Predicate RCU: An RCU for Scalable Concurrent Updates

Maya Arbel Adam Morrison

Computer Science Department
Technion—Israel Institute of Technology

Abstract

Read-copy update (RCU) is a shared memory synchronization mechanism with scalable synchronization-free reads that nevertheless execute correctly with concurrent updates. To guarantee the consistency of such reads, an RCU update transitioning the data structure between certain states must *wait for the completion of all existing reads*. Unfortunately, these waiting periods quickly become a bottleneck, and thus RCU remains unused in data structures that require scalable, fine-grained, update operations.

To solve this problem, we present *Predicate RCU* (PRCU), an RCU variant in which an update waits *only for the reads whose consistency it affects*, which are specified by a user-supplied *predicate*. We explore the trade-offs in implementing PRCU, describing implementations that reduce wait times by 10–100× with varying overhead on reads on modern x86 multiprocessor machines.

We demonstrate the applicability of PRCU by applying it to two RCU-based concurrent algorithms—the CITRUS binary search tree and a resizable hash table—and show experimentally that PRCU significantly improves the performance of both algorithms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Concurrent data structures; Synchronization; RCU

1. Introduction

Concurrent data structures strive to be both scalable and fast. Scalability requires increasing the amount of operations running concurrently—in particular, allowing concurrent execution of *reads* (operations that do not change the state of the data structure, such as hash table lookups) and *updates* (e.g., insertions and deletions). Being fast entails minimal synchronization overhead, especially for reads, which are very common operations [15]. Unfortunately, most synchronization mechanisms either prevent read/update concurrency—e.g., read/write locks [5]—or impose synchronization overhead on reads, such as blocking [23] or validation checks and retries [3, 7].

The *read-copy update* (RCU) mechanism is a notable exception, as it allows scalable (synchronization-free) reads that execute concurrently with updates. RCU places the burden of maintaining correctness solely on the updates, which must guarantee that readers observe a consistent state of the data structure. To achieve this goal, an update must sometimes ascertain that a modification it makes is *globally visible*. For example, without knowing that the removal of a node from a linked list is visible to all concurrent reads, it is not safe to reclaim the node's memory. Updates obtain this guarantee using a wait-for-readers primitive provided by RCU, which blocks until the completion of all reads that started before the wait-for-readers invocation. This ensures that no read which missed the modification—e.g., read a pointer to the node before it was removed—remains.

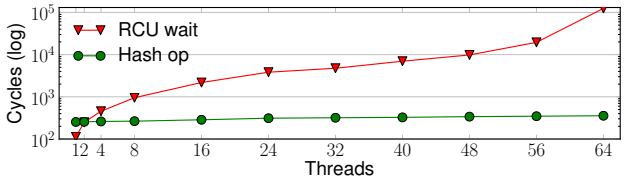

Figure 1: **RCU wait-for-readers time compared to typical data structure operation time:** Latency (log scale) of concurrent hash table lookup (read-only workload, load factor is 2) vs. wait-for-readers, on a four-processor AMD Opteron x86 machine (64 hardware threads).

Unfortunately, RCU appears unsuitable for use in data structures with scalable, fine-grained, update operations. The problem, which RCU-based data structures inevitably run into [2, 16, 27], is that wait-for-readers becomes a dominating bottleneck even if executed rarely. For example, Figure 1 shows that wait-for-readers can take 300× the time of a typical data structure operation like a hash table lookup. Consequently, a thread performing a mix of reads and updates (as usually happens) in which even 1% of operations invoke wait-for-readers will spend 75% of its time in wait-for-readers. This bottleneck stands in the way of wider RCU adoption, which would allow many use-cases to benefit from its appealing properties—particularly its simple interface, which enables writing simple and clean algorithms. Indeed, RCU use in practice remains confined to memory reclamation and to essentially read-only data structures (i.e., billion-to-one read vs. update ratios [22]).

To solve this problem and open the door to broader RCU usage, this paper presents *Predicate RCU* (PRCU), an RCU variant offering 10–100× shorter wait-for-readers times. PRCU builds on the insight that an update needs to wait *only for the reads whose consistency it might affect* and not *all* existing reads. Frequently, no such reads will be running, allowing PRCU to *avoid any waiting*. For example, a delete() of key x from a chained hash table—with a linked list for each hash bucket—only needs to wait if reads are traversing the bucket x hashes to.

PRCU uses a concise and opaque method of identifying such affected reads: a read associates itself with an (algorithm-specific) value—e.g., a key being looked-up—and wait-for-readers receives

PPoPP'15, February 7–11, 2015, San Francisco, CA, USA.

ACM 978-1-4503-3205-7/15/02…$15.00.
http://dx.doi.org/10.1145/2688500.2688518

This version corrects the proceedings version, which omits a memory fence in Algorithms 1– 3 and mistakenly attributes the algorithm in § 5.1 to [27].
Last updated: February, 2015

a user-defined *predicate*, $\mathcal{P}$, that identifies which reads to wait for—e.g, those whose hash value equals x's hash.

We present several PRCU implementations, exploring trade-offs between wait-for-readers time and read overhead. In our first implementation, EER-PRCU, wait-for-readers (e)valuates $\mathcal{P}$ for (e)ach (r)eader and waits only for those readers $\mathcal{P}$ holds for, which requires readers to post their value to memory. EER-PRCU reduces wait-for-readers time by $10\times$ with read overhead comparable to that of existing RCU algorithms [6]. However, its wait-for-readers time complexity is linear in the number of threads, *even if no waiting is needed*.

Our second implementation, D-PRCU, aims at breaking this barrier by exploiting the (d)omain of values presented to PRCU by the data structure. Here, a read interprets its value, v, as an index to a table, C, of shared counters, incrementing $C[v]$ when starting and decrementing it on completion. A wait-for-readers then essentially waits until the counters of values for which $\mathcal{P}$ holds (i.e., $\{C[v] \mid \mathcal{P}(v) = 1\}$) are zero. The idea is to provide PRCU with a predicate that holds over few values, thus drastically reducing wait-for-readers work, particularly in cases in which no waiting is needed. For example, a hash table can use a key's bucket as its value, with an update working in bucket b invoking wait-for-readers with a predicate that holds only for b. In such cases, wait-for-readers time decreases by $100\times$. In update-heavy workloads, this more than compensates for the cost of reader counter updates, yielding an overall performance gain.

Our final implementation, DEER-PRCU, represents a middle ground between EER-PRCU and D-PRCU: Like D-PRCU, it uses the domain of data structure values, but it maintains a table of counters *for each reader*. DEER-PRCU thus has linear wait-for-readers time complexity like EER-PRCU, but lower read overhead than D-PRCU. In addition, DEER-PRCU alleviates cache coherency-related *ping pongs* that occur in EER-PRCU because DEER-PRCU readers do not always post their presence to the same memory location.

We demonstrate the applicability of PRCU by applying it to two RCU-based concurrent algorithms—the CITRUS binary search tree [2] and a resizable hash table—including defining predicates suitable for D-PRCU. We show experimentally that PRCU significantly improves the performance of both algorithms compared to standard RCU, to the point of exceeding the performance of non-RCU algorithms in some cases.

2. Background: RCU

Read-copy update (RCU) is a synchronization mechanism allowing concurrent readers and updates. RCU strives to minimize the synchronization overhead on readers, by placing the burden of maintaining correctness on updates. To this end, updates *atomically* transition the data structure between *consistent* states, thereby guaranteeing that a reader always observes *some* consistent state. Typically, this means that an object is updated by *copying* it, updating the copy, and atomically swinging a pointer from the old object to the new (updated) object—hence the name *read-copy update*. Reclaiming the memory of the old copy is the original motivation for RCU's wait-for-readers primitive. (The idea is that the copy can be reclaimed after all existing reads, which might hold a reference to it, have completed.) However, wait-for-readers is useful as a general algorithmic building block [2, 16, 27].

2.1 RCU Interface and Terminology

An RCU-protected read operation is delimited by wait-free rcu_enter and rcu_exit operations and is referred to as a *read-side critical section*. (For simplicity, we do not consider nested read-side critical sections.) Times when a reader is not inside a read-side critical section are denoted *quiescent states*. A time period during which every thread goes through a quiescent state is called a *grace period*. The RCU wait-for-readers method waits for a grace period to occur, i.e., it does not return before all pre-existing readers have exited their critical section by calling rcu_exit. The RCU *safety property* (Figure 2) formalizes this: if the return from a rcu_enter precedes the call to wait-for-readers than any operation of the read side critical section precedes the return from wait-for-readers [10, 11]. Notice that there is no promise of returning *soon* after a grace period occurs, and RCU implementations exploit this to obtain zero-overhead reads at the price of extremely long wait-for-readers times.

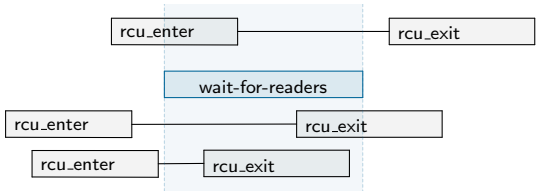

Figure 2: RCU safety property: semantics of wait-for-readers.

Asynchronous wait-for-readers RCU overcomes the wait-for-readers bottleneck for memory reclamation, its main current use case, by providing a method, call_rcu, that *defers* the reclamation to the end of the grace period without blocking the calling thread. Internally, call_rcu records the desired action (passed as a callback function) and periodically checks whether a grace period has occurred, at which point it invokes the callback. Still, wait-for-readers must be used when strict memory bounds are required, since call_rcu can accumulate unbounded amounts of unreclaimed memory until a grace period occurs. In this paper, we focus on wait-for-readers, since fast wait-for-readers time can be easily translated into faster grace period detection for call_rcu.

2.2 RCU Implementations

Below we describe existing RCU implementations and their performance characteristics (cf. § 6).

Tree RCU (Linux kernel implementation [20]) Conceptually, Tree RCU maintains a bit string with a bit per thread. A wait-for-readers operation sets all the bits, and a thread entering a quiescent state clears its bit. (wait-for-readers operations are serialized.) A grace period occurs when all bits are clear. The bit string is implemented in a hierarchical tree manner: A leaf packs several thread bits, and each level up contains a bit per child, indicating if all the bits of the child leaf are cleared. The last thread to clear its bit in a leaf clears the leaf's bit in the parent, repeating this up the tree as long as is keeps clearing the last bit on the path to the root. wait-for-readers detects the grace period by polling the root node.

Performance: Tree RCU leverages its in-kernel implementation to obtain zero overhead rcu_enter and rcu_exit. It achieves this by defining special code locations—such as during context switch code—as quiescent states, relegating the bit checking to this code only. However, wait-for-readers times become long—possibly tens of milliseconds—since they are tied to the OS scheduling. Tree RCU is not suitable for general purpose userspace code, in which quiescent states cannot be easily identified.

URCU The userspace RCU library [6] uses a *global grace period counter*. Each thread maintains a variable indicating whether or not it is inside a read-side critical section together with a snapshot of the grace period counter at the beginning of the critical section. A thread calling wait-for-readers acquires a global lock protecting the grace period counter, increments the counter and waits for each thread that is currently inside a read-side critical section that started before the current grace period.

Performance: URCU abandons the zero read overhead approach; its rcu_enter writes to memory and issues a memory fence. In exchange for this it gains faster wait-for-readers time and generality. However, URCU's global wait-for-readers lock prevents wait-for-readers performance from scaling [2].

Batching Both Tree RCU and URCU employ *batching* to compensate for the serialization of wait-for-readers operations: if a wait-for-readers operation, w, arrives while another waiter, w_0, is running but before w_0 has seen any quiescent state, then w can "piggyback" on w_0's grace period, instead of waiting after w_0 completes. However, URCU batches waiters in a FIFO queue, which is itself a contended bottleneck, and thus URCU wait-for-readers remains unscalable. In addition, batching cannot decrease the fundamentally long grace period time of Tree RCU.

Distributed counters RCU Arbel and Attiya's RCU implementation [2] supports multiple RCU waiters without synchronization among them. Instead of a global grace period counter, each thread maintains a local critical section counter. A thread calling wait-for-readers waits for each thread to increment its local counter or indicate that it is no longer inside a read-side critical section.

Performance: The read overhead is comparable to URCU—a write and memory fence. wait-for-readers operations scale much better than in URCU, since they are read-only.

3. Predicate RCU (PRCU)

PRCU aims to address the inherent lack of scalability of wait-for-readers, by providing RCU-using algorithms with an interface to specify *which* reads a wait-for-readers should wait for. The observation behind this approach is that an update invoking wait-for-readers only needs to wait for reads whose consistency it might affect [2, 16, 27], but is forced to wait for all reads because of the conservative wait-for-readers interface. This section defines the PRCU interface, which enables an algorithm to express precisely which reads it wishes to wait for.

3.1 PRCU Interface

Values For simplicity, we describe PRCU methods as accepting values from a data structure-specific domain, $\mathcal{D}$. However, these values are opaque to PRCU, and so we envision PRCU implementations accepting a generic encoding of values (say, 64-bit integers) that can be applied to different algorithms.

Methods Similarly to the RCU interface, the PRCU interface consists of the methods below:

1. prcu_enter(v) and prcu_exit(v), where $v \in \mathcal{D}$. Similarly to RCU, the interval between a prcu_enter(v) and a matching prcu_exit(v) is named a *read-side critical section on* v. For simplicity, we do not allow nesting of read-side critical sections, but it is straightforward to support.
2. wait-for-readers($\mathcal{P}$), where $\mathcal{P}\colon \mathcal{D} \to \{0, 1\}$ is referred to as the *predicate*. We say that $\mathcal{P}(v)$ *holds* if $\mathcal{P}(v) = 1$.

We say that a thread *enters* the read-side critical section when its prcu_enter(v) completes, and *exits* the critical section when it invokes prcu_exit(v).

PRCU safety A PRCU implementation must satisfy the PRCU *safety* property: if a read-side critical section on v, r_v, is entered before the invocation of a wait-for-readers($\mathcal{P}$) and $\mathcal{P}(v)$ holds, then r_v is exited before the return of the wait-for-readers($\mathcal{P}$). That is, wait-for-readers($\mathcal{P}$) blocks until the completion of all existing read-side critical sections of values v for which $\mathcal{P}(v)$ holds.

Encoding predicates To facilitate efficient predicate evaluation, we require the implementation of wait-for-readers($\mathcal{P}$) to accept its predicate in the form of a function in the underlying programming language, e.g. a C function pointer. PRCU assumes such a predicate-encoding function has no side effects, and wait-for-readers($\mathcal{P}$) may invoke $\mathcal{P}$ any number of times during its execution. (Users should thus strive to implement predicates efficiently.)

Specialized predicates A PRCU implementation may wish to exploit structural properties of $\mathcal{P}$ to implement wait-for-readers($\mathcal{P}$) more efficiently—for example, D-PRCU (§ 4.2) depends on being able to quickly iterate over the values $\mathcal{P}$ holds for. To this end, an implementation may accept additional *specialized* predicate encodings that succinctly expose the desired properties. Our implementations support two such specialized predicates: (1) *singletons*, which hold only for a single value and are encoded as that value, and (2) *iterable predicates*, which hold over some set of values $\{v_1, v_2 = \text{next}(v_1), v_3 = \text{next}(v_2), \ldots, v_k\}$, where $\text{next}\colon \mathcal{D} \to \mathcal{D}$ is referred to as the *iterator*. Iterable predicates are encoded as (v_1, v_k, next), where the iterator is passed as a function object. (Note that a singleton is an iterable predicate, but for simplicity we distinguish the two.)

RCU fallback Our experience (§ 5) is that domain values for read-side critical sections that wrap concurrent data structure operations map naturally to the semantics of the data structure, and are therefore straightforward to define. In general, however, it may not be possible to a priori define a domain value. In such cases, programmers can define a *wildcard* value for which every predicates always holds. Wildcards enable PRCU to "fall back" into standard RCU mode, in which a wait-for-readers($\mathcal{P}$) waits for all readers.

4. PRCU Implementations

This section describes several PRCU implementations with different trade-offs—which we discuss in detail—between wait-for-readers time and overhead imposed on read operations.

For simplicity, we assume a standard sequentially consistent (SC) memory model. In practice, an implementation must prevent hardware and compiler reordering of memory operations. x86 processors—our evaluation platform—implement a TSO memory model [25] that can only reorder a load with a prior store to a different address. Our pseudo code thus indicates where *memory fence* instructions should be placed to prevent such reordering. We omit *compiler fences* used to prevent instructions from escaping out of read-side critical sections and across wait-for-readers($\mathcal{P}$) calls. We leave to future work the placement of fences for memory models weaker than TSO, such as those of ARM and POWER processors.

4.1 EER-PRCU

In EER-PRCU, wait-for-readers($\mathcal{P}$) ⓔvaluates $\mathcal{P}$ for ⓔach existing ⓡeader and waits only for those readers $\mathcal{P}$ holds for. To this end, EER-PRCU maintains an array of single-writer multi-reader nodes—one for each thread. A prcu_enter(v) writes v into a field in the node associated with the invoking thread, while a wait-for-readers($\mathcal{P}$) scans the array and waits for any thread whose node contains a value for which $\mathcal{P}$ holds.

We implement waiting using *time-based* quiescence detection (a generalization of *epoch-based* detection [9]) which works by waiting until a thread posts in memory that it has entered or exited a read-side critical section at a time after wait-for-readers($\mathcal{P}$)'s invocation time. Time-based quiescence detection requires some *global clock*, represented as a monotonically increasing time() method, which we discuss how to implement shortly. (Memory models weaker than the SC model we use require more involved time() properties [26], but the implementations we discuss below are compatible with these properties as well.)

Algorithm 1 presents the EER-PRCU pseudo code. In addition to the value field, a thread maintains a time field, which either

Algorithm 1 EER-PRCU: code for thread T_i

```
1:  Node type:
2:      struct { value : 64-bit int, time : 64-bit int }
3:  function prcu_enter(v)
4:      Nodes[i].value ← v
5:      Nodes[i].time ← time()
6:      ▷ TSO requires a memory fence here.
7:  function prcu_exit(v)
8:      Nodes[i].time ← ∞
9:  function wait-for-readers(P)
10:     ▷ TSO requires a fence here, to make updater's writes visible
11:     t0 ← time()
12:     for each thread Tj ≠ Ti do
13:         if P(Nodes[j].value) then
14:             while true do                          ▷ Wait for Tj
15:                 t ← Nodes[j].time
16:                 if t > t0 then break
```

contains a time value read in the execution interval of prcu_enter, or—if the thread is not inside a read-side critical section—contains ∞. To wait for a reader, a wait-for-readers($\mathcal{P}$) spins, reading this time field until it reads a value $t > t_0$, where t_0 is the time in which the wait-for-readers($\mathcal{P}$) started (Line 11). The waiting process is read-only, and thus scales well to concurrent invocation of wait-for-readers($\mathcal{P}$) that occur in algorithms with scalable updates.

Proposition 1. *EER-PRCU satisfies the PRCU safety property.*

Proof. Let w be a wait-for-readers($\mathcal{P}$) invocation by thread T_i, and let r_v be a read-side critical section on v, where $\mathcal{P}(v)$ holds. Suppose, towards a contradiction, that r_v's prcu_enter completes before w's invocation, and yet w completes before r_v invokes prcu_exit. Thus, r_v is invoked by thread $T_j \neq T_i$ and so T_i reads v from Nodes[j].value (Line 13) (since r_v has not completed at this point). w therefore runs Line 16 until reading a time t strictly greater than w's t_0. Since r_v still has not completed at this point, t is the value written by r_v's prcu_enter. This is a contradiction, since time() is monotonically increasing. □

Trade-offs Compared to OS kernel-level RCU [21], in which rcu_enter is a no-op with zero overhead, EER-PRCU imposes additional overhead by updating the node, which in practice requires issuing a memory fence to prevent reordering of the node updates with the subsequent reads done in the critical section. (Comparable read overhead exists in other RCU implementations, which have abandoned the zero overhead approach to provide userspace functionality [6] or shorter grace periods [19]). In exchange for this small read overhead, EER-PRCU obtains a read-only wait-for-readers($\mathcal{P}$) implementation which thus scales as the number of concurrent wait-for-readers($\mathcal{P}$) instances grows. However, the time complexity of wait-for-readers($\mathcal{P}$) is linear in the number of threads in the system, even if waiting turns out to be unnecessary.

Clock implementation Our evaluated EER-PRCU implementation (§ 6) implements time() by reading the machine's *timestamp counter* (TSC), a cheaply-accessible x86 hardware counter that is architecturally defined as suitable for time-keeping purposes [1]. Other approaches exist for architectures without such a hardware counter. For example, the clock can be implemented based on an *epoch count* [9], or one can use thread-private clocks instead of a global clock, similar to the CITRUS tree's RCU implementation [2].

4.2 D-PRCU

D-PRCU tries to exploit the (d)omain of the predicate to reduce the number of memory locations a wait-for-readers($\mathcal{P}$) scans when looking for relevant readers—e.g., to $O(1)$ locations. This makes wait-for-readers($\mathcal{P}$) scalable in the number of threads, because

Algorithm 2 D-PRCU with iterable predicate: code for thread T_i

```
1:  C node type:
2:      struct { gate : bit, readers[2] : 64-bit int, lock : mutex }
3:  Thread-local variable: b : bit
4:  function prcu_enter(v)
5:      b ← C[h_rcu(v)].gate
6:      fetch-and-add(C[h_rcu(v)].readers[b], 1)
7:      ▷ TSO atomic operation also acts as a memory fence.
8:  function prcu_exit(v)
9:      fetch-and-add(C[h_rcu(v)].readers[b], -1)
10: function wait-for-readers(P=(v1, vk, next))
11:     ▷ TSO requires a fence here, to make updater's writes visible
12:     for each v ∈ {v1, v2 = next(v1), ..., vk} do
13:         drain(h_rcu(v))
14: function drain(j)
15:     C[j].lock()
16:     g ← C[j].gate
17:     await(C[j].readers[¬g] = 0)
18:     C[j].gate ← ¬g
19:     await(C[j].readers[g] = 0)
20:     C[j].unlock()
```

frequently no waiting is required, which turns the act of *detecting* this case into the critical path of wait-for-readers($\mathcal{P}$).

The idea in D-PRCU is to track all readers with the same value v in the same memory location $C[v]$, allowing wait-for-readers($\mathcal{P}$) to scan only locations $C[v]$ for which $\mathcal{P}(v)$ holds. To do this efficiently, D-PRCU requires a *specialized* predicate that succinctly encodes $\mathcal{P}^{-1} \doteq \{v \mid \mathcal{P}(v) = 1\}$. Such predicates often arise naturally when applying PRCU to a scalable data structure (see § 5).

D-PRCU aggressively trades off short wait-for-readers($\mathcal{P}$) times with higher read overhead. This trade-off pays well for update-heavy workloads (§ 6), but is likely not appropriate elsewhere. (We discuss this later.)

Implementation Conceptually, a D-PRCU reader interprets v as an index to a table, C, of shared counters, atomically incrementing $C[v]$ in prcu_enter and decrementing it in prcu_exit. [1] A wait-for-readers($\mathcal{P}$) then scans $C[v]$ for each $v \in \mathcal{P}^{-1}$, and waits until $C[v]$ becomes zero. To implement this scanning, we assume an iterable predicate; dealing with a general predicate is described later.

Due to practical concerns, the implementation (Algorithm 2) deviates from the above description in two ways. First, we use a more complex waiting protocol—explained below—to prevent wait-for-readers($\mathcal{P}$) from waiting forever at some $C[v]$, which can happen if prcu_enter(v) invocations during the wait-for-readers($\mathcal{P}$) execution interval keep $C[v] > 0$ at all times. Second, since the domain $\mathcal{D}$ is opaque to PRCU (and may be huge), we use a hash function $h_{\text{rcu}} : \mathcal{D} \to [|C|]$ to map values into C. [2]

Waiting protocol We use a protocol similar to that of SRCU [19]. A node $C[v]$ contains two counters, readers[i], $i \in \{0, 1\}$, and a *gate* bit that specifies which counter readers should increment. A reader remembers the gate value, b, it observes when entering the read-side critical section (Line 5), and uses it to decrement the same counter when exiting the critical section. To wait at node $C[v]$, wait-for-readers($\mathcal{P}$) reads g, the value of the gate, and then waits

[1] The pseudo code implements atomic counter updates using fetch-and-add, which is available on x86 machines. On architectures without fetch-and-add, counter can be updated using a compare-and-swap loop.

[2] Hashing introduces the possibility of multiple values in $\mathcal{P}^{-1}$ mapping to the same C element, but wait-for-readers($\mathcal{P}$) can easily avoid waiting at the same node twice. We omit this detail from the pseudo code.

for readers[$\neg g$] to be 0 (Lines 16–17). This "drains" any readers that read $\neg g$ from the gate, since new readers arriving do not increment readers[$\neg g$]. Next, wait-for-readers($\mathcal{P}$) toggles the gate and then waits for readers[g] to be 0 (Lines 18–19). This now drains readers that read g from the gate, ensuring that—pending enough steps by readers—wait-for-readers($\mathcal{P}$) does not wait forever. (Note that concurrent wait-for-readers($\mathcal{P}$) operations synchronize using a lock in each C node.) The following establish the safety of the waiting protocol.

Lemma 1. *Let w be a wait-for-readers($\mathcal{P}$) operation, and let r_v be a read-side critical section on v entered before w's invocation. Then, if w reads 0 from $C[j]$.readers[0] and from $C[j]$.readers[1], where $j = h_{rcu}(v)$, r_v is exited before w completes.*

Proof. Let i be the value r_v reads from $C[j]$.gate in prcu_enter (Line 5). Because r_v's prcu_enter(v) completes before w starts, $C[j]$.readers[i] > 0 when w starts. Then w reading 0 from $C[j]$.readers[i] implies that r_v has decremented $C[j]$.readers[i] by that time, i.e., r_v has invoked prcu_exit during w's execution. □

Proposition 2. *D-PRCU satisfies the PRCU safety property.*

Proof. Immediate from Lemma 1. □

Optimistic waiting Our evaluated D-PRCU implementation adds an *optimistic waiting* optimization to Algorithm 2. The idea is to hope that readers will drain naturally, and thus avoid acquiring the lock and toggling the gate bit. Thus, a wait-for-readers($\mathcal{P}$) waits at node $C[v]$ by first spinning for a while, reading both of $C[v]$'s counters, until it has observed 0 in each counter. It moves to the full protocol only after timing out in this loop. Lemma 1 implies the safety of this optimization. Optimistic waiting increases concurrency among wait-for-readers($\mathcal{P}$) operations and reduces their latency, particularly when no waiting is required, i.e., both counters are zero to begin with.

Trade-offs The amount of read overhead D-PRCU imposes on reads depends on the workload. Concurrent readers accessing the same value turn the relevant counter into a contended bottleneck, as with standard read/write locks [4]. In contrast, when readers access disjoint values, a counter update becomes an uncontended atomic operation, which is relatively cheap on modern machines. (However, as different values map to distinct counters, the chance of the counter update entailing a cache miss increases—as opposed to EER-PRCU, in which a reader updates one memory location.) In exchange for this, wait-for-readers($\mathcal{P}$) time becomes almost negligible, which can more than compensate for the read overhead in an update-heavy workload (see § 6).

General predicate support For completeness, we describe how D-PRCU supports general predicates. Given a non-specialized $\mathcal{P}$, wait-for-readers($\mathcal{P}$) applies the waiting protocol at each node of C. This likely obviates the benefit of D-PRCU over EER-PRCU, as we expect $|C|$ to be greater than the number of threads, to reduce chance of contended counters.

Further optimizations D-PRCU can benefit from a couple of standard optimizations, which we leave for future work: First, expanding the C table, to address hash collisions that lead to contention on a counter. (Doing this requires a global wait-for-readers, to drain readers from the old table.) Second, *batching* in the wait protocol [19]. Here, if a wait-for-readers($\mathcal{P}$) operation, w, finds the lock of node $C[v]$ taken but the lock holder, w_0, has not yet read 0 from any of the counters, then w can "piggyback" on w_0 and avoid going through the wait protocol once the lock is released.

4.3 DEER-PRCU

The DEER-PRCU implementation incorporates the idea of exploiting a specialized predicate's domain into EER-PRCU, with the goal of alleviating a cache coherency-related *ping pong* problem in EER-PRCU: A reader and wait-for-readers($\mathcal{P}$) require conflicting rights to the cache line of the reader's node—exclusive access for the reader as opposed to read access for the wait-for-readers($\mathcal{P}$). As a result, both reader and wait-for-readers($\mathcal{P}$) incur a cache miss when accessing the node.

Algorithm 3 DEER-PRCU with iterable predicate: thread T_i code

```
1:  C_i node type:
2:      struct { value : 64-bit int, time : 64-bit int }
3:  function prcu_enter(v)
4:      C_i[h_rcu(v)].value ← v
5:      C_i[h_rcu(v)].time ← time()
6:      ▷ TSO requires a memory fence here.
7:  function prcu_exit(v)
8:      C_i[h_rcu(v)].time ← ∞
9:  function wait-for-readers(P=(v_1, v_k, next))
10:     ▷ TSO requires a fence here, to make updater's writes visible
11:     t_0 ← time()
12:     for each thread T_j ≠ T_i do
13:         for each v ∈ {v_1, v_2 = next(v_1), ..., v_k} do
14:             while true do                          ▷ Wait for T_j
15:                 t ← C_j[h_rcu(v)].time
16:                 if t > t_0 then break
17:             ▷ T_j entered CS while we are running?
18:             if t ≠ ∞ then break
```

DEER-PRCU uses specialized predicates to alleviate cache line ping pongs. Each reader r maintains an array C_r of EER-PRCU-style nodes, updating the value and time in node $C_r[h_{\text{rcu}}(v)]$ when entering a read-side critical section on v. (We update the value to support general predicates, as explained below.) A wait-for-readers($\mathcal{P}$) scans only the times in nodes $C_r[h_{\text{rcu}}(v)]$, $v \in \mathcal{P}^{-1}$, for each reader r. Consequently, a reader and wait-for-readers($\mathcal{P}$) that do not conflict semantically also do not conflict at the memory operation level. Algorithm 3 shows the complete implementation. The waiting protocol differs from EER-PRCU in that it terminates only after observing $t_0 < t \neq \infty$, as that implies that any pre-existing read-side critical section has completed. We thus have:

Proposition 3. *DEER-PRCU satisfies the PRCU safety property.*

DEER-PRCU exploits the fact that the C_r arrays are single-writer to support general predicates more efficiently than D-PRCU. A reader r writes its current value, v, to $C_r[h_{\text{rcu}}(v)]$, which allows wait-for-readers($\mathcal{P}$) to evaluate $\mathcal{P}$ for each $C_r[j]$ node and wait if required. Unlike D-PRCU, these arrays can be small as there is no concern of hash collisions due to concurrent accesses—we use 16 elements in our DEER-PRCU implementation, which enables quick scanning by wait-for-readers($\mathcal{P}$). (We omit general predicate support from the code.)

DEER-PRCU is especially effective on systems with older Westmere Intel processors, which appear to serialize cross-processor get-read-ownership coherency transactions. In EER-PRCU, this causes the latency of the load instructions issued by wait-for-readers($\mathcal{P}$) to read the readers' time field to *increase* with the amount of concurrent wait-for-readers($\mathcal{P}$) invocations, because the time field is frequently written to by the reader. In DEER-PRCU, however, a reader and wait-for-readers($\mathcal{P}$) access to the same time field less frequently—only when they conflict semantically—and so the time fields are usually in shared state in the caches. Consequently, on these systems wait-for-readers($\mathcal{P}$) in DEER-PRCU is about $4\times$ faster than in EER-PRCU.

5. Applying PRCU to Algorithms

This section provides two examples demonstrating the process of replacing RCU with PRCU in RCU-based algorithms.

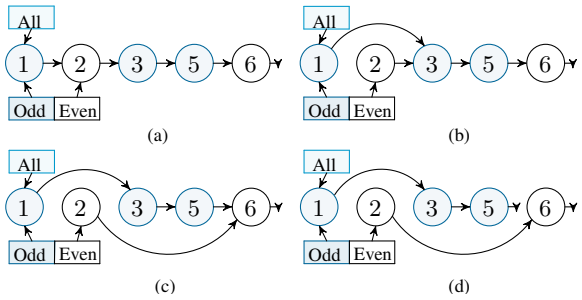

Figure 3: Expanding a hash table with a single bucket (*All*) into a table with two buckets.

5.1 Resizable Hash Table

We consider a closed addressing hash table that implements buckets using RCU-based linked lists, as in the algorithm of Triplett et al. [27]. Lookups can thus safely traverse the buckets concurrently to updates, which synchronize via per-bucket locks. Our table expansion algorithm, described below, differs from the one described by Triplett et al. [27] in that it uses wait-for-readers in a more fine-grained manner. Since insert()s are prevented during expansion [27], excessive expansion time due to wait-for-readers calls poses a problem in our table expansion variant.

Table expansion The table uses a modulo-table-size hash function to ensure that during an expand operation each original bucket is split into two new buckets. The expand operation creates a new array of buckets. It then links each new bucket to an existing bucket's linked list, pointing to the first node that matches the new bucket (Figure 3a). Next, the expand splits the old bucket into two lists, one for each bucket (Figure 3b– 3d). The expand calls wait-for-readers before every pointer change in this splitting process. This makes sure that lookups do not traverse the wrong linked list. For example, in Figure 3, if a lookup of 2 reaches the node 1 before the new bucket array is created (before (a)), then gets delayed until (b), it will incorrectly fail to find 2. Similarly, if a lookup of 6 starts at time (b) and reaches node 3, then gets delayed until (d), it will incorrectly miss 6. Using wait-for-readers before every unlinking step prevents these problems.

Applying PRCU An expand of bucket b needs to wait only for operations accessing nodes linked from b. Thus, we define the values for read-side critical section as its bucket. Then, an expand splitting bucket b_{old} needs to wait for reads accessing b_{old} and b_{new}, the bucket containing items being split. This is naturally expressed passing $\mathcal{P}(x) = (x = b_{\text{old}} \vee x = b_{\text{new}})$ to wait-for-readers($\mathcal{P}$), which is also an iterable predicate.

5.2 The CITRUS Tree

CITRUS is a concurrent binary search tree providing a wait-free contains() operation that can run concurrently with tree updates. CITRUS uses fine-grained locking to synchronize updates, and RCU to protect tree traversal—both contains() queries and the traversal prefixes of insert() and delete() updates.

Citrus implements an *internal* search tree, i.e., each node holds a *key* (henceforth, we will refer to nodes and keys interchangeably). Therefore, CITRUS must support deletion of an internal node with two children, as such a node cannot be merely linked out of the tree. Deleting a node k with two children requires replacing k with its successor k', that is, moving the leftmost node in k's right subtree, T, upwards (see left side of Figure 4). However, simply replacing k with k' can cause a contains() to return the wrong result. For example, moving k' up may cause a contains(k') concurrently traversing T to incorrectly return `false`.

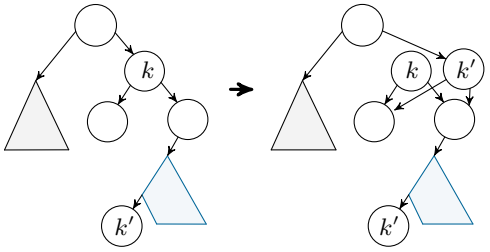

Figure 4: CITRUS: Deletion of internal node with key k.

In general, replacing k with k' *shrinks* the interval of keys that can be present in T—specifically, any search for key in $(k, k']$ would formerly enter T, but with k' at T's root would turn left instead. It may seem that because k' is the successor of k, this only affects contains(k')—after all, a contains() for a different value that ends up at the old parent of k' and returns `false` still returns the correct semantic result. However, an insert() in CITRUS performs its initial search just like a contains()—optimistically, in a read-side critical section. Consequently, an insert(k^*), $k^* \in (k, k']$, that is traversing T, will stop at the old parent of k' and insert k^* as its child, leading to a *lost update*.

CITRUS uses RCU to avoid these problems. Instead of moving the successor, k', to the new location, it uses a copy of k' (see Figure 4) and removes the original k' only after calling wait-for-readers. Thus, new operations find k' in its new location, while the original k' disappears only after every pre-existing traversal finishes. In addition, CITRUS prevents pre-existing insert()s from inserting new keys as children of the original k' by *marking* this node [2]. The wait-for-readers thus guarantees consistency of CITRUS operations.

Applying PRCU CITRUS needs to wait only for searches on keys in $(k, k']$. (Its correctness proof [2, Lemma 4] shows this formally.) This can be easily expressed using a PRCU predicate $\mathcal{P}(x) = x > k \wedge x \leq k'$, coupled with having operations pass their search key as the value in prcu_enter/prcu_exit.

To apply D-PRCU, we need to express $\mathcal{P}$ as an iterable predicate. If the key domain, $\mathcal{D}$, is finite or countably infinite, $\mathcal{P}$ can be expressed as $(\text{succ}_{\mathcal{D}}(k), k', \text{succ}_{\mathcal{D}})$, where $\text{succ}_{\mathcal{D}}$ is the successor function over $\mathcal{D}$—e.g., $\text{succ}_{\mathcal{D}}(x) = x + 1$ over the integers. However, because D-PRCU requires $\mathcal{P}$ to hold over few values to obtain sub-linear wait-for-readers time (§ 4.2), this naive approach works only if the interval $(k, k']$ is small.

In many cases, we can obtain good D-PRCU performance by presenting it with a *compressed* domain, in which the intervals will usually be small. The idea is to hash the keys so that both of an interval's endpoints likely fall into the same bucket. For example, with integer keys we can divide $\mathcal{D}$ to equally sized intervals, mapping all keys in each interval to the same bucket. That is, a search for k calls prcu_enter/prcu_exit with value $v = \lfloor k/S \rfloor$, and a delete() that needs to wait for searches in $(k, k']$ uses the predicate $\mathcal{P}(x) = x \in (\lfloor k/S \rfloor, \lfloor k'/S \rfloor]$. The factor S can be fixed, derived from properties of the workload, or with the assistance of PRCU. Our evaluation uses the latter approach, using $S = |C|$, the size of D-PRCU's counter table.

Alternative to PRCU? CITRUS can seemingly avoid the wait-for-readers bottleneck by using call_rcu; this is correct because a delete()'s linearization point occurs before the wait-for-readers. However, a delete() can release its locks only after the grace period, which in existing call_rcu implementations is detected by dedicated threads. The increased lock hold times resulting in this approach decrease throughput by 2×–100×, depending on the level of contention in the tree.

6. Evaluation

This section evaluates the impact of applying PRCU to the CITRUS tree and resizable hash table (as described in § 5).

Platform We use a 64-core x86 system, consisting of 4 AMD Opteron 6376 (Abu Dhabi) processors, each with 16 2.3 GHz cores. We use C implementations of the algorithms and RCU, compiled with `gcc` 4.6.3 at `O3` optimization level. To prevent memory allocation from being a bottleneck, we use the scalable `jemalloc` multi-threaded memory allocator.

RCU implementations We compare our PRCU implementations to three RCU implementations. First, **URCU**, Desnoyers and McKenney's userspace RCU implementation [6], an optimized RCU implementation available as open source. Second, **Tree RCU**, our implementation of the Linux kernel hierarchical RCU algorithm [20]. While Tree RCU is not suitable for general userspace code, we manage to apply it in our restricted setting by treating the states between data structure operations as quiescent (cf. §2). (Note that as a result, our Tree RCU has significantly shorter grace periods than the Linux Tree RCU.) Finally, **Time RCU**, which uses time-based quiescence detection—similarly to RCU implementation used in the CITRUS tree [2], but optimized to use the system's timestamp counter (TSC) as the PRCU implementations do. Time RCU serves two purposes: First, it is essentially EER-PRCU without the predicate evaluation, which allows teasing apart the impact of using the predicates. Second, Time RCU performs better than Tree RCU and URCU on workloads with updates [2], and so enables a fairer comparison against RCU.

PRCU parameters The D-PRCU implementation uses a 1024-counter table. The DEER-PRCU implementation uses a 16-entry node array for each thread.

6.1 CITRUS Tree

Methodology We follow prior work on concurrent search trees (e.g., [3, 17, 24]) and measure the throughput of the trees using a benchmark in which threads repeatedly invoke operations following a specified distribution, with integer keys selected uniformly from a given range. We use the following operation distributions, which simulate various common workloads: **read-dominated** (98% contains(), insert() and delete() each 1%), **mixed** (70% contains(), insert() and delete() each 15%) and **write-dominated** (50% insert() and 50% delete()). To understand RCU read overhead, we additionally use a **read-only** distribution (100% contains()). Initially, the tree contains $K/2$ random keys, where K is the size of the key space; the equal insert()/delete() probability keeps the tree at roughly this size throughout the experiment. We show results for $K \in \{2 \times 10^4, 2 \times 10^6\}$—results for other key ranges are similar. Each experiment runs for 3 seconds, and we report the median of 5 experiments (all experiments have negligible variance).

Non-RCU trees We do not claim PRCU-based CITRUS to be the best performing search tree, but rather a demonstration of the performance gains achievable by replacing RCU with PRCU. Nevertheless, as a performance yardstick, we evaluate **Opt-Tree**, the optimistic relaxed balance AVL tree of Bronson et al. [3] [3] and **LF-Tree**, the recent lock-free tree of Natarajan and Mittal [24]. However, we report only the Opt-Tree results, to maintain legibility of the plots as LF-Tree usually outperforms Opt-Tree and CITRUS by 2×. Note that Opt-Tree is the more meaningful apples-to-apples comparison, as it is a *lock-based internal* tree like CITRUS, as opposed to the *lock-free* and *external* LF-Tree.

[3] Implemented in C by Philip W. Howard: `https://github.com/philip-w-howard/RP-Red-Black-Tree`.

Throughput Figure 5 shows the throughput of the tested implementations, exposing several trends. First, EER-PRCU outperforms RCU, obtaining 1.2× better throughput than Time RCU (the best of the RCU variants) and 1.5×–3× higher throughput than URCU, the main RCU library available today. Second, D-PRCU CITRUS gains advantage as the update rate increases—it is worse than Time RCU on the read-dominated workload, equals EER-PRCU in the mixed workload, and outperforms EER-PRCU on the write-dominated workload by 1.45× (large tree) to 1.74× (smaller tree). In the write-dominated workload, D-PRCU even outperforms Opt-Tree—which is the best performer in the other workloads—by ≈ 10%. Third, EER-PRCU and DEER-PRCU perform comparably, with an advantage to EER-PRCU in read-dominated workloads that shrinks as the update rate increases, shifting to an advantage to DEER-PRCU in the write-dominated workload—about 1.3× on the smaller tree. Fourth, when using Tree RCU and URCU, CITRUS exhibits poor performance in the non read-dominated workloads.

To explain these performance trends, Figure 6 by depicts the wait-for-readers latency and the resulting time CITRUS spends in wait-for-readers in the smaller tree. In the read-dominated workload, compared to Time RCU, EER-PRCU has 14× shorter wait-for-readers times, D-PRCU is 25× shorter, and DEER-PRCU is 4× shorter. Consequently, the overall time PRCU-based versions spend in wait-for-readers reduces by these factors. However, these versions spend little time in wait-for-readers—which is rarely invoked in this workload (< 1%)—and so throughput improvement is modest compared to wait-for-readers time decrease. (The exceptions are URCU and Time RCU, which are 2.5× and 40× worse than Time RCU.) Moreover, D-PRCU's shortest wait-for-readers time do not translate to the best performance because of the read overhead it imposes.

These trends change in the write-dominated workload, with EER-PRCU obtaining 3× shorter wait-for-readers times than Time RCU, while D-PRCU becomes 100× better than Time RCU and DEER-PRCU 4× better. As a result, the D-PRCU version spends negligible amount of time in wait-for-readers and obtains the best performance. In contrast, the throughput advantage of DEER-PRCU over EER-PRCU is not explained by wait-for-readers time, since DEER-PRCU spends only 7% less time in wait-for-readers compared to EER-PRCU. EER-PRCU's throughput suffers because of increased read overhead due to cache coherency read/write ping pongs on EER-PRCU's book-keeping data (§ 4.3), which DEER-PRCU alleviates. This effect is pronounced in this workload because of the high rate of wait-for-readers invocations.

Read overhead Overhead that RCU/PRCU impose on reads consists of two factors: First, the cost of rcu_enter and rcu_exit, unrelated to any interaction with wait-for-readers operations. We quantify this cost by comparing the *read-only* throughput of the tested implementations (Figure 7, which we analyze below). Second, *cache coherency related* costs, that occur as a result of wait-for-readers accessing book-keeping data that a reader updates. This cost is not observable in a read-only workload. We measure the cache coherency costs using *simulated* wait-for-readers versions of the tested implementations, in which wait-for-readers performs no memory accesses and only waits the same average number of cycles that the real version waits for. The throughput of the simulated wait-for-readers version thus accounts only for the cost imposed by *waiting* in wait-for-readers, without the cache coherency overhead created by accesses to book-keeping data.

In the following, we analyze the overhead imposed by both of the above factors.

Read-only cost The read-only cost is fixed, and so the overhead it imposes diminishes as read operations become longer. Figure 7 quantifies this by comparing the *read-only* throughput of the tested implementations. On the smaller tree (10K nodes), Tree RCU—

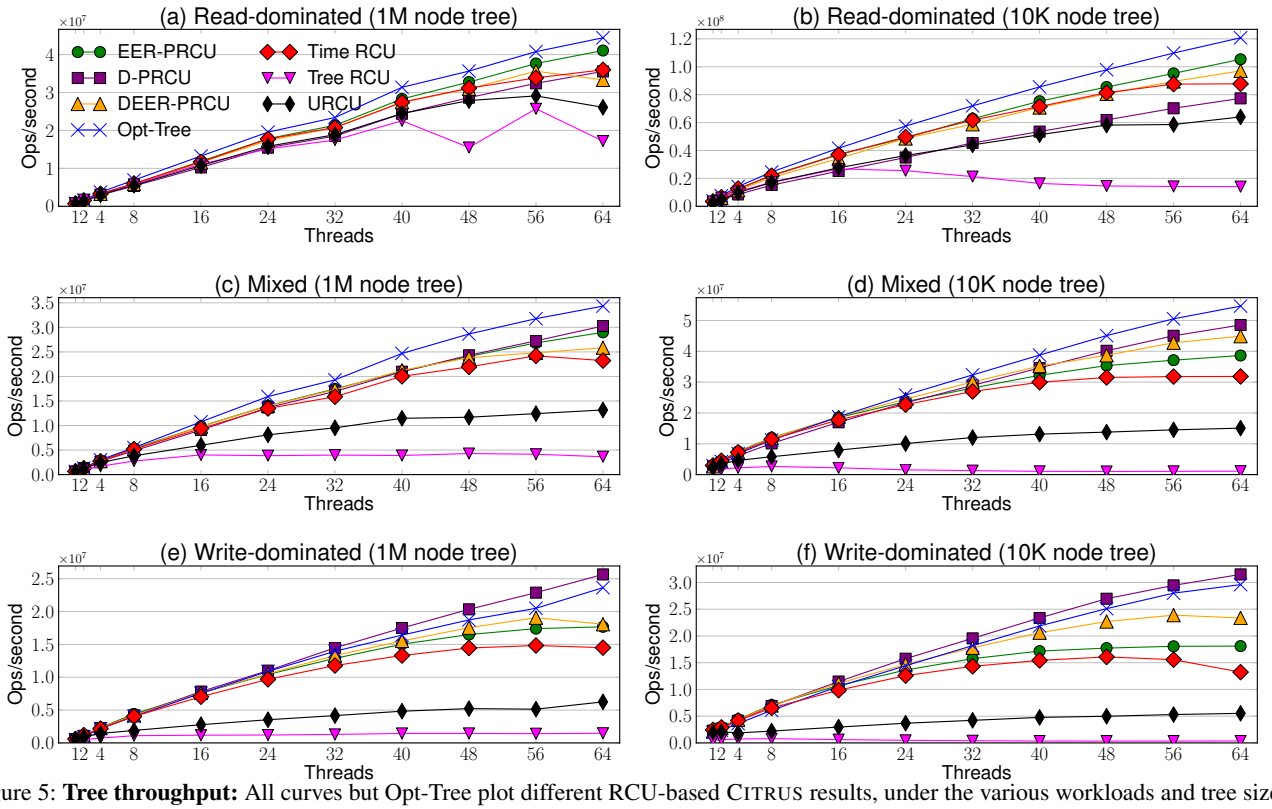

Figure 5: **Tree throughput:** All curves but Opt-Tree plot different RCU-based CITRUS results, under the various workloads and tree sizes.

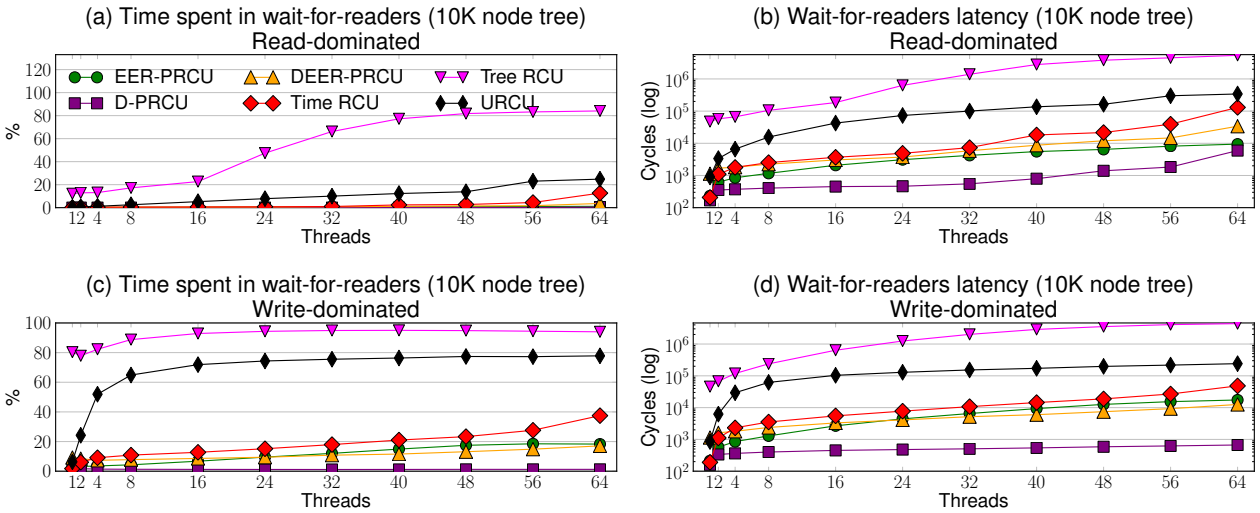

Figure 6: **Tree wait-for-readers overhead:** Overall time spent in wait-for-readers and individual wait-for-readers latency.

which has no read-only cost—is the best performer, with 9% better throughput than Opt-Tree, which imposes modest read-only cost due to validation checks its traversal code performs. Time RCU, EER-PRCU and DEER-PRCU have comparable overhead, about 10% worse than Tree RCU. URCU performs worse than them—0.65× worse than Time RCU. This is due to its implementation, which performs several thread-local storage accesses in rcu_enter and rcu_exit. Finally, D-PRCU obtains 0.5× of Tree RCU's throughput, because its reads update the shared counter table. In contrast, on the larger (1 million nodes) tree, D-PRCU's obtains only 13% worse throughput than Time RCU and EER-PRCU, which in turn are only 5% worse than Tree RCU and Opt-Tree.

Cache coherency related costs Figure 8 depicts the throughput of each implementation, normalized to its simulated wait-for-readers variant. Throughput decrease thus constitutes the overhead imposed by cache misses due to reader/wait-for-readers interaction in the real implementation. In the read-dominated workloads, this overhead is negligible for all implementations. As the update rate—and with it, wait-for-readers rate—increases, so does the cache-related cost. Time RCU imposes an overhead of 15% (large tree)

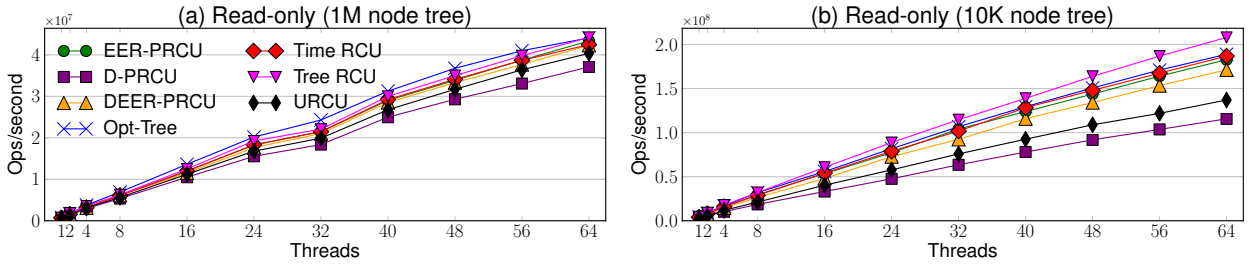

Figure 7: **Tree read-only overhead:** Throughput differences in a read-only workload expose any overhead on reads.

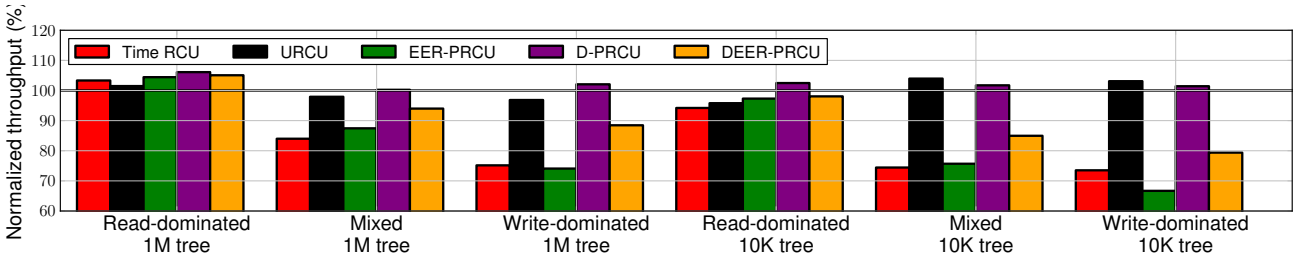

Figure 8: **Cache coherency related read-side overhead:** To isolate overhead due to reader cache misses resulting from communication with wait-for-readers, we show the throughput of each RCU-based version, normalized to when wait-for-readers waits for the same average amount of cycles, but without performing any memory operations.

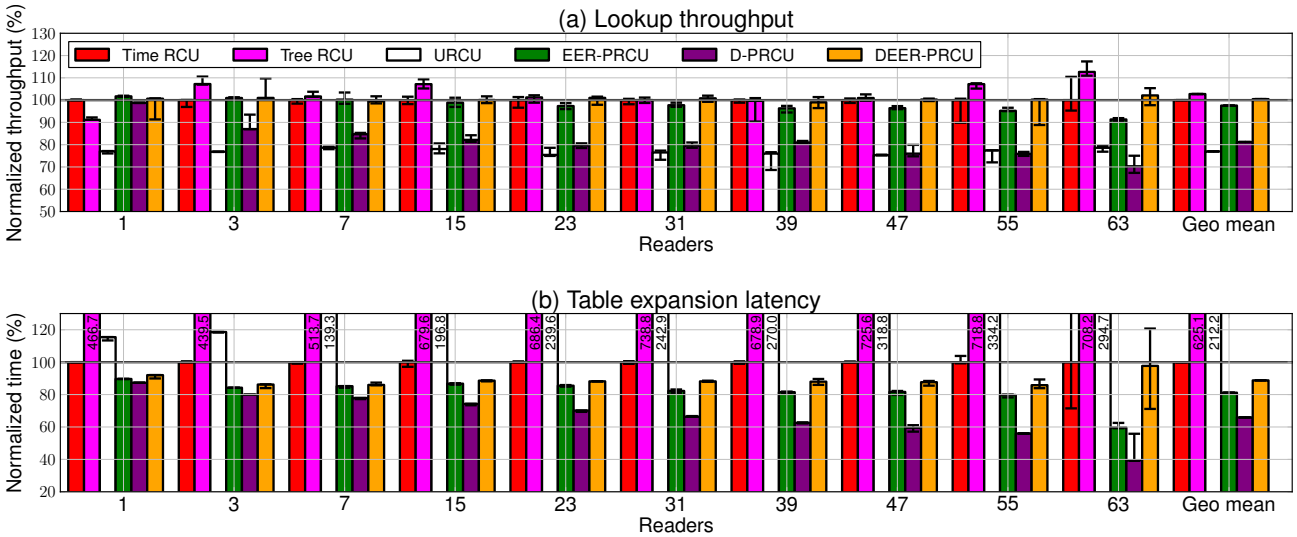

Figure 9: **Resizable hash table performance** (normalized to Time RCU): N readers performing lookups concurrently with an expand.

to 25% (small tree). EER-PRCU has the same overhead, because a wait-for-readers performs the same memory accesses as in Time RCU—the predicate use only affects the decision of whether to wait. URCU obtains lower overhead than Time RCU because its wait-for-readers process is slower, and thus invalidates reader's data less often. In contrast to these implementations, D-PRCU reads pay virtually no cost for interacting with wait-for-readers. This is because D-PRCU pays the cache coherency costs in its read-only mode due to the shared counter updates.

6.2 Resizable Hash Table

Here, we use a benchmark simulating an overloaded hash table (load factor 4), with 10^6 elements, being expanded. The benchmark consists of N reader threads repeatedly performing lookups of uniformly random keys selected from a range of size 2×10^6. Concurrently to this, another thread performs a resizing expand operation. We report median reader throughput and resize latency from 5 experiments.

Figure 9 depicts the results (normalized to Time RCU), which demonstrate the RCU tradeoffs. Similarly to CITRUS, Tree RCU obtains the best read (lookup) throughput, outperforming Time RCU by 10%. However, its resize latency is 7× that of Time RCU. In contrast, resizing with EER-PRCU and DEER-PRCU takes 15%–20% less time than in Time RCU in most concurrency levels, with EER-PRCU resize taking 0.6× that of Time RCU at 63 readers. Despite this, lookup throughput with EER-PRCU and

DEER-PRCU is comparable to with Time RCU, although above 31 readers EER-PRCU lookups underperform Time RCU. Finally, D-PRCU shows the fastest resize time—up to $0.4\times$ that of Time RCU. But it pays for this with an average 20% worse lookup throughput.

7. Related Work

Synchronization mechanisms RCU enables read/write concurrency in which readers are implemented simply—essentially as in a sequential implementation—and with negligible synchronization overhead. Alternative synchronization mechanisms fail to provide these properties: Read/write locks [5] do not provide read/write concurrency. Transactional memory [14] imposes aborts and retries on readers [7]. Hand-crafted applications of locking or nonblocking synchronization [13] can provide read/write concurrency [3, 12, 15], but result in complex implementations that are difficult to prove correct.

RCU implementations We describe several commonly-used RCU implementations in § 2.2. In addition, for code requiring shorter grace periods, the Linux kernel uses SRCU [19]. SRCU restricts waiting by subsystem (e.g, filesystem code need not wait for networking code). In contrast, PRCU can decrease waiting time within a subsystem, by leveraging its semantics. The Mindicator [18] is a nonblocking quiescence detection algorithm. It detects only global quiescence, as with wait-for-readers. Like EER-PRCU, both SRCU and the Mindicator require readers to post their arrival in memory and issue a memory fence.

RCU-based algorithms RCU has been used to implement search trees [2, 4, 16] and hash tables [27]. The CITRUS tree [2] is targeted at supporting concurrent updates with RCU, demonstrating the applicability of RCU as a generic synchronization mechanism.

Predicate use The use of predicates in order to maintain consistency was originally suggested for databases. A *predicate lock* enables a transaction to lock all the tuples that satisfy the predicate [8]. Since predicate locks are used to enforce mutual exclusion, the main difficulty in implementing them is the need to efficiently determine if two predicates agree on a common value, i.e., *conflict*, as conflicting locks cannot be held at the same time. In contrast, we use predicates to manage synchronization between updates and concurrent reads. In PRCU, conflicting predicates imply that two wait-for-readers calls need to wait for common readers and so require no special handling.

8. Conclusion

We have presented PRCU, an RCU variant that allows an update to wait only for reads whose consistency it affects. PRCU thus significantly reduces update overheads, facilitating the use of RCU synchronization in concurrent data structures that require high-throughput and scalable updates.

We have described several PRCU implementations, highlighting trade-offs between read overhead and short wait-for-readers time. Understanding these tradeoffs is interesting future work: can one devise PRCU algorithms with short wait-for-readers time with less read overhead, or is this provably impossible?

Acknowledgements

We thank Hagit Attiya for her encouragement, support, and for the many insightful discussions. We thank the anonymous reviewers for their thoughtful remarks.

This work was supported by the Israel Science Foundation (grants 1227/10 and 1749/14) and by Yad-HaNadiv foundation. Maya Arbel is supported in part by the Technion Hasso Platner Institute (HPI) Research School. Adam Morrison is supported in part at the Technion by an Aly Kaufman Fellowship.

References

[1] Intel 64 and IA-32 Architectures Software Developers Manual, Volume 3: System Programming Guide, June 2013.

[2] M. Arbel and H. Attiya. Concurrent Updates with RCU: Search Tree As an Example. In *PODC*, 2014.

[3] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A Practical Concurrent Binary Search Tree. In *PPoPP*, 2010.

[4] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *ASPLOS*, 2012.

[5] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent Control with "Readers" and "Writers". *CACM*, 14(10), Oct. 1971.

[6] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. User-Level Implementations of Read-Copy Update. *IEEE TPDS*, 23(2), 2012.

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC*, 2006.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *CACM*, 19 (11), Nov. 1976.

[9] K. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, Computer Laboratory, February 2004.

[10] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, 2013.

[11] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole. The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal*, 47(2):221–236, May 2008.

[12] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, and N. Shavit. A lazy concurrent list-based set algorithm. In *OPODIS*, 2006.

[13] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, Jan. 1991.

[14] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[15] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[16] P. W. Howard and J. Walpole. Relativistic red-black trees. *Concurrency and Computation: Practice and Experience*, 2013.

[17] S. V. Howley and J. Jones. A Non-blocking Internal Binary Search Tree. In *SPAA*, 2012.

[18] Y. Liu, V. Luchangco, and M. Spear. Mindicators: A Scalable Approach to Quiescence. In *ICDCS*, 2013.

[19] P. E. McKenney. Sleepable RCU. `http://lwn.net/Articles/202847/`, October 2006. Linux World News.

[20] P. E. McKenney. Hierarchical RCU. `http://lwn.net/Articles/305782/`, November 2008. Linux World News.

[21] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, 1998.

[22] P. E. McKenney, H. J. Boehm, and L. Crowl. C++ Data-Dependency Ordering: Atomics and Memory Model. Technical Report N2664, ISO/IEC JTC1 SC22 WG21, 2008.

[23] T. Nakaike and M. M. Michael. Lock Elision for Read-only Critical Sections in Java. In *PLDI*, 2010.

[24] A. Natarajan and N. Mittal. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP*, 2014.

[25] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: X86-TSO. In *TPHOLs*, 2009.

[26] W. Ruan, Y. Liu, and M. Spear. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM TACO*, 10(4), Dec. 2013.

[27] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX ATC*, 2011.