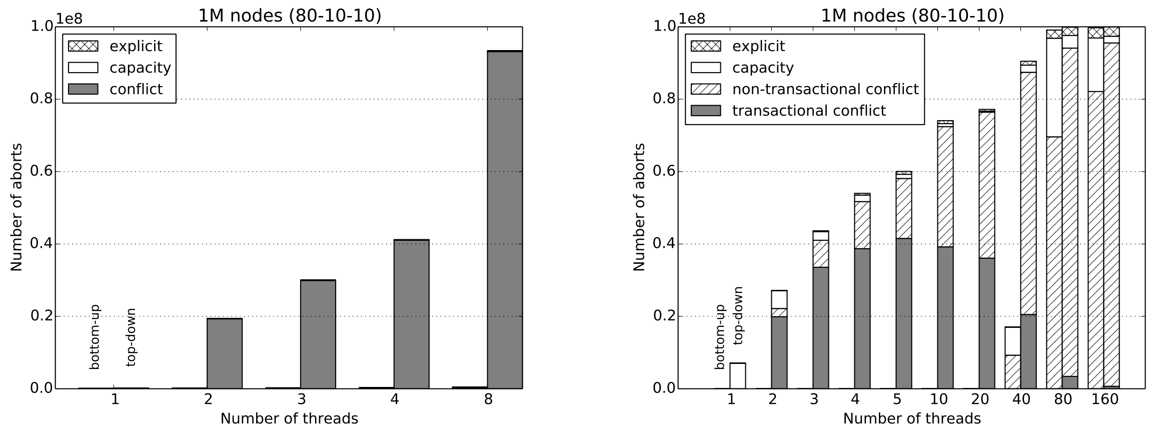
We first compare the HTM-based RBTs with their lock-based counterparts. Figure 3 presents the throughput of the four concurrent implementations for a tree with 1 million nodes and a workload with only 20% of the operations being lookups. Such a write-intensive workload is not ideal for HTM as a large amount of conflicts is expected. As expected, *bu_cg_lock* does not scale even for a low number of threads, as all the accesses on the RBT are serialized. Moreover, when threads are spread over multiple numa nodes (more than 5 threads on Power8), a performance degradation is observed, due to the coherence traffic imposed by reads and writes on the shared global lock.

In contrast to the coarse-grained lock implementation, *td_fg_lock* scales for up to 8 and 5 threads, on Haswell and Power8 respectively. However, coherence traffic is again high due to reads and writes on the locks of each node in the tree, resulting in performance loss for more than 5 threads on Power8.



(a) Haswell (b) Power8

Figure 5. Throughput of bottom-up and top-down HTM-based implementations for various workloads.



(a) Haswell (b) Power8

Figure 6. Aborts' breakdown for bottom-up and top-down HTM-based implementations.

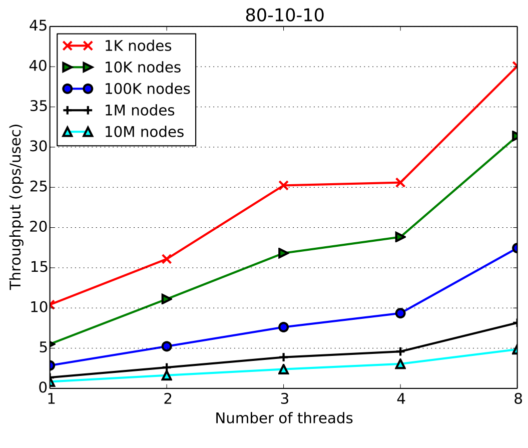
HTM-based implementations exhibit completely opposite behaviors. *bu_cg_htm* scales for up to 20 threads and outperforms the lock-based implementations offering around 11 and 15 times better throughput compared to the coarse and fine-grained lock implementations respectively. On the other hand, *td_cg_htm* does not scale and is even worse than the lock-based implementations.

The 1M nodes tree provides a large number of different paths on which threads can concurrently operate. It is thus a tree that favors HTM in the sense that conflicts are expected to be rare. Figure 4 depicts the throughput of our implementations for a smaller tree with 4K nodes, which is expected to suffer from more conflicts. It is evident that *bu_cg_htm* still performs better than the rest.

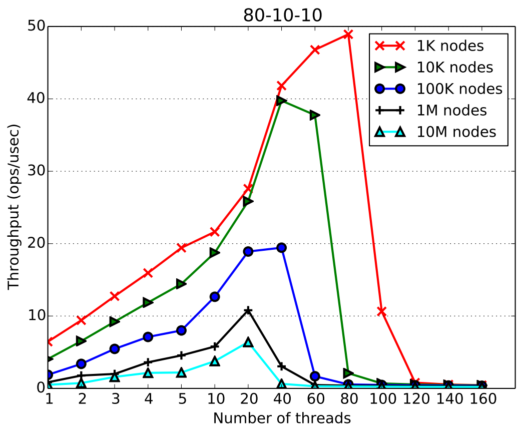
5.3 Bottom-up vs Top-down HTM

In this section we further analyze the performance of the two HTM-based implementations. Figure 5 presents their throughput for a tree with 1M nodes and various workloads. For all the workloads, the bottom-up approach outperforms the top-down one, achieving up to almost 4.5 and 17 times better performance for 8 and 20 threads on Haswell and Power8 respectively.

To offer an insight on the different performance of the two approaches, Figure 6 depicts the number of aborts and their breakdown for the 80-10-10 workload. It is obvious that the top-down approach suffers significantly more aborts, which hurt its performance. This is expected, as the top-down approach, due to its pessimistic nature, results in more tree restructures and thus more data conflicts. On the contrary, in



(a) Haswell



(b) Power8

Figure 7. Throughput of bottom-up HTM-based implementation for various tree sizes.

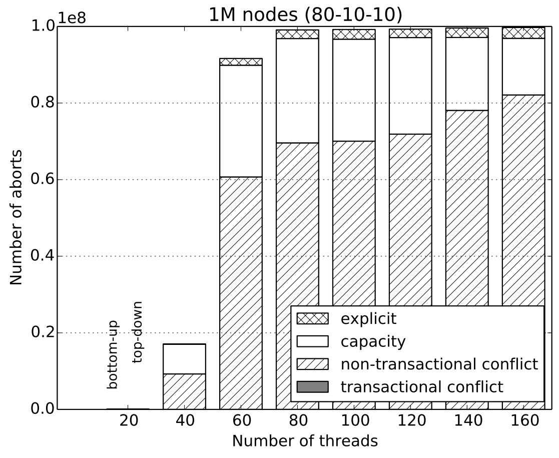


Figure 8. Aborts' breakdown for bottom-up HTM in Power8.

the bottom-up approach appear to be almost no data conflicts for up to 20 threads on the Power8.

However, as threads keep increasing, performance deteriorates due to a significant rise of aborts. Figure 8 presents the aborts' breakdown for the bottom-up implementation for 20 threads or more for the Power8. It is evident that, as we employ more SMT threads, capacity aborts increase, which result in the threads acquiring the global lock more often in order to perform their operations. This is illustrated by the huge increase of non-transactional and explicit aborts.

5.4 HTM Limitations

To further investigate the impact of capacity limits on the bottom-up approach, we evaluate its behavior for various tree sizes and present the results in Figure 7. In Haswell, even the largest tree with 10M nodes manages to scale for up to 8 threads. This is due to the fact that Haswell provides a larger transactional read buffer than Power8, as it utilizes the L1 cache. At the same time, the resources are shared by only two threads.

On the other hand, as the tree gets bigger we can efficiently utilize fewer SMT threads for Power8. This can be attributed to two reasons. First, as the tree size increases, so does the transaction footprint, making it more possible for a capacity abort to occur. Second, as SMT threads share the transactional resources, the allowed size for a transaction effectively shrinks.

5.5 Global-lock fallback

As explained before, in order to ensure forward progress, the threads acquire a global lock after a number of retries. As aborts increase for high numbers of threads, so does the possibility of a thread following the non-transactional path, which in turn causes more aborts, which are illustrated in Figures 6 and 8 as non-transactional and explicit aborts.

Figure 9 presents the achieved throughput together with the percentage of operations completed with the lock acquired for the bottom-up HTM-based implementation executing a 80-10-10 workload on a tree with 1M nodes, while varying the number of attempts to successfully complete a transaction before acquiring the global lock. It is obvious that as more threads share the transactional resources, the number of retries must increase in order for a thread to succeed in executing its operation in transactional mode. However, there is a trade-off, as for a high number of retries the threads spend their time retrying the same transaction again and again, achieving no real progress and thus lowering their throughput.

6. Conclusions and future work

In this paper we used two commodity HTM implementations, Intel's TSX and IBM's Power8 HTM, to implement

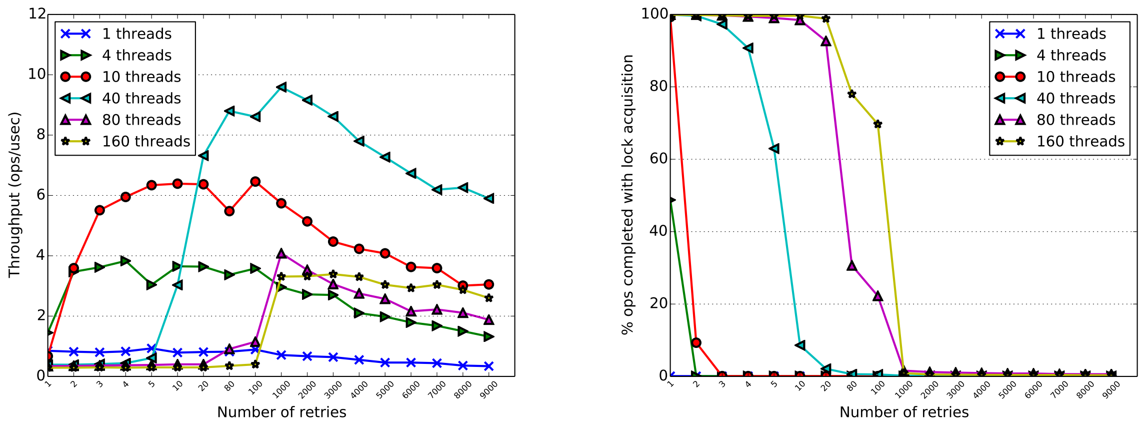


Figure 9. Throughput and percentage of operations completed with locks for different number of retries.

and evaluate concurrent bottom-up and top-down RBTs. We first compared the HTM-based implementations with their lock-based counterparts, and found out that HTM can provide up to 15 times better throughput for a relatively high number of threads. Next, we investigated how the choice between bottom-up and top-down approaches affects the performance of HTM. The results we obtained showed that the top-down approach, which is the appropriate choice for a lock-based implementation, does not fit to HTM because its pessimistic nature leads to many tree restructures and as a result the majority of transactions fail to commit due to conflicts. On the other hand, the bottom-up approach fully exploits HTM for up to a high number of threads, until a point when HTM resources are shared by too many threads and the performance deteriorates due to capacity aborts.

In our future work we intend to tackle the HTM limitations by employing various optimizations. First, we will attempt to split the coarse-grained transactions used in our implementations to consecutive fine-grained transactions, in order to reduce capacity aborts. Second, instead of using a global lock in the fallback path, which aborts all running transactions, a per-cpu lock could be used to only abort threads executing in the same core. This way, threads that would otherwise abort due to resource sharing, will be given the chance to execute using all the available resources, while threads on other cores will keep executing concurrently.

Acknowledgments

This research was funded by project I-PARTS: ”Integrating Parallel Run-Time Systems for Efficient Resource Allocation in Multicore Systems” (code 2504) of Action ARISTEIA, co-financed by the European Union (European Social Fund) and Hellenic national funds through the Operational Program Education and Lifelong Learning (NSRF 2007- 2013). The IBM 8247-22L servers on which we have evaluated our implementations have been kindly provided by IBM Hellas.

References

[1] M. Herlihy and J. E. B. Moss, “Transactional memory: Architectural support for lock-free data structures,” in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA ’93, (New York, NY, USA), pp. 289–300, ACM, 1993.

[2] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, “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, (New York, NY, USA), pp. 19:1–19:11, ACM, 2013.

[3] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, “Robust architectural support for transactional memory in the power architecture,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA ’13, (New York, NY, USA), pp. 225–236, ACM, 2013.

[4] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, “Evaluation of blue gene/q hardware support for transactional memories,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT ’12, (New York, NY, USA), pp. 127–136, ACM, 2012.

[5] R. Sedgewick, “Left-Leaning red black trees.” `http://www.cs.princeton.edu/~rs/talks/LLRB/LLRB.pdf`.

[6] L. J. Guibas and R. Sedgewick, “A dichromatic framework for balanced trees,” in *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pp. 8–21, Oct 1978.

[7] R. Bayer and M. Schkolnick, “Readings in database systems,” ch. Concurrency of Operations on B-trees, pp. 129–139, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 3rd ed., 2009.

[9] R. A. Tarjan, “Efficient top-down updating of red-black trees,” Tech. Rep. TR-006-85, Department of Computer Science, Princeton University, 1985.