An HTM-Based Update-side Synchronization for RCU on NUMA systems

SeongJae Park
Amazon
sjpark@amazon.com

Paul E. McKenney
Facebook
paulmck@kernel.org

Laurent Dufour
IBM Linux Technology Center
ldufour@linux.ibm.com

Heon Y. Yeom
Seoul National University
yeom@snu.ac.kr

Abstract

Read-copy update (RCU) can provide ideal scalability for read-mostly workloads, but some believe that it provides only poor performance for updates. This belief is due to the lack of RCU-centric update synchronization mechanisms. RCU instead works with a range of update-side mechanisms, such as locking. In fact, many developers embrace simplicity by using global locking. Logging, hardware transactional memory, or fine-grained locking can provide better scalability, but each of these approaches has limitations, such as imposing overhead on readers or poor scalability on non-uniform memory access (NUMA) systems, mainly due to their lack of NUMA-aware design principles.

This paper introduces an RCU extension (RCX) that provides highly scalable RCU updates on NUMA systems while retaining RCU's read-side benefits. RCX is a software-based synchronization mechanism combining hardware transactional memory (HTM) and traditional locking based on our NUMA-aware design principles for RCU. Micro-bench-marks on a NUMA system having 144 hardware threads show RCX has up to 22.6 times better performance and up to 145 times lower HTM abort rates compared to a state-of-the-art RCU/HTM combination. To demonstrate the effectiveness and applicability of RCX, we have applied RCX to parallelize some of the Linux kernel memory management system and an in-memory database system. The optimized kernel and the database show up to 24 and 17 times better performance compared to the original version, respectively.

CCS Concepts • **Software and its engineering → Concurrency control**; *Operating systems*; Virtual memory.

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04.
https://doi.org/10.1145/3342195.3387527

Keywords RCU, synchronization, transactional memory, parallel programming, operating systems

ACM Reference Format:
SeongJae Park, Paul E. McKenney, Laurent Dufour, and Heon Y. Yeom. 2020. An HTM-Based Update-side Synchronization for RCU on NUMA systems. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387527

1 Introduction

RCU can provide ideal read-side performance and scalability, further providing concurrent forward progress among readers and updaters [48], in contrast to traditional read-centric synchronization mechanisms such as reader-writer locks [10]. However, RCU imposes relatively heavy overhead on updates and further defers to other synchronization mechanisms to synchronize between concurrent updates. This design choice allows RCU to work with whatever update-side synchronization mechanism is the best for the case at hand, or even to work with an ad-hoc mechanism devised for a particular special case [47]. The downside of this tradeoff is the lack of clear guidelines. As a result, many RCU users use simple synchronization mechanisms, such as global locking, which do not scale well, thus leading some to incorrectly believe that poor update-side scalability is inherent to RCU.

One response to this situation has been to develop widely used RCU-protected concurrent data structures, including hash tables [7, 58] and binary trees [1]. These data structures have been successfully adopted, for example by a distributed file system [6] and by an in-memory database [59]. That said, hash tables and search trees do not always suffice: Ad-hoc data structures are often required.

Another response makes use of techniques similar to software transactional memory (STM) such as read-log update (RLU) [41, 47], hardware transactional memory (HTM) [23, 34, 39, 49, 51], and HTM in conjunction with RCU [55]. In Sections 2.2 and 2.3 we show that these approaches suffer from several limitations, including scalability limitations due to NUMA-obliviousness and due to naïve fallback mechanisms for HTM failures. We revisit the straightforward conjunction of RCU and fine-grained NUMA-aware scalable

locks [11–13, 35], which emphasizes the problem of the TM based approaches. This revisitation is covered by Sections 2.3 and 4.2.

To mitigate these problems, we introduce another synchronization mechanism combining HTM and locking, along with an RCU extension called RCX that incorporates this mechanism. RCX's advantages stem from its five design principles for read-mostly workloads on HTM-enabled NUMA systems, which is summarized in Table 1.

1.	Do fine-grained synchronization.
2.	Use pure RCU read mechanism for the nearly-ideal read performance.
3.	Utilize HTM for minimal overhead.
4.	Isolate the working set of HTM from dominating readers.
5.	Avoid the use of HTM across remote NUMA nodes.

Table 1. The five design principles of RCX.

Fundamentally, RCX is a fine-grained synchronization scheme. Reader threads use RCU, thus attaining RCU's traditional ideal performance and scalability. It utilizes HTM-based synchronization for minimal overhead but isolates the meta-data for update-side synchronization from readers to eliminate HTM aborts induced by large numbers of readers. It also restricts HTM to threads in the same NUMA node to limit HTM abort rates. Updater threads within a given NUMA node compete using HTM, and then winners from each node compete using traditional locking. This traditional locking scales well in RCX due to the limited number of contending threads.

To evaluate our approach in detail, we implemented linked lists and hash tables protected by various RCU extensions, including state-of-the-art mechanisms and RCX. In addition to illustrative micro-benchmarks, we applied RCX to a portion of the Linux-kernel virtual memory system and a popular in-memory database system to obtain a realistic system-level RCX evaluation.

In short, the contributions of this paper can be summarized by three main points. This paper

- investigates state-of-the-art RCU extensions on NUMA systems (Sections 2 and 3.1),
- introduces our update-side synchronization mechanism for RCU on NUMA systems (Section 3.2)
- and presents adoption of the mechanism to data structures, a virtual memory system, and a database system (Sections 3.3 and 4).

```
1 rcu_read_lock();
2 p = rcu_dereference(head);
3 do_something(p);
4 rcu_read_unlock();
```

Listing 1. RCU read critical section.

```
 1 void insert(node, prev, next)
 2 {
 3     node->next = next;
 4     rcu_assign_pointer(prev->next, node);
 5 }
 6
 7 void remove(node, prev, next)
 8 {
 9     rcu_assign_pointer(prev->next, next);
10     synchronize_rcu();
11     kfree(node);
12 }
```

Listing 2. RCU update code for a linked list (single updater thread).

2 Background and Related Work

Because power-consumption and heat-dissipation issues have limited CPU core clock frequencies [44], systems with hundreds of cores have been common, pushing parallel programming to the fore. However, Amdahl's Law dictates scalability limitations for real applications. Furthermore, the use of expensive synchronization primitives [3, 19] often causes synchronization costs to increase with the number of CPUs [29, 35]. No single state-of-the-art concurrency control mechanism works best for all workloads [28].

2.1 Read-Copy Update

RCU protects shared data from concurrent threads based on two core concepts. **1) RCU read-side critical sections:** A region of code enclosed by `rcu_read_lock()` and `rcu_read_unlock()`. RCU guarantees accesses to RCU-protected data from within such a region is safe. **2) Grace periods:** The time until every *pre-existing* RCU read-side section has completed. RCU provides a `synchronize_rcu()` primitive that waits for a grace period to complete.

Example pseudocode for reads from RCU-protected data is shown in Listing 1. The aforementioned `rcu_read_lock()` and `rcu_read_unlock()` calls are shown on lines 1 and 4 respectively. The `rcu_dereference()` call on line 2 suppresses compiler and hardware (in the case of DEC Alpha [20]) optimizations that might otherwise allow readers to access pre-initialization garbage in newly added data items. RCU read-side critical sections should be short and must not block, though some RCU variants permit blocking, such as SRCU [43] and userspace RCU [21].

Example pseudocode for RCU updates is shown in Listing 2, which provides update-side synchronization via a

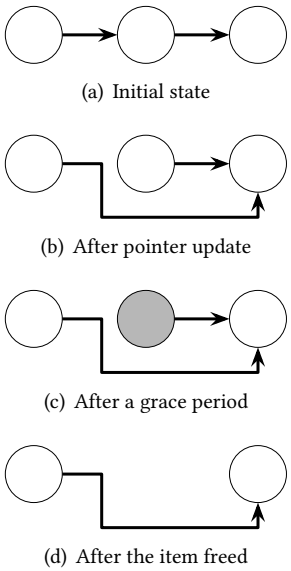


Figure 1. RCU-protected data item removal.

single updater thread. The `insert()` function on lines 1-5 initializes the new item (line 3) then uses `rcu_assign_pointer()` (line 4) to link the new item into a linked list. This function is similar to a store-release operation [9], ensuring that the initialization will take place before the linking, and also that concurrent readers will see either the old pointer or the new one, but either way ensuring that they see a properly formatted linked list.

The `remove()` function on lines 7-12 of the listing removes an item from a linked list, whose initial state is shown in Figure 1(a). The `rcu_assign_pointer()` function on line 9 redirects the first item's `->next` pointer to the third item, as shown in Figure 1(b). The newly removed item cannot be safely freed because pre-existing RCU readers might still have a reference to it. The `synchronize_rcu()` function on line 10 waits for pre-existing readers, after which no reader has a reference to the newly removed item, as shown in Figure 1(c). At this point, it is safe for line 11 to invoke `kfree()`, resulting in the state shown in Figure 1(d).

Another widely used read-mostly synchronization mechanism is reader-writer locking [10, 18], which provides mutual exclusion such that readers exclude updaters, but not other readers, and updaters exclude both readers and other updaters. This can degrade the performance of read-mostly workloads because readers must wait for updaters. In contrast, RCU readers never wait at all. Further, in server-class (preemption-disabled) builds of the Linux kernel running on systems other than DEC Alpha, RCU readers might execute exactly the same sequence of machine instructions that would be executed by a single-threaded reader. RCU readers therefore enjoy the ideal performance and linear scalability, besting all other synchronization mechanisms.

In contrast, the update-side code in Listing 2 does require synchronization to avoid corrupting the linked list. RCU does not provide update-side synchronization mechanisms, instead requiring that users choose some other mechanism to synchronize their concurrent updates. This design choice allows users to choose (or invent) the best synchronization mechanism for their specific situation. However, users often use simple mechanisms such as global locking, so that RCU algorithms often have poor update-side performance and scalability [37, 41, 47]. The code in the listing is even simpler, having chosen the trivial synchronization solution of a single updater thread.

2.2 Hardware Transactional Memory

HTM allows programs to group multiple memory accesses into atomic transactions, which in turn provides a simple program environment and high performance compared to that of software-based synchronization mechanisms. Several CPU vendors are supplying HTM-enabled products, either in production [34, 49, 51] or as a research prototype [23]. Those have been successfully adopted by several workloads [39, 55, 60–63]. Among those, we use Intel's implementation [50].

Intel's HTM maintains a read data set and a write data set in the CPU cache and checks for data conflict between CPUs based on the cache coherency protocol. If a transaction is writing a data item that is read or written by other concurrent CPUs, HTM aborts and rolls back the intermediate changes of transactions involved in the conflict. Intel suggests HTM users use HTM in two ways, hardware lock elision (HLE) and restricted transactional memory (RTM). HLE replaces memory operations for lock acquisition and release with HTM operations and falls back again to software-based traditional locking if the transaction is aborted. HLE is easy to adopt because only the source code for the lock implementation needs modification. RTM is more analogous to traditional transaction systems. Users should explicitly use HTM transactions and provide fallback mechanisms for aborts. In short, compared to HLE, RTM is harder to adopt but is highly optimizable.

Although HTM normally incurs lower overhead and scales better compared to software-based synchronization mechanisms, careless uses of HTM can result in insufficient or even worse performance because most HTM implementations in real products have many restrictions [39], including transaction size limitations and unexpected aborts for various reasons.

2.3 Related Work

Ramadan *et al.* present TxLinux, which applies transactions to an early version of the Linux kernel [52–54]. To transactionalize the Linux kernel as opposed to interacting with RCU, they took the pragmatic approach of aborting any transactions containing accesses conflicting with an RCU

read-side critical section. This work did not involve a commercial HTM implementation and did not address NUMA performance issues.

Howard *et al.* present an STM that is enhanced to accommodate RCU readers, which they call "relativistic readers" [30, 32]. They extended SwissTM [25] to commit writes in program order and to respect ordering directives, including memory barriers and RCU grace-period-wait operations. As far as we know, this is the first work permitting concurrent forward progress among RCU readers and TM writers. However, it does not involve HTM, and nor address NUMA issues.

McKenney describes a number of additional strategies for interfacing TM to RCU readers [44, Section 16.2.3.3], including delaying RCU readers to avoid conflicts, converting RCU readers to transactions, and finally restricting RCU to linked structures and allowing RCU readers to see pre-commit values. There is also the null strategy of prohibiting combining TM with RCU.

Matveev *et al.* [41] introduce an RCU-like mechanism called read-log-update (RLU) which is an STM with explicitly marked readers. RLU provides good update-side performance and reasonable scalability, at least for an STM [47]. RLU also provides the simple programmability expected from an STM but with the added requirement that the developer explicitly marks RLU readers. However, RLU incurs read-side overhead that is unacceptable for some workloads [47], a result corroborated by our work. A recently proposed multi-version variant of RLU improves update performance, but still lacks read-side improvements [37].

Siakavaras *et al.* [55] developed a concurrent binary search tree that is protected by a combination of RCU and HTM (RCU-HTM). Because RCU-HTM readers are the same as those of RCU, RCU-HTM enjoys excellent read-side performance and scalability. Updaters synchronize among themselves and readers using HTM. Updaters retry upon transaction failure, but if the number of retries exceeds a predefined limit, the updater falls back to locking. Updaters thus avoid locking if their transactions succeed and in that happy situation provide good performance and scalability. It also provides simple update-side programmability. However, we will show that RCU-HTM does not address NUMA performance issues.

Although some researchers evaluate RCU only with globally locked updates [41, 55], fair evaluations include fine-grained locking [48], non-blocking synchronization [21, 42], and TM [31, 52, 55]. Some claim that RCU is hard to use [37, 41], but others note that RCU is no harder than other fine-grained synchronization mechanisms [47, 55]. In practice, RCU eases creations of highly performant and scalable subsystems in the Linux kernel [4]. Where read-side consistency is required (e.g., atomic updates of multiple objects), RCU can easily provide it [2, 45, 46]. Also, NUMA-aware scalable locking [11–13, 24, 35] has progressed over several decades and

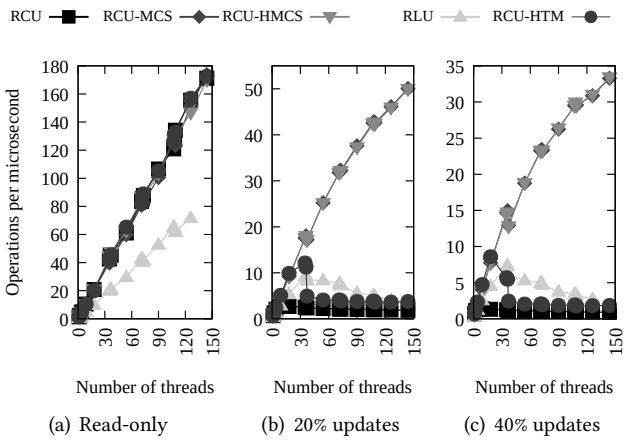


Figure 2. Performance of RCU, RLU, and RCU-HTM for linked lists with 256 initial nodes.

is widely used, including in the Linux kernel [17]. This suggests highly scalable RCU updates with fine-grained locking on NUMA systems, which is covered by this paper.

2.4 Existing Approaches on NUMA Systems

Figure 2 compares the performance of RCU adopting a global lock (RCU), fine-grained NUMA-aware scalable locks (RCU-MCS and RCU-HMCS for MCS and HMCS lock, respectively), RLU, and RCU-HTM on linked lists for a varying number of threads for read-only, 20%, and 40%-update workloads. We choose MCS lock and HMCS lock among various NUMA-aware locks because MCS lock is already adopted in the Linux kernel [17] and HMCS is a hierarchical version of MCS that is known to perform well in many cases [12]. Note that we represent the RCU utilizing a global lock as RCU simply because other closely related works did [41, 55]. The evaluation was performed on an Intel Xeon system with 4-way 18-core hyperthreading for a total of 144 hardware threads. To limit contention between the 144 threads, we insert 256 initial nodes in the list before starting each of the workloads. This allows a reasonable probability of each thread accessing a different target node, thus allowing the fine-grained synchronization mechanisms to scale to 144 threads. RLU results for 144 threads are omitted due to a bug in the RLU original implementation we retrieved. For a detailed implementation and evaluation setup, see Sections 3.3 and 4.1.

Figure 2(a) shows that RLU fails to provide a good read performance due to its log-lookup overhead, which discourages the use of RLU for read-mostly workloads. Contrarily, other variants provide almost ideal performance and scalability. Figures 2(b) and 2(c) show the update-heavy scalability of variants. RCU shows the worst scalability due to the single update-side lock. RCU-HTM scales best until 19 threads but collapses after 36 threads. RLU's scalability degradation

is less dramatic but still decreases to that of RCU-HTM at 126 threads. Both of MCS and HMCS show similar performance because HMCS's benefit becomes subtle under a low contention, which is natural for the read-mostly workloads. These NUMA-aware locks show the best performance, which is even much higher than those of state-of-the-art RCU extensions (RLU and RCU-HTM). In other words, the state-of-the-art extensions perform worse mainly due to their NUMA-obliviousness. This also means that a NUMA-aware use of HTM would achieve better performance. How could it be optimized and how big the improvement will be? These are the questions this paper answers.

3 An RCU Extension for NUMA Systems

The following sections describe the root causes of poor RCU-HTM performance, then describe the design and implementation of a novel improved mechanism.

3.1 Root Cause of RCU-HTM Performance Degradation

HTM users should minimize the number of aborts and provide an efficient fallback mechanism because frequent transaction aborts degrade performance. However, RCU-HTM neither provides an efficient fallback mechanism nor minimizes aborts for NUMA systems. If an HTM transaction aborts, RCU-HTM retries the transaction up to a predefined limit (which defaults to 10), and then falls back to a global locking. This fixed limit cannot accommodate all dynamic workloads, and the global locking-based fallback does not scale well.

We therefore investigated three straightforward fallback mechanisms for RCU-HTM. The first one (FORGIVE) just forgives the failure, so that if the transaction aborts, a failure indication is returned to the user and processing resumes with other operations. We can expect this mechanism to provide near-optimal performance because it imposes nearly zero fallback overhead. Of course, it has the disadvantage of forcing applications to handle transaction aborts. The second one (RETRY) unconditionally retries the transaction until it succeeds. This mechanism will have reasonable performance if transaction abort rates are low, which of course requires that all transactions fit within any applicable hardware constraints. The third fallback mechanism (HWA) bases its retry decision on HTM feedback, retrying if this feedback indicates that a retry is likely to be successful, and otherwise falling back to locking. Please note that these are designed for experimental purposes for HTM on NUMA systems, not for real use.

Figure 3 shows throughput and the number of HTM aborts per 100 updates of the variants for linked lists with 256 initial nodes. Note that the y-axis for abort rates is in log-scale. The performance of all variants is similar in the read-only case (Figure 3(a)) because this case uses RCU without HTM,

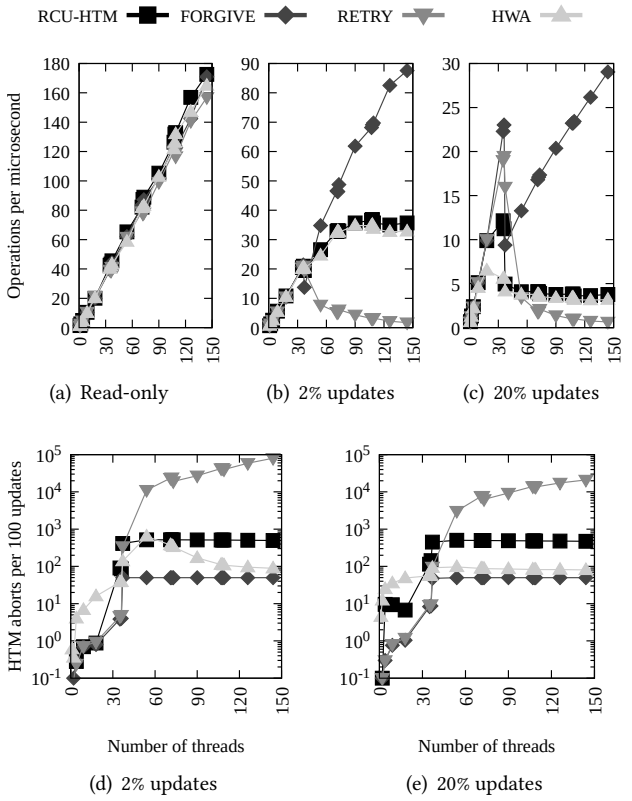
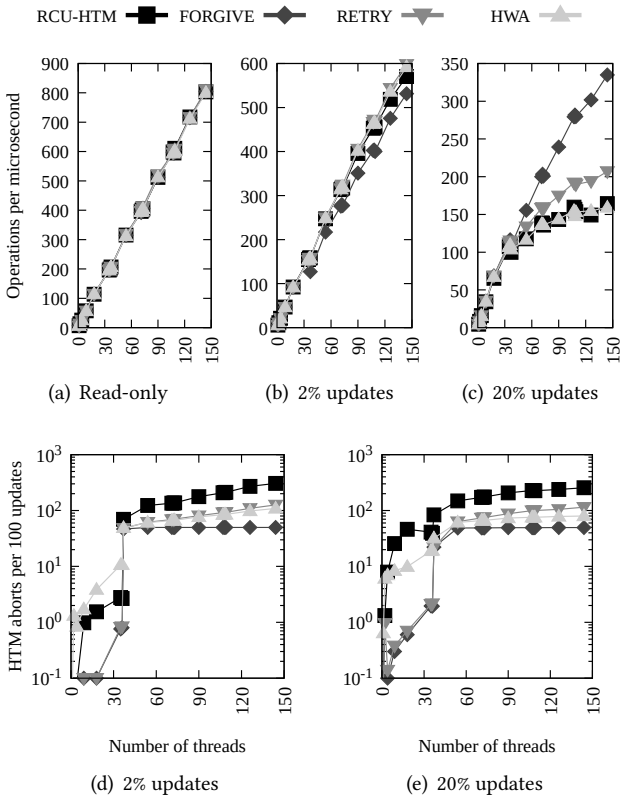


Figure 3. Performance and HTM abort rates of RCU-HTM variants utilizing different fallback mechanisms for linked lists with 256 initial nodes.

and thus without any HTM aborts. In cases with updates and a large number of threads, FORGIVE performs the best, followed by RCU-HTM and HWA. RETRY has the worst performance due to high abort rates. The gap of abort rates between FORGIVE and RETRY explains their performance difference. Contrarily, the abort rates gap between RCU-HTM and HWA does not explain their similar performance. RCU-HTM slightly outperforms HWA despite HWA having 5 times lower abort rates with 144 threads. This result is due to the use of global locking as the fallback mechanism, which is expensive and non-scalable. In short, HWA's reduced abort rate is overshadowed by the high cost of the global-locking fallback.

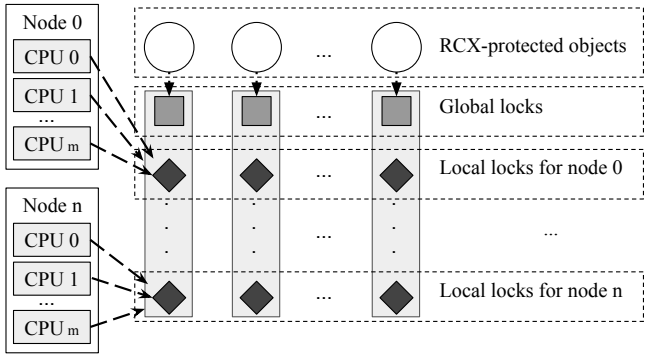
Figure 4 also shows the performance and the number of HTM aborts per 100 updates, but for hash tables with 128 buckets, each initially containing eight nodes. Conflict probability has decreased from that of the single linked list because the total number of data items has increased from 256 to 1024, greatly improving the performance and scalabity of all four variants. Also, per-bucket locking greatly reduces the performance penalty incurred by the fallback over that of the globally locked linked list, providing better scalability for variants using fallbacks (that is, all but RETRY), even at 20%



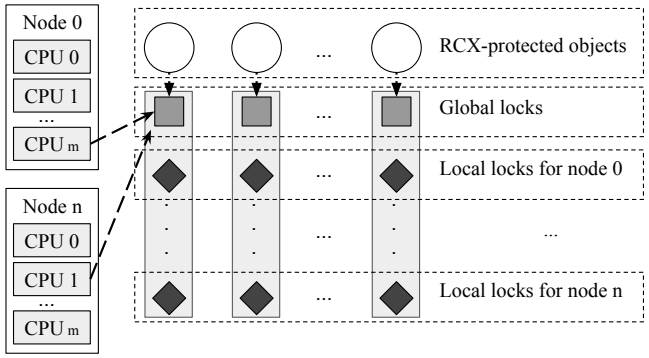
(a) Read-only (b) 2% updates (c) 20% updates

(d) 2% updates (e) 20% updates

Figure 4. Performance and HTM abort rates of RCU-HTM variants utilizing different fallback mechanisms for hash tables with 128 buckets and 8 initial nodes per bucket.



(a) Per-NUMA node winner thread election



(b) Global locking

Figure 5. Hierarchical synchronization of RCX.

update rates. FORGIVE still performs the best, but RETRY outperforms RCU-HTM and HWA when updates induced.

In both figures, the performance scaling of every variant except FORGIVE suffers from poor performance and increased abort rates beyond 36 threads. Because threads are placed to occupy a minimal number of NUMA nodes (see Section 4.2 for detail), this means that scalability degrades as soon as multiple NUMA nodes are in use. The abort rates increase due to higher communication latencies among CPUs in different NUMA nodes. These higher latencies increase transaction duration, in turn increasing abort probabilities. Similar findings have been reported [8], but this is the first investigation from the perspective of read-mostly situations.

Interestingly, RETRY and RCU-HTM abort rates increase with *decreasing* update rates. For example, abort rates of RETRY for lists using 144 threads with 2% updates and 20% updates are 80,378 and 20,909 per 100 updates, respectively. This is because reads are non-transactional, thus unconditionally aborting concurrent updates. Therefore, beyond a certain point, increasing the probability of reads (thus decreasing the update rate) increases the abort rate.

We have identified three root causes of poor RCU-HTM performance. First, high HTM abort rates and expensive fallbacks degrade performance. Second, HTM abort rates increase rapidly for transactions spanning multiple NUMA nodes. Third, read-intensive workloads can result in high update-side HTM abort rates.

3.2 Design of RCX

This section describes the design of RCX, which addresses the root causes identified in the previous section. Threads reading RCX-protected objects use RCU read-side primitives (Listing 1), thus minimizing read-side overhead.

RCX updater threads synchronize via per-object meta-data comprising one global lock and a set of per-NUMA node locks, all of which are cache aligned to avoid false sharing. This updater synchronization is carried out in two phases. The first phase elects a winner thread from each NUMA node (Figure 5(a)), the winner being the thread holding the HTM-based per-NUMA-node locks on that thread's node for the objects to be updated. The second phase uses traditional software-based global locks acquired by the winner threads (Figure 5(b)). Only one thread can successfully pass these two phases and exclusively modify the objects.

RCX enjoys a few benefits by using HTM for the per-NUMA node lock acquisition. First, HTM provides good

scalability within a single NUMA node as the previous evaluation results show. Second, hardware-based synchronization such as HTM normally generates lower overhead compared to software-based synchronizations such as highly optimized NUMA-aware locks, owing to the use of dedicated hardware. Third, the per-NUMA node lock, which is the only data that HTM transaction accesses, is explicitly isolated from the RCX readers. This isolation avoids the HTM transaction aborts incurred by readers in read-mostly workloads (see Section 3.1 for details). Finally, HTM transaction abort amplification incurred by the involvement of remote NUMA nodes is avoided because only threads on the same NUMA node contend for each per-NUMA node.

The process also means that each update acquires two kinds of locks, thus incurring additional overhead compared to single-lock-acquisition mechanisms. However, this additional overhead is small due to reduced lock contention. Figure 5 assumes a system utilizing n NUMA nodes and m CPU cores in each node. If the mechanism is based on a single-lock acquisition or a single HTM transaction, $n \times m$ threads contend in the worst case. However, in our hierarchical synchronization, worst case contention for the per-NUMA node lock and the global lock is limited to m and n, respectively. Moreover, the number of nodes (n) is usually much smaller than that of CPU cores in each node (m) for common multicore systems. The contention for the global lock acquisition is thus greatly reduced, allowing traditional spinlocks to work well.

To show the impact of these factors in detail, we additionally designed two variants of RCX, namely RCX-HL and RCX-HHL. RCX-HL skips the per-NUMA node lock acquisition and acquires the per-entry global locks using HTM. This variant will show the impact of the aborts caused by readers. RCX-HHL uses HTM for the per-entry global lock acquisition, instead of a traditional spinlock. This variant will show the impact of NUMA awareness and the overhead of the RCX's traditional spinlock.

In addition to the logical description, we present an implementation of an RCX-protected sorted linked list with pseudocode. Pseudocode for the insertion of an item into the list is shown in Listings 3, 4, and 5. Listing 3 shows the data structure for the list, which is embedding the local locks (`pnodelocks`) and the global lock (`global_lock`) for RCX, which are cache-aligned, and Listing 4 shows helper macros for redundant code. The core logic for the insertion is in Listing 5. Once an RCX updater thread finds data items to update (lines 4–10), it should prepare a new item to insert (lines 11–13) and acquire per-NUMA node locks for the items. The thread first checks whether the locks are free and spins on the locks until the locks are marked as free (line 14). If the locks are all free, the thread starts a hardware transaction (line 15). In the transaction, it checks whether a concurrent transaction has already acquired the locks before this transaction starts, and if so, aborts this transaction (lines 16–17).

```
1 struct entry {
2     int val;
3     node_t *next;
4     int removed;
5     struct rcu_head rcu;
6     cacheline_t pnodelocks[nr_numa_nodes];
7     spinlock_t global_lock;
8 } entry_t;
```

Listing 3. RCX protected data item.

```
1 #define pndlock(e)                      \
2     (e->pnodelocks[numa_node_id()])
3 #define pnd_lock(e) pndlock(e) = 1;
4 #define pnd_ulock(e) pndlock(e) = 0;
5 #define glb_lock(e)                     \
6     spin_lock(e->global_lock);
7 #define glb_ulock(e)                    \
8     spin_unlock(e->global_lock);
```

Listing 4. RCX list: add helper macros.

```
 1 int rcx_list_add(list, val)
 2 {
 3 retry:
 4     prev = rcu_deref(list->head);
 5     next = rcu_deref(prev->next);
 6     while (next->val < val) {
 7         prev = next;
 8         next = rcu_deref(prev->next);
 9     }
10     if (val == next->val) return 1;
11     new_entry = rcx_new_entry();
12     new_entry->val = val;
13     new_entry->next = next;
14     while (pndlock(prev) || pndlock(next));
15     if (_xbegin() == _XBEGIN_STARTED) {
16         if (pndlock(prev) || pndlock(next))
17             _xabort();
18         pnd_lock(prev); pnd_lock(next);
19         _xend();
20     } else { /* Transaction aborted */
21         kfree(new_entry);
22         goto retry;
23     }
24     glb_lock(prev); glb_lock(next);
25     if (prev->removed || next->removed ||
26             rcu_deref(prev->next) != next)
27         goto unlock_retry;
28     rcu_assign(prev->next, new_entry);
29     success = 1;
30 unlock_retry:
31     glb_ulock(next); glb_ulock(prev);
32     pnd_ulock(prev); pnd_ulock(next);
33     if (success) return 0;
34     goto retry;
35 }
```

Listing 5. RCX list: add function.

If not, it acquires the locks and completes the transaction (lines 18–19). Because both lookup and acquisition of the locks are done in an HTM transaction, only one thread in the NUMA node can hold the lock. If the transaction is aborted, it means that there was a concurrent transaction to the data items and that the locks are not acquired by this thread. The thread then frees the allocated memory and restart by finding items to update (lines 20–23). If the transaction is successful, it means the thread is the only thread holding the per-NUMA node locks for updating items in the current NUMA node. This thread then acquires the global locks of the updating items (lines 24). After this acquisition, it checks whether there were concurrent updates to the target data items (lines 25 and 26). If so, it releases every lock it acquired and restarts by finding items to update (line 27 and lines 30–34). If the global locks are acquired and no other concurrent updates to given data items occurred, the thread can safely update the items using the pointer assignment and necessary memory barriers (`rcu_assign()`) (line 28). Once the update is done, it releases every lock it acquired and returns (lines 29–33).

Listings 6 and 7 show pseudocode for the removal of an item from the list. We skip a detailed description of the code because the basic process is the same for the insertion, although it has additional tasks including data item invalidation (line 32).

If each object is embedding a set of locks for it as this simplified pseudocode shows, the memory footprint of RCX can be arbitrarily increased as the number of RCX-protected objects becomes greater. Nevertheless, because RCX requires each object to have only a reference to corresponding locks, a set of objects can share their locks, thus reducing the memory footprint. This is illustrated by the lock-multiplexing scheme shown in Figure 6. This scheme first allocates a number of locks and maps a set of objects to each lock. As long as the number of locks is sufficiently larger than the number of threads, the contention remains low. Thus, this scheme effectively maintains a low memory footprint while preserving scalability. Alternatives include hierarchical scalable locks [35], but our prototype implementation simply gives each object its own lock, thus allowing this paper to focus on performance and scalability. We hypothesize that the suggested memory-footprint approaches are sound.

Note that RCX's target workload is read-intensive. This reduces RCX's update-side contention, enabling a straightforward design and implementation of its update-side locks. Furthermore, because concurrent updates to a single object are typically avoided in such workloads, per-object contention is also low. RCX, therefore, does not need sophisticated optimizations such as the ownership transfer or back-offs often required by state-of-the-art NUMA-aware scalable locks. Dispensing with these optimizations reduces software overhead and greatly simplifies RCX's implementation.

```
1  #define pnd_locks(e1, e2, e3)        \
2      pnd_lock(e1); pnd_lock(e2);      \
3      pnd_lock(e3);
4  #define pnd_ulocks(e1, e2, e3)       \
5      pnd_ulock(e1); pnd_ulock(e2);    \
6      pnd_ulock(e3);
7  #define glb_locks(e1, e2, e3)        \
8      glb_lock(e1); glb_lock(e2);      \
9      glb_lock(e3)
10 #define glb_ulocks(e1, e2, e3)       \
11     glb_ulock(e1); glb_ulock(e2);    \
12     glb_ulock(e2);
```

Listing 6. RCX list: remove helper macros.

```
1  int rcx_list_remove(list, val)
2  {
3  retry:
4      p = rcu_deref(list->head);
5      n = rcu_deref(p->next);
6      while (n->val < val) {
7          p = n;
8          n = rcu_deref(p->next);
9      }
10     if (val != n->val) return 0;
12     e = n;
13     n = rcu_deref(n->next);
14     while (pndlock(p) ||
15            pndlock(e) || pndlock(n));
16     if (_xbegin() == _XBEGIN_STARTED) {
17         if (pndlock(p) ||
18                pndlock(e) || pndlock(n))
19             _xabort();
20         pnd_locks(p, e, n);
21         _xend();
22     } else { /* transaction aborted */
23         goto retry;
24     }
25     glb_locks(p, e, n);
26     if (p->removed || e->removed ||
27             n->removed)
28         goto unlock_retry;
29     if (rcu_deref(p->next) != e ||
30         rcu_deref(e->next) != n)
31         goto unlock_retry;
32     rcu_assign(p->next, n)
33     e->removed = 1;
34     kfree_rcu(e, rcu);
35     success = 1;
36 unlock_retry:
37     glob_ulocks(n, e, p);
38     pnd_ulocks(n, e, p);
39     if (success) return;
40     goto retry;
41 }
```

Listing 7. RCX list: remove function.

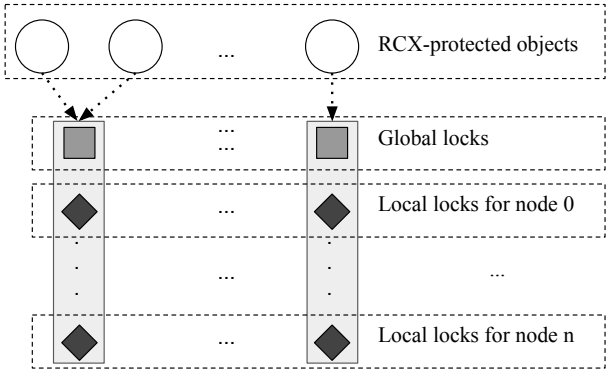


Figure 6. Lock multiplexing for memory footprint reduction.

Finally, note that this pseudocode implementation provides no software handler for situations involving endless HTM aborts. Because of HTM's transaction-size limit, such a handler is required to guarantee forward progress in the general case. The FORGIVE and RETRY versions of RCU-HTM (Section 3.1) also lack such a handler, which can be problematic for transactions with unbounded memory footprints. However, because RCX's HTM transactions access only the per-NUMA node locks of the updated data items, their size is bounded above by the number of updated items times the size of a single cache line. Thus, the limited-size RCX updates in this paper should not fail due to cache overflow. If the number of updated items could exceed the limit, we suggest splitting the items into several groups and acquiring the per-NUMA node and global locks separately for each group. This workaround avoids the problem, though it will incur additional overhead and might be hard to apply for unknown transaction-size limits. Also, note that this assumption is dependent on the detailed implementation of the underlying HTM.

3.3 Implementation

For proof of concept and a micro and macro evaluation, we implemented RCX-protected data structures including sorted linked lists and hash tables (Section 4.2) and optimized a part of the virtual memory management system in an operating system kernel (Section 4.3) and an in-memory database system (Section 4.4) using RCX. All implementations except the database system work in kernel space because RCU works best and most reliably in this space whereas userspace RCU implementations are not optimally tuned. This also means that the global lock of RCX for the data structures and the virtual memory system is an MCS lock because the default spinlock of the Linux kernel is the MCS lock.

The implementation of the linked lists and hash tables is based on the RLU kernel space micro-benchmark implementation source code, which is available at Github (https://github.com/rlu-sync/rlu). Based on the code, we implemented linked lists and hash tables protected by each variant this paper described. Linked lists protected by RCU-MCS or RCU-HMCS are analogous to the RCX-protected one, but use the qspinlock of the Linux kernel [17], which is based on MCS lock, in single-level (RCU-MCS) or two-level as RCX does (RCU-HMCS). The current implementation of RCX-protected linked lists and hash tables consists of about 200 lines of code. Note that the current implementations are made to be compatible only with Intel Haswell or later architectures. The source code of the mechanisms and programs described in this paper is available at https://github.com/rcx-sync.

4 Evaluation

This section describes the evaluation system configuration and then presents results from a set of microbenchmarks and system-level benchmarks.

4.1 Evaluation Setup

We use a large NUMA system for our evaluation. The system includes 4-way Intel Xeon E7-8870 v3 processors. Each processor utilizes 18 CPU cores, and each core provides two hardware threads (Intel hyperthreading is enabled). In total, the system provides 144 hardware threads. The system is running Ubuntu Server 16.04.3 and the v4.19 Linux kernel.

To minimize measurement errors, we repeated every evaluation five times and averaged the results. Deviations for repeated runs were negligible.

4.2 Micro-benchmarks

To show overall performance and to see whether our intended design works as expected in detail, we use simple but widely used data structures, linked lists and hash tables. For this evaluation, we measure throughput and HTM abort rates of those data structures protected by RCU, RCU-HMCS, RLU, RCU-HTM, RCX-HL, RCX-HHL, and RCX in kernel space. Each of these variants protects updates using global locking, fine-grained hierarchical locking, STM, HTM, HTM-based locking, hierarchical HTM-based locking, and our algorithm, respectively. Refer to Section 2.4 and 3.2 for the detailed definitions of the first four variants and the last three, respectively. Table 2 compares each of these variants by enumerating their protection mechanisms and their adherence to the five design principles of RCX (Table 1). "O" means that the variant adheres to the corresponding design principle, while "X" means it does not. The fourth and fifth principles depend on the third principle (HTM utilization), so variants not adhering to this principle have their fourth and fifth principles marked with "-". The fifth principle (NUMA-global HTM use avoidance) in RCX-HHL is marked as "△" because although RCX-HHL allows NUMA-global HTM transactions, its NUMA-aware hierarchy reduces NUMA-global contention.

As noted in Section 3.3, we implemented the data structures based on those in the RLU paper [41]. However, we

Variant	Protection mechanism	Principles				
		1	2	3	4	5
RCU	Global locking	X	O	X	-	-
RCU-HMCS	Fine-grained hierarchical locking	O	O	X	-	-
RLU	STM	O	X	X	-	-
RCU-HTM	HTM	O	O	O	X	X
RCX-HL	HTM-based locking	O	O	O	O	X
RCX-HHL	Hierarchical HTM-based locking	O	O	O	O	△
RCX	Our mechanism	O	O	O	O	O

Table 2. The protection mechanisms and adoption of the five design principles (1. finer-granularity, 2. pure RCU read mechanism, 3. HTM utilization, 4. HTM working set isolation, and 5. NUMA-global HTM use avoidance).

made only small changes to the data structures protected by RCU and RLU. Hash tables are constructed with multiple buckets, and each bucket is a linked list described in Listings 5 and 7. We omit RCU-MCS because its performance is almost the same as RCU-HMCS as we already described in Section 2.3. RCU uses global locking while RCU-HMCS uses fine-grained locking for the update-side synchronization.

For each evaluation, we specify the desired number of threads, the number of buckets (b), the number of initial items in each bucket (n), and the update rate (u). We construct a hash table instance with b buckets and insert $b \times n$ random integers ranging from zero to $2 \times b \times n$. After the initialization, we induce lookup, insert, and remove operations with random keys ranging from zero to $2 \times b \times n$ for 3 seconds and measure the number of successfully processed operations and the number of HTM aborts. Insertions and removals are induced with rate $u \div 2$. For example, if u is 20%, 80% of operations do lookup, 10% do insert, and the remaining 10% do remove.

We pin working threads onto a minimal number of NUMA nodes and vary the number of threads across the set { 1, 2, 4, 9, 18, 35, 36, 37, 54, 71, 72, 73, 90, 107, 108, 109, 126, 144 }. For example, if the number of threads is 36, every hardware thread in the first NUMA node works. We run evaluations with 35, 37, 71, 73, 107 and 109 threads to highlight the effect of the NUMA architecture. RLU performance for 144 threads is omitted due to a bug in the officially available RLU implementation.

Figures 7(a)–7(d) show the performance of the linked lists with 256 initial nodes for read-only, 2%, 20%, and 40% updates. With read-only workload (Figure 7(a)), only RLU shows the log-seeking overhead while others show almost ideal performance because their read operations are the same as that of RCU. With 2% updates (Figure 7(b)), the results of RCU, RCU-HMCS, RLU, and RCU-HTM are the same as that of Section 2.4. RCX variants show the best performance because

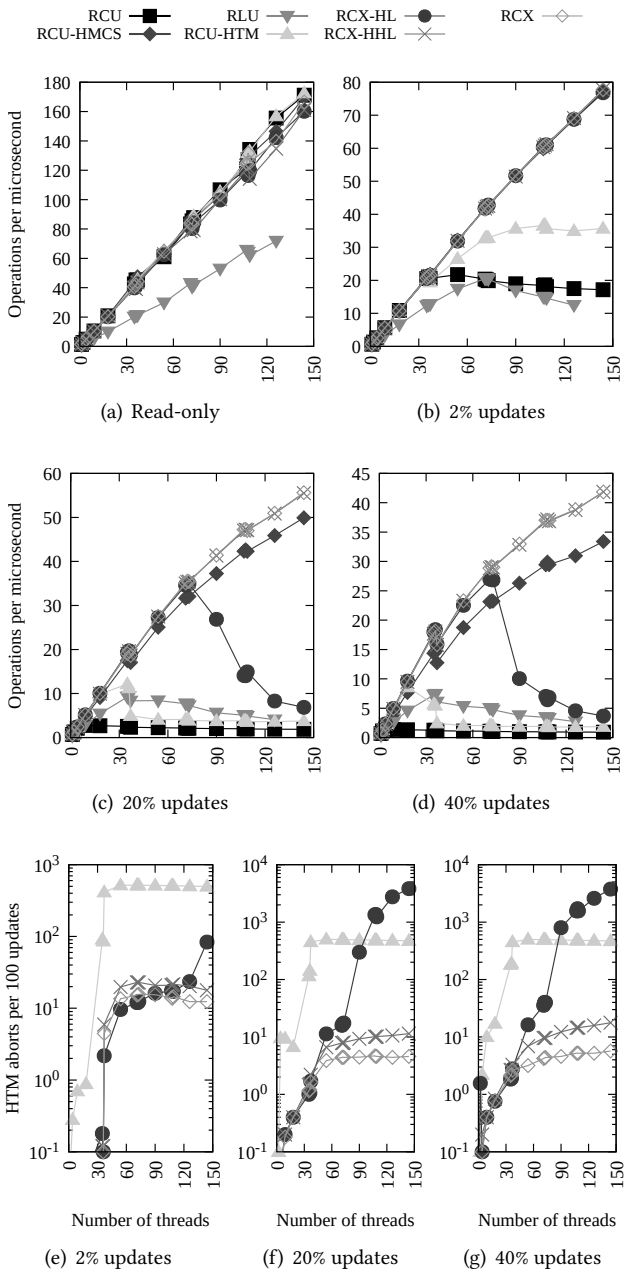


Figure 7. Performance and HTM abort rates of RCU variants for linked lists with 256 initial nodes.

those variants avoid the readers caused aborts. As update rates increase (Figure 7(c) and 7(d)), even RCU-HL stops scaling from 72 threads and collapses as the number of threads increases due to its NUMA-oblivious design. Though RCU-HMCS keeps scaling, it shows clearly lower performance compared to RCX due to its software overhead. However, RCX keeps the best performance and scaling towards 144 threads owing to its efficient and NUMA-aware use of HTM. RCX also performs as same as RCX-HHL; this means the

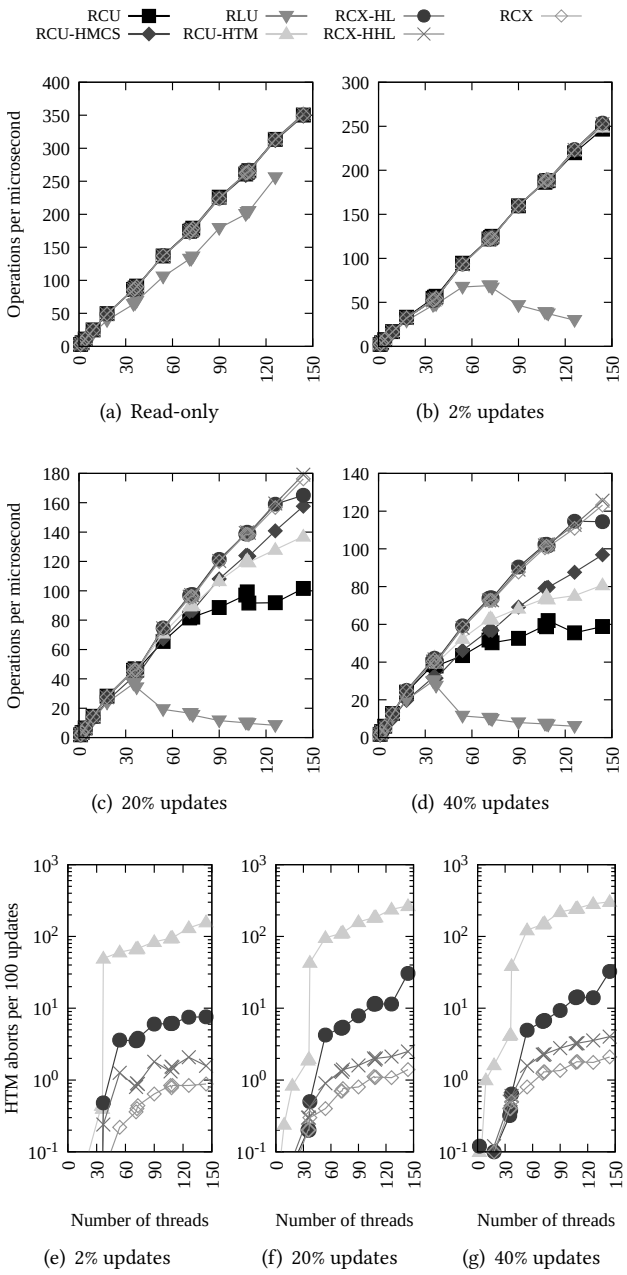


Figure 8. Performance and HTM abort rates of RCU variants for hash tables with 128 buckets and 32 initial nodes.

overhead of the traditional spinlock of RCX is as low as HTM.

Figures 7(e)–7(g) show abort rates of the linked lists for the workload above in log-scale. Because RCU, RCU-HMCS, and RLU do not use HTM at all, the variants always show 0% abort rates.

Results of RCU-HTM are the same as that of Section 3.1. The abort rates for RCU-HTM gradually grows as the number of threads increases until 36 threads, but it suddenly shows an exponential increase with 37 threads and then stops growing above 500% regardless of the number of threads and update rates. RCX-HL abort rates clearly explain its performance results. Those are as low as RCX and RCX-HHL with 2% updates but rapidly increases with higher updates. RCX-HHL abort rates are higher than RCX due to its extended HTM use. This also justifies the use of the traditional spinlock of RCX. Meanwhile, RCX maintains only up to 10% abort rate for every case regardless of update rates due to its NUMA-aware and safe-from-reads synchronization.

Figures 8(a)–8(d) show the performance of hash tables with 128 buckets and 32 initial nodes. Because the number of items is 64 times bigger than that of previous evaluations of linked lists, the data conflict rate is significantly lower. In particular, because the RCU-protected hash table is doing locking per bucket, it can scale with updates.

The read-only results are similar to those of linked lists. RLU shows lookup overhead, and others show the ideal scalability. Due to the hash table scalability, every variant except RLU shows a similar and almost ideal performance with a 2% updates. RLU does not scale above 72 threads, even in this case. Nevertheless, with increased update rates (Figures 8(c) and 8(d)) and increased number of threads, RCX and RCX-HHL show clear performance enhancement upon others. With 40% update rates and 144 threads, the RCX-protected hash table outperforms RCU-HTM by 1.55 times.

Figures 8(e)–8(g) show abort rates for the hash tables workloads. The results are similar to those in Figures 7(e)–7(g). RCX still shows about two orders of magnitude lower abort rates compared to RCU-HTM.

4.3 Scalable Virtual Memory System

To further show whether RCX can be used for large, complicated systems and enhance real-world applications, we optimized the v4.19 Linux kernel with RCX for one well-known scalability bottleneck issue [15, 16] in virtual memory address space manipulation for multi-threaded applications.

RCU-based approaches for the issue, namely RCUVM [15] and Speculative Page Faults (SPF) [26, 33], have been proposed, though the approaches cannot scale with concurrent address space modifications due to their coarse-grained-locked updates. We therefore further modified the virtual memory system to process a few common code paths of address space modification in parallel with RCX protection. In detail, we modified a part of the `mprotect()` system call that adjusts only a part of the address space (virtual memory areas) by applying RCX to protect each virtual memory area so that most `mprotect()` system calls can proceed in parallel. We call the modified kernel RCXVM. In total, the use of RCX modified about 1900 lines of Linux kernel code. For the concurrent address space lookup of RCXVM, we applied the SPF patchset [26] instead of RCUVM [15] because these two approaches are similar and SPF is optimized for recent Linux kernels.

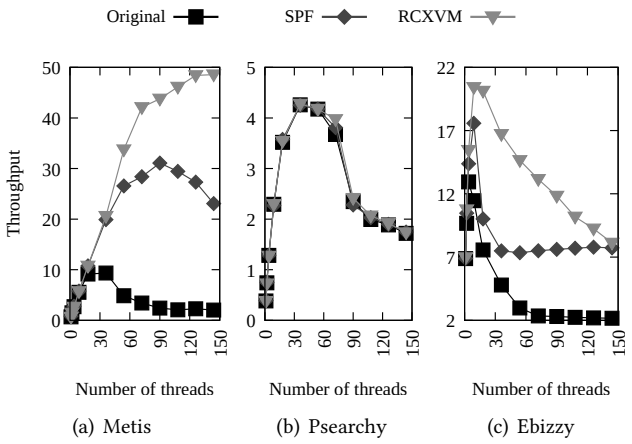


Figure 9. Performance of realistic workloads on modified virtual memory systems. The number of jobs per minute was measured for Metis and Psearchy while the number of records per millisecond was measured for Ebizzy.

To show the performance of the optimized kernel for realistic workloads, we run and measure the throughput of *Metis*, *Psearchy*, and *Ebizzy*. Metis is an in-memory multi-threaded map-reduce application that computes a word position index. Psearchy is a parallel version of Searchy [56]. We configure Psearchy to index files in a v4.19.10 Linux kernel source code directory. These two workloads are retrieved from the MOSBENCH suite [4], which is developed to compare the scalability of the kernels, and widely used for similar studies [14–16, 35, 37]. Ebizzy [5] is a benchmark resembling general webserver workloads and frequently used for memory management systems evaluations [26, 36].

Figure 9 shows the performance of the applications on three kernels. To summarize, the kernels are as below:

- *Original* is the v4.19 Linux kernel protecting the address space using the global reader-writer lock.
- *SPF* is the SPF patchset [26] applied to the v4.19 Linux kernel protecting the address space using the RCU with coarse-grained update locking.
- *RCXVM* is our optimization applied to the v4.19 Linux kernel protecting the address space using RCX.

With Original, Metis suffers from bad scalability. SPF significantly improves the performance and scalability of Metis due to concurrent page fault handlings. However, SPF performance also decreases from 90 threads. Our investigation using the perf tool [27] found the bottleneck is the `mprotect()` system call. The CPU portion of `mprotect()` kernel space execution is only 2.43% with 36 threads but grows to 73.22% with 144 threads. Meanwhile, the CPU portion for `strcmp()` userspace execution is 22.57% with 36 threads but decreases to only 3.35% with 144 threads. SPF cannot help this problem because the bottleneck is serialized execution of concurrent `mprotect()`, not page fault handling. RCXVM further improves the performance of Metis for many threads due to its address space adjustment parallelization. With 144 threads, RCXVM shows 24.03 times and 2.10 times higher performance compared to Original and SPF, respectively.

All three kernels show poor scalability for Psearchy. Our profiling results suggest that the bottleneck of Psearchy is memory mapping incurred TLB flushes. It is widely known that a range of complicated scalability bottlenecks [16] exists in virtual memory management systems. Because RCXVM is parallelizing only a part of the address space modification and lookup, it cannot address the TLB flush's bottleneck and thus cannot improve the performance of Psearchy.

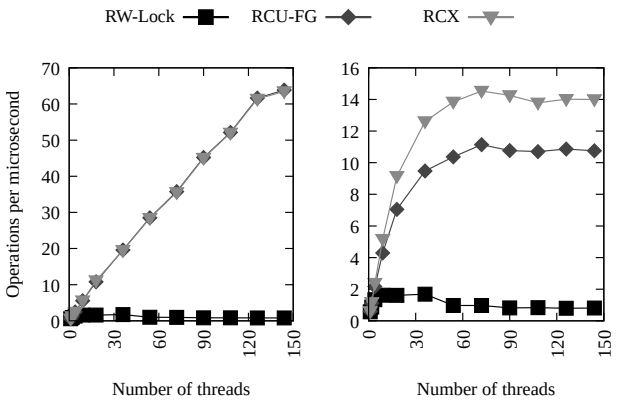
Ebizzy results are worse than Metis but better than Psearchy because the intensiveness of the `mmap()` call of Ebizzy is higher than Metis but lower than Psearchy, according to our profiling results. Original system's performance scales only until 4 threads. It shows a performance that is even worse than that of single thread from 36 threads. Meanwhile, SPF keeps scaling until 9 threads with much better scalability. With 9 threads, SPF shows 1.53 times higher performance compared to Original. After that, SPF starts to rapidly degrade. Nevertheless, SPF keeps higher performance compared to Original and achieves 3.57 times higher throughput in the best case (144 threads). RCXVM keeps scaling until 18 threads but starts to drop in performance with a higher number of threads. That said, RCXVM's performance scaling is fastest and its degradation is slowest among the three systems. In the best cases, RCXVM performs 2.23 times higher compared to SPF (36 threads), 5.60 times higher compared to Original (72 threads).

Further optimizing the kernel and RCX for Psearchy and Ebizzy is outside the scope of this paper. Nevertheless, these results suggest that RCX can improve kernel scalability.

4.4 Kyoto Cabinet Cache DB

Kyoto Cabinet CacheDB [38] is a popular in-memory database implementation which is the focus of several read-mostly researches [14, 22, 37, 41]. The database is constructed with multiple slots, and each slot is again split into a number of buckets. To find the corresponding slot and bucket, Kyoto CacheDB utilizes two-level hashing of a given key. The first hash result finds the corresponding slot, and the second hash result is used to find its bucket. To protect each slot and the database from the concurrent threads, it utilizes per-slot locks and a global reader-writer lock, respectively. Dice *et al.*[22] found that the global reader-writer lock is the main scalability bottleneck of Kyoto CacheDB, and several following works also confirmed that [37, 41].

We apply fine-grained locking uses of RCU and RCX to Kyoto CacheDB by substituting the global reader-writer lock with RCU read-side critical sections and use the existing slot locks or RCX locks for updates. After each read, reader thread checks whether there was any concurrent update to



(a) Read-only (b) 10% updates

Figure 10. Performance of KyotoCabinet (20 million records) protected by the original global reader-writer lock (RW-Lock), fine-grained locking based RCU (RCU-FG) and RCX.

the data it read. If so, it acquires the lock (in case of the fine-grained locking based RCU) or follows RCX update-side task for the slot and retries the read. Our implementation is based on Kyoto Cabinet version 1.2.77. Because Kyoto CacheDB is in userspace, we also implemented RCX on userspace by modifying the official userspace RCU [21, 40] of version 0.9.6. In total, we modified about 500 lines of code of those.

For the evaluation, we create a Kyoto CacheDB instance containing 20 million records (about 1.4 GB in total), randomly induce *get()*, *set()*, and *remove()* operations for three seconds, and measure the number of operations processed during the three seconds. We set the number of slots and the total number of buckets as 16 and 1,048,583 respectively because those are default values. Figure 10 shows the measurement results for three Kyoto CacheDB instances each protected by the three different protection mechanisms.

- *RW-Lock* is the original Kyoto CacheDB. It protects the data using the global reader-writer lock.
- *RCU-FG* is our straightforward optimization of the Kyoto CacheDB using RCU. It protects the data using RCU with a fine-grained update-side synchronization as described above.
- *RCX* is our Kyoto CacheDB optimization using RCX. The detailed application of RCX is as described above.

Both RCU-FG and RCX achieves the near-ideal performance for read-only workload, while the global reader-writer lock based version (RW-Lock) does not scale. With 10% updates, RCX shows about 17.28 times and 1.3 times higher performance compared to the original reader-writer lock and fine-grained locking based RCU, owing to its efficient use of HTM for read-mostly workloads on NUMA systems. Though, both RCU-FG and RCX stops scaling from 72 threads. This is due to the small number of slots, which is only 16. Increasing the number will effectively reduce the contention to each slot and improve the scalability.

5 Discussions

Because RCX is designed for read-mostly workloads, it is natural to ask what happens when RCX is applied to real-world workloads that are either consistently or intermittently update intensive. As shown by our update-intensive experiments with Psearchy (Section 4.3) KyotoCabinet (Section 4.4), RCX can experience scalability limitations or even significant scalability collapse. Nevertheless, RCX performed at least as well as its alternatives in these cases.

Future work includes a number of straightforward solutions for these scalability issues. One approach uses the sophisticated backoff and priority inheritance mechanisms that have been applied to other NUMA-aware scalable locks. However, care is required to avoid degrading RCX's fast-path performance. We therefore hypothesize that other mechanisms will prove to be more beneficial. Such mechanisms include adaptive approaches based on the use of offline measurements of the workload's read-intensity or use of dynamic techniques to monitor other workload characteristics [57].

Nonetheless, we reiterate that in the update-intensive evaluations in this paper, RCX always performed best, scaling to 144 threads with up to 40% updates. These evaluations clearly cannot be characterized as read-intensive.

One important strength of RCX compared to another state-of-the-art HTM-based RCU extension (RCU-HTM) is RCX's small HTM transaction size. This is due to the fact that RCX's HTM transactions access only the per-NUMA node locks. That said, it is reasonable to ask whether there are corner cases that might increase transaction size such that transaction-size aborts become commonplace, and what happens in that case. For example, consider the RCX-protected linked list that acquires two per-NUMA node locks (`prev` and `next`) in one HTM transaction when adding a new object (Listing 5). As shown in Figure 7, this works quite well. However, one could further consider graphs that might have an arbitrarily large number of links, thus requiring an arbitrarily large number of per-NUMA node locks to be acquired. At some point, acquiring all of these locks in a single HTM transaction results in unconditional transaction aborts.

However, it turns out that it is not necessary to acquire all of the per-NUMA node locks in a single HTM transaction. Because the locks can be acquired independently, they may be acquired in multiple groups such that each group is small enough to remain within the size limit, thus permitting the use of per-group HTM transactions. RCX exports both per-NUMA node locking functions (`pnd_lock()` and `pnd_ulock()`) and also global locking functions (`glb_lock()` and `glb_ulock()`), thus allowing developers users to tune their use of HTM to their particular situation.

6 Conclusion

This paper provided empirical evaluations and profiling of a few state-of-the-art RCU extensions running on NUMA systems and introduced their significant shortcomings. We further investigated the root cause of these shortcomings, and then developed a new synchronization mechanism called RCX that combines RCU and HTM to avoid these shortcomings.

We created micro-benchmarks evaluating RCX on simple data structures, resulting in up to 22.6 times higher performance and up to 145 times lower HTM transaction abort rates, both compared to an HTM-based state-of-the-art RCU extension. We also carried out a system-level evaluation of RCX by parallelizing a portion of the v4.19 Linux kernel, resulting in RCXVM. RCXVM provides up to 24.03 times the performance of the vanilla v4.19 Linux kernel when running a multi-threaded map-reduce application. Finally, we optimized the userspace Kyoto Cabinet in-memory database system by applying RCX and achieved up to 17.28 times speedup. This work demonstrates the performance and scalability benefits of carefully conceived combinations of RCU and HTM, as exemplified by RCX.

Acknowledgments

This work was done prior to the first author joining Amazon and while the second author was at IBM. The last author is the corresponding author of this paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2015M3C4A7065646).

References

[1] Maya Arbel and Hagit Attiya. 2014. Concurrent updates with RCU: search tree as an example. In *Proceedings of the 2014 ACM symposium on Principles of Distributed Computing*. ACM, 196–205.

[2] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. 2003. Using Read-Copy Update Techniques for System V IPC in the Linux 2.5 Kernel. In *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*. USENIX Association, 297–310. https://www.usenix.org/legacy/publications/library/proceedings/usenix03/tech/freenix03/full_papers/arcangeli/arcangeli.pdf

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M Michael, and Martin Vechev. 2011. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. *ACM SIGPLAN Notices* 46, 1 (2011), 487–498.

[4] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. 2010. An Analysis of Linux Scalability to Many Cores.. In *OSDI*, Vol. 10. 86–93.

[5] Rodrigo Rubira Branco. 2008. ebizzy 0.3 released. (January 2008). https://lwn.net/Articles/263706/.

[6] Neil Brown. 2017. [PATCH 00/20] staging: lustre: convert to rhashtable. (2017). https://lkml.org/lkml/2018/4/11/1341.

[7] Neil Brown. 2018. The rhashtable documentation I wanted to read. (2018). https://lwn.net/Articles/751374/.

[8] Trevor Brown, Alex Kogan, Yossi Lev, and Victor Luchangco. 2016. Investigating the performance of hardware transactions on a multi-socket machine. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 121–132.

[9] Davidlohr Bueso. 2015. acquire/release semantics in the kernel. (October 2015). https://blog.stgolabs.net/2015/10/acquirerelease-semantics-in-kernel.html.

[10] Davidlohr Bueso. 2017. locking: Introduce range reader/writer lock. (2017). https://lwn.net/Articles/719182/.

[11] Milind Chabbi, Abdelhalim Amer, Shasha Wen, and Xu Liu. 2017. An efficient abortable-locking protocol for multi-level NUMA systems. *ACM SIGPLAN Notices* 52, 8 (2017), 61–74.

[12] Milind Chabbi, Michael Fagan, and John Mellor-Crummey. 2015. High performance locks for multi-level NUMA systems. *ACM SIGPLAN Notices* 50, 8 (2015), 215–226.

[13] Milind Chabbi and John Mellor-Crummey. 2016. Contention-conscious, locality-preserving locks. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 22.

[14] Haibo Chen, Heng Zhang, Ran Liu, Binyu Zang, and Haibing Guan. 2016. Fast consensus using bounded staleness for scalable read-mostly synchronization. *IEEE Transactions on Parallel and Distributed Systems* 27, 12 (2016), 3485–3500.

[15] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable address spaces using RCU balanced trees. *ACM SIGPLAN Notices* 47, 4 (2012), 199–210.

[16] Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 211–224.

[17] Jonathan Corbet. 2014. MCS locks and qspinlocks. (2014). https://lwn.net/Articles/590243/.

[18] P. J. Courtois, F. Heymans, and D. L. Parnas. 1971. Concurrent Control with "Readers" and "Writers". *Commun. ACM* 14, 10 (October 1971), 667–668. https://doi.org/10.1145/362759.362813

[19] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2013. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 33–48.

[20] Will Deacon. 2017. [PATCH v2 0/5] Get rid of lockless_dereference(). (2017). http://lkml.kernel.org/r/1508840570-22169-3-git-send-email-will.deacon@arm.com.

[21] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23 (2012), 375–382. https://doi.org/10.1109/TPDS.2011.159

[22] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. 2014. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures*. ACM, 188–197.

[23] Dave Dice, Yossi Lev, Mark Moir, and Dan Nussbaum. 2009. Early Experience with a Commercial Hardware Transactional Memory Implementation. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*. Washington, DC, USA, 157–168. https://doi.org/10.1145/1508244.1508263

[24] Dave Dice, Virendra J. Marathe, and Nir Shavit. 2011. Flat-combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/1989493.1989502

[25] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. 2009. Stretching Transactional Memory. *ACM SIGPLAN Noticeis* 44, 6 (June 2009), 155–165. https://doi.org/10.1145/1543135.1542494

[26] Laurent Dufour. 2017. [v5,00/22] Speculative page faults. (2017). https://patchwork.ozlabs.org/cover/824425/.

[27] Brendan Gregg. [n.d.]. perf Examples. ([n. d.]). http://www.brendangregg.com/perf.html.

[28] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. 2016. Multicore Locks: The Case Is Not Closed Yet.. In *Proceedings of the 2016 USENIX Annual Technical Conference*. 649–662.

[29] Hyuck Han, SeongJae Park, Hyungsoo Jung, Alan Fekete, Uwe Rohm, and Heon Y Yeom. 2014. Scalable serializable snapshot isolation for multicore systems. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 700–711.

[30] Phil Howard. 2012. *Extending Relativistic Programming to Multiple Writers*. Ph.D. Dissertation. Portland State University.

[31] Philip W. Howard and Jonathan Walpole. 2011. A Relativistic Enhancement to Software Transactional Memory. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11)*. USENIX Association, Berkeley, CA, USA, 1–6. http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf

[32] Philip W. Howard and Jonathan Walpole. 2013. Relativistic Red-Black Trees. *Concurrency and Computation: Practice and Experience* (2013), 29. https://doi.org/10.1002/cpe.3157

[33] Nur Hussein. 2017. Another attempt at speculative page-fault handling. (2017). https://lwn.net/Articles/730531/.

[34] Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional memory architecture and implementation for IBM System z. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 25–36.

[35] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Scalable NUMA-aware blocking synchronization primitives. In *Proceedings of the 2017 USENIX Annual Technical Conference*.

[36] Kris Kennaway. 2008. Memory allocation performance on FreeBSD and Linux with ebizzy. (March 2008). https://people.freebsd.org/~kris/scaling/ebizzy.html.

[37] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. 2019. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 779–792.

[38] FAL Labs. 2011. Kyoto Cabinet: a straightforward implementation of DBM. (2011). https://fallabs.com/kyotocabinet/.

[39] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2014. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 580–591.

[40] Paul E. McKenney Mathieu Desnoyers. [n.d.]. Userspace RCU. ([n. d.]). https://liburcu.org/.

[41] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. ACM, 168–183.

[42] Paul E. McKenney. 2004. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. Ph.D. Dissertation. OGI School of Science and Engineering at Oregon Health and Sciences University. http://www.rdrop.com/~paulmck/RCU/RCUdissertation.2004.07.14e1.pdf

[43] Paul E. McKenney. 2006. Sleepable RCU. (2006). https://lwn.net/Articles/202847/.

[44] Paul E. McKenney. 2014. *Is Parallel Programming Hard, And, If So, What Can You Do About It? (First Edition)*. kernel.org, Corvallis, OR, USA. https://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e1.html

[45] Paul E. McKenney. 2016. Beyond the Issaquah Challenge: High-Performance Scalable Complex Updates. In *CppCon2016*. https://cppcon2016.sched.com/.

[46] Paul E. McKenney. 2016. High-Performance and Scalable Updates: The Issaquah Challenge. In *Applicative 2016*. ACM.

[47] Paul E. McKenney and Aravinda Prasad. 2015. Some more details on Read-Log-Update. (2015). https://lwn.net/Articles/667720/.

[48] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*. 509–518.

[49] Rick Merritt. 2011. IBM plants transactional memory in CPU. (August 2011). http://www.eetimes.com/electronics-news/4218914/IBM-plants-transactional-memory-in-CPU.

[50] James R. 2012. Transactional Synchronization in Haswell. (2012). https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell.

[51] Ravi Rajwar and Martin Dixon. 2012. Intel transactional synchronization extensions. In *Intel Developer Forum San Francisco*, Vol. 2012.

[52] Hany E. Ramadan, Christopher J. Rossbach, Donald E. Porter, Owen S. Hofmann, Aditya Bhandari, and Emmett Witchel. 2007. MetaTM/TxLinux: transactional memory for an operating system. *SIGARCH Comput. Archit. News* 35, 2 (June 2007), 92–103. https://doi.org/10.1145/1273440.1250675

[53] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. 2007. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 87–102. https://doi.org/10.1145/1294261.1294271

[54] Christopher J. Rossbach, Owen S. Hofmann, Donald E. Porter, Hany E. Ramadan, Bhandari Aditya, and Emmett Witchel. 2007. TxLinux: Using and Managing Hardware Transactional Memory in an Operating System. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 87–102. https://doi.org/10.1145/1323293.1294271

[55] Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2017. RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees. In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 1–13.

[56] Jeremy Stribling, Jinyang Li, Isaac G Councill, M Frans Kaashoek, and Robert Morris. 2006. OverCite: A Distributed, Cooperative CiteSeer.. In *NSDI*, Vol. 6. 11–11.

[57] Dixin Tang and Aaron J Elmore. 2018. Toward coordination-free and reconfigurable mixed concurrency control. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 809–822.

[58] Josh Triplett. 2012. *Relativistic Causal Ordering a Memory Model for Scalable Concurrent Data Structures*. Ph.D. Dissertation. Portland, OR, USA. Advisor(s) Walpole, Jonathan. AAI3502650.

[59] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. ACM, 18–32.

[60] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling Concurrent Search Trees Under Contention Using HTM. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 385–399. https://doi.org/10.1145/3018743.3018752

[61] Xin Wang, Weihua Zhang, Zhaoguo Wang, Ziyun Wei, Haibo Chen, and Wenyun Zhao. 2017. Eunomia: Scaling Concurrent Search Trees Under Contention Using HTM. *SIGPLAN Not.* 52, 8 (Jan. 2017), 385–399. https://doi.org/10.1145/3155284.3018752

[62] Zhaoguo Wang, Hao Qian, Jinyang Li, and Haibo Chen. 2014. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the 9th ACM European Conference on Computer Systems*. ACM, 26.

[63] Yingjun Wu and Kian-Lee Tan. 2016. Scalable In-Memory Transaction Processing with HTM. In *Proceedings of the 2016 USENIX Annual Technical Conference*. 365–377.