Concurrent Immediate Reference Counting

JAEHWANG JUNG, KAIST, Korea
JEONGHYEON KIM, KAIST, Korea
MATTHEW J. PARKINSON, Azure Research, Microsoft, United Kingdom
JEEHOON KANG, KAIST, Korea

Memory management for optimistic concurrency in unmanaged programming languages is challenging. Safe memory reclamation (SMR) algorithms help address this, but they are difficult to use correctly. Automatic reference counting provides a simpler interface, but it has been less efficient than SMR algorithms. Recently, there has been a push to apply the optimizations used in garbage collectors for managed languages to elide reference count updates from local references. Notably, Fast Reference Counter, OrcGC, and Concurrent Deferred Reference Counting use SMR algorithms to protect local references by deferring decrements or reclamation. While they show a significant performance improvement, their use of deferral may result in growing memory usage due to slow reclamation of linked structures, and suboptimal performance in update-heavy workloads.

We present *Concurrent Immediate Reference Counting* (CIRC), a new combination of SMR algorithms with reference counting. CIRC employs deferral like other modern methods, but it avoids their problems with novel algorithms for (1) immediately reclaiming linked structures recursively by tracking the reachability of each object, and (2) applying decrements immediately and deferring only the reclamation. Our experiments show that CIRC's memory usage does not grow over time and is only slightly higher than the underlying SMR. Moreover, CIRC further narrows the performance gap between the underlying SMR, positioning it as a promising solution to safe automatic memory management for highly concurrent data structures in unmanaged languages.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Software and its engineering** → *Garbage collection.*

Additional Key Words and Phrases: automatic memory reclamation, reference counting, concurrent data structures

ACM Reference Format:
Jaehwang Jung, Jeonghyeon Kim, Matthew J. Parkinson, and Jeehoon Kang. 2024. Concurrent Immediate Reference Counting. *Proc. ACM Program. Lang.* 8, PLDI, Article 153 (June 2024), 40 pages. https://doi.org/10.1145/3656383

1 INTRODUCTION

The performance and scalability of modern software systems often depend on highly concurrent non-blocking data structures. These data structures can optimistically access memory, which makes manual memory management challenging: knowing when memory will no longer be accessed and is safe to reuse is difficult.

Authors' addresses: Jaehwang Jung, KAIST, Daejeon, Korea, jaehwang.jung@kaist.ac.kr; Jeonghyeon Kim, KAIST, Daejeon, Korea, jeonghyeon.kim@kaist.ac.kr; Matthew J. Parkinson, Azure Research, Microsoft, Cambridge, United Kingdom, mattpark@microsoft.com; Jeehoon Kang, jeehoon.kang@kaist.ac.kr, KAIST, Daejeon, Korea.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/6-ART153
https://doi.org/10.1145/3656383

To address this challenge, algorithms for *safe memory reclamation* (SMR) have been developed for non-blocking data structures. There are many forms of SMR, such as hazard pointers (HP) [Michael 2002b, 2004], pass-the-buck [Herlihy et al. 2005], read-copy-update (RCU) [McKenney and Slingwine 1998], and epoch-based reclamation (EBR) [Fraser 2004]. The interfaces to these systems require a deep understanding of both the client data structure and the SMR algorithm. This makes their correct usage challenging. In fact, Anderson et al. [2021] report several usage bugs in the benchmark suite of several SMR algorithms that lead to use-after-free and memory leaks.

An alternative to SMR is automatic reference counting. With C++20, concurrent reference counting is exposed in the standard library as `atomic<shared_ptr>` and provides a considerably easier programming model. Unfortunately, in the presence of optimistic memory access getting the implementation correct and efficient is tricky. Standard implementations typically involve either locks or split reference counts [Williams 2019]. These implementations negate many of the scaling benefits of the non-blocking data structures.

On the other hand, developments in garbage collectors (GCs) for managed languages have shown that reference counting can be fast. A key optimization in high-performance reference counting GC is to avoid eagerly counting the references from the local variables (*i.e.*, stacks and registers) and let the collector check them during the collection routine. There are two seminal approaches to this optimization. In Deutsch and Bobrow [1976]'s method, objects that reach the zero count (*i.e.*, no references from other heap objects) are first added to the *zero-count table* (ZCT), deferring their reclamation. The GC occasionally pauses all the other threads, scans the local variables to temporarily mark objects referenced by them (*e.g.*, by incrementing), and reclaims all unmarked objects in ZCT. In Bacon et al. [2001]'s method, decrements are deferred by logging them in a thread-local buffer.[1] The GC pauses each thread one by one to fetch their logs and scan their local variables. Then the GC temporarily increments the referents of scanned local variables and executes the decrements logged sufficiently long ago, reclaiming the objects that reach zero count.

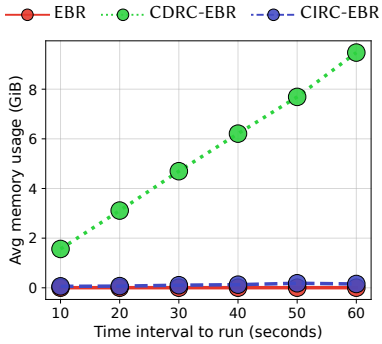
Recently, there has been a push to apply such optimizations to concurrent reference counting for unmanaged languages. Modern algorithms such as Fast Reference Counter (FRC) [Tripp et al. 2018], OrcGC [Correia et al. 2021], and Concurrent Deferred Reference Counting (CDRC) [Anderson et al. 2021, 2022] use SMR to protect uncounted local references, replacing the GC's role of scanning local references. Specifically, OrcGC delays the reclamation of the zero-count object protected by hazard pointer, following Deutsch and Bobrow [1976]; FRC delays decrements and temporarily increments objects protected by hazard pointer, following Bacon et al. [2001]; and CDRC generalizes the deferred decrement approach to other SMRs by delaying decrements to an object until it is no longer protected by the SMR.

1.1 Problems of Deferral-Based Methods

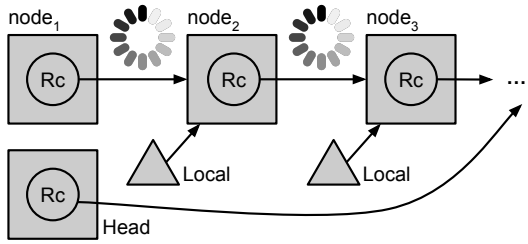
SMR-based deferral has enabled efficient concurrent reference counting in unmanaged languages. However, it is also a weakness.

Slow progress of reclamation. The delay between the release of the last reference and reclamation can lead to the algorithm not being able to keep up with the application's rate of creating garbage. For instance, Fig. 1a shows that the memory usage of a linked-list-based concurrent queue using CDRC grows over time. To see why this occurs, consider dequeuing a series of nodes node_1 to node_n from the queue, where each node_i references node_{i+1} (Fig. 1b). Only after node_i is reclaimed can node_{i+1} be considered for reclamation. But reclaiming node_{i+1} requires checking the protected local references. Since this process is executed in batches by the underlying SMR, the rate of collection may fall behind the rate of dequeuing, resulting in a buildup of garbage backlog. Other

[1] To be precise, increments are deferred too. See §7 for details.



(a) Average memory usage for varying running times.



(b) After the counted reference (Rc) from $node_i$ to $node_{i+1}$ is destroyed, reclaiming $node_{i+1}$ requires checking its Local references, delaying its reclamation.

Fig. 1. Slow reclamation in linked-list-based queue using CDRC.

modern reference counting methods face the same issue as they utilize some form of deferred processing. For example, Correia et al. [2021] observed that the memory footprint of the lock-free skiplist [Fraser 2004; Shavit et al. 2011] using OrcGC can be very large, because the multiple levels of links increase the likelihood of forming garbage chains.

Several solutions exist, but they either compromise the performance or the safe interface. The first straightforward solution is to eagerly attempt to execute the deferred tasks. For example, OrcGC scans the entire hazard pointer slots whenever an object reaches zero count. However, this incurs a big overhead. In fact, Anderson et al. [2021] report that OrcGC is consistently outperformed by CDRC, more than twice slower in some cases. Our evaluation of a linked-list-based queue with an eager variant of CDRC shows a similar increase in the overhead (Fig. 8a). And yet, this solution does not completely fix the memory usage problem, as observed in OrcGC's skiplist benchmark and our queue benchmark (Fig. 8b).

The second solution employed by OrcGC, called *poisoning*, is to manually apply preemptive decrements to the successors of garbage objects to eliminate the dependency between the reclamation of linked objects. For example, after dequeuing $node_i$, it immediately decrements $node_{i+1}$. However, for safety, the links from the poisoned objects must not be followed. This is done by marking the links as poisoned when applying preemptive decrements, and restarting the operations that encounter poisoned links. This not only disallows uninterrupted traversal of linked structures, but also makes its application as difficult as manual SMR schemes. Specifically, programmers must ensure that they poison only the detached objects in order to maintain the correctness of data structure operations.

Finally, we believe the deferred decrement approaches (FRC and CDRC) can enable prompt recursive reclamation by temporarily incrementing the protected local references and immediately applying recursive decrements during the collection routine, similarly to the original algorithm by Bacon et al. [2001].[2] However, this is not compatible with the fastest variants of CDRC, because they are based on RCU/EBR-like SMR schemes that do not announce each local reference. They instead use critical sections that protect all references inside it, which is the key factor to their superior performance. Adding per-pointer protection would negate their performance advantage.

[2]FRC does temporary increments, but does not immediately apply recursive decrements in the collection routine.

Performance overhead of deferred decrement. CDRC may incur significant performance overhead when the objects have many counted references and are updated frequently. Since CDRC schedules a deferred task for each decrement, it creates a large number of deferred tasks in such cases. Frequent scheduling of deferred tasks imposes a non-negligible burden on the underlying SMR, as it increases the frequency of scanning local protections. In addition, it would also increase global synchronization overhead if CDRC were implemented on top of real-world SMR implementations such as Folly's HP and RCU [Meta 2023] and Crossbeam's EBR [Crossbeam Developers 2023], because they use shared data structures to distribute the reclamation workload and to be *transparent* [Nikolaev and Ravindran 2021], *e.g.*, supporting dynamic (un)registration of threads.

1.2 Our Solution for Fast and Safe Recursive Reclamation

We present *Concurrent Immediate Reference Counting* (CIRC), a new combination of an SMR scheme with reference counting. CIRC employs deferral like other modern methods, but it avoids their problems without incurring significant overhead or resorting to an unsafe interface. In throughput, CIRC generally outperforms CDRC and incurs almost none to modest overhead over the underlying SMR. The key idea of CIRC is to track when each object was last reachable so that it can immediately and recursively reclaim the objects that have been unreachable for a sufficiently long time. CIRC realizes this idea in a safe interface and an efficient implementation, underpinned by two novel algorithms.

First, CIRC automatically tracks reachability without inputs from the programmer. To do so, CIRC divides the time into epochs [Bacon et al. 2001; Fraser 2004], and attaches the few-bit representation of the epoch number to pointer fields and reference counts, which are updated along with pointer writes and immediate decrements, respectively. The updates are done in a way that when an object reaches the zero count, its reference count epoch is the one at which the object was last reachable. Specifically, this reachability information is propagated through recursive decrements: in Fig. 1b, if node_i was dequeued long ago and is being destroyed now, node_{i+1} could not have been accessed through node_i. This knowledge allows immediate recursive destruction of the node_{i+1}.

Second, CIRC follows the immediate decrement style (*i.e.*, deferred reclamation) and handles concurrency in a simple yet efficient way. Zero-count objects must be checked for local references, but concurrency complicates the problem. If a zero-count object is incremented away from zero and decremented back to zero while a collector is running concurrently, its destruction must be canceled because there can be new local references missed by the collector. To detect such cases, OrcGC uses an additional sequence number to track how many updates have been applied to the reference count. Our approach is surprisingly simple: if an increment moves the count away from zero, then it must apply a second increment. Thus, when the delayed destruction is processed, if the reference count is still zero, then it must be safe to destruct the object. Otherwise, the destruction is canceled, and the additional reference count is removed.

The combination of two key ideas in CIRC allows memory to be reclaimed considerably more quickly than CDRC. Fig. 1a shows that CIRC's memory usage is only slightly higher than the underlying SMR, and more importantly is not growing with time. At the same time, CIRC performs comparably to EBR in read-most workloads, and in the worst case of heavily contended workloads with large amounts of reference count updates, CIRC performs within 35% of EBR. In addition, the HP version of CIRC without recursive destruction shows up to 55% throughput improvement over the CDRC's counterpart thanks to the reduced number of deferred tasks.

The rest of the paper is structured as follows. §2 provides the background on SMR algorithms and the basic structure of deferral-based concurrent reference counting algorithms. §3 shows how to immediately apply decrements, and §4 presents the algorithm to allow immediately applying recursive destruction. §5 extends CIRC with support for weak references to handle reference

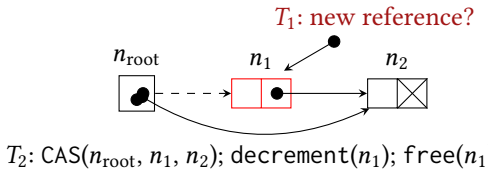


Fig. 2. The race between reference creation and reclamation. Thread T_1 is attempting to get a reference to n_1. At the same time, thread T_2 detaches n_1, decrements it, and reclaims it.

cycles. §6 presents an experimental evaluation of CIRC against the underlying SMRs and CDRC. §7 discusses related work in detail and concludes with future work.

2 BACKGROUND

A naive implementation of reference counting does not work in non-blocking concurrent programs because of the race between reference creation and reclamation. Consider the scenario illustrated in Fig. 2, where two threads access a shared non-blocking concurrent linked list. The first thread T_1 reads the root n_{root} to obtain a reference to the first node n_1. But just before T_1 increments the count of n_1, another thread T_2 unlinks n_1 from n_{root}, decrements the count, reaching 0, and thus destructs and deallocates it. Then it is not safe for T_1 to increment n_1 as it will lead to use-after-free. Note that making the increment atomic, *e.g.*, using the *fetch-and-add* (FAA) instruction, is not enough for preventing this issue, because it stems from the fact that obtaining a pointer value and incrementing the pointed object's count are not atomic.

In this section, we review how the manual concurrent reclamation algorithms handle such problems (§2.1) and the common aspects of the modern concurrent reference counting algorithms for unmanaged languages that utilize the manual reclamation methods under the hood (§2.2).

2.1 Manual Concurrent Reclamation Schemes

Manual memory reclamation methods for non-blocking concurrency, often called *safe memory reclamation* (SMR) schemes, require some manual effort to defer the reclamation until it is safe. In particular, they provide an interface consisting of a function to *protect* a local reference from the reclamation of its referent, and a function to *retire* a pointer, *i.e.*, schedule its reclamation, so that it can be reclaimed later when it is no longer protected. There are two classic approaches for implementing this interface: pointer-based methods represented by hazard pointers (HP) [Michael 2004] and critical-section-based approaches represented by read-copy-update (RCU) [McKenney and Slingwine 1998].

Hazard pointers. In HP, each thread owns *hazard pointer slots* in which they announce (store) a pointer to protect. For example, in Fig. 2, T_1 should write n_1 to its hazard slot before accessing it. On the other hand, T_2 calls the retire() function with n_1 after unlinking it from n_{root}. The retire() function occasionally triggers the reclamation procedure, which takes a pointer from the retired pointer list, checks if it is protected by any of the hazard pointer slots, and if not, reclaims it. Reclamation is usually done in batch (or incrementally) to amortize (or distribute) the cost of scanning the protection slots.

One important subtlety in HP is that writing to a hazard slot itself does not guarantee the safety of access. For example, in Fig. 2, T_2 may retire and reclaim n_1 just before T_1 protects n_1, leading to the same race problem discussed above. To ensure safety, the protector should *validate* that the pointer has not yet been retired. Since a pointer is retired only after it is detached (*i.e.*, made unreachable)

from the data structure's entrypoints, validation can be done by checking the reachability of the object. For example, T_1 should check that n_{root} still points to n_1 after writing to a hazard slot.

Read-copy-update. RCU provides protection based on *critical sections*. A critical section protects all references that can be obtained inside it. More precisely, if a pointer had not been retired before the beginning of a critical section, then it is protected in the said critical section. For example, in Fig. 2, after T_1 starts a critical section, it can freely traverse the list nodes even if some of them are detached and retired while T_1 is traversing.

RCU can be implemented with *epoch-based* technique [Fraser 2004], which maintains a monotonically increasing epoch counter that represents time. When a thread enters a critical section, it announces the current epoch. When an object is retired, the current epoch is recorded in the object. A retired object can be reclaimed if its retirement epoch is smaller than the minimum epoch active critical sections. Such algorithms are called epoch-based reclamation (EBR), and are usually the fastest among the SMR schemes.

Difficulties in using manual schemes. While the manual schemes are more performant than other methods such as traditional implementations of concurrent reference counting, they are known to be difficult to use even for experienced programmers. Correctly applying them requires a deep understanding of the client data structure, and sometimes non-trivial changes should be made. For instance, manual methods require retiring all and only the objects that have been globally detached from the data structure. This is difficult in data structures such as lock-free skiplists [Fraser 2004; Shavit et al. 2011] where a logically deleted node can be physically inserted back. To decide the safety of retirement, the C++ and Rust implementations of lock-free skiplists we are aware of incorporate manual reference counting alongside an SMR method. For another example, HP requires validation and handling its failure, which is not compatible with many concurrent data structures [Brown 2015]. Anderson et al. [2021] report usage bugs in the benchmark suite of several manual schemes that lead to use-after-free and memory leaks.

2.2 Basics of Deferral-Based Concurrent Reference Counting

FRC [Tripp et al. 2018], OrcGC [Correia et al. 2021], CDRC [Anderson et al. 2021, 2022], and our new algorithm CIRC use SMR schemes to implement efficient concurrent reference counting with a safe interface. They share the same key idea: using the SMR to protect objects from being reclaimed while incrementing them (Fig. 2), and exposing the SMR's local reference protection to allow short-lived accesses without updating reference counts. Algorithm 1 shows the interface and implementations that are common to them in a pseudocode with a Rust-style ownership type system. Our presentation largely follows the generalized version of CDRC [Anderson et al. 2022].

Preliminaries. Object<T> (line 1) represents objects of type T managed by reference counting. It extends T with a count for strong (normal) references and another for weak references. (Weak references are discussed in §5.) Rc<T> (line 4) is a smart pointer type for reference-counted pointer to an object of type T, and Atomic<Rc<T>> (line 6) represents a mutable field that contains an Rc<T>.

A Snapshot<T> (line 8) is a local reference protected by the backend SMR scheme.[3] It consists of the pointer value and a Guard that represent the per-pointer protection from the backend SMR, provided by the generalized interface called *acquire-defer* (AD, lines 11 to 16).[4] For example, a guard in HP is a pointer to the hazard pointer slot, and in RCU it is the zero-sized unit type. To account for

[3]The counterpart of Snapshot references is called PrivatePointer in FRC and orc_ptr in OrcGC.

[4]This interface is originally called acquire-retire in Anderson et al. [2021, 2022], but we renamed it to avoid confusion with the retire function SMR schemes. Also, the interface presented here is slightly simplified and generalized for scheduling arbitrary deferred tasks.

Algorithm 1 Interface and implementation common to deferral-based concurrent reference counting libraries. The parts that require algorithm-specific implementation are highlighted purple.

```
1:  struct Object<T>
2:      data: T
3:      strong, weak: Atomic<uint>
4:  struct Rc<T>, Weak<T>
5:      ptr: Object<T>*
6:  struct Atomic<Rc<T>>, Atomic<Weak<T>>
7:      inner: Atomic<Object<T>*>
8:  struct Snapshot<T>
9:      ptr: Object<T>*
10:     guard: Guard

11: function AD::begin_CS()
12: function AD::end_CS()
13: function AD::acquire<T>(src: &Atomic<Rc<T>>)
        → (Object<T>*, Guard)
14: function AD::acquire_raw<T>(ptr: Object<T>*)
        → Guard
15: function AD::release(guard: Guard)
16: function AD::defer<T>(ptr: T*, fn: T* → ())

17: global strongAD, weakAD: AD

21: function Atomic<Rc<T>>::get_snapshot(&self) → Snapshot<T>
22:     (ptr, guard) ← strongAD.acquire(&self.inner)
23:     return Snapshot { ptr, guard }
24: function Rc<T>::from_snapshot(s: &Snapshot<T>) → Rc<T>
25:     if s.ptr ≠ null then increment s.ptr
26:     return Rc { ptr: s.ptr }
27: function Atomic<Rc<T>>::load(&self) → Rc<T>
28:     (ptr, guard) ← strongAD.acquire(&self.inner)
29:     if ptr ≠ null then increment ptr
30:     strongAD.release(guard)
31:     return Rc { ptr }
32: function Atomic<Rc<T>>::cas(&self, expected: Object<T>*, desired: Rc<T>)
        → Result<Rc<T>, (Object<T>*, Rc<T>)>
33:     match self.inner.cas(expected, desired.ptr)
34:         case OK(_) then forget(desired); return Ok(Rc { ptr: expected })
35:         case Err(cur) then return Err((cur, desired))
36: function Atomic<Rc<T>>::store(&self, desired: Rc<T>)
37:     old ← self.inner.swap(desired.ptr)
38:     if old ≠ null then decrement old
39: function Atomic<Rc<T>>::drop(&mut self)
40:     ptr ← self.inner.load()
41:     if ptr ≠ null then decrement old
```

RCU-style protection, acquire-defer provides the begin_CS and end_CS functions to manage critical sections. Critical sections must be active throughout a client operation, and especially Snapshots must not escape from the critical section they are created in. For HP, critical section functions are no-op. We collectively call the per-pointer protection and the critical section *snapshot protections*.

The acquire function creates a guard and protects the pointer loaded from src. For HP, it obtains a hazard slot, announces the loaded pointer, and validates the protection by checking that src has not changed (if changed, repeat). This validation is always safe in the context of reference counting (unlike in HP), because a pointer loaded from Atomic<Rc> is backed by a reference count, which ensures that the object is live. This also means that a Snapshot can be acquired directly from an Rc without validation, using the acquire_raw function. For RCU, acquire does nothing other than loading the pointer. A guard is destroyed by the release function.

The defer function (renamed from "retire" to avoid confusion) schedules a task associated with ptr. A deferred task is executed after all the snapshot protections acquired before its scheduling are

Algorithm 2 Immediate decrement in CIRC without weak references

```
51: function Object<T>::decrement_strong(&self)
52:     if self.strong.faa(-1) = 1 then // No more counted references. Now check the snapshots.
53:         strongAD.defer(self, try_reclaim)
54:     Periodically trigger execution of deferred tasks
55: // Caller must have acquired a snapshot protection to the object.
56: function Object<T>::increment_strong(&self)
57:     if self.strong.faa(1) = 0 then  // If true, it created a permission to schedule try_reclaim again.
58:         self.strong.faa(1)  // The actual increment.
59: function try_reclaim<T>(ptr: Object<T>*)
60:     if (*ptr).strong.load() = 0 then // No new Rc is created, and thus there is no new snapshot.
61:         destruct(&(*ptr).data); free(ptr)
62:     else
63:         (*ptr).decrement_strong()
```

released. CDRC uses this function to schedule decrements, while OrcGC and CIRC schedule the reclamation of zero-count objects.

High-level implementation. Atomic<Rc>::get_snapshot (line 21) is a thin wrapper around acquire. Given a snapshot, Rc::from_snapshot (line 24) increments the count, the implementation of which differs across the different reference counting algorithms. For example, CDRC simply increments the count with FAA, while it is more involved in CIRC and OrcGC. Atomic<Rc>::load (line 27) is a combination of get_snapshot and from_snapshot, but releases the guard at the end.

Atomic<Rc>::cas (line 32) implements the atomic compare-and-swap (CAS) operation with the CAS for the underlying raw pointer. The expected raw pointer value is taken from an Rc or a Snapshot. If the CAS is successful, the desired Rc is forget-ed to prevent running its destructor, so that the ownership of its count is transferred to the Atomic<Rc>; and the old pointer value is returned as an Rc reference, receiving the ownership of the count. If the CAS fails, the current raw pointer value and the input Rc are returned.

The store function (line 36) atomically swaps the new pointer value with the old value, and applies the implementation-specific decrement method to it. For example, CDRC schedules a deferred decrement, while OrcGC and CIRC immediately apply a decrement. Similarly, the destructor of Atomic<Rc> (line 39) loads the pointer value and requests its decrement.

3 IMMEDIATE DECREMENT

CIRC utilizes deferral like other modern methods, but it attempts to apply operations immediately to resolve the problems of the other methods (§1.1). Specifically, CIRC **(1)** follows the immediate decrement style that (in most cases) schedules a single task after the count reaches zero; and **(2)** identifies a chain of garbage objects that can be immediately reclaimed. This section focuses on the first aspect, immediate decrement. Its core design challenge is coordinating snapshots and the zero count: even if the count has become zero, it should be possible to create an Rc reference out of a Snapshot reference. Algorithm 2 presents the immediate decrement algorithm for CIRC without weak references.[5]

Immediate decrement. The decrement_strong function (line 51) is used for functions such as Atomic<Rc>::drop and Atomic<Rc>::store. It first decrements the count with the FAA instruction

[5]Our implementation uses SeqCst memory ordering for accesses to reference counts (loads, FAAs, and CASes). However, we believe many of them can be relaxed. Our hazard pointer implementation uses asymmetric fences [Dice et al. 2001; Goldblatt 2022] to reduce the protection cost.

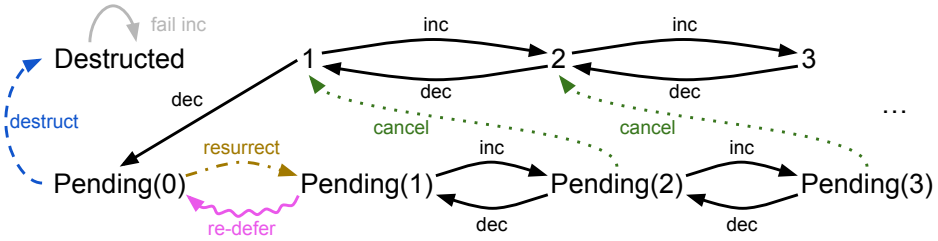


Fig. 3. The state machine of the strong reference count in CIRC. The numbers mean the physical value of the counter. The "inc" and "dec" transitions are normal increments and decrements. The "destruct" transition corresponds to the "if" branch of try_reclaim (Algorithm 2) and try_destruct (Algorithm 3). The "re-defer" and "cancel" transitions correspond to the "else" branch. The "resurrect" transition corresponds to resurrection. "fail inc" happens only in Algorithm 3, when attempting to increment already destructed object.

(which returns the old value). If the count has reached zero, it schedules a deferred execution of the try_reclaim function (line 59), which is invoked after all the existing snapshot protections are withdrawn. To ensure that the deferred try_reclaim is eventually invoked in scenarios such as Fig. 1b, it occasionally triggers the execution of deferred tasks (line 54).

However, it is not safe to directly reclaim the object when the deferred function is invoked, because a new Rc reference could have been created from a (now destroyed) Snapshot reference. So, try_reclaim checks if the count is still zero, ensuring that there is no new Rc. If so, there cannot be a new Snapshot, either. Therefore, it is safe to reclaim the object.

Resurrection. But if the count has increased, the reclamation process should be canceled and retried later. Naively re-scheduling a deferred execution of try_reclaim is not correct, because just before it is invoked again (after checking snapshot protections), a new Snapshot might be derived from an Rc. If all the Rc references are removed after the creation of the Snapshot and before the invocation of try_reclaim, the re-invoked try_reclaim sees zero count, triggering the reclamation. So, using the new Snapshot reference may result in use-after-free.

To fix this, we let the increment_strong function (line 56) increment twice if the count was zero, and try_reclaim call decrement_strong if the count was non-zero. In increment_strong, what actually grants a count for a new Rc is the second increment (line 58). Intuitively, the first increment (from zero to one, line 57) tells the pending try_reclaim that the object has been *resurrected* and thus must be checked again for new snapshot protections. Since increment_strong is called with a snapshot protection, the pending try_reclaim is invoked only after increment_strong completes. Then, try_reclaim will remove this resurrection count, and re-schedule itself if the count has become zero. The try_reclaim function cannot immediately reclaim the object even if it has decremented to zero, because a new Snapshot could have been created from a (now destroyed) Rc.

Fig. 3 summarizes the algorithm with a state transition diagram of the count. The count starts in the state 1, and it moves along the states in the upper row by increments and decrements. When it becomes zero, it enters Pending(0) state, waiting for the invocation of try_reclaim. It moves along the Pending(n) states with $n \geq 0$ by increments from increment_strong and decrements from usual decrement_strong not called by try_reclaim. These transitions cannot move the count to Pending(0) state, because of the extra resurrection increment of increment_strong from Pending(0). Only try_reclaim can move the count from Pending(1) to Pending(0). This guarantees that there is no concurrent execution of try_reclaim. For Pending(n) with $n \geq 2$, try_reclaim returns the count to one of the normal states. If try_reclaim is invoked in Pending(0), the count enters the final Destructed state.

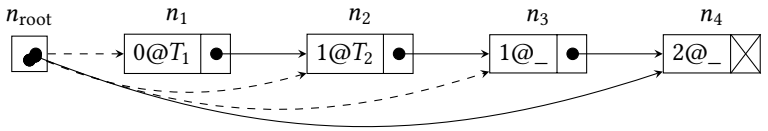


Fig. 4. n_1, n_2, and n_3 are detached from the list one by one, concurrently. n_1 is decremented to zero at time T_1, and n_2 is decremented to one at time T_2.

4 IMMEDIATE RECURSIVE DESTRUCTION

The design up to §3 still suffers from the slow reclamation of long links due to deferred destruction (§1.1). For example, suppose consecutive nodes n_1, n_2, and n_3 are detached from a linked list one at a time as depicted in Fig. 4. After n_1 is destructed and n_2's count is decremented to zero, try_reclaim(n_2) is scheduled for deferred execution. To immediately destruct n_2, we should immediately check if n_2 has snapshot references. However, this cannot be done efficiently. In the HP version, one can scan the hazard slots, but it will degrade the performance. Alternatively, one can temporarily increment the referents of the snapshots and immediately decrement during collection, but it is not possible in the RCU version since it does not track the protection of each individual object.

In this section, we develop the second component of CIRC: immediate recursive destruction based on epoch-based RCU. We first introduce a generic idealized algorithm that tracks the time each object was last reachable (§4.1) and refine it to an efficient algorithm leveraging epochs (§4.2).

4.1 Key Idea

In the above example, we observe that checking the safety of destructing n_2 can be split into two parts based on the time of snapshot creation.

(Snap-Old) Check if there are *old* snapshots for n_2 by reusing the result of the protection scan performed just before the invocation of try_reclaim(n_1). SMR schemes usually cache the scan result to test multiple retired objects in batches. For example, HP first copies the set of protected pointers, and epoch-based RCU computes the minimum epoch of active critical sections. If this check started at time T_s, then the cache tells us whether the snapshots created before T_s are gone.

(Snap-New) Check if n_2 has *new* snapshots that are created after T_s. (Such a snapshot could have been created from n_{root} when it was pointing to n_2.)

If n_2 passes these checks and is destructed, n_3 is decremented to zero and undergoes the same procedure with the same protection cache. This procedure continues until reaching a node with a non-zero count after decrement.

However, Snap-New is difficult to implement efficiently. For example, naively recording the time of each snapshot creation would incur significant overhead, defeating the purpose of snapshot references. Also, relying on user-provided information such as explicit retirement is not an option since such an interface is inherently unsafe, defeating the purpose of automatic reference counting.

Tracking the upper bound of snapshot creation time. To tackle this challenge, we first consider how to track the upper bound of the snapshot creation time, which can be used to implement Snap-New by checking if this upper bound is smaller than T_s. We aim to maintain *object timestamps* for each object such that when the object's count reaches zero, its timestamp is the upper bound of snapshot creation time. Specifically, the object timestamp should adhere to the following invariant:

Fig. 5. Installing an Rc reference to a zero-count node. (a) n_{root} initially points to n_1. Thread 1 gets a Snapshot to n_1, creates a new node n_2 (getting an Rc), and gets a Snapshot to n_2. Thread 2 updates n_{root} to null, decrementing n_1 to zero at time T_1. (b) Thread 1 installs the Rc reference to n_2 on n_1 at time T_i. When n_1 is destructed, we want to know if n_2 can be immediately destructed.

(Invariant) The object timestamp is the upper bound of the time at which a snapshot might have been derived from the object's *destroyed* Rc references.

When destroying an Rc reference, *i.e.*, decrementing a count, Snapshot references can no longer be derived from that Rc reference. Therefore, object timestamps are updated as follows:

(Dec-Base) When an object is non-recursively decremented (*e.g.*, in Fig. 4, decrementing n_1 after directing n_{root} from n_1 to n_2), its object timestamp is updated to the current time.

Applying Dec-Base to recursive decrements (*e.g.*, decrementing n_2 due to the destruction of n_1) precludes immediate recursive destruction, because the recursive decrements are performed after the scan (at T_s). Therefore, it needs special treatment in order to precisely track the snapshot time. The key observation is that if no one had access to the Rc being destroyed since long ago, then no one could have created a snapshot from it. We proceed by case analysis on how a thread that created a snapshot to n_2 was accessing the Rc from n_1 to n_2 that is being recursively destroyed.

- The snapshot is created after the Rc reference is stored in an Atomic<Rc> field of n_1. In this case, creating the snapshot requires a reference to the n_1, either an Rc or a Snapshot. Therefore, the time at which a snapshot to n_2 can be derived this way is bounded by the time by which all the references to n_1 are destroyed. Since the destruction time of Rc references to n_1 are tracked by its object timestamp, we are left with the obligation of tracking the destruction time of the snapshots to n_1.
- The snapshot is created before the Rc is stored in n_1, *e.g.*, when it is directly owned by a thread. Fig. 5 depicts such a scenario: n_1 is decremented to zero at T_1, a thread that holds an Rc reference to n_2 creates a snapshot to n_2, and this thread installs the Rc reference to n_1 at T_i.[6] The upper bound of the time when a snapshot to n_2 can be derived this way is T_i.

Summing up the above analysis, the following rules enforce the object timestamp invariant.

(Link) Each mutable pointer field (Atomic<Rc>) is associated with a *link timestamp*. Whenever a reference is written, the link timestamp is updated together to the current time.

(Dec-Rec) When an object O is decremented due to the destruction of a predecessor P, O's object timestamp is updated to $\max(T_P, T'_P, T_{P\to O}, T_O)$, where T_P is P's object timestamp, T'_P is the time by which all snapshots to P are destroyed, $T_{P\to O}$ is the link timestamp of the link from P to O that is currently being destroyed, and T_O is the current object timestamp of O.

[6]This does not happen in most non-blocking data structures as their correctness relies on the invariant that detached node's pointer fields are not modified. However, automatic reference counting should not rely on such an assumption.

Problems. There are two main problems in implementing those rules. **(1)** In Dec-Rec, how do we know when all the snapshots are destroyed (T'_P)? For efficiency, this should not rely on additional scanning or tracking individual snapshots. **(2)** How do we atomically get the current time, update the count (resp. link), and update the object (resp. link) timestamp? If these operations are done non-atomically, the updated object (resp. link) timestamps are outdated, thwarting the correctness.

4.2 An Efficient Epoch-Based Algorithm

We design an efficient epoch-based algorithm that addresses the problems discussed above.

Epoch-based RCU. Our algorithm builds on a variant of epoch-based RCU (EBR) algorithm by Parkinson et al. [2017]. This scheme maintains an invariant that the difference between epochs of overlapping critical sections is at most 1. In other words, if a thread's critical section is assigned epoch e, then the upper bound on the epoch of other threads' critical section is $e + 1$. This ensures that the end of a critical section with epoch e happens before epoch $e + 2$. In this scheme, an object retired at e can be reclaimed safely at epoch $\geq e+3$, because the maximum epoch of critical sections that have snapshot to the object ("snapshot epoch" in short) is $e + 1$, and the end of such a critical section happens before epoch $(e + 1) + 2$.

Replacing timestamps with epochs. We use the epochs in place of the timestamps from the idealized algorithm outlined in §4.1. Specifically, each object is associated with an *object epoch*, and each mutable pointer field is associated with a *link epoch*. The flavor of EBR we use enables this algorithm because a snapshot at epoch e is guaranteed to be destroyed before epoch $e + 2$, and the staleness of the epoch is bounded by one even if the read of the current epoch and the update of the object and link epochs are done non-atomically.

We start with the object epoch invariant that leverages the EBR's invariant. If an Rc is destroyed at e, then a thread at $e + 2$ cannot access it. This leads to the following invariant.

(Invariant) If the object epoch is e, then the upper bound of the epoch of snapshots derived from the object's destroyed Rc reference is $e + 1$.

This bound allows destructing zero-count objects with epoch e if the current epoch is at least $e + 3$.

The rules for updating object and link epochs should not simply overwrite the epoch to the current epoch because an epoch e can co-exist with $e - 1$ and $e + 1$. For example, if an object O has epoch e_O, the thread that decrements O could be in epoch $e_O - 1$. Updating the object epoch to $e_O - 1$ violates the Invariant because there can be a snapshot to O at epoch $e_O + 1$. Therefore, the rules should ensure that they do not decrease the object epoch and link epoch:

(Dec-Base) When an object with epoch e_O is non-recursively decremented by a thread at epoch e, the object's epoch is updated to $\max(e_O, e)$.
(Link) When a reference is written to a mutable pointer field with link epoch e_l by a thread at epoch e, the link epoch is updated to $\max(e_l, e)$.

Finally, we consider recursively decrementing an object O due to the destruction of a predecessor P. The problematic part of Dec-Rec in §4.1 was T'_P, the maximum destruction time of snapshots to P. Note that it suffices to track the maximum destruction time of Snapshots that *can* be created before T_P. Thanks to the EBR version of Invariant, if P reaches zero count with object epoch e_P, it is guaranteed that the maximum snapshot epoch of P is $e_P + 1$. Therefore, we have the following rule:

(Dec-Rec) When an object O is decremented due to the destruction of a predecessor P, O's object epoch is updated to $\max(e_P, e_{P\to O}, e_O)$, where e_P is P's object epoch, $e_{P\to O}$ is the epoch of the link from P to O that is currently being destroyed, and e_O is the current object epoch of O.

Truncating epochs. The algorithm so far assumes using a dedicated field for each object epoch and link epoch. To reduce the overhead of an extra word, epochs can be truncated into a few bits and packed into the reference count for an object epoch and the most significant bits (MSBs) of the pointer value for a link epoch.

The key ideas are that it is safe to over-approximate the object and link epochs because it only prevents destruction; and that if the current epoch is C, then the upper bound of the epoch value in the whole system is $C + 1$, bounding the over-approximation. Specifically, for b-bit truncated epoch e, *i.e.*, the b least significant bits (LSBs) of the real epoch, if the current untruncated epoch is C, we define the over-approximation function: $\mathrm{ceil}_{b,C}(e) = \max\{E \mid E = e + 2^b n, n \in \mathbb{N}, E \leq C + 1\}$.

Then we define two operators required for our algorithm: the $\mathrm{expired}_{b,C}(e)$ predicate that tests if the real untruncated epoch has no snapshot; and the $\max_{b,C}$ function that over-approximates the maximum of over-approximations of truncated epochs. Formally, they should satisfy the following properties.

$$\mathrm{expired}_{b,C}(e) \implies \mathrm{ceil}_{b,C}(e) \leq C - 3, \quad \max(\mathrm{ceil}_{b,C}(e_1), \mathrm{ceil}_{b,C}(e_2)) \leq \mathrm{ceil}_{b,C}(\max_{b,C}(e_1, e_2))$$

The key idea for implementing these operators is to operate in the *translated* epoch space such that $C + 1$—the upper bound on the epochs in the system—corresponds to the maximum value in the space of unsigned b-bit numbers. That is, $\mathrm{trans}_{b,C}(\mathrm{LSB}_b(C + 1)) = 2^b - 1$. This can be implemented as follows: $\mathrm{trans}_{b,C}(e) := \mathrm{LSB}_b(e - C - 2)$ $(= \mathrm{LSB}_b(e + ((2^b n - 1) - (C + 1))))$, and $\mathrm{untrans}_{b,C}(e) := \mathrm{LSB}_b(e + C + 2)$. Therefore, the operators are implemented as follows:

$$\max_{b,C}(e_1, e_2) := \mathrm{untrans}_{b,C}(\max(\mathrm{trans}_{b,C}(e_1), \mathrm{trans}_{b,C}(e_2)))$$
$$\mathrm{expired}_{b,C}(e) := \mathrm{trans}_{b,C}(e) \leq \mathrm{trans}_{b,C}(C - 3) \quad \text{where } b \geq 3$$

We experimentally found that $b = 4$ provides sufficient progress of the recursive destruction.

5 SUPPORTING WEAK REFERENCES

In this section, we add weak references to CIRC to support data structures that contain reference cycles. Our algorithm largely follows CDRC [Anderson et al. 2022]'s approach,[7] but it is adapted to our immediate decrement algorithm and incorporates Parkinson et al. [2023]'s optimization. Algorithm 3 presents the immediate decrement algorithm extended with weak references.

Background. Normally, automatic reference counting cannot reclaim cyclic structures due to the cyclic dependency of reclamation. Breaking the dependency requires at least one edge in the cycle to be an uncounted reference. To handle this with a safe interface, reference counting schemes usually come with *weak references* (represented by the Weak<T> type) associated with *weak counts*. Weak references make reclaiming an object a two-step process: when an object has no incoming strong references, then it can be destructed, which removes all its outgoing references; and when an object has no incoming weak (and strong) references, its memory block can be deallocated. This allows an object pointed only by weak references in a cycle to initiate destruction, which eventually deallocates it after the destruction of the cycle removes its weak references. At the same time, programmers can safely check whether the referent of a weak reference has not yet been destructed and obtain a dereferenceable strong reference (called *upgrading*).

The standard strategy for implementing the two-stage reclamation for concurrent reference counting is to give an implicit weak count to undestructed objects. That is, the weak count of an object is the number of weak references plus one if the strong count is non-zero. This allows detecting the absence of both strong and weak references atomically.

[7] CDRC's weak reference algorithm has a bug, which is later fixed by Parkinson et al. [2023]. See §7 for details.

Algorithm 3 CIRC with weak references. Notable differences from strong-only CIRC and the corresponding parts in CDRC are highlighted green.

```
71: function Rc<T>::downgrade(&self) → Weak<T>
72:     if self.ptr = null then return Weak { ptr: null }
73:     (*self.ptr).increment_weak(); return Weak { ptr: self.ptr }
74: function Weak<T>::upgrade(&self) → Rc<T>
75:     if self.ptr ≠ null && (*self.ptr).increment_strong() then return Rc { ptr: self.ptr }
76:     return Rc { ptr: null }
77: const DESTRUCTED = 1 << (BITS - 1);
78: function try_destruct<T>(ptr: Object<T>*)
79:     // Since try_destruct is exclusive, CAS fails only when resurrected.
80:     if (*ptr).strong.cas(0, DESTRUCTED).is_ok() then
81:         destruct(&(*ptr).data); (*ptr).decrement_weak()
82:     else
83:         (*ptr).decrement_strong()
84: function Object<T>::increment_strong(&self) → bool
85:     val ← self.strong.faa(1)
86:     if (val & DESTRUCTED) ≠ 0 then return false
87:     if val = 0 then self.strong.faa(1)
88:     return true
89: function Atomic<Weak<T>>::get_strong_snapshot(&self) → Snapshot<T>
90:     loop
91:         (ptr, weak_guard) ← weakAD.acquire(&self.inner)
92:         strong_guard ← strongAD.acquire_raw(ptr)
93:         if ptr ≠ null && ¬(*ptr).is_destructed() then
94:             weakAD.release(weak_guard)
95:             return Snapshot { ptr, guard: strong_guard }
96:         else
97:             strongAD.release(strong_guard); weakAD.release(weak_guard)
98:             if ptr = null || self.inner.load() = ptr then
99:                 return Snapshot { ptr: null, guard: null }
100: function Object<T>::is_destructed(&self) → bool
101:     val ← self.strong.load()
102:     if val = 0 then // Try resurrection.
103:         match self.strong.cas(0, 1)
104:             case OK(_) then return false
105:             case Err(cur) then val ← cur
106:     return (val & DESTRUCTED) ≠ 0
```

Managing weak counts. A Weak reference to an object is constructed from an Rc reference by the Rc::downgrade function (line 71). Similarly to Rc, a Weak can be stored to and loaded from an Atomic<Weak>. Updating the weak count is done in the same manner as the strong count of the strong-only CIRC, but using weakAD, the instance of acquire-defer for protecting the object from deallocation but not destruction (omitted in the algorithm).[8] For example, the increment_weak function (omitted) resurrects the count if the count was zero, and the decrement_weak function

[8]Conceptually, CIRC uses two separate instances of acquire-defer: strongAD for managing destruction and weakAD for deallocation. However, implementations may use a single instance for both tasks.

(omitted) schedules a deferred execution of the try_dealloc function (omitted), which deallocates the memory block if the count was not resurrected.

The try_reclaim function from Algorithm 2 is renamed to try_destruct (line 78) and modified to remove the implicit weak count by invoking decrement_weak when the count has not been resurrected. However, the resurrection check should be modified to take account of the interaction between weak references and strong references. We discuss this change below.

Increment-if-not-destructed. Weak::upgrade (line 74) creates an Rc reference by atomically incrementing the strong count if the referenced object has not been destructed yet. Therefore, increment_strong should be modified to fail (return false) when the object is already destructed. Traditional reference counting method implements this operation (sometimes called increment-if-not-zero or sticky counter) as a simple CAS loop that tries adding one to the count when the count is non-zero. However, this approach is not compatible with Algorithm 2 as it results in spurious failures. This is because the count's physical value becoming zero is not the linearization point of the destruction of the object. Even if the count is zero, there can be snapshots preventing the destruction and the count can even be resurrected. In other words, it is not possible to distinguish the Pending(0) state and Destructed state in Fig. 3 just by looking at the count value. Therefore, we need a new *increment-if-not-destructed* operation.

We adopt the idea of stealing a bit from the count to indicate the count's state used in CDRC [Anderson et al. 2022] and Parkinson et al. [2023]'s wait-free increment-if-not-zero algorithm. The DESTRUCTED bit indicates whether the count is in Destructed state. try_destruct tries setting DESTRUCTED bit by a CAS from 0 (line 80). If successful, the count transitions from the Pending(0) state to Destructed state, allowing it to proceed to destruction. If the CAS failed, then it must be due to resurrection, because try_destruct cannot be invoked concurrently. So, it calls decrement_strong as in the strong-only version.

The modified increment_strong function first increments the count with FAA, and then checks the DESTRUCTED bit (line 86). If set, the operation fails. This corresponds to the self transition of the Destructed state in Fig. 3, which changes the physical value but not the logical state. If DESTRUCTED was not set and the count value was zero (line 87), then the previous increment has resurrected the count, so the count should be incremented again.

Loading Snapshot from Atomic<Weak>. With the interface introduced so far, obtaining a dereferenceable reference from a mutable weak pointer field (Atomic<Weak>) involves at least two increments: one in Atomic<Weak>::load and at least one in Weak::upgrade. Following CDRC's *weak snapshots*, CIRC supports the Atomic<Weak>::get_strong_snapshot function (line 89), which creates a Snapshot if and only if the object is not destructed, without updating the counts in most cases.[9]

The get_strong_snapshot function starts by acquiring a protection in weakAD to prevent deallocation of the referent. Then it uses the acquire_raw function to initiate the acquisition of a protection in strongAD (corresponding to writing to a hazard slot in HP, and no-op in RCU). The protection is validated (line 93) by checking that the object's strong count is not in the Destructed state, using the is_destructed function (line 100).

For this validation to be sound, *i.e.*, the object does not get destructed while the snapshot is active, the is_destructed function must ensure the scheduled try_destruct (if exists) will fail when it returns false. To this end, if the strong count was in the Pending(0) state, it resurrects the count with a CAS from zero to one (line 103). If failed, it re-checks whether the count transitioned to the Destructed state (line 106).

[9]CDRC's weak snapshot is weaker than the normal snapshot. See §7 for details.

The validation failure of get_strong_snapshot does not necessarily mean that the object pointed by the source Atomic<Weak> is destructed, because it may now point to another object. Therefore, instead of unconditionally returning null (which would not be linearizable), it retries from the beginning if the source points to a different object (lines 90 and 98).

Optimization for objects without weak references. Algorithm 3 introduces non-negligible overhead from the deferred execution of try_dealloc when the weak count becomes zero. However, deferred deallocation is not necessary if the object has never had a weak reference, because the object could not have been acquired in weakAD. Following Parkinson et al. [2023], we optimize such cases by taking another bit from the count to indicate whether the object has ever had a weak reference. The application is straightforward: the increment_weak function attempts to set this bit if it is not already set, and try_destruct can immediately deallocate the memory if it is not set. We present the full algorithm in the appendix [Jung et al. 2024].

Interaction with immediate recursive destruction. The immediate recursive destruction algorithm (§4) is modified to work with try_destruct function. Note that the deferred executions of try_delloc do not cause the garbage chain problem, since they do not depend on one another.

6 EXPERIMENTAL EVALUATION

We implemented CIRC as a Rust library and evaluated it on a synthetic benchmark suite[10] to demonstrate that it is resistant to the long garbage chain problem and introduces only a minor overhead to underlying SMR schemes, while keeping the simple interface of reference counting.

The benchmark suite includes the following reclamation schemes: **NR**: the baseline that does not reclaim memory; **EBR**: epoch-based RCU; **HP**: hazard pointers with asymmetric fence optimization [Dice et al. 2001; Goldblatt 2022]; **CIRC-EBR**: the full CIRC with EBR; **CIRC0-HP**: the HP flavor of CIRC without immediate recursive destruction; **CDRC-EBR** and **CDRC-HP**: the EBR and HP flavor of CDRC[11]. CIRC and CDRC share the same code for the underlying reclamation schemes. In the implementation of the reclamation schemes, configuration parameters are tuned to adequately balance the throughput and memory usage.

The benchmark suite consists of the following lock-free map and queue data structures, which we believe represent most use cases of `atomic<shared_ptr<T>>` libraries. **HMList**: Harris-Michael linked list [Michael 2002a], as an example of long sequence of read-only operations; **HashMap**: chaining hash table using HMList [Michael 2002a], as an example of a large number of root locations with short link chains; **NMTree**: Natarajan-Mittal tree [Natarajan and Mittal 2014]; **SkipList**: skiplist [Shavit et al. 2011], as an example of complex data structures with high node indegree; and **DoubleLink**: doubly-linked queue [Ramalhete and Correia 2017b], as an example of a long reclamation dependency chain and the use of weak pointers for back references. CIRC-EBR-based NMTree, SkipList, and DoubleLink is about 500, 400, and 140 lines of code respectively.

The benchmark suite was compiled with Rust nightly-2023-04-21 with default optimization and link-time optimization. We used jemalloc [Evans 2006] for the memory allocator. We conducted experiments on two machines: **AMD64T**: single-socket AMD EPYC 7543 (2.8 GHz, 32 cores, 64 threads) with 8×32 GiB DDR4 DRAMs (256 GiB), and **INTEL96T**: dual-socket Intel Xeon Gold 6248R (3.0 GHz, 48 cores, 96 threads) with 12×32 GiB DDR4 DRAMs (384 GiB). The machines run Ubuntu 22.04 and Linux 5.15 with the default configuration. The results from the two machines exhibit a similar trend, so we mainly discuss AMD64T results here. For full results, see the appendix [Jung et al. 2024].

[10] Available at the project website [Jung et al. 2024].
[11] Our CDRC implementation does not fix the weak reference bug discussed in §7.

Methodology. For map data structures, each thread repeatedly calls get(), insert(), and remove() methods randomly. We measured throughput (operations per second) and the peak memory usage for **(1)** varying number of threads: 1, 8, 16, 24, ⋯, 128 (twice the number of hardware threads); **(2)** three types of workloads: **write-heavy** (50% inserts and 50% removes), **read-write** (50% reads and 50% writes), and **read-most** (90% reads and 10% writes); and **(3)** fixed time: 10 seconds. The key ranges for HMList are 1K and 10K, and the key ranges for the others are 100K and 100M. The data structures are pre-filled to 50%. Figs. 6 and 7 show representative results from this benchmark.

For DoubleLink, each thread repeatedly enqueues an element and then dequeues an element. We measured the throughput (operations per second) of a pair of operations and the peak memory usage for **(1)** varying number of threads: 1, 2, 4, 8, 16, 24, ⋯, 128 (twice the number of hardware threads); and **(2)** fixed time: 10 seconds. For this benchmark, we additionally evaluate a variant of CDRC, **CDRC-EBR-Flush**, which flushes its thread-local deferred tasks after dequeuing an element. Fig. 8 shows the representative results from this benchmark.

Throughput. CIRC adds only a moderate performance overhead to the backend SMR scheme, and outperforms CDRC in write-heavy workloads thanks to the reduced number of deferred tasks, and adds only a moderate performance overhead to the backend SMR scheme. The additional overhead of maintaining object and link epochs for recursive destruction was negligible.

The read-most HMList benchmark (Fig. 7) compares the overhead in traversing long data structures. For the EBR backend, CIRC and CDRC introduce negligible overhead to EBR, showing equal throughput. For the HP backend, CIRC and CDRC can be slower than HP. Since the size of a Snapshot is two words in HP backend (a pointer value and a Guard, *i.e.*, pointer to a hazard pointer), the cost of swapping Snapshots for hand-over-hand acquisition is bigger. CIRC0-HP is slightly slower than CDRC-HP in this benchmark, but it is slightly faster in INTEL96T (see appendix).

In the high-throughput low-indegree data structure benchmarks (HashMap and NMTree, Figs. 6a and 6b) CIRC and CDRC introduce up to 20% and 30% overhead to the underlying SMR scheme, respectively. In SkipList, a high-throughput and high-indegree data structure (Fig. 6c), CIRC-EBR, CIRC0-HP, and CDRC-EBR introduce up to 35% performance overhead to the underlying SMR scheme. CDRC-HP shows a significantly larger overhead, up to 58%. Note that CDRC exhibits a much larger memory footprint (discussed below), suggesting that its performance would substantially degrade if this problem is fixed.

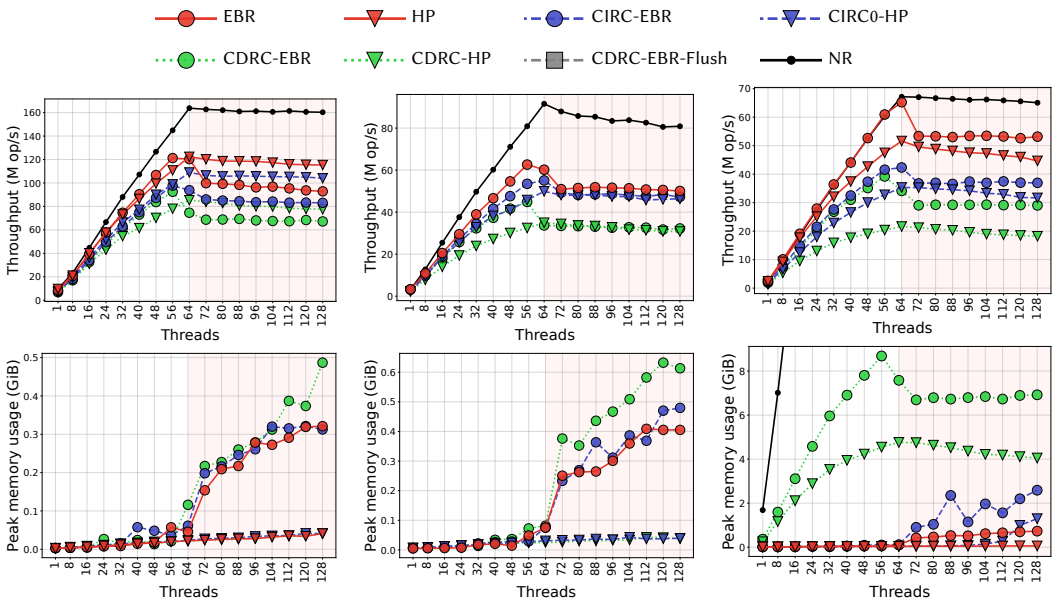
In DoubleLink (Fig. 8a), CIRC-EBR is up to 35% slower than EBR due to the extra deferred task required for weak references. CDRC-EBR shows better throughput than EBR, because its progress in reclamation is very slow (explained below).

In the map benchmarks with big key ranges (see appendix), the throughput gap is smaller due to reduced contention: the gap among EBR, CDRC-EBR, and CIRC-EBR is within 10% and the gap among HP, CDRC-HP, and CIRC-HP is within 30%.

INTEL96T benchmarks show a similar trend (see appendix), but the throughput gap between RC schemes and the underlying SMR is generally larger than that of AMD64T (at most 45% between EBR and CIRC-EBR). We believe that reference count updates incur more overhead in multi-socket machines.

Memory usage. CIRC exhibits a memory footprint similar to its underlying SMR scheme, because it schedules only a single deferred task for reclaiming an object in the common case, and it can recursively destruct a long chain of unreachable objects without scheduling a deferred task.

While CDRC shows a similar trend in HMList, HashMap, and NMTree (Figs. 6a, 6b and 7), SkipList (Fig. 6c) and DoubleLink (Fig. 8b) benchmark demonstrate that CDRC cannot promptly reclaim long linked structure. DoubleLink is a linked list where elements are dequeued from the head and enqueued to the tail, which naturally forms a long chain of detached objects (§1.1). In SkipList,



(a) HashMap (b) NMTree (c) SkipList

Fig. 6. Write-heavy map benchmarks with key range 100K. Throughput (top, higher is better) and peak memory usage (bottom, lower is better) for a varying number of threads.

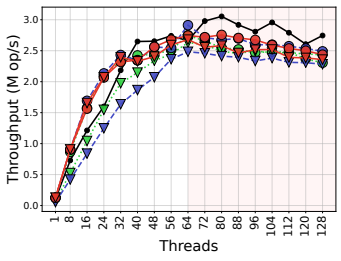
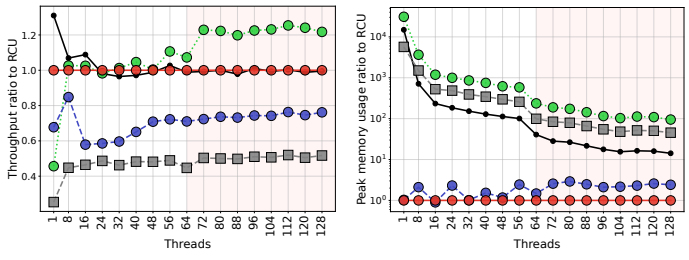


Fig. 7. Throughput (higher is better) of read-most HMList with key range 10K.



(a) Throughput ratio to EBR (higher is better) (b) Peak memory usage ratio to EBR (lower is better)

Fig. 8. DoubleLink benchmark

even if nodes are detached from arbitrary positions in the list, it is likely to form dense unlinked chains due to the multiple levels of links. In such cases, eagerly flushing the batch of deferred tasks (CDRC-EBR-Flush in Fig. 8) at the cost of severe performance degradation only slightly improves the memory footprint, because one flush can only destruct a single node (the first node in unlinked node chain), which is still not guaranteed due to the existence of snapshots.

Under heavy contention, CIRC-EBR's memory footprint may grow up to 4× larger than EBR's. We believe that this is due to the dependency in the reclamation of linked objects. Even if CIRC allows a thread to quickly reclaim long chains, the chain can be reclaimed sequentially by only one thread at a time because of the dependency. On the other hand, manual SMR schemes can reclaim the chain in parallel thanks to the assumption that all retired objects are unreachable.

7 RELATED AND FUTURE WORK

In this section, we compare CIRC and related algorithms in detail, introduce other related work, and conclude with the future work.

Reference counting GCs. Reference counting is a key component in some modern high-performance GCs such as LXR [Zhao et al. 2022]. As discussed in §1, one of the key optimizations is to avoid eagerly counting the local references and defer the job to the GC.

Deutsch and Bobrow [1976]'s method defers reclamation of zero-count objects by putting them into the zero-count table (ZCT) data structure. The GC occasionally initiates a *stop-the-world* pause where all the other threads are stopped, scans all local variables and temporarily marks the objects referenced by them, and reclaims all unmarked objects in ZCT. Pausing all the threads at once is crucial for correctness. For example, if the GC pauses and scans each thread one by one, it may miss the local references newly created by a thread that was already scanned, and the objects decremented to zero after the creation of local reference can be prematurely reclaimed.

On the other hand, Bacon et al. [2001]'s method does not require stop-the-world pause, because it enforces the invariant that the zero-count objects do not have any incoming references by deferring decrements to the next round of the collection cycle. To do so, the time is divided into *epochs*, and the reference count updates (including increments) are first logged in a thread-local buffer with the current epoch number. The GC interrupts each thread one at a time to fetch their logs, scan their local variables, and increment their epoch. After checking all the threads, the GC applies the increments from the current epoch, temporarily increments the scanned local variables, and executes the decrements from the *previous* epoch, reclaiming the objects that reach the zero count. Deferred increment is not crucial for correctness, but it allows the designated GC to update the counts non-atomically, avoiding expensive synchronization.

Another prominent optimization is *coalescing* by Levanoni and Petrank [2001]. Essentially, this approach considers only the difference between the heap states at each epoch boundary. This eliminates the redundant reference count updates for the intermediate referents of frequently modified heap objects. This idea is implemented by logging the first update to each field. The GC collects the logs from each thread one by one without stop-the-world. To resolve the inconsistency in the data due to concurrency, the GC checks each thread multiple times. In addition, the GC *snoops* the new references while it is running by letting each thread record the objects whose reference is written to another heap object. This information is also used for correctly handling the ZCT. CIRC's resurrection mechanism can be considered as a variant of snooping, where only increment-from-zero is recorded. CIRC and other concurrent reference counting methods for unmanaged languages we are aware of do not utilize coalescing.

These optimizations are not easy to apply to unmanaged languages. Since unmanaged languages use raw pointers that are ambiguous with non-pointer values, automatic garbage collectors for them usually resort to conservative methods [Boehm and Weiser 1988; Shahriyar et al. 2014]. Furthermore, scanning the local variables still requires stopping the thread. Recent work such as FRC, OrcGC, and CDRC, and our algorithm CIRC use SMR method's local reference protection to replace the GC's role of scanning the local references.

Deferred decrement in unmanaged languages. FRC [Tripp et al. 2018] builds on the buffered reference counting method by Bacon et al. [2001]: the processing of decrements is deferred to the collector, and the collector scans and temporarily increments the local references. After loading a local reference, it should be explicitly announced, similarly to hazard pointers (HP) [Michael 2004].

CDRC uses deferred decrements too, but the time at which the decrements are applied is governed by an underlying SMR. The initial version of CDRC [Anderson et al. 2021] was based on HP, but it

was generalized to utilize any standard SMR as its backend [Anderson et al. 2022]. Conceptually, the underlying SMR protects a count of the given object. The deferred decrement is implemented with the retire function modified to decrement the object when the count is no longer protected. Since the SMR prevents decrements, the object can be immediately reclaimed when the count hits zero, and the local references do not need to be temporarily incremented during the collection routine. This allows using critical-section-based protection SMR methods such as Read-Copy-Update (RCU) [McKenney and Slingwine 1998] or epoch-based reclamation (EBR) [Fraser 2004] as the backend, which usually are the fastest SMRs.

An advantage of CDRC over CIRC is that it trivially allows conditional store of Snapshot with lazy increment, *i.e.*, passing a Snapshot to Atomic<Rc>::cas as desired and incrementing the count after the successful CAS. This is not allowed in CIRC because its Snapshot does not guarantee a non-zero count, and thus it may lead to a negative count, which is not compatible with the resurrection mechanism. For example, before Atomic<Rc>::cas increments, another thread may overwrite the pointer (*e.g.*, with store) and decrement the count to -1. Therefore, CIRC only takes Rc as the argument of Atomic<Rc>::cas, and as a result, it may have to run a pair of a redundant increment and decrement when it fails. This affects operations that make an object point to another object in a data structure, *e.g.*, removing a node from a linked list. On the other hand, CDRC's Snapshot guarantees a non-zero count because it protects the count itself, allowing lazy increment. However, our evaluation (§6) shows that the advantage of immediate decrements usually outweighs the disadvantage of lacking this feature when implementing concurrent data structures. Such operation failures are less common, and defer is called only "1 + (the number of resurrections)" times in CIRC while on the other hand it is called "1 + (the number of all increments)" times in CDRC.

Deferred reclamation in unmanaged languages. OrcGC [Correia et al. 2021] is closer to CIRC and Deutsch and Bobrow [1976]'s ZCT-based approach in that decrements are immediately applied and the zero-count objects enter a special state that handles their potential reclamation. A variant of HP called pass-the-pointer is used for managing this state: hazard pointers protect local references, and the retire function is modified to scan the hazard pointers and reclaim the unprotected zero-count objects.

As discussed above, algorithms following this style should handle concurrency carefully. For example, suppose a zero-count object is incremented and decremented back to zero while the reclamation procedure for the object is running. There are two problems: (1) the reclamation process should not be invoked again for that object to avoid double-free; and (2) if a new local reference is created during that period, the reclamation should be canceled since the scan may have missed the new reference. For (1), OrcGC uses a bit in the count to indicate that the collector is checking the object. For (2), each reference count is combined with a version number that is increased whenever the count is updated. This allows for detecting when a reference count has remained zero for a period of time. If there was no Snapshot during this period, the object is safe to reclaim.

This combination of methods also tolerates negative counts and thus allows conditional store of Snapshot with lazy increment. We believe CIRC's resurrection mechanism can be modified to tolerate negative counts by using bit flags instead of additional increments.

Unlike the reference count epoch we use in §4, an overflow in OrcGC's version numbers can lead to unsoundness in the algorithm: incorrectly believing the count remained zero. As this must be packed into a single atomically updatable location, there is a trade-off between potential unsoundness and the number of possible incoming edges to an object.

Other automatic concurrent memory management methods. Isolde [Yang and Wrigstad 2017] takes a similar approach to the EBR variant of CDRC. Its memory management method is based on delaying decrements of objects until EBR says that there are no snapshots. In addition, it uses a separate heap for implementing lock-free data structures, and an ownership type system to ensure that the correct annotations can be added to the data structure. We believe Isolde will suffer from the problems of deferral discussed in §1.1.

Parkinson et al. [2017] take a similar approach to FRC, but instead of using reference counts, they use unique owning pointers. Using unique pointers means that once the owning reference goes away it cannot come back, which simplifies the collection. This, however, severely limits the types of data structures that can be represented.

Cohen and Petrank [2015] take a different approach to implementing automatic memory reclamation in a lock-free setting. They effectively build a simple GC for an individual data structure with multiple phases to mark and sweep the live objects, and a custom allocator to track which regions of memory are associated with the data structure. The approach requires that the individual operations on the data structure are restartable, which limits the applicability of the approach to tailored data structures, unlike the more general reference-counting-based approaches.

Reclamation of linked structures. Michael [2020] considers a combination of hazard pointer and reference counting in data structures with *immutable* links (*e.g.*, Michael-Scott queue [Michael and Scott 1996]), in which only the first node needs to be protected by a hazard pointer, allowing efficient traversal. Such a combination normally requires a hazard pointer scan for destructing each link, resulting in either performance degradation or slow reclamation (§1.1). However, Michael observes that if links are immutable, the result of the scan can be reused for descendants without worrying about new protections, which allows immediate recursive destruction. Our immediate recursive destruction algorithm (§4) generalizes this observation and dynamically checks the condition with an epoch-based method.

Manual SMR algorithms. RCU-style SMR algorithms such as EBR are fast thanks to critical-section-based protection, but their reclamation can get stuck if a thread does not deactivate its critical section (*e.g.*, suspended for a long time), leading to unbound memory usage. On the other hand, HP-style SMR algorithms bound the memory usage at the cost of per-object protection. Some recent algorithms such as Hazard Eras (HE) [Ramalhete and Correia 2017a] and Interval-based Reclamation (IBR) [Wen et al. 2018] mix two approaches to provide good throughput and bounded memory usage. In HE and IBR, a bounded number of objects can be allocated or retired in each epoch (incremented when the bound is reached), and each thread protects a bounded set of epochs, which is announced less frequently than per-object protection. The CDRC and our immediate decrement algorithm can use such methods as a backend. However, it is unclear whether our recursive destruction can be applied to them.

Weak references. CIRC's weak reference algorithm consists of two components: protecting an object from being deallocated while loading Atomic<Weak>, and increment-if-not-destructed for upgrading to a strong reference. The first is a direct adaptation of the deferral-based reference counting (§2.2), and the second takes the idea from the wait-free increment-if-not-zero in the weak reference algorithms by Anderson et al. [2022] and Parkinson et al. [2023].

Parkinson et al.'s algorithm is similar to Anderson et al.'s, but it fixes a bug in the latter. Both algorithms use a bit from the count word to represent whether the count is closed (permanently zero). FAA can be used to manipulate the count without affecting the bit, and when the count reaches zero, the closed bit should be set with a CAS. A subtlety in this algorithm is that it is possible for the count to be incremented away from zero before the bit is set, and this was the source of the

bug in Anderson et al.'s algorithm. Multiple threads may decrement to zero, attempt to set the closed bit, and if successful, decrement the implicit weak count. If the thread that successfully sets the closed bit also deallocates the object, the other threads attempting to set the bit will access the freed memory. Parkinson et al.'s solution is to associate the implicit weak count with the strong count's physical value being zero instead of it being closed. When a weak reference holder increments from zero, it should increment the weak count, because the weak reference's weak count is logically converted to the implicit weak count. This protects the other threads from deallocation.

In CIRC, the implicit weak count is still associated with the object's destruction, but it does not suffer from this problem because its resurrection mechanism guarantees try_destruct is exclusive. In fact, CIRC's resurrection count plays a similar role as Parkinson et al.'s implicit weak count.

CIRC's Atomic<Weak>::get_strong_snapshot is based on CDRC's *weak snapshot*. CDRC's weak snapshot has the same goal, but it is weaker than normal Snapshot in that creating an Rc reference out of it may fail. To implement weak snapshot, CDRC uses two acquire-defer instances for destruction: strongAD defers decrements, and disposeAD defers destruction after the count reaches zero. In CDRC, the protection in strongAD cannot be validated by checking that the count is non-zero (line 93), because even if the count value is currently non-zero, there might be a deferred decrement scheduled earlier. Therefore, the count can be closed after obtaining a weak snapshot, after which no new Rc reference can be created. On the other hand, validating the disposeAD protection with a non-zero count does guarantee that the object is not destructed, because the count being non-zero implies that the destruction has not been scheduled yet.

Conclusion and future work. We have designed *Concurrent Immediate Reference Counting* (CIRC), a safe concurrent reference counting method for unmanaged languages that promptly reclaims long linked structures without additional per-pointer announcements. Our evaluation shows that CIRC performs competitively with the fastest manual methods in highly concurrent data structures.

As future work, we would like to explore recursive destruction algorithms that support other SMR techniques. CIRC adds EBR to the list of SMRs supporting recursive destruction (previously only HP, as discussed in §1.1), but EBR-based reference counting has a limitation that it does not bound the memory usage. To achieve high performance and bounded memory usage at the same time, a recursive destruction algorithm for hybrid SMRs such as HE and IBR is needed.

We also plan to formally verify CIRC; apply our techniques to GCs for managed languages; and adopt more optimizations from the GC literature such as coalescing to unmanaged languages.

ACKNOWLEDGMENTS

We thank the PLDI 2024 reviewers for their valuable feedback. Jaehwang Jung, Jeonghyeon Kim and Jeehoon Kang are supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT2201-06.

DATA AVAILABILITY STATEMENT

The implementation of CIRC and the appendix for this paper can be found in [Jung et al. 2024].

REFERENCES

Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2021. Concurrent Deferred Reference Counting with Constant-Time Overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 526–541. https://doi.org/10.1145/3453483.3454060

Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2022. Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design*

and Implementation (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 61–75. https://doi.org/10.1145/3519939.3523730

David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. 2001. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 92–103. https://doi.org/10.1145/378795.378819

Hans-Juergen Boehm and Mark Weiser. 1988. Garbage Collection in an Uncooperative Environment. 18, 9 (1988), 807–820. https://doi.org/10.1002/spe.4380180902

Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (Donostia-San Sebastián, Spain) *(PODC '15)*. Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2767386.2767436

Nachshon Cohen and Erez Petrank. 2015. Automatic Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 260–279. https://doi.org/10.1145/2814270.2814298

Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. OrcGC: Automatic Lock-Free Memory Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 205–218. https://doi.org/10.1145/3437801.3441596

Crossbeam Developers. 2023. Crossbeam. https://github.com/crossbeam-rs/crossbeam

L. Peter Deutsch and Daniel G. Bobrow. 1976. An Efficient, Incremental, Automatic Garbage Collector. *Commun. ACM* 19, 9 (sep 1976), 522–526. https://doi.org/10.1145/360336.360345

Dave Dice, Hui Huang, and Mingyao Yang. 2001. Asymmetric Dekker Synchronization. http://web.archive.org/web/20080220051535/http://blogs.sun.com/dave/resource/Asymmetric-Dekker-Synchronization.txt

Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD.

Keir Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation. University of Cambridge, Computer Laboratory.

David Goldblatt. 2022. P1202R5: Asymmetric Fences. https://wg21.link/p1202r5.

Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking Memory Management Support for Dynamic-Sized Data Structures. *ACM Trans. Comput. Syst.* 23, 2 (may 2005), 146–196. https://doi.org/10.1145/1062247.1062249

Jaehwang Jung, Jeonghyeon Kim, Matthew J. Parkinson, and Jeehoon Kang. 2024. Concurrent Immediate Reference Counting (artifact and appendix). https://doi.org/10.5281/zenodo.10806736 Project webpage: https://cp.kaist.ac.kr/gc.

Yossi Levanoni and Erez Petrank. 2001. An On-the-Fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa Bay, FL, USA) *(OOPSLA '01)*. Association for Computing Machinery, New York, NY, USA, 367–380. https://doi.org/10.1145/504282.504309

P. E. McKenney and J. D. Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *PDCS '98*.

Meta. 2023. Folly: Facebook Open-source Library. https://github.com/facebook/folly

Maged M. Michael. 2002a. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Winnipeg, Manitoba, Canada) *(SPAA '02)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/564870.564881

Maged M. Michael. 2002b. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing* (Monterey, California) *(PODC '02)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/571825.571829

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

Maged M. Michael. 2020. Brief Announcement: Hazard Pointer Protection of Structures with Immutable Links. In *Proceedings of the 39th Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC '20)*. Association for Computing Machinery, New York, NY, USA, 230–232. https://doi.org/10.1145/3382734.3405738

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256

Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 987–1002. https://doi.org/10.1145/3453483.3454090

Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. 2017. Project Snowflake: Non-Blocking Safe Manual Memory Management in .NET. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 95 (oct 2017), 25 pages. https://doi.org/10.1145/3141879

Matthew J. Parkinson, Sylvan Clebsch, and Ben Simner. 2023. Wait-Free Weak Reference Counting. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management* (Orlando, FL, USA) *(ISMM 2023)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/3591195.3595271

Pedro Ramalhete and Andreia Correia. 2017a. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures* (Washington, DC, USA) *(SPAA '17)*. Association for Computing Machinery, New York, NY, USA, 367–369. https://doi.org/10.1145/3087556.3087588

Pedro Ramalhete and Andreia Correia. 2017b. DoubleLink - A Low-Overhead Lock-Free Queue. https://concurrencyfreaks.blogspot.com/2017/01/doublelink-low-overhead-lock-free-queue.html

Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. 2014. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 121–139. https://doi.org/10.1145/2660193.2660198

Nir N Shavit, Yosef Lev, and Maurice P Herlihy. 2011. Concurrent lock-free skiplist with wait-free contains operator. https://patentcenter.uspto.gov/applications/12191008 US Patent 7,937,378.

Charles Tripp, David Hyde, and Benjamin Grossman-Ponemon. 2018. FRC: A High-Performance Concurrent Parallel Deferred Reference Counter for C++. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (Philadelphia, PA, USA) *(ISMM 2018)*. Association for Computing Machinery, New York, NY, USA, 14–28. https://doi.org/10.1145/3210563.3210569

Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-Based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3178487.3178488

Anthony Williams. 2019. *C++ Concurrency in Action,2E* (2 ed.). Manning Publications, New York, NY.

Albert Mingkun Yang and Tobias Wrigstad. 2017. Type-Assisted Automatic Garbage Collection for Lock-Free Data Structures. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management* (Barcelona, Spain) *(ISMM 2017)*. Association for Computing Machinery, New York, NY, USA, 14–24. https://doi.org/10.1145/3092255.3092274

Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-Latency, High-Throughput Garbage Collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 76–91. https://doi.org/10.1145/3519939.3523440

A OPTIMIZATION FOR OBJECTS WITHOUT WEAK REFERENCES

Algorithm 4 CIRC with optimization for objects without weak references.

```
1:  struct Object<T>
2:      data: T
3:      state: Atomic<uint64> // For simplicity, counts are merged into a single state word.
4:  const DESTRUCTED, WEAKED // Bits indicating whether the object is destructed and whether it has ever had a weak reference
5:  const STRONG, WEAK // Bitmask for strong and weak counts
6:  const COUNT, WEAK_COUNT // Bit representation of a single count
7:  // Called from Rc::downgrade (the only case where weak is created first), Atomic<Weak>::load, Weak::clone.
8:  function Object<T>::increment_weak(&self)
9:      val ← self.state.load()
10:     while val & WEAKED = 0 do
11:         match self.state.cas(val, (val | WEAKED) + WEAK_COUNT)
12:             case Ok(_) then return
13:             case Err(cur) then val ← cur
14:     if self.state.faa(WEAK_COUNT) & WEAK = 0 then
15:         self.state.faa(WEAK_COUNT)
16: function Object<T>::decrement_weak(&self)
17:     if self.state.faa(-WEAK_COUNT) & WEAK = WEAK_COUNT then
18:         weakAD.defer(self, try_dealloc)
19: function try_destruct<T>(ptr: Object<T>*)
20:     val ← (*ptr).state.load()
21:     loop
22:         if val & STRONG > 0 then
23:             (*ptr).decrement_strong(); return
24:         match (*ptr).state.cas(val, val | DESTRUCTED)
25:             case Err(cur) then val ← cur
26:             case OK(_) then
27:                 destruct(&(*ptr).data)
28:                 if val & WEAKED = 0 then free(ptr)
29:                 else (*ptr).decrement_weak()
30:                 return
31: function Object<T>::increment_strong(&self) → bool
32:     val ← self.state.faa(COUNT)
33:     if val & DESTRUCTED ≠ 0 then return false
34:     if val & STRONG = 0 then self.state.faa(COUNT)
35:     return true
36: function Object<T>::decrement_strong(&self)
37:     if self.state.faa(-COUNT) & STRONG = COUNT then
38:         strongAD.defer(self, try_destruct)
39: function Object<T>::is_destructed(&self) → bool
40:     val ← self.state.load()
41:     while val & (DESTRUCTED | STRONG) = 0 do
42:         match self.state.cas(val, val + COUNT)
43:             case OK(_) then return true
44:             case Err(cur) then val ← cur
45:     return val & DESTRUCTED = 0
```

B AMD64T FULL EXPERIMENTAL RESULTS

B.1 Write-heavy & Small Key Ranges (1K for Lists and 100K for Others)

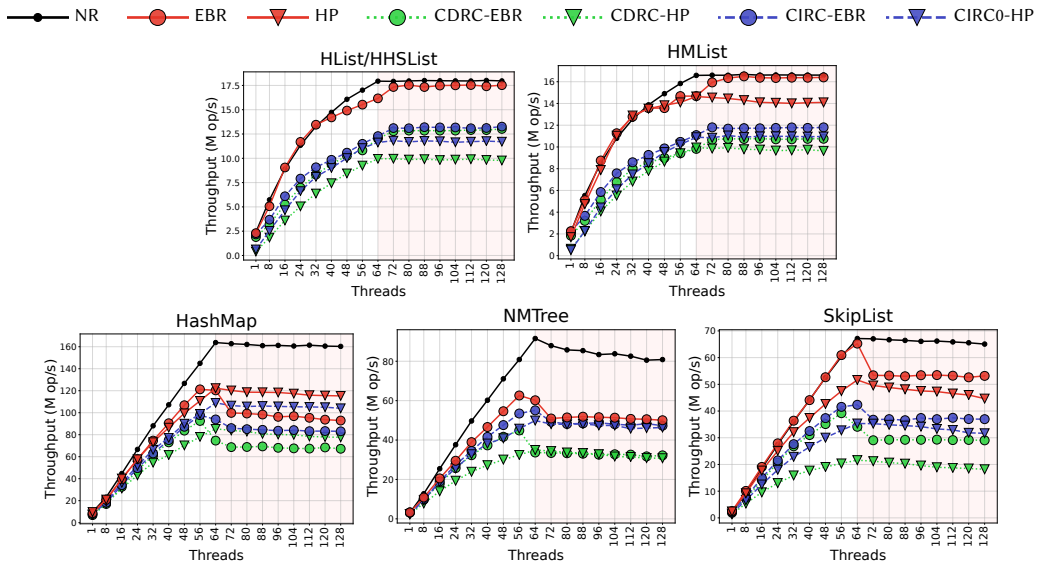


Fig. 9. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with small key ranges.

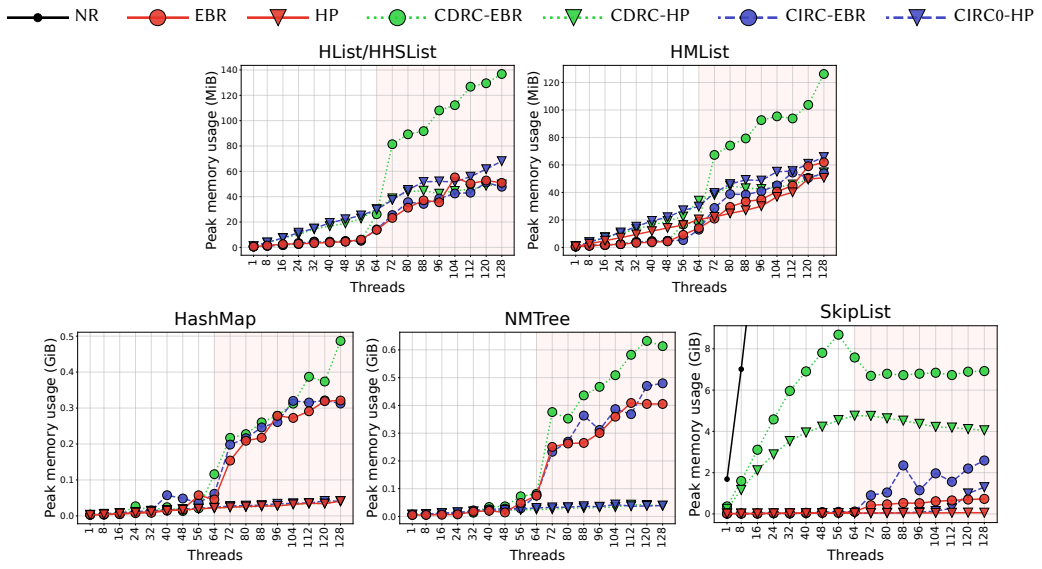


Fig. 10. Peak number of memory usage of write-heavy workloads for a varying number of threads with small key ranges.

B.2 Write-heavy & Large Key Ranges (10K for Lists and 100M for Others)

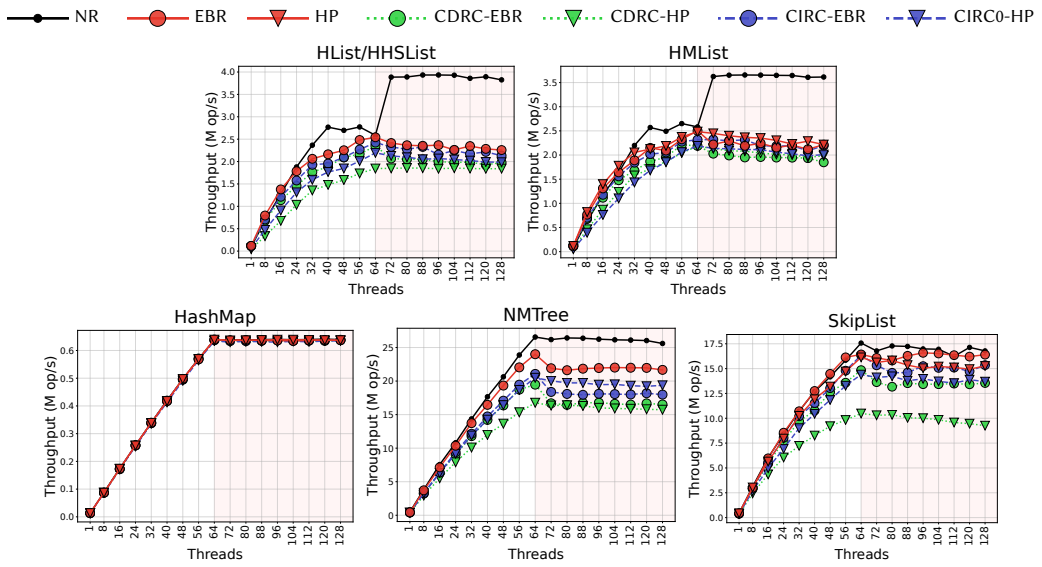


Fig. 11. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with large key ranges.

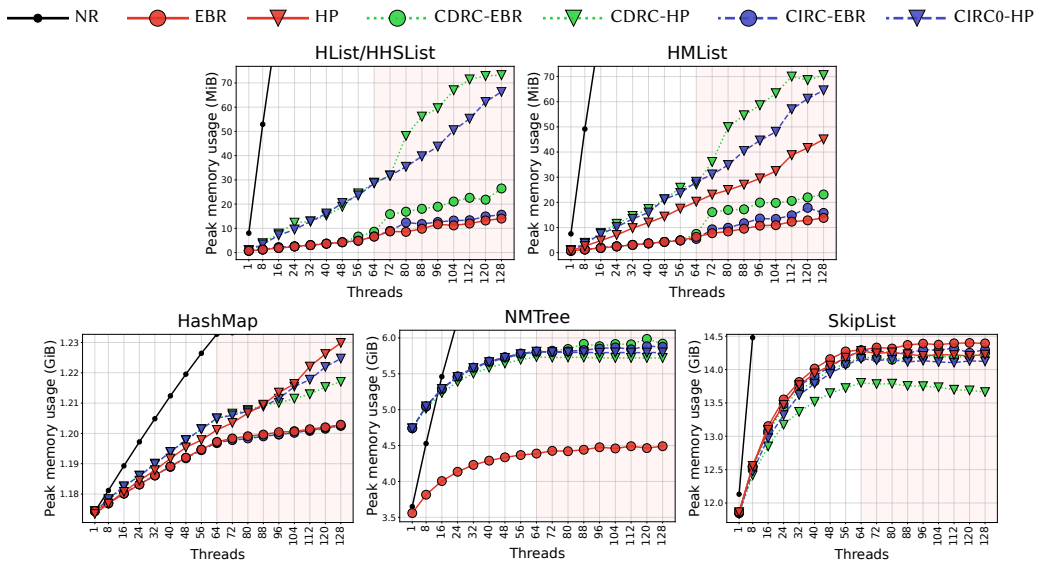


Fig. 12. Peak number of memory usage of write-heavy workloads for a varying number of threads with large key ranges.

B.3 Read-write & Small Key Ranges (1K for Lists and 100K for Others)

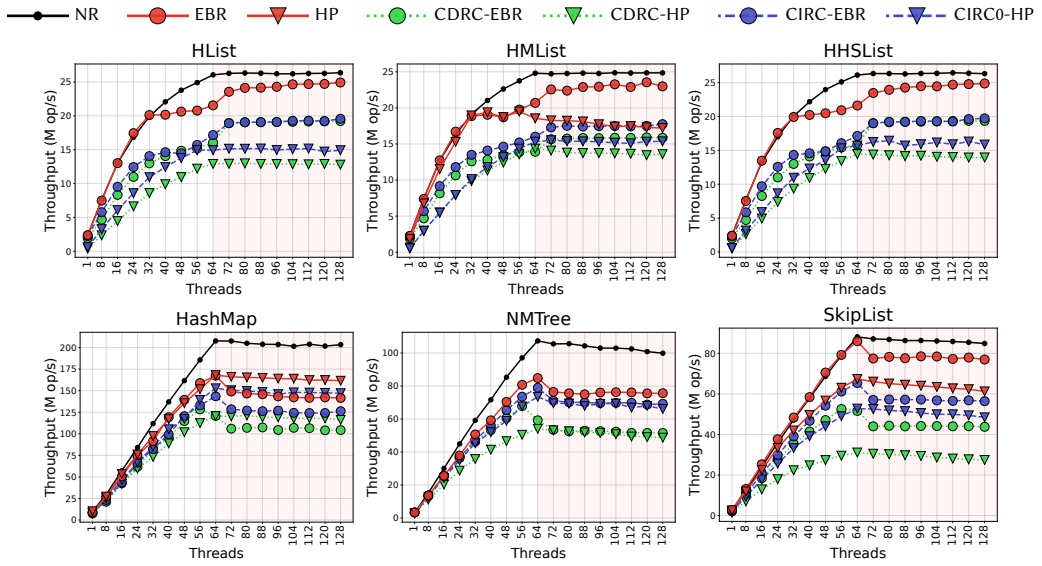


Fig. 13. Throughput (million operations per second) of read-write workloads for a varying number of threads with small key ranges.

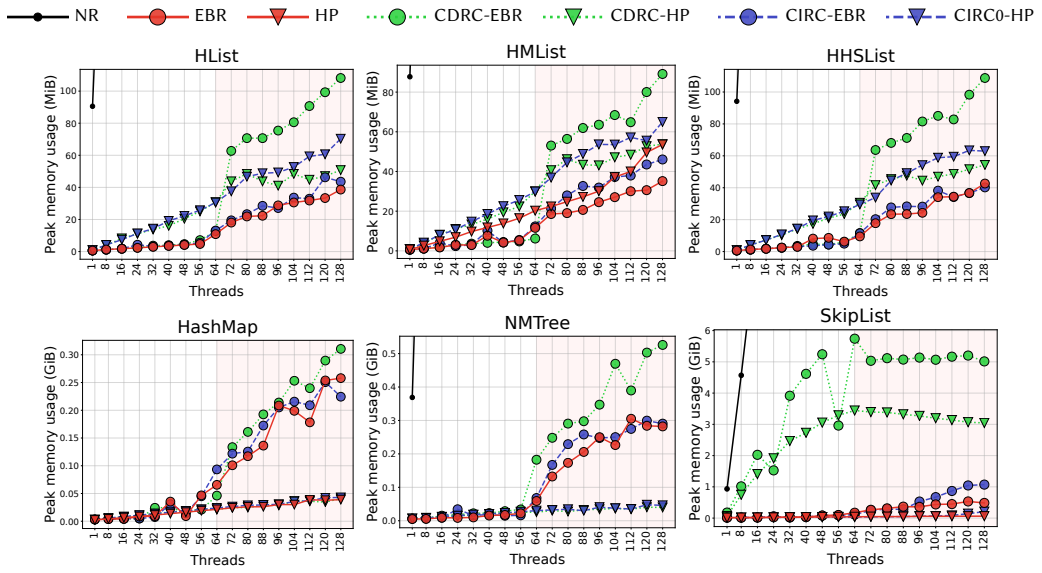


Fig. 14. Peak number of memory usage of read-write workloads for a varying number of threads with small key ranges.

B.4 Read-write & Large Key Ranges (10K for Lists and 100M for Others)

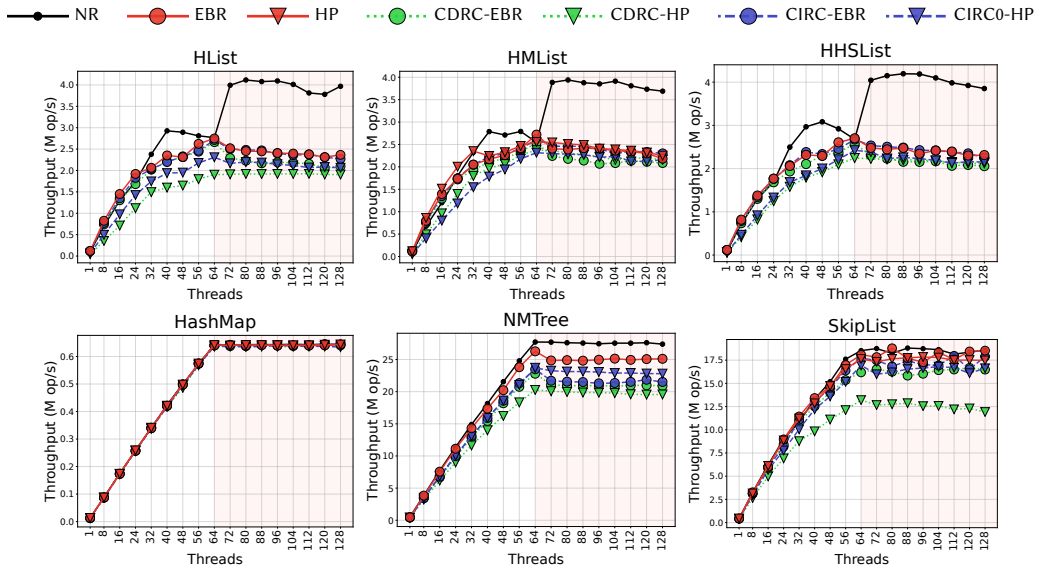


Fig. 15. Throughput (million operations per second) of read-write workloads for a varying number of threads with large key ranges.

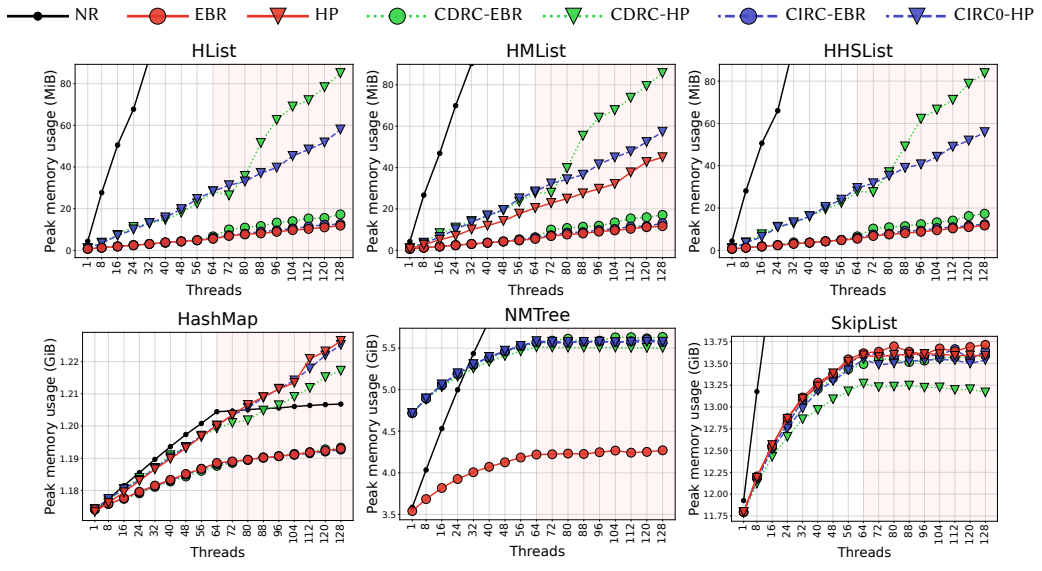


Fig. 16. Peak number of memory usage of read-write workloads for a varying number of threads with large key ranges.

B.5 Read-most & Small Key Ranges (1K for Lists and 100K for Others)

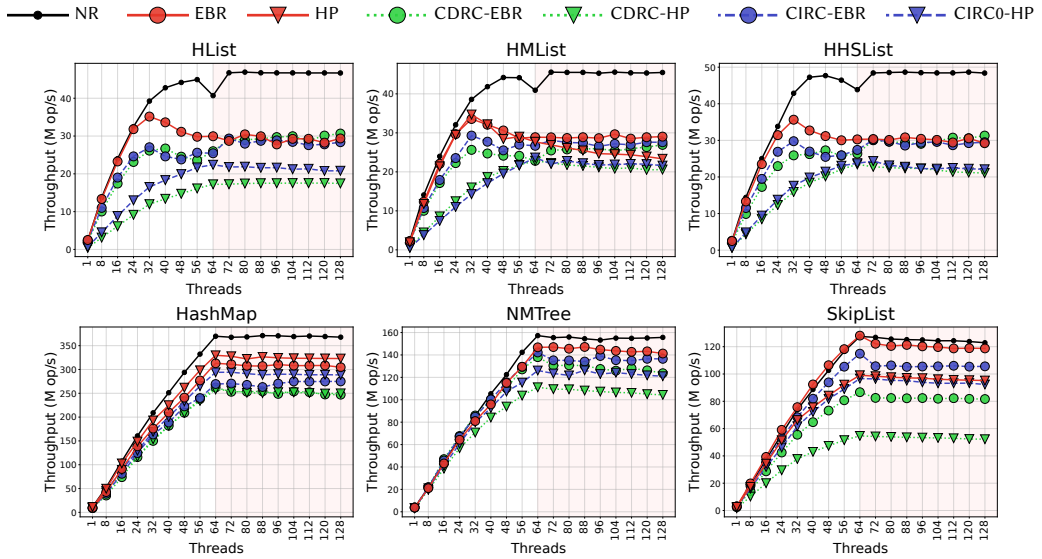


Fig. 17. Throughput (million operations per second) of read-most workloads for a varying number of threads with small key ranges.

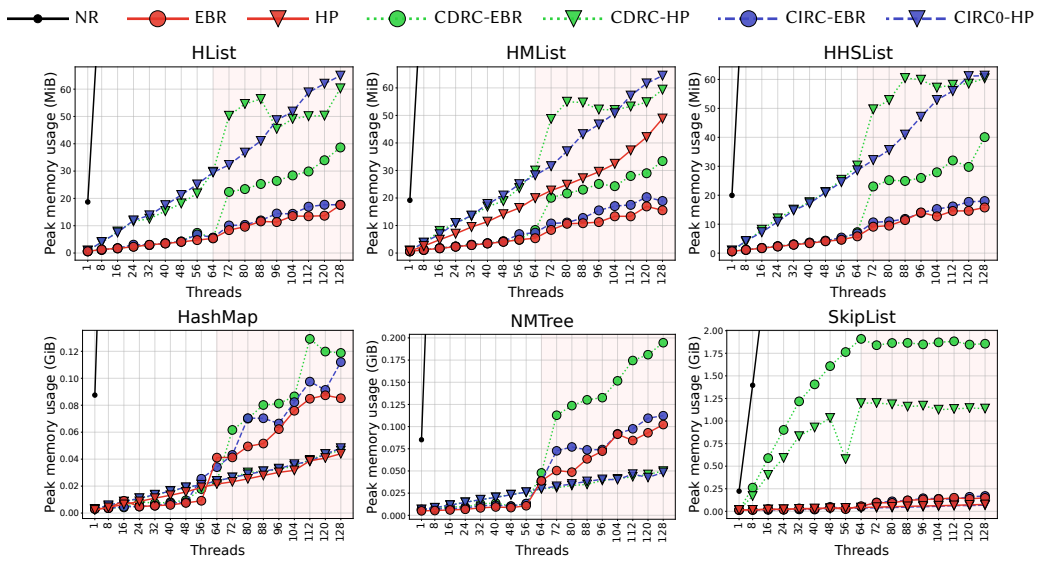


Fig. 18. Peak number of memory usage of read-most workloads for a varying number of threads with small key ranges.

B.6 Read-most & Large Key Ranges (10K for Lists and 100M for Others)

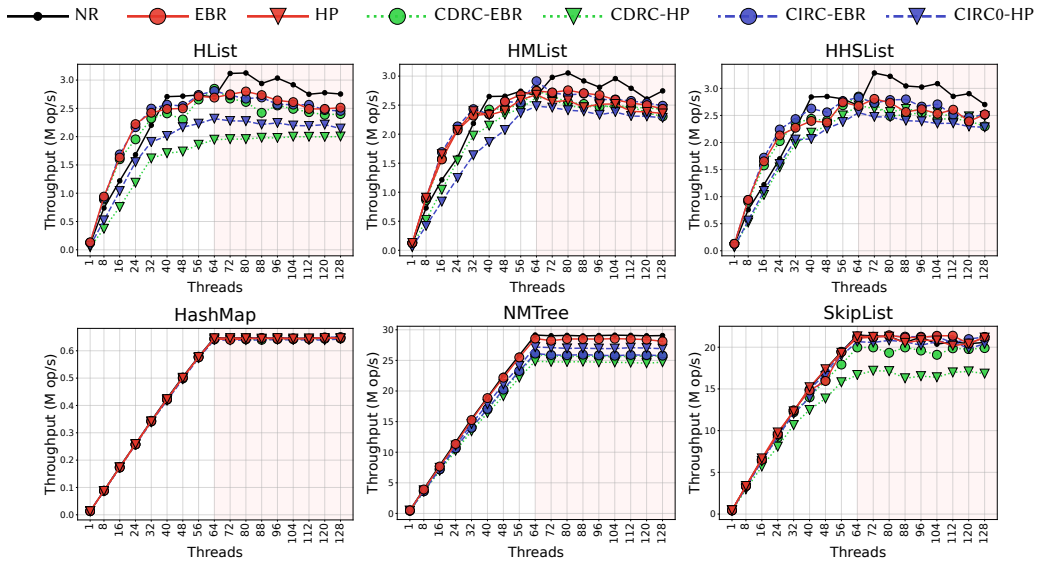


Fig. 19. Throughput (million operations per second) of read-most workloads for a varying number of threads with large key ranges.

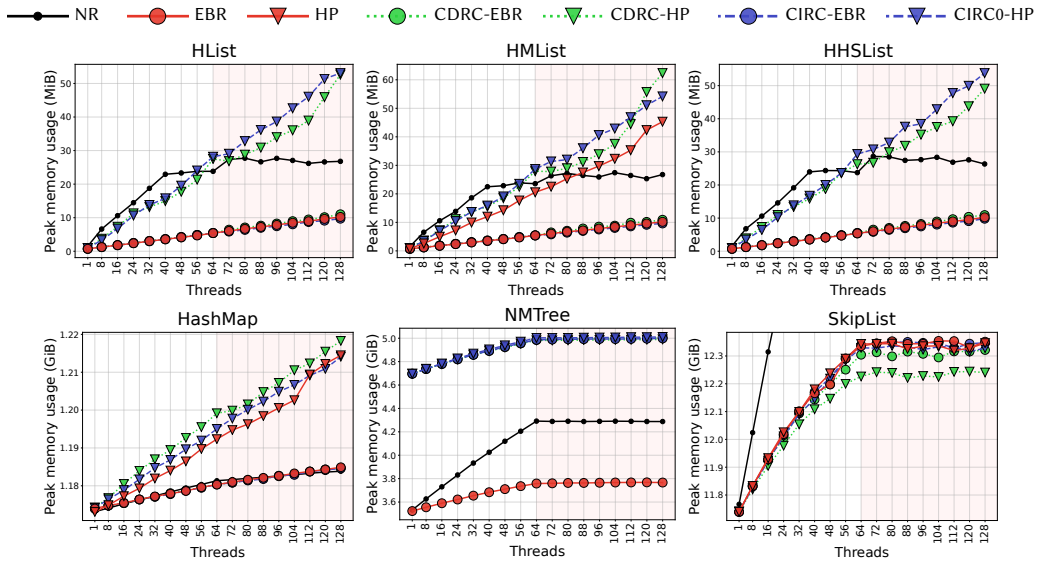


Fig. 20. Peak number of memory usage of read-most workloads for a varying number of threads with large key ranges.

B.7 DoubleLink

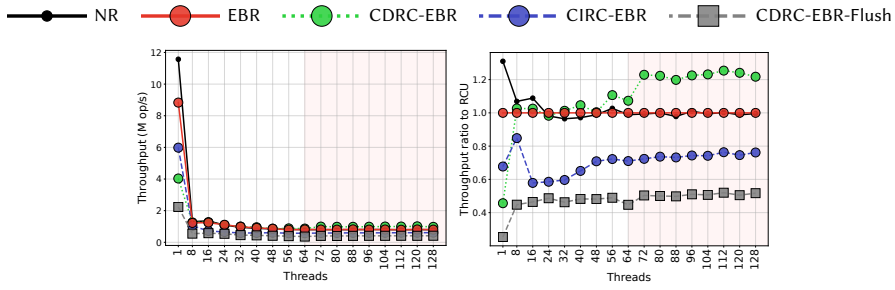


Fig. 21. Throughput (million operations per second) of DoubleLink workloads for a varying number of threads.

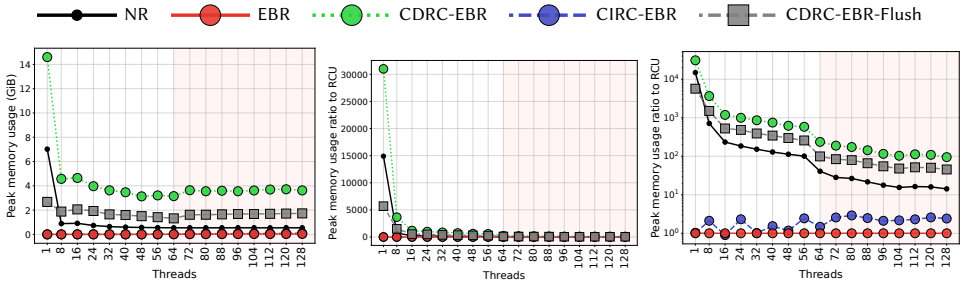


Fig. 22. Peak number of memory usage of DoubleLink workloads for a varying number of threads.

C INTEL96T FULL EXPERIMENTAL RESULTS

C.1 Write-heavy & Small Key Ranges (1K for Lists and 100K for Others)

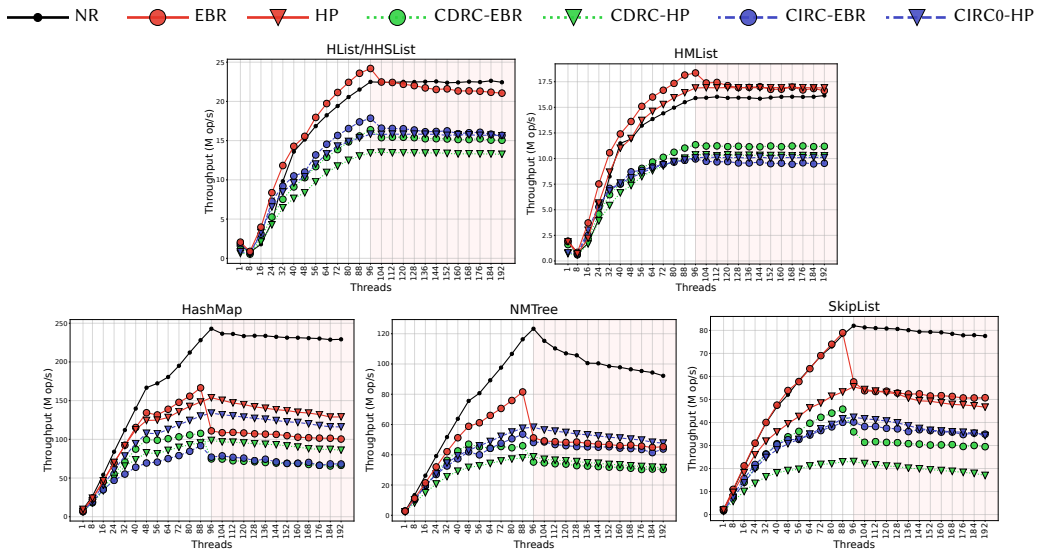


Fig. 23. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with small key ranges.

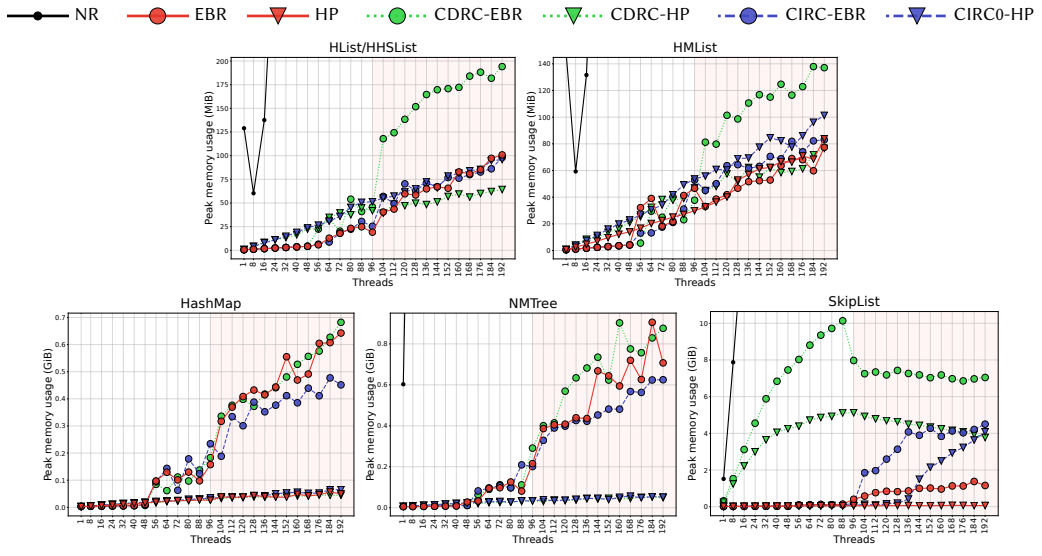


Fig. 24. Peak number of memory usage of write-heavy workloads for a varying number of threads with small key ranges.

C.2 Write-heavy & Large Key Ranges (10K for Lists and 100M for Others)

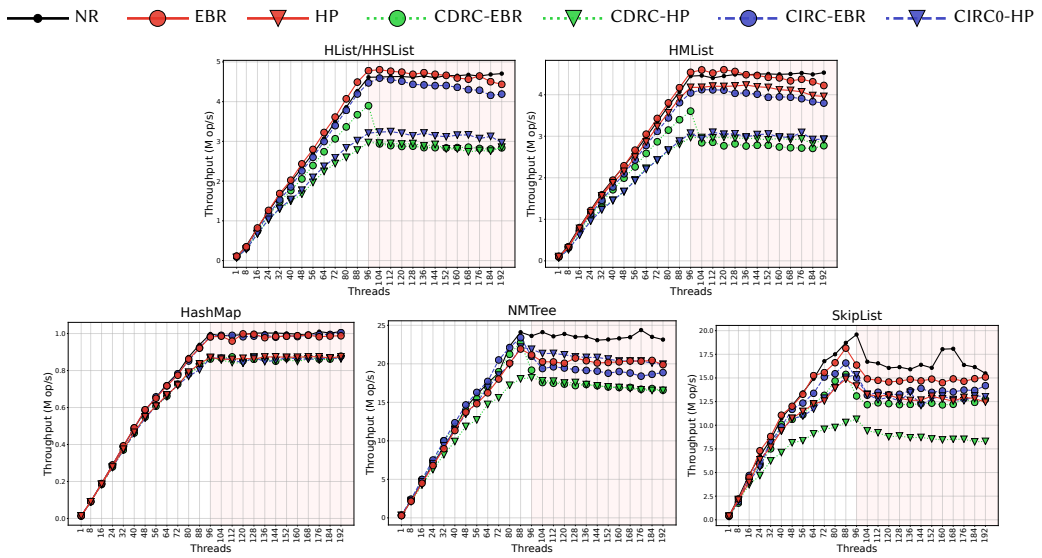


Fig. 25. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with large key ranges.

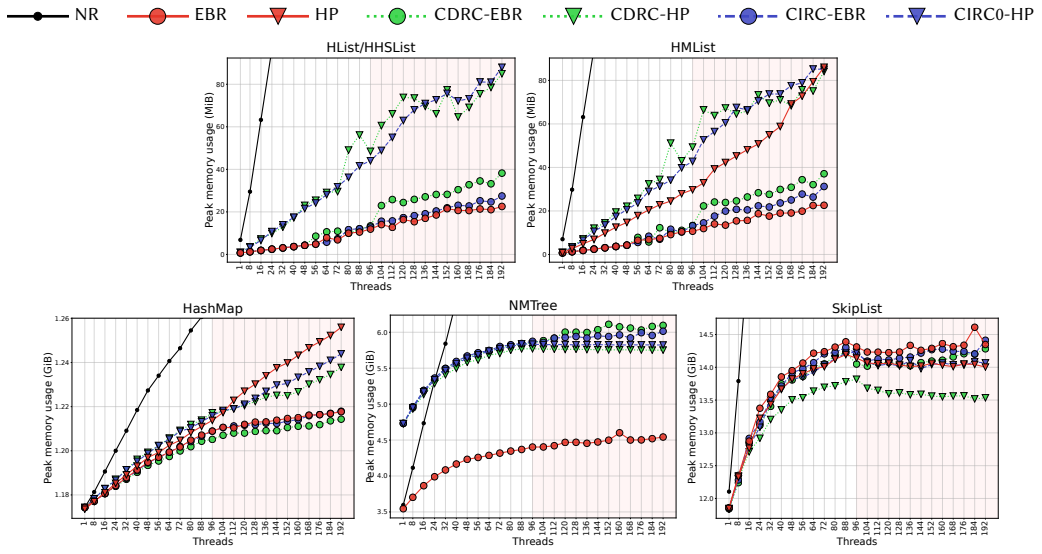


Fig. 26. Peak number of memory usage of write-heavy workloads for a varying number of threads with large key ranges.

C.3 Read-write & Small Key Ranges (1K for Lists and 100K for Others)

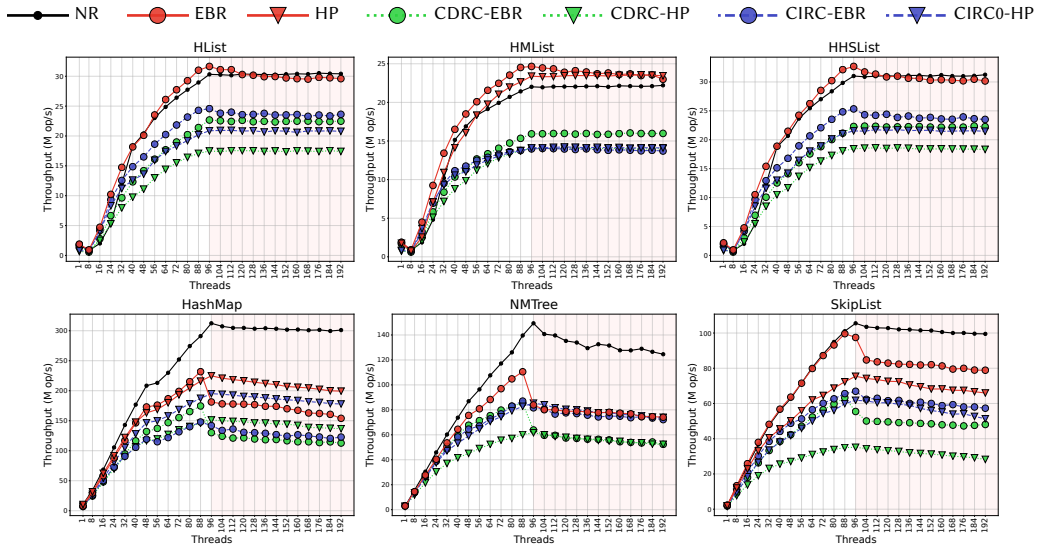


Fig. 27. Throughput (million operations per second) of read-write workloads for a varying number of threads with small key ranges.

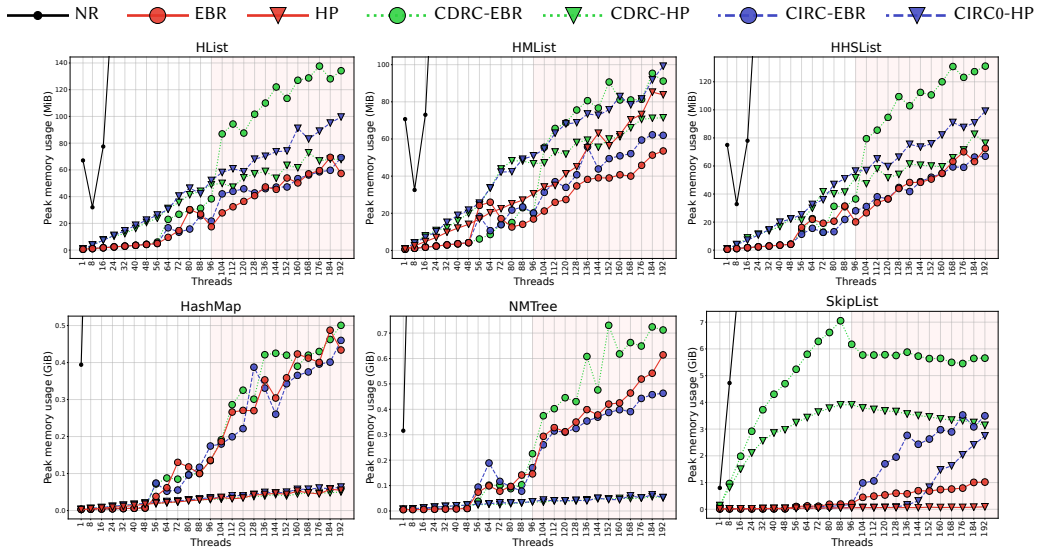


Fig. 28. Peak number of memory usage of read-write workloads for a varying number of threads with small key ranges.

C.4 Read-write & Large Key Ranges (10K for Lists and 100M for Others)

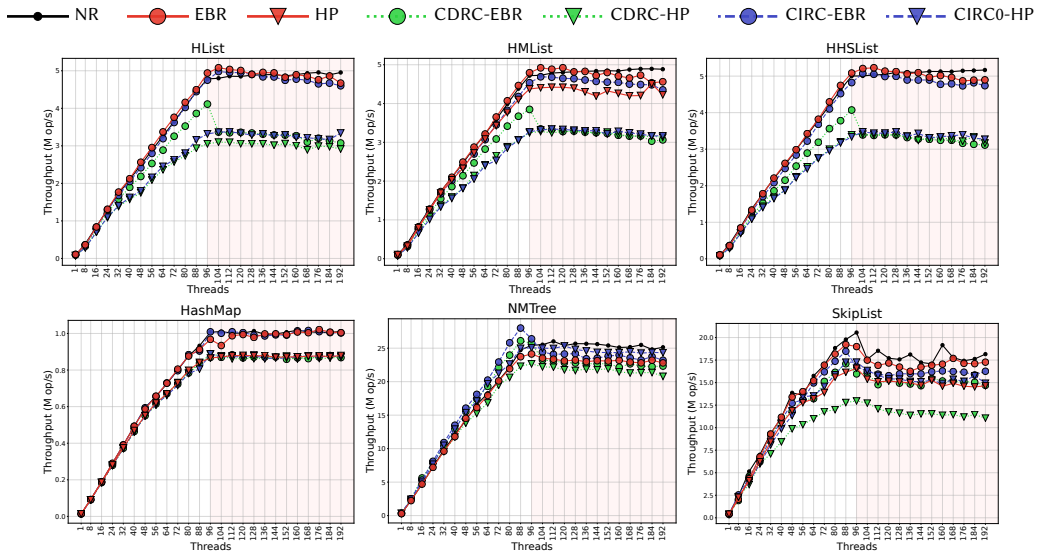


Fig. 29. Throughput (million operations per second) of read-write workloads for a varying number of threads with large key ranges.

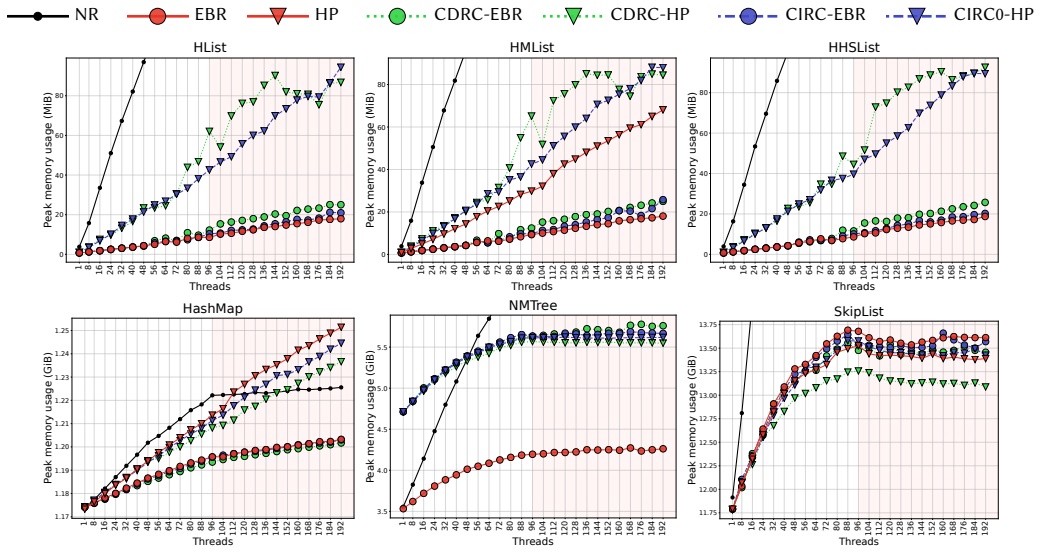


Fig. 30. Peak number of memory usage of read-write workloads for a varying number of threads with large key ranges.

C.5 Read-most & Small Key Ranges (1K for Lists and 100K for Others)

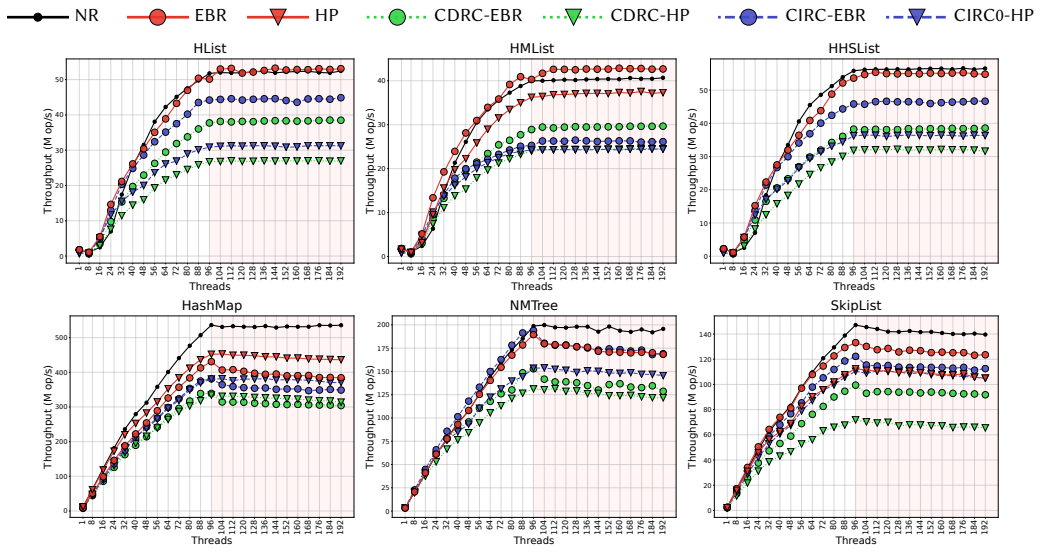


Fig. 31. Throughput (million operations per second) of read-most workloads for a varying number of threads with small key ranges.

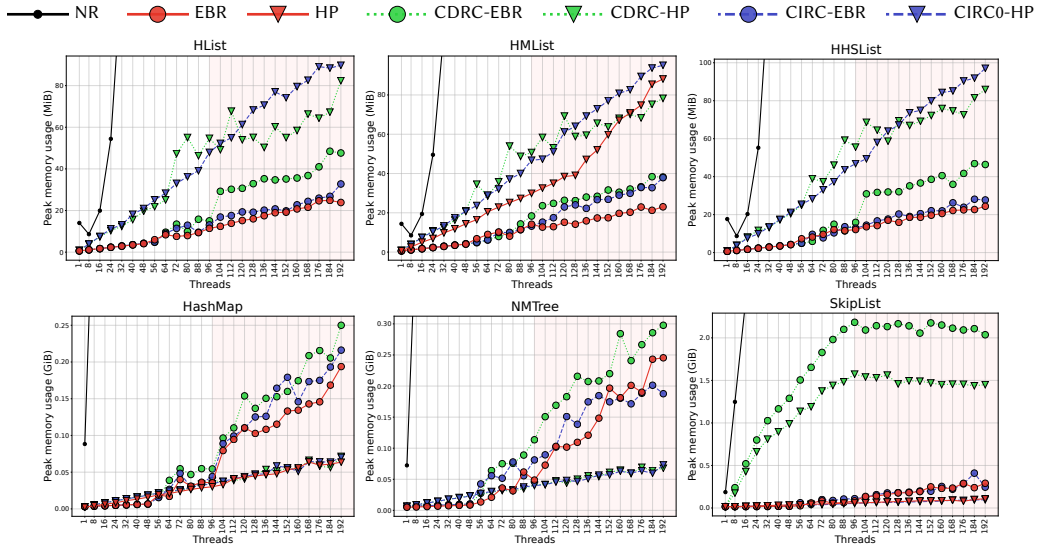


Fig. 32. Peak number of memory usage of read-most workloads for a varying number of threads with small key ranges.

C.6 Read-most & Large Key Ranges (10K for Lists and 100M for Others)

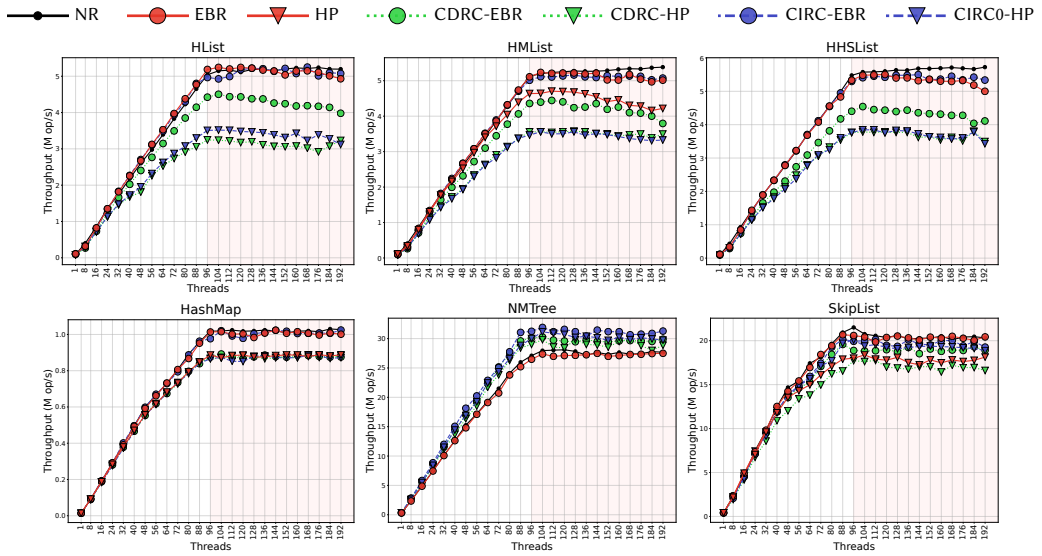


Fig. 33. Throughput (million operations per second) of read-most workloads for a varying number of threads with large key ranges.

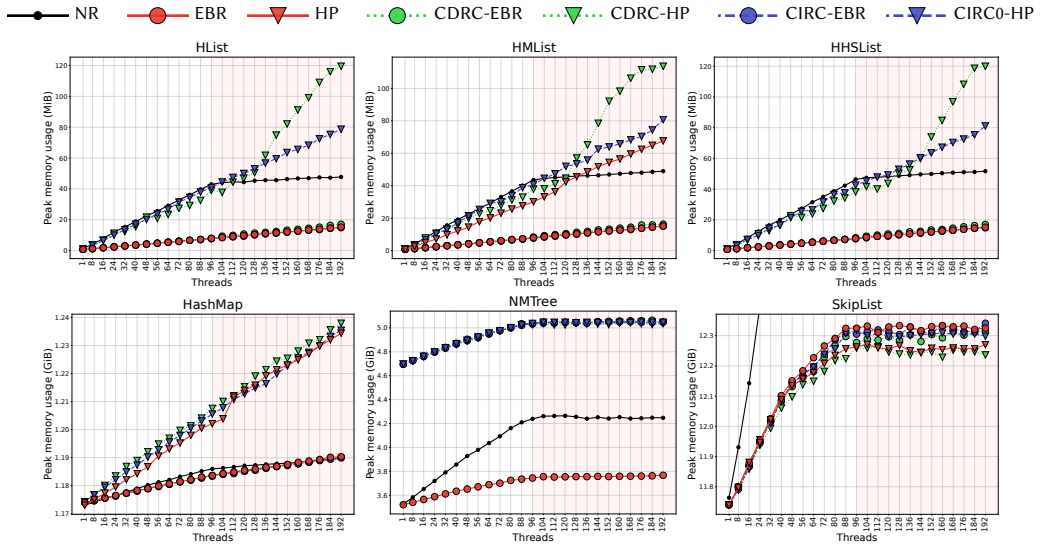


Fig. 34. Peak number of memory usage of read-most workloads for a varying number of threads with large key ranges.

C.7 DoubleLink

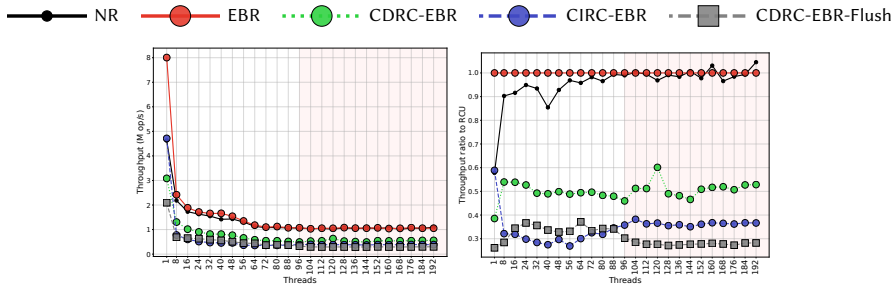


Fig. 35. Throughput (million operations per second) of DoubleLink workloads for a varying number of threads.

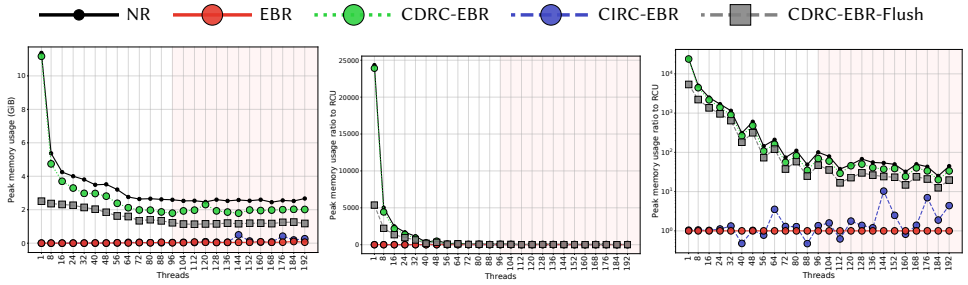


Fig. 36. Peak number of memory usage of DoubleLink workloads for a varying number of threads.

Received 2023-11-16; accepted 2024-03-31