Massively Concurrent Red-Black Trees with Hardware Transactional Memory

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*—Hardware Transactional Memory (HTM) is nowadays available in several commercial and HPC targeted processors and in the future it will likely be available on systems that can accommodate a very large number of threads. Thus, it is essential for the research community to target on evaluating HTM on as many cores as possible in order to understand the virtues and limitations that come with it.**

In this paper we utilize HTM to parallelize accesses on a classic data structure, a red-black tree. With minimal programming effort, we implement a red-black tree by enclosing each operation in a single HTM transaction and evaluate it on two servers equipped with Intel Haswell-EP and IBM Power8 processors, supporting a large number of hardware threads, namely 56 and 160 respectively. Our evaluation reveals that applying HTM in such a simplistic manner allows scalability for up to a limited number of hardware threads. To fully utilize the underlying hardware we apply different optimizations on each platform.

I. Introduction

Multi-core systems are nowadays ubiquitous and the number of cores comprising a processor chip is growing at a fast pace. However, multi-threaded applications that can take full advantage of this growing number of cores are hard to implement, as the programmer needs to carefully synchronize concurrent accesses to shared data.

The most common approach to synchronization is locking, where only one thread is allowed to enter a critical section at any time, a mechanism known as mutual exclusion. By controlling the granularity of such critical sections, a programmer can trade-off performance for 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 often 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 also very hard to implement.

Transactional Memory (TM) [1] is a programming paradigm that aims to combine the simplicity of coarse-grained locking with the performance of fine-grained locking. Specifically, with TM a programmer annotates regions of code, called *transactions*, that should be executed atomically; the underlying TM system executes multiple transactions in parallel and serializes them only if a conflict is detected. In a non-conflicting execution a transaction *commits*, otherwise it *aborts*. To enable conflict detection all reads and writes performed inside a transaction are tracked in the *read-set* and *write-set* respectively. The combination of these two sets is the transaction's *footprint*.

Until recently, most of the research on TM has concentrated on software implementations (STM) or simulated hardware. STM generally scales well, however, it imposes high single thread overheads because all shared memory accesses inside a transaction are instrumented. Nowadays, Hardware TM (HTM) is avalaible in Intel Haswell [2] (and successors), IBM Power8 [3], Blue Gene/Q [4], and zEC12 [5] processors. HTM has very low overhead, but it comes with two significant limitations. First, the transaction's footprint is limited by the hardware buffers that maintain the read and write sets. Second, apart from conflicts a transaction may abort for several other reasons, e.g. a timer interrupt or a system call. As a consequence, all currently available HTM systems implement a *best-effort* TM, which means that no progress guarantees are provided and the programmer is responsible for implementing a non-transactional execution path (*fallback path*) to guarantee progress. The most common fallback path is using a single global lock (SGL) to serialize all accesses.

In this paper we utilize HTM to parallelize the access to red-black trees (RBTs), a classic data structure, widely used to implement the *dictionary* abstract data type. We first implement a straightforward concurrent RBT by just enclosing each operation in a single HTM transaction and using an SGL fallback path to ensure forward progress. Such a simplistic implementation is representative of the simplicity of the TM programming model. However, as our evaluation reveals, to achieve scalability for high number of threads, the programmer needs to take several things under consideration.

We perform our experimental evaluation on two multi-core servers, one equipped with Intel's Haswell-EP accommodating 56 hardware threads and one with IBM's Power8 providing 160 hardware threads. In order to enable scalability for these high number of threads, we face different challenges and resort to different solutions on each processor. On Haswell-EP the large transactional buffers provided allow the straightforward

HTM RBT to scale to a large number of threads. However, the programmer needs to carefully tune the number of transactional retries before serializing on the global lock. On the other hand, on Power8 the sharing of transactional resources between several hardware threads hurts the performance severely. In this case, increasing the number of transactional retries does not help. Therefore, we propose two ways to cope with the problem: an alternative fallback path that uses a per-cpu lock and a method to split the large transactions to a sequence of smaller fine-grained ones.

The rest of the paper is organized as follows. Section II presents the characteristics of the two HTM implementations that we have used in our study and Section III provides background information on RBTs. In Section IV we explain the details of the straightforward HTM RBT implementation. Sections V and VI provide the optimizations that we applied for each HTM implementation. Finally, we discuss related work in Section VII and draw conclusions on Section VIII.

II. Hardware Transactional Memory

Intel and IBM have recently shipped processors with HTM support, namely Haswell [2] and Power8 [3]. The ISA of each of the two processors has been extended with a set of instructions that allows a programmer to use the HTM infrastucture. The instructions provided are the following (*Haswell*/*Power8*):

- *xbegin/tbegin*: Marks the start of a transaction.
- *xend/tend*: Marks the end of a transaction.
- *xabort/tabort*: Explicitly aborts a transaction. In both processors a representative code can be passed to the abort instruction to enable the distinction among different abort reasons.

The basic TM characteristics of the two HTM implementations are similar:

- *Lazy data versioning:* Both implementations use lazy versioning. All memory writes performed inside a transaction become visible to other threads only after the transaction successfully commits.
- *Eager conflict detection:* Upon the detection of a conflict the transaction immediately aborts.
- *Cache line granularity:* Both Haswell and Power8 detect conflicting operations at a cache line granularity. This can result in false conflicts when concurrent threads modify disjoint parts of a cache line.
- *Strong isolation:* Both processors provide strong isolation, meaning that a conflict is detected even if the conflicting access occurs in non-transactional code.
- *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.

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

- *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. While Power8 distinguishes between transactional and non-transactional conflicts, Haswell just reports both as conflicts.
- *Capacity abort:* When the transaction's footprint has exceeded a size limit imposed by the HTM implementation. Table I presents the hardware transactional buffers' sizes of each HTM [6], [7], which we have also validated with micro-benchmarks on our servers. It is evident that Haswell-EP can support larger transactions. Note also that in Haswell-EP these buffers are shared among 2 hyperthreads in each core, while in Power8 between 8 SMT contexts.
- *Explicit abort:* When the programmer explicitly aborts the transaction.
- *Other:* A transaction may abort due to several other reasons including interrupts, unsupported instructions, system calls etc.

TABLE I
The Size of the Transactional Buffers of the HTMs.

	Haswell-EP	**Power8**
Read set (Total / Per HW thread)	4MB / 2MB	8KB / 1KB
Write set (Total / Per HW thread)	22KB / 11KB	8KB / 1KB

III. Red-Black Trees

A. Definition

Red-black trees [8] (RBTs) are a class of height-balanced binary search trees. In addition to the properties of a binary search tree, an RBT must also satisfy the following which guarantee that the tree remains balanced:

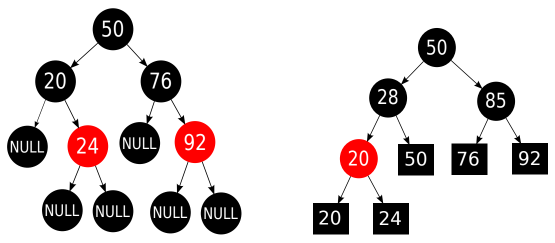
1) A node is either red or black.
2) The root is always black.
3) All leaves are black.
4) Every red node has two black child nodes.
5) Every path from a given node to any of its descendant leaves contains the same number of black nodes.

RBTs are most commonly used as the underlying implementation of the *dictionary* abstract data type, where key-value pairs are stored in the nodes of the tree. Three operations are supported on the set of key-value pairs:

- *Lookup:* Searches for a node containing a given key.
- *Insert:* Adds a node with a given key.
- *Delete:* Unlinks the node containing a specific key from the tree.

B. Implementation details

1) Internal/External: Depending on the underlying organization of the key-value pairs in the nodes of the tree, RBTs are categorized as *internal* or *external*. Internal RBTs store key-value pairs on every node. On the other hand, external RBTs store the values in the leaves, while the internal nodes are used only for routing purposes. An RBT implemented both ways is shown in Figure 1. In the external tree square shapes denote the leaves, which contain the key-value pairs. All the



(a) Internal (b) External

Figure 1. An RBT tree in internal and external format.

other nodes contain only keys and are used for routing to the appropriate leaf.

The two formats entail different requirements and challenges in their implementation. An important one is that in order 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 implementation this operation would require exclusive access to every node between these two nodes. On the other hand, a deletion in external RBTs involves only leaf nodes. To avoid such complex situations we focus in this work on external RBTs.

2) Bottom-Up/Top-Down: The classic RBT implementation [9] is called *bottom-up*. Insertion and deletion consist of two phases: first a traversal of the tree in a top-down manner, i.e. from the root towards the leaves, to locate the place where the node with the new key is going to be inserted or the node that is going to be unlinked from the tree; second, if needed, a traversal of the tree in a bottom-up manner, i.e. from the leaf towards the root, modifying parts of the tree by recoloring and/or rotating nodes to restore the RBT properties and rebalance the tree.

Bottom-up RBTs are very efficient for serial configurations but fine-grained concurrency is very hard to be applied, as multiple threads might traverse the tree in opposite directions acquiring locks in their way, possibly leading to deadlock. Tarjan [10] proposed a *top-down* RBT implementation which performs insertion and deletion in a single top-down pass. To achieve this, while traversing the tree from the root to the appropriate leaf, modifications are proactively performed in order to guarantee that no bottom-up traversal of the tree is required. In this case, in a concurrent configuration, all threads acquire locks in the same direction avoiding the possibility of deadlock. However, as top-down implementations perform typically more tree modifications for each operation, they impose more overhead.

IV. An HTM-based RBT

A. Implementation

HTM is expected to provide a performance boost, with little programming effort, in cases where there is enough parallelism and threads are expected to rarely modify the same data. RBTs are an example of such a data structure as they provide a large number of disjoint paths that can be concurrently modified. The straightforward way to parallelize RBTs with HTM is to enclose each operation of a bottom-up RBT in a single transaction and use a single global lock (SGL) fallback path to guarantee forward progress, as presented in Figure 2. We refer to this implementation as *bu-cg-htm*.

```
 1: int aborts = MAX_TX_RETRIES; /* Maximum number of transactional retries. */
 2:
 3: while (SGL is taken)
 4:     /* Avoid lemming effect. */ ;
 5:
 6: /* Begin transaction. On abort we return here with status != OK. */
 7: int status = TX_BEGIN();
 8: if (status == OK) {
 9:     /* Transactional mode. */
10:     /* Put SGL into read-set, abort if already taken. */
11:     if (SGL already taken) {
12:         TX_ABORT();
13:     }
14:     rbt_insert_key(...); /* Execute operation in transactional mode. */
15:     TX_END(); /* Commit transaction. */
16: } else {
17:     /* Non-transactional fallback path. Executed on abort. */
18:     if (− − aborts > 0) {
19:         goto line 3; /* Retry in transactional mode. */
20:     }
21:     acquire_lock(SGL);
22:     rbt_insert_key(...); /* Execute operation in non-transactional mode. */
23:     release_lock(SGL);
24: }
```

Figure 2. Insertion in *bu-cg-htm*.

The while loop (lines 3–4) is necessary to avoid the lemming effect [11]. In transactional mode (lines 8–15) we first add the SGL into the read set so when it is acquired all concurrent transactions are aborted. If SGL is already taken, we explicitly abort the transaction. Otherwise, we execute the operation and commit the transaction. If a transaction aborts, the fallback path is executed (lines 16–24), where we either retry the operation in transactional mode or, if we have exceeded the maximum number of retries, the SGL is acquired and the operation is executed non-transactionally.

B. Evaluation

For our experiments we have used two dual socket servers, equipped with Intel's Haswell-EP and IBM's Power8 processors. The main characteristics of the two systems are shown in Table II. In all the experiments we first utilize any empty physical cores of each system before employing hyperthreads or SMT contexts.

We compare *bu-cg-htm* against the following concurrent RBTs:

- *bu-cg-lock*: A bottom-up RBT with each operation protected by a single global lock. This version does not scale as it serializes all accesses on the tree and we use it only as a baseline.
- *td-fg-lock*: A top-down RBT [10] on which we have applied fine-grained locking.
- *td-wf*: A wait-free RBT implementation [12] based on Tarjan's [10] top-down approach.

To evaluate the concurrent RBT implementations, we perform random operations varying the number of threads, the

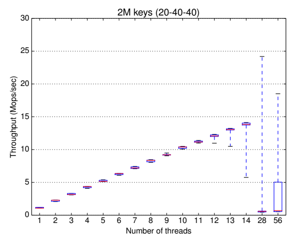
TABLE II
PLATFORMS' CONFIGURATION

Name	**Haswell-EP**	**Power8**
Processors	2 x Intel Xeon E5-2697 v3	2 x Power8
# Cores	2 x 14	2 x 10
# Threads	56	160
Core clock	2.6 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, 128B block size
L3	20-way, 35 MB, 64B block size (shared per die)	8-way, 80 MB, 128B block size (shared per die)
Memory	64 GB	256 GB
OS	Debian 8.1	Ubuntu 14.04
Linux Kernel	4.0.4	3.16.0
GCC	4.9.2 with -O3 optimization	4.9.1 with -O3 optimization

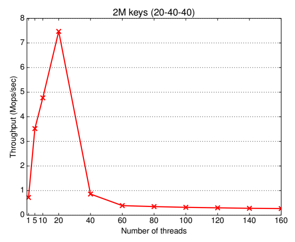
proportion of lookup, insert and delete operations as well as the range of the values that the keys are selected from, in the following way:

- Each run lasts 15 seconds, during which each thread performs randomly chosen operations.
- Each software thread is pinned to a hardware thread.
- Unless otherwise noted, for each transaction we set the number of retries in transactional mode to 10.
- To avoid false conflicts we use padding and alignment so that each node of the tree occupies exactly one cache line.
- To test our implementations under various contention levels we use three workloads, namely 80-10-10, 50-25-25 and 20-40-40, with 80%, 50% and 20% of operations respectively being lookups in the tree, i.e. read-only traversals, while the rest are equally divided between insertions and deletions. These workloads represent a read-dominated, read-write and write-dominated access pattern on the tree respectively.
- As the key range effectively determines the size of the tree, we evaluate our implementations for ranges of 2M, 20M and 100M keys, which represent medium to large-sized trees. At the start of each run the tree is initialized to contain half the keys of the selected range. Due to space limitations, in the rest of the paper we will only provide the results of the 2M key range as the other ranges lead to similar conclusions.
- All reported results are the average of 20 independent executions. In cases where a large variation is observed we provide box plots with median, minimum and maximum values.

Figure 3 presents the throughput achieved by *bu-cg-htm* on our servers. It is evident that the straightforward HTM implementation fails to utilize efficiently the underlying hardware



(a) Haswell-EP



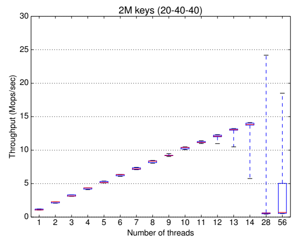
(b) Power8

Figure 3. Total throughput of *bu-cg-htm* on the two servers for the write-dominated workload.

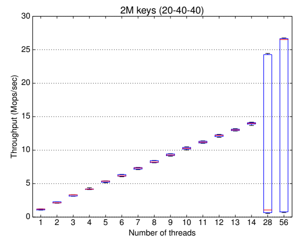
resources, and exhibits different behavior on the two systems. On Haswell-EP, as we increase the number of threads, performance becomes unstable and most of the executions exhibit very low throughput. On the other hand, on Power8, when multiple SMT contexts are employed, performance collapses.

V. Optimizing for Haswell-EP

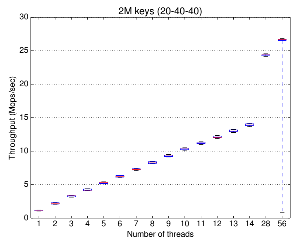
Figure 4(a) presents the results obtained from the execution of *bu-cg-htm*, on Haswell-EP, with the number of transactional retries set to 10. It scales up to 13 threads but after that point it has unstable behavior with high variability on total throughput. Stable performance was achieved by increasing the number of transactional retries. Figures 4(b), 4(c) and 4(d) depict the total throughput for 20, 30 and 50 transactional retries respectively.



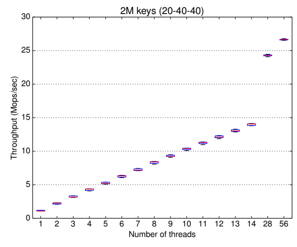
(a) *bu-cg-htm* with 10 retries



(b) *bu-cg-htm* with 20 retries



(c) *bu-cg-htm* with 30 retries



(d) *bu-cg-htm* with 50 retries

Figure 4. Execution of *bu-cg-htm* with 2M key range and write-dominated workload with various number of transactional retries.

Figure 5 presents the performance of all concurrent RBTs on Haswell-EP. The HTM-based RBT scales up to 56 threads and outperforms all other implementations. This is due to the conflict free nature of RBTs and the large transactional buffers of Haswell-EP that permit almost all transactions to commit, thus increasing parallelism.

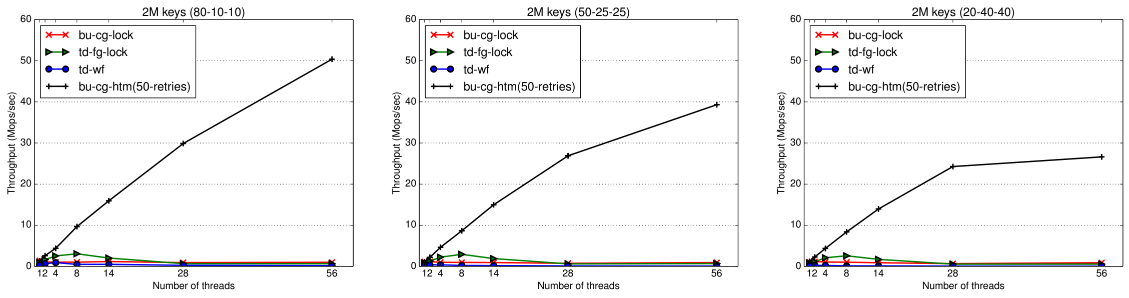


Figure 5. Throughput of concurrent RBT implementations for 2M key range and the three workloads on Haswell-EP.

VI. Optimizing for Power8

Figure 6(a) depicts the throughput achieved by *bu-cg-htm*, on Power8, for the 2M key range and the three operation mixtures. We observe that *bu-cg-htm* scales up to 20 threads, but after that point the performance collapses. To investigate the reasons for this, Figure 6(b) depicts the breakdown of aborts per RBT operation for the write-dominated workload. Up to 20 threads, *bu-cg-htm* suffers close to zero aborts per operation, which allows the performance to scale. For more threads, when we employ more SMT contexts per core, the sharing of the hardware transactional buffers between multiple concurrent transactions causes a huge increase of capacity aborts. Repetitive capacity aborts lead the transactions to the SGL fallback path, which causes the non-transactional conflict aborts.

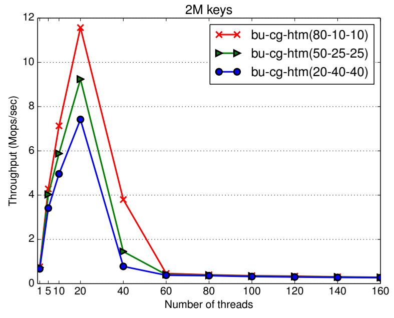
Our first attempt in order to avoid the observed performance degradation was to try different number of transactional retries, similar to Haswell-EP. However, this offered no performance improvement. Therefore, we devised two ways to cope with the hardware limitations of Power8 HTM infrastructure: an alternative to the SGL fallback path and a method to split the large transactions of *bu-cg-htm* to smaller fine-grained ones.

A. Per-cpu lock fallback

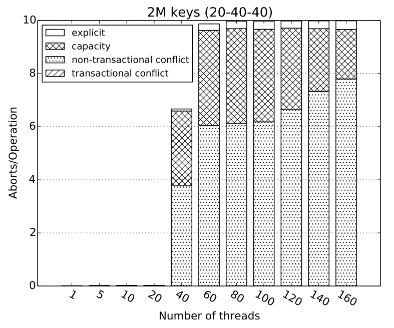
As explained, the HTM buffers of Power8 are large enough to fit a single transaction of *bu-cg-htm* but cannot support multiple transactions from multiple concurrent SMT threads. When using the SGL fallback path, repetitive capacity aborts lead to the acquisition of SGL, aborting all concurrent transactions, even those that execute on different cores and do not share resources with the one that resorts to the SGL fallback path. Here, we implement and evaluate a different approach which we refer to as *per-cpu lock (PCL)* fallback.

The main characteristics of our proposed PCL fallback path are the following:

- Apart from SGL, we maintain a global array of locks, with one entry for each physical core. Each lock entry is padded to the size of the cache line to avoid false sharing.
- When a transaction begins, it adds to its read set the SGL together with the corresponding PCL, i.e. the lock for the physical core where the transaction is executed. If either of them is owned by another thread, the transaction explicitly aborts.



(a) Total throughput.

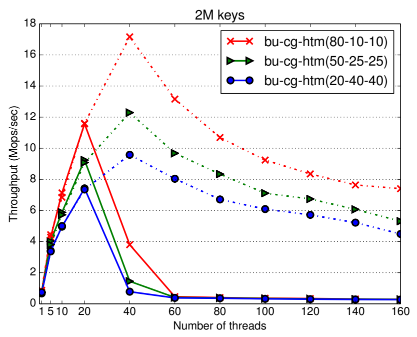


(b) Breakdown of aborts per RBT operation.

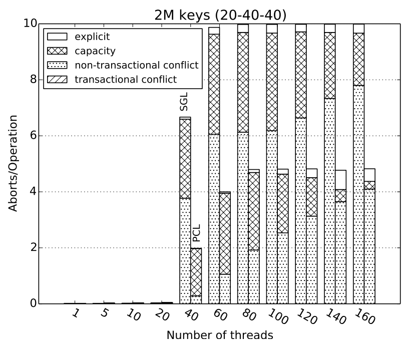
Figure 6. Performance characteristics of *bu-cg-htm* on Power8.

- When repetitive capacity aborts occur, the corresponding PCL is acquired causing other transactions executing on the same core to abort. The transaction is then retried, while the rest of the SMT contexts wait for it to complete. It is essential to retry in transactional mode to ensure consistency with concurrent transactions running on other cores.
- If capacity aborts still occur after the acquisition of the PCL, we resort to the SGL fallback path and the operation is executed in non-transactional mode.
- If a transaction aborts for any other reason, we resort to the SGL fallback path.

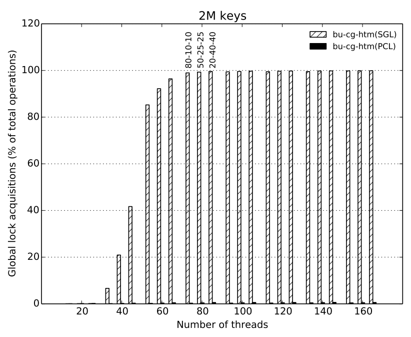
In order for the PCL to be effective, threads need to be pinned at hardware threads. Otherwise, a thread that has acquired the PCL of core C1 could be migrated by the OS scheduler to another core C2. This would allow other threads to execute on core C2 while core C1 would remain idle.



(a) Throughput (Solid lines: SGL, Dotted lines: PCL).



(b) Breakdown of aborts.



(c) Percentage of non-transactional operations.

Figure 7. Performance characteristics of *bu-cg-htm* with PCL fallback.

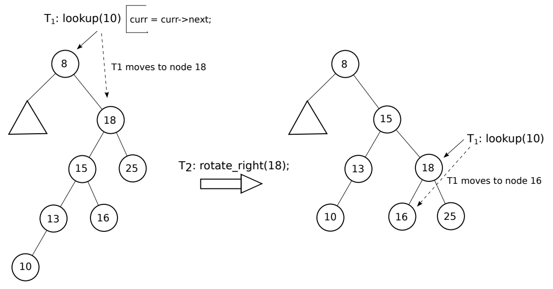
The PCL fallback allows a transaction to execute using all the available HTM resources of a core without affecting threads executing on other cores. The dotted lines of Figure 7(a) show the achieved throughput for the three different workloads when the PCL fallback is used. The PCL is acquired after 5 capacity aborts and if 5 more occur we resort to SGL. Figure 7(b) presents the breakdown of aborts for 2M key range and the write-dominated workload for both SGL and PCL cases. It is evident that PCL decreases the aborts per operation, which in turn results in performance gains. Finally, Figure 7(c) depicts the operations percentage that resort to SGL acquisition and execute in non-transactional mode for all workloads. With PCL, almost every operation completes in transactional mode.

However, we still fail to utilize efficiently all the available SMT threads, as capacity aborts in a core lead to only one of the SMT contexts operating in transactional mode, while the others wait for it to finish. Therefore, our implementation with PCL fallback still fails to scale for more than 40 threads, although it manages to provide better throughput than when using the SGL fallback.

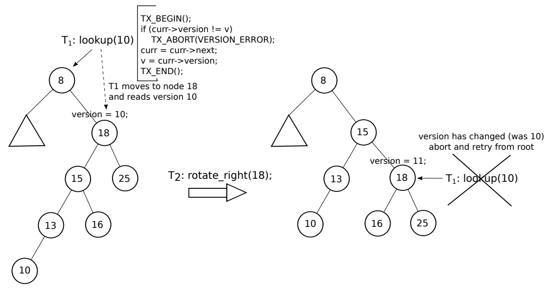
B. Fine-grained transactions

To achieve full SMT thread utilization and scalability we need to split the large transactions of the coarse-grained HTM RBT to a sequence of smaller fine-grained ones.

The larger portion of an RBT operation is the lookup phase, i.e. the top-down traversal that starts from the root and ends at the appropriate leaf. Therefore, it makes sense to try and split this phase in several transactions. Doing so while preserving the correctness of the traversal is challenging. To clarify this, we need to explain why synchronization is required in the lookup phase. In general, a thread traversing the tree performs a set of distinct steps until a leaf is reached. At each step, the thread holds a reference to the node that it currently examines (the current node) and forwards the current node variable to point to the next appropriate child. If no synchronization is used, a concurrent rotation around the current node could lead the traversing thread to a wrong subtree (rotations on upper or lower levels of the tree do not affect the correctness). Figure 8(a) illustrates such an erroneous traversal. Thread T1



(a) A lookup driven to the wrong direction by a concurrent rotation.


```
TX_BEGIN();
if (curr->version != v)
    TX_ABORT(VERSION_ERROR);
curr = curr->next;
v = curr->version;
TX_END();
```

(b) Concurrent rotation causes fine-grained transactions to retry the operation.

Figure 8.

is in the middle of a lookup for key 10. It is currently at node with key 8 and moves to the node with key 18. At that point thread T2 performs a right rotation around node 18 as a result of an insertion or deletion. Thread T1 is unaware of that rotation and ends up traversing the wrong subtree.

Coarse-grained HTM RBT avoids such situations by adding every traversed node in the transaction's read-set. This way, when a rotation is performed that includes a traversed node, the transaction aborts and the traversal restarts from the root. However, this approach has two disadvantages: first, the footprint of the transaction is too large and causes a high number of capacity aborts; second, a conflict is detected not only when the rotation affects the current node, but also nodes that have been previously traversed on upper levels. As rotations on these levels of the tree do not affect the correctness of the traversal, such conflict aborts could be avoided.

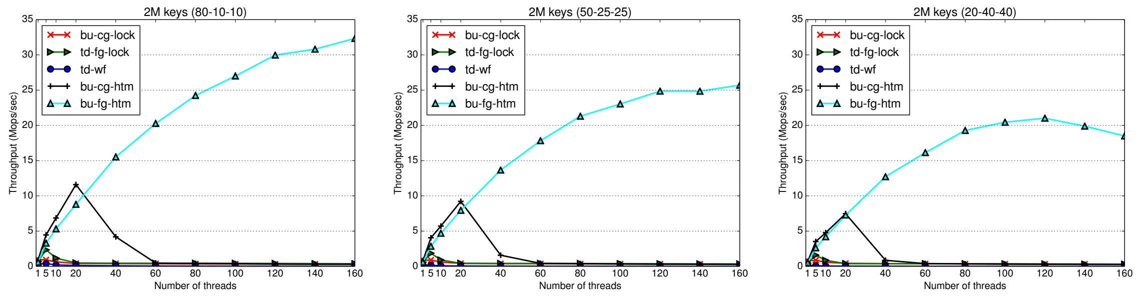


Figure 9. Throughput of concurrent RBT implementations for 2M key range and the three workloads on Power8.

Here we describe a way to split the large traversal phase into a series of small fine-grained transactions while preserving the correctness of the operation. Our approach is similar to the hand-over-hand validation [13], but we employ HTM instead of locking to isolate concurrent modifications. Each node of the tree has a version number, which is increased every time the node is modified by a rotation. To search for a key in the tree, a thread starts by executing an HTM transaction, which sets the local current node variable to point to the root of the tree and reads the root's version. Then, the following steps are performed until a leaf is reached:

1) *Begin an HTM transaction and validate that the version of the current node is the same as the version read by the previous transaction*. If the validation fails the transaction explicitly aborts with a code that indicates a version error and the operation restarts from the root of the tree.
2) *Move to the next node in the access path*. This step can be extended to traverse several nodes in a single transaction. This way we can control the granularity and consequently the footprint size of each transaction.
3) *Read the version of the reached node in a local variable to be used for validation from the next transaction and commit the transaction.*

Figure 8(b) shows an example of a fine-grained tree traversal that is aborted as soon as it realizes that its current node's version has been modified, signaling that the traversal leads to the wrong path.

For insertion and deletion, after the appropriate leaf is reached, we execute a final HTM transaction, in which the insertion (or deletion respectively) of the new node is performed along with the bottom-up rebalance phase. The footprint size of this last transaction is proportional to the number of tree levels that are modified during the rebalance phase. To investigate how large the footprint size can be, we have conducted a series of benchmarks on RBTs with different sizes. The results are presented in Table III. For every tree size, around 75% of the rebalance operations modify only 1 tree level and almost 97% of them less than 3. This indicates that the final transaction typically has small footprint.

Our retry policy for the fine-grained transactions is the following:

- To ensure correct execution of the traversal, when a transaction explicitly aborts due to a version validation error, the traversal restarts from the root of the tree.
- To guarantee forward progress, when a given threshold of aborts, other than validation errors, is encountered, the traversal restarts from the root of the tree. When an operation has restarted for a given number of times, we resort to the SGL fallback path and execute the operation in non-transactional mode.

TABLE III
DISTRIBUTION OF REBALANCE PHASES BASED ON NUMBER OF MODIFIED TREE LEVELS.

Tree Levels	**Key Range**			
	2M	**20M**	**100M**	**200M**
1	77.12%	75.96%	73.37%	73.08%
2	19.52%	18.66%	18.73%	18.79%
3	2.78%	3.69%	5.30%	5.47%
>3	0.56%	1.69%	2.59%	2.66%

Figure 9 shows the achieved throughput of all concurrent RBT implementations on Power8, including the fine-grained transactions version (*bu-fg-htm*) and the *bu-cg-htm* with PCL fallback. For each transaction of *bu-fg-htm* we have set the threshold before restarting the operation from the root to 10. An operation is restarted maximum 10 times before resorting to the SGL fallback path. It is evident that *bu-fg-htm* manages to overcome the limitations imposed by the Power8 HTM infrastructure and scale up to 160 threads, or 120 for the write-dominated workload. For up to 20 threads, transactions fit in the HTM buffers and coarse-grained HTM performs better than the fine-grained version. This is due to the overhead imposed by executing a large number of small transactions.

Splitting the large transactions of *bu-cg-htm* to smaller fine-grained ones has the advantage of keeping the footprint of the transactions small, independently of the size of the tree. To further illustrate this, Table IV presents the aborts per operation of the two implementations for the write-dominated workload and for all tree sizes. The coarse-grained implementation suffers much more aborts per operation than fine-grained, which has only 1 or less aborts per operation for every tree size.

TABLE IV
ABORTS PER OPERATION FOR COARSE & FINE-GRAINED HTM RBT.

#Threads	*bu_cg_htm*			*bu_fg_htm*		
	2M	**20M**	**100M**	**2M**	**20M**	**100M**
20	0.02	0.23	0.27	0.001	0.002	0.007
40	6.2	9.7	9.9	0.001	0.003	0.007
60	9.8	9.9	9.9	0.002	0.004	0.01
80	9.9	9.9	9.9	0.009	0.008	0.012
120	9.9	9.9	9.9	0.24	0.17	0.11
160	9.9	9.9	9.9	1.15	1.07	0.85

VII. RELATED WORK

With the advent of HTM support on commercial processors, several researchers have evaluated their performance using a limited number of hardware threads [14], [4], [2], [6], [15]. Our work extends these evaluations using a significantly higher number of threads and especially [14] by providing specific optimizations for different hardware platforms.

The importance of tuning the number of transactional retries on Intel Haswell has been noted in several papers [2], [16], [15]. However, while all previous efforts aim at gaining performance, we have found that robustness can also be achieved.

COP [17] and ParT [18] are generic methods for splitting large transactions in smaller fine-grained ones, and use RBT as a use case. They both employ an unsynchronized lookup phase followed by an HTM transaction that validates the lookup phase's outcome and updates the RBT. This is a *read-validate-update* approach. Our method, on the other hand, validates the lookup phase step-by-step. In the future we plan to compare our work with these two approaches.

VIII. CONCLUSION

Transactional Memory has emerged as an attractive alternative to lock-based approaches. Its main goal is to simplify concurrent programming while at the same time providing high performance. Our evaluation validates that a straightforward HTM implementation of a classic data structure, red-black tree, can outperform lock-based and wait-free alternatives. However, to enable scalability to high numbers of threads, the programmer needs to be aware of the underlying HTM system's limitations and optimize the code appropriately.

Our evaluation on two HTM systems with different characteristics has led us to different sets of optimizations for each one. Specifically, on the Intel Haswell-EP server we only tuned the number of transactional retries to achieve scalability and robustness for up to 56 threads. On the other hand, on Power8 we dealt with the problem of capacity aborts due to the limited transactional buffers and we proposed two methods to overcome these limitations.

ACKNOWLEDGMENT

This research was supported 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). We would also like to thank Intel Corporation and IBM Hellas for kindly providing the two servers on which we have evaluated our implementations.

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] "z/architecture principle of operations. sa22-7832-09."

[6] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari, "Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, (New York, NY, USA), pp. 144–157, ACM, 2015.

[7] H. Le, G. Guthrie, D. Williams, M. Michael, B. Frey, W. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, pp. 8:1–8:14, Jan 2015.

[8] 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.

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

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

[11] A. Kleen, "Tsx anti patterns in lock elision code."

[12] A. Natarajan, L. Savoie, and N. Mittal, "Concurrent wait-free red black trees," in *Stabilization, Safety, and Security of Distributed Systems* (T. Higashino, Y. Katayama, T. Masuzawa, M. Potop-Butucaru, and M. Yamashita, eds.), vol. 8255 of *Lecture Notes in Computer Science*, pp. 45–60, Springer International Publishing, 2013.

[13] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 257–268, ACM, 2010.

[14] D. Siakavaras, K. Nikas, G. Goumas, and N. Koziris, "Performance analysis of concurrent red-black trees on htm platforms," *TRANSACT*, 2015.

[15] N. Diegues, P. Romano, and L. Rodrigues, "Virtues and limitations of commodity hardware transactional memory," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, (New York, NY, USA), pp. 3–14, ACM, 2014.

[16] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *11th International Conference on Autonomic Computing (ICAC 14)*, (Philadelphia, PA), pp. 209–219, USENIX Association, June 2014.

[17] L. Xiang and M. L. Scott, "Software partitioning of hardware transactions," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 76–86, ACM, 2015.

[18] H. Avni and B. C. Kuszmaul, "Improving htm scaling with consistency-oblivious programming," *TRANSACT*, vol. 6, p. 5, 2014.