Revisiting Partial Tracing for Safe, Efficient, and Concurrent Garbage Collection in Unmanaged Languages

JEONGHYEON KIM, KAIST, Korea
JONGSE PARK, KAIST, Korea
YOUNGJIN KWON, KAIST, Korea
JEEHOON KANG, FuriosaAI, Korea

Garbage collection (GC) remains a desirable yet elusive goal in unmanaged languages like C/C++ and Rust, where concurrent memory reclamation must be achieved without compiler or runtime support. Existing techniques face fundamental trade-offs among *efficiency*, *safety*, and *ease of integration*: *tracing collectors* like BDWGC incur costly stop-the-world pauses and unsafe conservative scanning, while *reference counting* schemes like CIRC are safe but introduce high overhead and require manual handling of cyclic data.

We present a safe, efficient, and easy-to-integrate concurrent GC library, revisiting *partial tracing* (PT), a concept initially conceived by Bacon et al. 22 years ago. PT is a hybrid approach that maintains reference counts for roots, ensuring safety through precise root identification, and traces from objects with non-zero counts, offering ease of integration by handling cyclic garbage. Although PT has historically been considered inefficient due to the high cost of root mutation, we overcome this limitation in two design steps. First, *Concurrent Partial Tracing* (CPT) introduces *phase consensus*, enabling concurrent phase coordination without mutator suspension and eliminating most reference-count updates during traversal. Second, *Concurrent Deferred Partial Tracing* (CDPT) further reduces overhead by replacing atomic root updates with a lightweight, *hazard pointer* (HP)-based mechanism safeguarded by a *phase barrier*. Our design matches the performance of manual schemes like RCU and HP while outperforming automatic collectors like BDWGC and CIRC.

CCS Concepts: • **Software and its engineering** → **Garbage collection**; *Allocation / deallocation strategies*; • **Theory of computation** → *Concurrent algorithms*.

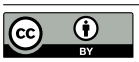
Additional Key Words and Phrases: garbage collection, partial tracing, reference counting, concurrent memory reclamation, Rust

ACM Reference Format:
Jeonghyeon Kim, Jongse Park, Youngjin Kwon, and Jeehoon Kang. 2026. Revisiting Partial Tracing for Safe, Efficient, and Concurrent Garbage Collection in Unmanaged Languages. *Proc. ACM Program. Lang.* 10, PLDI, Article 000 (June 2026), 44 pages. https://doi.org/10.1145/0000000.0000000

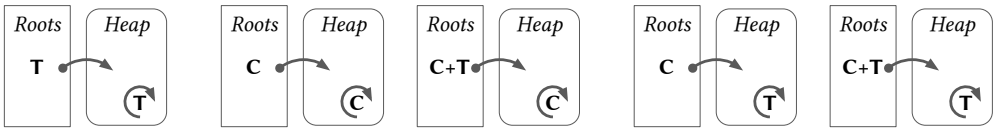
1 Introduction

Garbage collection (GC) is a robust and effective solution to the long-standing challenge of concurrent memory reclamation. GC is the de facto memory management strategy for *managed languages*, which provide an ideal environment for GC, as their compilers and runtimes offer built-in support enabling GC to operate safely and efficiently as a fundamental feature. Specifically, the compiler inserts *safepoints* [46] at strategic locations in the code. These safepoints allow the runtime to suspend mutator threads at known, safe states, enabling efficient and low-latency phase changes. Additionally, the compiler generates precise metadata, *e.g.*, *stack maps*, that informs the collector

Authors' Contact Information: Jeonghyeon Kim, KAIST, Daejeon, Korea, jeonghyeon.kim@kaist.ac.kr; Jongse Park, KAIST, Daejeon, Korea, jspark@casys.kaist.ac.kr; Youngjin Kwon, KAIST, Daejeon, Korea, yjkwon@kaist.ac.kr; Jeehoon Kang, jeehoon.kang@furiosa.ai, FuriosaAI, Seoul, Korea.



ACM 2475-1421/2026/6-ART000
https://doi.org/10.1145/0000000.0000000



(a) Tracing (b) RC (c) DRC[a] (d) PT (e) CDPT (§3)

Fig. 1. Schematic diagrams of garbage collection techniques by Bacon et al. [6, Figures 3–6]

[a]The original DRC diagram from Bacon et al. [6, Figure 5] considers only tracing for roots, but in modern algorithms the two often coexist, *e.g.*, a system may use RC for globally shared roots while relying on tracing for local roots.

exactly where pointers are located. Consequently, managed runtimes achieve GC that is *efficient* (low coordination overhead), *safe* (no missed roots), and *easy-to-integrate* at the same time.

Concurrent memory reclamation remains a fundamental challenge even in *unmanaged languages*. In highly concurrent workloads, shared resources often have neither a statically identifiable owner nor a statically determinable lifetime (*i.e.*, the program point at which an object becomes unreachable), making it difficult to reclaim memory safely. To address this, manual techniques such as *safe memory reclamation* (SMR) [37, 56–59] have been proposed, but they require programmers to explicitly track in-use pointers, a task that remains notoriously error-prone even for experts. Indeed, Anderson et al. report multiple misuse bugs in benchmark suites of existing SMR techniques, leading to *use-after-free* errors and memory leaks. Consequently, automatic GC remains highly desirable even in unmanaged languages, especially for highly concurrent workloads [3, 47, 78].

1.1 Problem: Trade-off of GC in Unmanaged Languages

However, unmanaged languages such as C/C++ and Rust lack integrated compiler and runtime support, *e.g.*, safepoints and stack maps. This absence forces GC techniques to rely on alternative mechanisms, which fundamentally constrain their design. Consequently, existing approaches inevitably make trade-offs among efficiency, safety, and ease of integration, as summarized in Table 1. We now review prior techniques for unmanaged languages in Bacon et al.'s taxonomy, as illustrated in Fig. 1: *tracing*, *reference counting* (RC), and their two hybrids: *deferred reference counting* (DRC) and *partial tracing* (PT).

Tracing. In unmanaged languages, the lack of integrated compiler and runtime support poses two fundamental challenges for implementing a tracing GC: **(1)** without compiler-inserted safepoints, coordinating GC phases across mutators and the collector becomes significantly more difficult, and **(2)** without precise stack maps or global maps, the collector cannot accurately identify roots.

As a result, tracing GCs for unmanaged languages often rely on conservative techniques to compensate for these limitations. *Conservative garbage collectors* (CGCs), such as the Boehm–Demers–Weiser collector (BDWGC) [14, 77], provide automatic reclamation in unmanaged languages by identifying reachable memory conservatively, offering a "drop-in" alternative to malloc/free.

However, this design sacrifices both efficiency and safety. To coordinate collection, BDWGC suspends all mutator threads via OS signals, incurring a *stop-the-world* (STW) pause that introduces substantial overhead and scales poorly [12, 34]. Indeed, our evaluation shows up to an 89% slowdown relative to the fastest manual scheme (§7). Moreover, during these pauses BDWGC performs conservative stack and global-memory scans, treating any pointer-like value as a valid pointer. This approach is prone to both false positives and false negatives [10, 11, 41]: encoded or misaligned pointers may be missed, and references from unmanaged objects (*e.g.*, a default-constructed std::vector<void*>) may go undetected, causing premature reclamation and potential use-after-free.

Table 1. Comparison of garbage collection techniques for unmanaged languages.

Types of GC (Bacon et al.)		**Efficiency** (*Mutator Perf.*)	**Safety** (*Precise Roots*)	**Ease-of-integration** (*Easy API & Cycle Coll.*)
Tracing (*e.g.*, BDWGC [14, 77])		✗	✗	✓
RC		✗	✓	✗
Deferred RC (*e.g.*, CIRC [47])		▲	✓	▲
Partial Tracing (Our Solutions)	CPT (§4)	▲	✓	✓
	CDPT (§5)	✓	✓	✓

Reference counting (RC). This design maintains a count for each object, enabling the collector to identify *anti-roots*, objects whose counts have fallen to zero, from which it can trace and reclaim dead objects. While modern unmanaged languages often provide *type-safe* interfaces for RC, a naive RC design remains notoriously inefficient due to frequent updates to reference counts. Furthermore, reclaiming cyclic garbage requires manual intervention via *weak references*, which complicates integration and incurs overhead for maintaining weak counts.

Hybrid #1: Deferred reference counting (DRC). According to Bacon et al., tracing and RC are *algorithmic duals*, and DRC [27] represents a practical hybrid of these two paradigms: RC maintains counts along heap edges, and the collector periodically traces from roots to reclaim zero-counted objects that are unreachable from them. By mitigating the cost of frequent RC updates caused by high root mutation rates, DRC achieves better performance than naive RC.

Recent variants for unmanaged languages [3, 4, 24, 47, 78] further improve performance by employing SMR techniques to protect *local roots*, the principal source of frequent mutations, allowing decrements and reclamation to be deferred. This technique enables efficient local root management and lets the collector identify anti-roots concurrently without suspending mutators.

However, these recent schemes still sacrifice efficiency and ease of integration in two fundamental ways: **(1)** They continue to maintain reference counts for most heap objects (*i.e.*, internal edges and global roots), which introduces significant overhead under write-intensive workloads (*e.g.*, CIRC incurs up to a 35% slowdown [47]). **(2)** They inherit RC's inability to automatically reclaim cyclic garbage, along with its associated integration challenges; for example, CIRC falls behind the fastest scheme by up to 55% on workloads containing cycles (§7).

Hybrid #2: Partial tracing (PT). Bacon et al. proposed partial tracing (PT) as the converse of DRC: instead of tracing from roots and maintaining reference counts among heap objects, it maintains reference counts only for roots and traces the heap starting from objects with non-zero counts, thereby avoiding stack and global scans. A GC based on this design would be *safe* by precisely identifying roots and *easy to integrate* by supporting cyclic garbage with tracing.

However, Bacon et al. argued that this approach would significantly suffer from the extremely high cost of root mutations. Indeed, to our knowledge, partial tracing has not been adopted in existing concurrent GC implementations for the 22 years since its publication.

1.2 Contributions: Concurrent Deferred Partial Tracing

We design a safe, efficient, and easy-to-integrate GC library for concurrent memory reclamation in unmanaged languages based on partial tracing. We base our solution on PT to achieve *safety* by replacing the fragile scanning heuristics of CGCs with precise reference counts for roots. Because every root pointer is explicitly reference-counted (as illustrated in Fig. 1d), the collector no longer scans unmanaged memory word by word; instead, it directly enumerates heap objects with non-zero

root counts, yielding an exact root set. By incorporating tracing, the library naturally handles cyclic garbage, thereby providing *ease of integration* without the need for manual cycle-breaking. Building on this foundation, we demonstrate that, with a highly concurrent design that avoids STW pauses and substantially reduces root-mutation overhead, PT can in fact be more *efficient* than prior CGC and DRC schemes. We advance this design perspective as follows.

In §4, we present our partial solution: the first *concurrent PT* (CPT) algorithm that eliminates most reference-count updates during object-graph traversal and coordinates GC phases without suspending mutators. We refer to this mechanism as *phase consensus*, inspired by *epoch consensus* [56, 57]. To access the managed heap, mutators enter *phase-critical sections*, which protect the acquisition of new local pointers and allow safe mutation of shared objects. Within these sections, mutators may create RC pointers (thereby incrementing reference counts) and then exit. Outside the critical section, these RC pointers can be freely dereferenced, enabling the collector to concurrently trace the object-graph and reclaim unreachable objects without interfering with mutator execution. Moreover, phase consensus also coordinates GC phases *without mutator suspension*, and it ensures the *weak tricolor invariant* (§2.1), enabling safe and efficient concurrent tracing.

In §5, we develop our full solution: the *concurrent deferred PT* (CDPT; Fig. 1e) that further reduces the reference-count updates by protecting local roots using a *single-writer–multiple-reader* (SWMR) pointer array, which is analogous to *hazard pointers* (HP) [58, 59] from the SMR literature. Importantly, unlike the original HP technique, our design does *not* require the subtle *validation* step [54]. The central idea is to replace RC-based protection, which requires an expensive atomic read-modify-write operation at the end of each critical section, with a lightweight HP-style protection implemented as an atomic store into a local array. The collector can then concurrently scan this array with precise knowledge of pointer alignment, and the safety of scanning these HP-protected roots is guaranteed by our *phase barrier*, which enables lock-free and correct root scanning. This protection strategy of CDPT resembles that of DRC, in that it optimizes a subset of the root set that was previously managed by reference counts by handling them through tracing instead.

In §6, we formalize four language-agnostic safety rules that an implementation must satisfy for the correctness of our algorithms. These rules are enforced at compile time in Rust via its ownership type system, while in C/C++ they can be upheld through a combination of API design and compiler-assisted static analysis (*e.g.*, Clang's thread safety analysis and lifetime annotations).

In §7, we demonstrate our library's *efficiency* and *ease of integration*: it is as efficient as manual reclamation schemes like RCU and HP while outperforming automated techniques like CIRC and BDWGC, and it is easier to integrate than CIRC, which requires weak pointers to manually break cycles. Such modifications may significantly alter an algorithm's original structure. We believe this work is a pivotal step toward advancing automatic memory management in unmanaged languages.

Outline. In §2, we review the background to motivate our solution. In §3, we present our GC interface. In §4–7, we detail and evaluate our GC design. In §8, we conclude with related and future work, such as extending our technique with a generational heap and memory defragmentation.

2 Background

We review tracing garbage collection (§2.1), manual reclamation algorithms (§2.2), and reference counting (§2.3), which together form the basis for our approach.

2.1 Tracing Garbage Collection

Tricolor invariant. For concurrent and on-the-fly GC, it is often preferable to satisfy the *weak* tricolor invariant rather than the strong one for performance [21, 85]. The weak invariant allows black-to-white pointers so long as each such white object remains *grey-protected*, *i.e.*, reachable

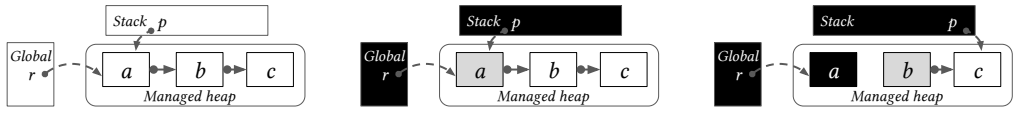


Fig. 2. Example scenario of a system satisfying the weak tricolor invariant with Yuasa's deletion barrier. (a) Initial setup. (b) The collector scans the global and local roots, shading them black. p is then pushed onto the mark stack, shading a grey. (c) The mutator detaches b from a and acquires a pointer to c. It is safe because Yuasa's barrier shades b upon deletion, ensuring that c remains *grey-protected*.

Algorithm 1 RCU interface.

```
struct Guard // RAII-type for CS
    function new() → Guard
    method destruct() // Close this CS
function Defer(task: fn())
```

Algorithm 2 HP interface.

```
struct HPSlot // Protector for a single pointer
    function new() → HPSlot
    method protect(ptr: void*)
function Retire<T>(ptr: T*)
```

from some grey object either directly or through a chain of white objects [46]. Under the strong tricolor invariant, a black (*i.e.*, already scanned) mutator must not hold a reference to a white object. This creates a problem in concurrent settings: a mutator may acquire a reference to a white object after the mutator's stack and registers have been scanned. Typical designs address this in one of two ways: **(1)** enforce a load barrier or read barrier that shades newly loaded white objects grey, which slows down hot paths, or **(2)** suspend all mutator threads and re-scan each mutator's stack and registers once tracing is believed to be complete, ensuring that no black-to-white edges were introduced, thereby increasing pause time. In contrast, the weak tricolor invariant allows mutators to temporarily acquire references to white objects, but ensures safety through write barriers such as Yuasa's deletion barrier, as explained below.

Write barriers. To maintain the tricolor invariants amid concurrent mutation, collectors employ *write barriers* [1, 13, 30, 31, 40, 74] that coordinate the interactions between mutators and the collector's wavefront. Two classical barrier styles are Dijkstra et al.'s insertion barrier and Yuasa's deletion barrier. Dijkstra et al.'s barrier, also known as the *incremental update* or grey mutator technique, preserves the strong invariant by shading the target object grey whenever a black object stores a reference to a white one. This ensures no black-to-white pointer is ever created at pointer insertion time. In contrast, Yuasa's barrier, a *snapshot-at-the-beginning* or black mutator technique, maintains the weak invariant by shading the target of a deleted pointer grey when a mutator overwrites or removes a reference. This ensures all white objects remain grey-protected at pointer deletion time, as illustrated in Fig. 2.

2.2 Manual Reclamation Algorithms

In contrast to the automatic GC, manual algorithms such as safe memory reclamation (SMR) achieve better efficiency through the programmer's explicit coordination.

Epoch-based RCU. Read-copy-update (RCU) [56, 57] is a general-purpose task scheduler that can also be used for reclamation, with its interface[1] illustrated in Algorithm 1. Each thread performs its

[1]We present all algorithms in Rust-style pseudocode.

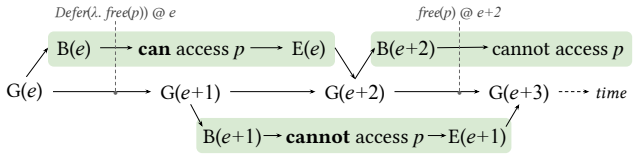


Fig. 3. Epoch consensus diagram. G(e): global epoch advancement to epoch e; B(e)/E(e): beginning and ending of a critical section.

operations within a *critical section* (CS), which begins with the creation of an RAII Guard. When a thread Defers a task, the task is executed only after all concurrent critical sections have completed.

The *epoch-based RCU* implementation [37, 39] ensures correctness using an *epoch*, a monotonically increasing integer. Fraser's implementation operates as follows: **(1)** a *global epoch* is shared among all threads; **(2)** each critical section starts by assigning the global epoch to its *local epoch*; and **(3)** the epochs of all concurrent critical sections differ by at most one. Fig. 3 illustrates an example of this epoch consensus in action. A node represents an event and an edge represents the *happens-before* relation [53]. Suppose a pointer p becomes unreachable from the entry points of the data structure, and its reclamation is deferred at global epoch e. Then, critical sections with local epoch e, denoted as B(e)$\rightarrow \cdots \rightarrow$E($e$), may still access p, whereas later critical sections cannot. Eventually, p becomes safe to reclaim at epoch $e + 2$, because by that time, all concurrent critical sections with local epoch e (or earlier) must have completed.

Hazard pointers. HP [58, 59] guarantees safety by individually protecting each thread's local pointers to prevent their premature reclamation. Its interface is illustrated in Algorithm 2. Each thread maintains a *single-writer–multiple-reader* (SWMR) pointer array that collectors can concurrently scan. Each HPSlot in this array serves as a protection slot for a local pointer to a shared object: a thread obtains a slot via new() and protects a pointer by atomically writing it into the chosen slot using protect(). However, even after protecting it, a pointer is still unsafe to dereference because a concurrent thread may already have Retired and reclaimed it. To ensure safety, the thread should *validate* that the protected pointer is not retired yet. The validation strategy drastically differs among data structures, and it is one of the most challenging aspects of using HP [54, 71].

Trade-off and hybrids. RCU and HP exemplify *coarse-grained* and *fine-grained* protection, highlighting a trade-off between *efficiency* and *robustness*. RCU is highly efficient and widely applicable, since it incurs no per-node overhead and supports optimistic traversal [16, 32, 39, 42, 48, 54, 60, 65]. Its limitation is poor robustness: a stalled or long-running thread can block reclamation of all nodes. HP offers robustness by protecting nodes individually with HPSlots, bounding memory consumption by the number of slots of all mutators. However, it suffers from per-node overhead because each traversed node must be written to a slot and later validated.

To balance these trade-offs, several works combine coarse-grained RCU-style traversal with fine-grained HP protection [17, 49, 50, 72]. The core idea is to alternate between RCU and HP phases, a strategy that Kim et al. refer to as *RCU-expedited traversal*: **(1)** A mutator performs most of its traversal within an RCU phase, then checkpoints selected pointers by protecting them with HP slots. **(2)** It then reinitializes the RCU phase, revalidates these checkpoints, and continues traversal. Because most traversal steps occur during RCU phases, the majority of HP's per-node overhead is eliminated. Meanwhile, accesses to HP-protected objects do not require RCU critical sections, allowing other nodes to be reclaimed concurrently. Despite these benefits, such schemes remain difficult to integrate because they require complex failure-recovery logic [50].

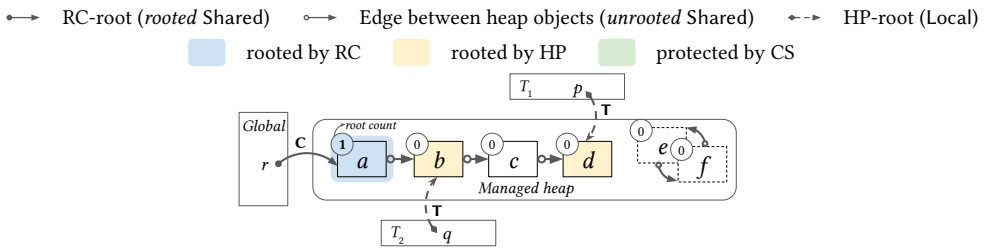


Fig. 4. Example of a concurrent linked list (from a to d) and reclaimable objects (e and f) under CDPT (§5).

2.3 Reference Counting

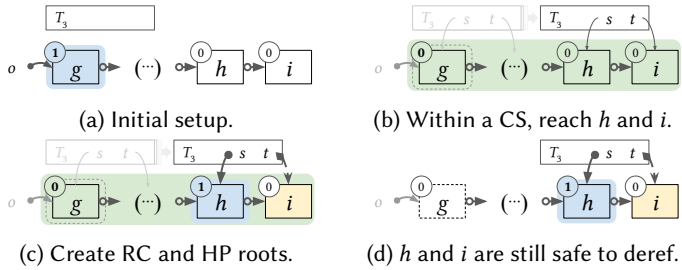
Reference counting (RC) automatically tracks the number of active references to each object. But concurrent RC libraries readily available in unmanaged languages, such as C++'s atomic<shared_ptr<T>> or Rust's Arc<T>, incur substantial performance overhead due to frequent atomic updates to counters. To reduce contention on the counters and improve scalability under concurrent workloads, recent *deferred reference counting* (DRC) algorithms [3, 4, 24, 27] such as CIRC [47] integrate SMR schemes as their backend to defer counter updates or object reclamation until it is safe.

Nevertheless, even with such optimizations, those schemes remain expensive for write-heavy applications due to frequent increments and decrements to counters. For instance, inserting a single node into a linked list, where every node's count will eventually be one, still requires three atomic updates: incrementing the successor's counter, incrementing the new node's counter, and finally decrementing the successor's counter again. Furthermore, workloads involving cyclic or back-edge references impose additional complexity due to the special handling required for weak references, causing nontrivial overhead and limiting the overall efficiency.

3 Interface

We first present the interface of our GC library designed through § 4 and 5. Fig. 4 illustrates an example of a concurrent linked list with mutators accessing its nodes, all managed by our library. The memory space follows the schematic diagram from Fig. 1e: the global region is *counted* (**C**), while mutator stacks are *traced* (**T**). Globally shared root pointers use reference counting (RC), while thread-local root pointers use the cheaper hazard pointer (HP) protection (§5). For example, the *shared* root pointer r in the global region maintains a reference count, called a *root count*, to indicate that the target object is a root, while mutators (T_1 and T_2) maintain their *local* root pointers in per-thread HP slots. The complete set of roots is therefore the union of all objects protected by hazard pointers and all objects with a non-zero root count. Both sets can be scanned concurrently, and the correctness of scanning HP slots is ensured by our *phase barrier* (§5.2), which enables safe and lock-free HP-root scanning.

Efficient traversals. Creating a new RC-root or HP-root is done within a *phase-critical section*, which is similar to an RCU critical section (§2.2) in that all reachable pointers within the section are safe to dereference without any explicit protections. Fig. 5 illustrates an example of an efficient traversal on a linked list. Fig. 5a: Suppose an entry point node g is currently rooted by RC or HP. Fig. 5b: A mutator first initiates its critical section and traverses the list starting from g, without updating RC or protecting HP. Note that the root pointer o to the entry point may be destroyed during the traversal, but the phase consensus guarantees that all nodes originally reachable from g are safe to dereference. Fig. 5c: Once the mutator reaches the destination nodes, h and i, it may create roots using either RC or HP. Fig. 5d: After the critical section ends, only the created root



(a) Initial setup. (b) Within a CS, reach h and i.

(c) Create RC and HP roots. (d) h and i are still safe to deref.

Fig. 5. Efficient traversal within a phase-critical section.

Algorithm 3 Definitions of managed pointers.

```
11: // Sync + Send, Atomically-mutable
12: pub struct AtomicShared<T>
13:     link: Atomic<Tagged<T>>
14: // Sync + Send, Immutable
15: pub struct Shared<T>
16:     inner: Atomic<Tagged<T>>
17: // Sync + !Send, Immutable
18: pub struct Local<'g, T>
19:     ptr: ManObj<T>*
20:     shield: Option<HPSlot> // None if CS-protected
21: struct ManObj<T>
22:     rc: AtomicU64 // Use MSB for marked (§4.2)
23:     item: T
24: enum PtrMeta // Representable in 2-bits
25:     Rooted,
26:     Unrooted(Color) // Color: 0 or 1
27: // Tagged pointer to ManObj<T> with PtrMeta
28: struct Tagged<T>
29:     method addr() → ManObj<T>*
30:     method meta() → PtrMeta
```

pointers are safe to dereference. This design generalizes the RCU-expedited traversal strategy used in prior SMR schemes (§2.2) while simplifying synchronization. In contrast to HP-BRCU [50], our approach obviates the need to *revalidate* a protected pointer before initiating a new traversal, as all objects transitively reachable from a root object are likewise protected from reclamation.

Managed pointer types. On this foundation, we provide managed pointer types in Rust that capture their semantic scopes (*e.g.*, global or local) and implement proper synchronization methods.

To model RC-roots (*e.g.*, r in Fig. 4) and internal edges between heap objects (*e.g.*, the next pointer of a), we provide Shared[2] and AtomicShared (lines 12 and 15), which can be shared (*i.e.*, Rust's Sync bound) or sent across mutators (*i.e.*, Rust's Send bound). Shared provides a method returning an immutable borrow &T, and AtomicShared is a cell that supports atomically reading and writing Shared. When first created, Shared and AtomicShared own a root count (*i.e.*, *rooted*), but when they become child edges of a heap object (*i.e.*, boxed), they no longer maintain a root count (*i.e.*, *unrooted*). Once a shared pointer is unrooted, it can never be rooted again because we do not support unboxing (*i.e.*, bringing a heap object back to the stack).[3]

To model HP-roots (*e.g.*, p and q in Fig. 4) and pointers within phase-critical sections (*e.g.*, s and t in Fig. 5b), we provide Local (line 18), which provides an immutable borrow &T, similar to Shared, but cannot be sent to other mutators (*i.e.*, !Send). Notably, these Local pointers are valid for a specific duration, *e.g.*, s and t in Fig. 5b remain valid until the end of the critical section. We capture this property using Rust's lifetime parameter, guaranteeing type-safety at compile-time.

User's data type. Because our library allows multiple mutators to hold shared, *immutable* borrows to the same object, the user-defined type T must be safe to share or transfer across threads, *i.e.*,

[2]Its inner pointer is an atomic variable (line 16) because the collector may update it concurrently (*e.g.*, during shading).
[3]This no-unboxing restriction is a limitation of the algorithm, not an artifact of Rust integration (§6).

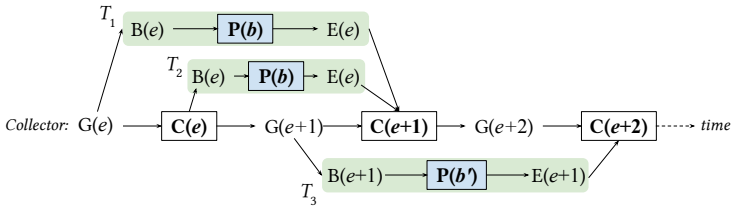


Fig. 6. Phase consensus diagram. The boxes highlight the additions to the epoch consensus (Fig. 3). P(b): protection of a pointer b, using either RC or HP (§5); C(e): *collection-work* for the epoch e.

it must satisfy Rust's Send + Sync bounds. Consequently, if T contains mutable state, that state must be protected by an appropriate thread-safe synchronization primitive (*e.g.*, RwLock<String>), which is the standard pattern for safe concurrent programming.

Additionally, since our collector does not conservatively scan heap objects, T must implement the TraceObj trait, which enumerates its Shared and AtomicShared fields. Concretely, it provides two methods, unroot_outgoings and shade_outgoings, which simply invoke unroot and shade on each shared pointer field, just as their names imply.[4] These routines are used to decrement the root counts of child objects when a parent object is boxed (see §4.5), and to shade child objects during tracing (see §4.2), respectively. Notably, these methods must not scan interior-mutable fields such as Mutex<Shared<T>>;[5] we formalize this and other safety conditions as language-agnostic rules in §6. We do not support finalizers [43] yet, so object destruction must use Rust's default destructor.

4 Concurrent Partial Tracing with Phase Consensus

We present our partial solution, *concurrent PT*, which is made possible by the *phase consensus* mechanism. We introduce the mechanism (§4.1), outline the collection cycle including marking and color management (§4.2), and describe its constituent phases (§4.3). We then analyze its properties (§4.4) and present the managed pointer implementation (§4.5).

4.1 Phase Consensus

Like conventional mark-and-sweep collectors, our collector operates in multiple phases within a single GC cycle, but replaces the usual STW pause with lock-free coordination. We introduce *phase consensus*, inspired by *epoch consensus* (§2.2).

Fig. 6 illustrates the key idea of phase consensus.[6] During an epoch e, mutators operate within their *phase-critical sections*, *e.g.*, $\mathrm{B}(e) \rightarrow \mathrm{P}(b) \rightarrow \mathrm{E}(e)$, where new managed pointers, *e.g.*, RC-protected, may be created. Within a critical section, mutators can dereference any acquired pointers. Outside a critical section, only managed pointers created in earlier sections are dereferenceable.

We abstract the collector's responsibility during each phase, *e.g.*, reclaiming previously identified garbage, as *collection-work*, denoted by $\mathrm{C}(e)$ in Fig. 6. The collector periodically advances the global epoch, denoted as $\mathrm{G}(e) \rightarrow \mathrm{G}(e{+}1)$, to initiate the next collection-work. When the epoch advances, each mutator entering a new critical section updates its local epoch to match the new global epoch and performs heap accesses according to the invariants of that phase (*e.g.*, allocating new objects as black during tracing). Once all mutators at e have completed their critical sections, the collector can safely perform the collection-work for the succeeding epoch $e{+}1$, denoted as $\mathrm{E}(e) \rightarrow \mathrm{C}(e{+}1)$.

[4]Because these methods are mechanical, our library provides a procedural macro that automatically derives them.
[5]This type is unnecessary anyway; AtomicShared should be used instead. We use it here merely as a representative example.
[6]In our diagram, $e{+}1$ denotes the logical successor of epoch e, rather than a simple arithmetic increment. See §4.2 for details.

Algorithm 4 Phase-critical section.

```
31: global variables
32:     EPOCH: Atomic<Epoch>
33:     HANDLES: ConcurrentList<Handle>
34: thread-local variable
35:     handle: &'h Handle // 'h: lifetime of mutator
36: struct Handle
37:     epoch: Atomic<Epoch>
38: impl Guard // RAII-type for CS
39:     function new() → Guard
40:         curr_epoch ← EPOCH.load(Acquire)
41:         loop
42:             handle.epoch.store(curr_epoch, Release)
43:             fence(SeqCst) // Sync with line 62
44:             new_epoch ← EPOCH.load(Acquire)
45:             if curr_epoch = new_epoch then break
46:             curr_epoch ← new_epoch
47:     method destruct()
48:         handle.epoch.store(⊥, Release)
```

Algorithm 5 Mutator: accessing the heap.

```
51: guard ← Guard::new()
52: Successfully initiated a CS.
53:     Freely traverse and mutate the heap,
        according to invariants for the current epoch.
54:     May update RC or create a new HP (§5).
55: guard.destruct() // Implicitly called
```

Algorithm 6 Collector: advancing the epoch.

```
61: next_e ← EPOCH.inc_phase() // e → e + 1
62: fence(SeqCst) // Sync with line 43
63: for handle ∈ HANDLES do
64:     loop
65:         handle_e ← handle.epoch.load(Acquire)
66:         if handle_e = ⊥ || handle_e = next_e then
67:             break
68: Do collection-work for e + 1.
```

Concurrency. As in epoch consensus, phase-critical sections are lock-free, allowing all mutators (*e.g.*, T_1, T_2, and T_3 in Fig. 6) to enter their critical sections and access the managed heap concurrently. Unlike classic epoch consensus, however, our design *enables* mutators to keep their critical sections *practically short*: they individually protect only the finally acquired pointers and then promptly exit. When the collector advances the epoch, it may need to wait for *pinned* mutators, *i.e.*, those currently inside a phase-critical section holding a valid local epoch, to unpin (*e.g.*, $\mathrm{E}(e) \to \mathrm{C}(e{+}1)$ in Fig. 6), but newly created sections start in the updated epoch, so no additional sections can delay the collector further (*e.g.*, $\mathrm{B}(e{+}1)$ does not block $\mathrm{C}(e{+}1)$).

Safety. Safe traversal within a critical section is ensured by the weak tricolor invariant, maintained by the hybrid barrier design (§4.3). We analyze this property in §4.4. The intuition is that any collector's work that could affect mutators already inside their critical sections is strictly ordered after those mutators' work.

Algorithm. Algorithm 4 illustrates the Guard type, which implements phase-critical sections following the RCU interface (Algorithm 1).[7] To begin a critical section (line 39), the mutator sets its local epoch to the global epoch within a validation loop (line 41), retrying if the global epoch changes. To end it (line 47), the mutator resets its local epoch to ⊥ (*i.e.*, unpinned). Algorithm 5 shows how mutators access the heap under this protection.

As in epoch consensus, *memory fences* (line 43 and line 62) enforce the required *happens-before* relations for correct synchronization between mutators and collectors. Specifically, this pair of fences guarantees that one of the following two orderings holds: **(1)** line 42 *happens before* line 65, ensuring that all pinned mutators are visible to collectors; or **(2)** line 61 *happens before* line 44, guaranteeing that a phase transition is observable by validating mutators. In addition, the *Release-Acquire* synchronization between line 48 and line 65 ensures that all RC updates performed by mutators become visible once the collector observes their unpinning.

[7] We do not implement Defer (line 4 in Algorithm 1) because unreachable objects are automatically identified by tracing.

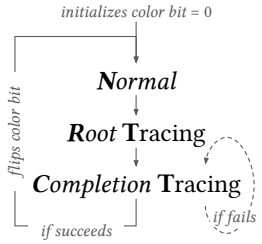


Fig. 7. Phase flowgraph.

			Epoch (*time* →)			
		Timestamp	0	1	2···	···
		Color bit	0	0	0	1 (= !0)
		Phase identifier	**N**	**RT**	**CT**	**N**
Mutator		Allocation color	white	**black**	**black**	white *= prev. black*
	Barriers	*Insertion* (§4.3)	**yes**			yes
		Deletion (§4.3)	no	**yes**		no
		Phase (§5.2)	no	**yes**	no	no

Fig. 8. Mutator-side invariants for each phase.

4.2 Collection Cycle

We outline the phases of a single collection cycle (Fig. 7) and the mutator-side invariants maintained across adjacent phases (Fig. 8).

Up to this point, we have treated the epoch as a simple integer. We now refine its meaning so that it represents the current phase, encoded as a tuple of three integers within a 64-bit word (from high to low bits): **(1)** a 61-bit *timestamp*, which monotonically increases at each phase transition; **(2)** a 1-bit *color bit*, which denotes the current white color; and **(3)** a 2-bit *phase identifier*, which specifies the current phase (*e.g.*, normal or tracing).

A single collection cycle in our design consists of three distinct phases, as illustrated in Fig. 7. **(1)** *Normal* (N; 00) represents a quiescent state where mutators execute without collector interference. **(2)** *Root Tracing* (RT; 01) marks the beginning of tracing, during which collectors identify and mark root objects, subsequently draining the mark stacks. **(3)** *Completion Tracing* (CT; 10) ensures that all mark stacks have been fully processed; if any remain non-empty, additional tracing steps are performed until global quiescence is achieved. Upon the successful completion of a collection cycle, the collector simply flips the color bit to alternate the meaning of white and black colors in the tricolor marking scheme, thereby resuming from the N phase for the next cycle.

Before delving into the details of each phase, we first introduce the setup for mark stacks and the color scheme of objects and pointers, which are managed across phases.

Mark stacks. Each mutator maintains a dedicated *local mark stack*, in addition to a *global mark stack* shared among all mutators and collectors. Both stacks are implemented using the Chase–Lev deque [18, 25], enabling concurrent work-stealing between collectors and mutators. All accesses to mark stacks occur within a critical section to ensure consistent meaning of colors. When a mutator encounters an object to be marked (*e.g.*, through a barrier), the corresponding marking task is pushed onto its local stack. If the local stack exceeds a threshold (*e.g.*, 4096 entries), the task is instead offloaded to the global stack. During tracing phases (*i.e.*, RT or CT), mutators may periodically help the collector based on a heuristic that accounts for the number of heap allocations and critical sections opened. A helping mutator first drains its own local mark stack; if empty, it attempts to steal work from the global stack.

Colors. Each managed object maintains an atomic color bit (marked; line 22) in its 64-bit header alongside the root count; objects start *white* and become *black* once fully scanned. Every shared pointer between heap objects (*e.g.*, the next pointer of *a* in Fig. 4) also carries a color bit in its metadata (line 26), utilizing unused high-order bits of the pointer value. The fundamental invariant is that if a shared pointer is black, the object it references must be grey or black. Building upon this foundation, the marking of a managed object proceeds as follows: **(1)** by invoking shade_outgoings (§3), iterate over all outgoing shared pointer fields and shade each pointer black; and **(2)** set the

Algorithm 7 Internal CAS implementations of AtomicShared using barriers.

```
71: function internal_cas_unrooted(
        self: &AtomicShared<T>, curr: Tagged<T>,
        new: Tagged<T>, cs: &Guard)
    → Result<Tagged<T>, Tagged<T>>
72:   // Precond: curr.meta(), new.meta() = Unrooted(_)
73:   loop // Retry if only color changed.
74:     c_color ← curr.meta().color()
75:     if c_color = cs.black_color() then
76:       new.shade_target() // Dijkstra-style
77:     new ← new.with_meta(Unrooted(c_color))
78:     result ← self.link.cas(curr, new)
79:     match result
80:       case Ok(_) then
81:         fence(SeqCst) // Sync with line 62
82:         if EPOCH.load().phase() ≠ N then
83:           curr.shade_target() // Yuasa-style
84:       case Err(actual) then
85:         if actual.addr() = curr.addr() then
86:           curr ← actual; continue
87:     return result

88: function internal_cas_rooted(
        self: &AtomicShared<T>, curr: Tagged<T>,
        new: Tagged<T>, cs: &Guard)
    → Result<Tagged<T>, Tagged<T>>
89:   // Precond: curr.meta(), new.meta() = Rooted
90:   match self.link.cas(curr, new)
91:     case Ok(_) then
92:       if !curr.is_null()
              && (*curr).fetch_sub_rc() = 1 then
93:         fence(SeqCst) // Sync with line 62
94:         if EPOCH.load().phase() ≠ N then
95:           curr.shade_target() // Yuasa-style
96:       return Ok(curr)
97:     case Err(actual) then return Err(actual)
```

object's marked bit to the current black color. Shading a shared pointer black is implemented as a compare-and-swap (CAS) loop that repeatedly reads the pointer, pushes it onto the mark stack if unmarked, and atomically updates the color of the source field to black.

4.3 Phases of a Collection Cycle

We present the details of each phase, with Fig. 8 summarizing the key invariants of mutators.

Normal. The N phase represents a quiescent state. When the global epoch indicates N, all reachable objects are considered *white*. The collector performs its collection-work by reclaiming objects that were identified as garbage in the previous cycle. The collector periodically transitions to the next tracing cycle based on heuristics derived from heap statistics. To this end, our implementation follows the strategy proposed by MemBalancer [51].

Root tracing. The RT phase marks the beginning of a new tracing cycle. The collector performs its collection-work by **(1)** marking every object that has a non-zero root count,[8] and **(2)** draining all mark stacks to recursively mark all transitively reachable objects.

Meanwhile, mutators allocate new objects as *black* and, when mutating an AtomicShared pointer, cooperate with the collector through two classical barriers: **(1)** Dijkstra et al.'s insertion barrier: whenever a mutator overwrites a black shared pointer, the new pointer must also be black (*i.e.*, the target is grey or black). **(2)** Yuasa's deletion barrier: whenever a mutator destroys or overwrites a shared pointer, the old target object must be shaded. Algorithm 7 presents the internal compare-and-swap (CAS) implementations of AtomicShared for both the unrooted and rooted cases.[9]

In the unrooted case (line 71), if the current pointer is black (line 75), the new pointer is shaded, following Dijkstra et al.'s insertion barrier. After a successful CAS operation (line 80), it also shades the overwritten pointer if tracing is ongoing, following Yuasa's deletion barrier. Note that for

[8] This initial marking step is revisited in §5 to support scanning HP slots.
[9] Recall from §3 that Shared and AtomicShared are either rooted (*i.e.*, owning a root count) or unrooted.

deletion barriers, we read the latest global epoch using a memory fence (line 81) instead of reusing the local epoch, to ensure that no reachable objects are missed even when mutators in the N and RT phases coexist due to ragged phase transitions. If the CAS fails (line 84) but the address remains unchanged (line 85), then only the color has changed (*e.g.*, due to concurrent shading by the collector), so the operation is retried. Note that an unrooted pointer never be re-rooted (§3).

In the rooted case (line 88), the insertion barrier is omitted since rooted shared pointers do not maintain a color. As in the unrooted case, the deletion barrier is applied (line 93), but only when the deletion causes the root count to drop to 0.

Once the collector observes that all mark stacks are empty, it *optimistically* assumes that tracing is complete (*i.e.*, no grey objects remain) and terminates the current RT phase. However, because this check is not performed atomically (*i.e.*, not within a STW pause), some mark stacks may still contain pointers to scan. This possibility is safely handled in the subsequent CT phase.

Completion tracing. The CT phase safely terminates tracing without suspending mutators. Tracing must only conclude after all mark stacks are verified to be empty, simultaneously; premature termination may lead to use-after-free errors. Conventional collectors [26, 36, 83] enforce termination via a STW pause; our design instead leverages *local modification timestamps* (ms_modified).

Specifically, during this phase, each mutator atomically updates ms_modified with the current timestamp immediately before modifying its mark stack.[10] The collector, in turn, advances to the next N phase if all of the following conditions hold: **(1)** all ms_modified values differ from the current timestamp, **(2)** all mark stacks are empty, and **(3)** all ms_modified values remain unchanged upon rechecking.[11] Otherwise, the collector drains mark stacks again and re-enters the CT phase. This loop enables correct and lock-free termination of tracing.

4.4 Correctness

We establish three correctness properties for our GC design, which apply to both CPT and CDPT (§5.3). Full proofs appear in the appendix [5].

- **Soundness**: Our GC maintains the weak tricolor invariant throughout RT and CT phases; no live object is reclaimed. This holds because the hybrid barrier (Dijkstra et al.'s insertion + Yuasa's deletion) ensures that neither a new black-to-white edge nor the deletion of the last grey-to-white path can go undetected [21].
- **Termination**: Every tracing cycle completes in finite steps, provided so does each mutator's phase-critical section. This follows from monotonic color transitions (white → grey → black), a finite heap, and the CT validation loop's double-checked protocol.
- **Reclamation**: Any unreachable object is reclaimed within a finite number of GC cycles. Unreachable objects remain white and are swept at cycle end; floating garbage from Yuasa's barrier during cycle k is reclaimed in cycle k+1.

4.5 Implementation of Managed Pointers

Algorithm 8 presents the implementations of the managed-pointer types from Algorithm 3, following the design developed so far. Each function enforces that the caller is within an active phase-critical section by requiring a reference to the RAII Guard. The constructors of Shared[12] (line 101) and Local (line 104) take a user-provided object item, which will be allocated on the heap (*i.e.*, boxed). After creating the managed object with the appropriate allocation color (as specified

[10] Both ms_modified and the mark stacks are accessed within a phase-critical section.
[11] The collector's reads and the mutator's writes of the ms_modified flag are performed with a memory fence.
[12] AtomicShared can be constructed in the same way.

Algorithm 8 Implementation of CPT (§4) and CDPT (§5).

```
101: function Shared<T>::new(item: T, cs: &Guard) → Shared<T>
102:     ptr = Tagged::alloc_rooted(item, cs.alloc_color()); (*ptr).unroot_outgoings() // Boxing (§3)
103:     return Shared { inner: Atomic::new(ptr) }
104: function Local<'g, T>::new(item: T, cs: &'g Guard) → Local<'g, T>
105:     ptr = Tagged::alloc_unrooted(item, cs.alloc_color()).addr(); (*ptr).unroot_outgoings() // Boxing (§3)
106:     return Local { ptr, shield: None }
107: method AtomicShared<T>::load<'g>(cs: &'g Guard) → Local<'g, T>
108:     return Local { ptr: self.link.load().addr(), shield: None }
109: method Local<'g, T>::as_shared() → Shared<T> // Creating an RC-root
110:     if self.ptr.is_null() then return Shared::null()
111:     (*self.ptr).fetch_add_rc()
112:     return Shared { inner: Atomic::new(Tagged::new(self.ptr, Rooted)) }
113: method AtomicShared<T>::cas<'g>(curr: &Local<T>, new: &Local<T>, cs: &'g Guard)
     ↦ Result<Local<'g T>, Local<'g T>>
114:     old ← self.link.load()
115:     if old.addr() ≠ curr.ptr then return Err(Local { ptr: old.addr(), shield: None }) // Trivial failure #1.
116:     if old.meta() = Rooted then
117:         new_rooted ← new.as_shared()
118:         match self.internal_cas_rooted(old, new, cs)
119:             case Ok(_) then forget(new_rooted); return Ok(Local { ptr: old.addr(), shield: None })
120:             case Err(actual) then
121:                 if actual.meta() = Rooted then return Err(Local { ptr: actual.addr(), shield: None })
122:                 else old ← actual // Fallback to the unrooted case (line 124).
123:         if old.addr() ≠ curr.ptr then return Err(Local { ptr: old.addr(), shield: None }) // Trivial failure #2.
124:     match self.internal_cas_unrooted(old, new, cs)
125:         case Ok(_) then return Ok(Local { ptr: old.addr(), shield: None })
126:         case Err(actual) then return Err(Local { ptr: actual.addr(), shield: None })
127: method Local<'g, T>::protect(hd: &'h Handle) → Local<'h, T> // Creating an HP-root (§5)
128:     slot ← HPSlot::new(); slot.protect(self.ptr)
129:     return Local { ptr: self.ptr, shield: Some(slot) }
```

in Fig. 8) and initializing its root count (1 for Shared), the constructor invokes the user-defined unroot_outgoings method (§3) to unroot any outgoing child pointers.

AtomicShared provides a load method (line 107) that enables efficient object-graph traversal within a phase-critical section. This method simply performs an atomic load and returns a Local whose lifetime is tied to the Guard. The returned Local can be upgraded to an RC root via the as_shared method (line 109), becoming independent of the phase-critical section.

The compare-and-swap (CAS) operation (line 113) is implemented using the internal CAS procedures defined in Algorithm 7. It first loads the current pointer along with its metadata (line 114) to determine whether the AtomicShared is currently rooted. If it is rooted, the algorithm pre-increments the root count of the new pointer by creating a temporary Shared (line 117) and then attempts the CAS. If the CAS succeeds (line 119), the temporary Shared is *forgotten* to avoid running its destructor (which would otherwise decrement the root count), and the old pointer is returned. If the CAS fails (line 120), the algorithm inspects the cause: **(1)** If the location is still rooted but the pointer has changed (line 121), the operation fails and returns the actual current pointer. **(2)** If the location has become unrooted (line 122), the algorithm falls back to the unrooted CAS path (line 124). Other write operations, *e.g.*, store, can be implemented by looping this CAS.

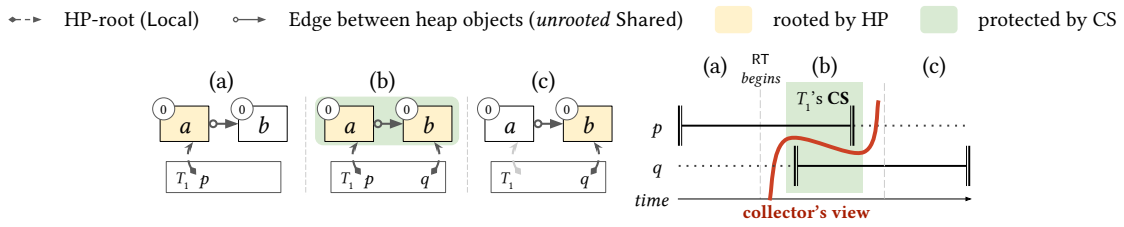


Fig. 9. List traversal using local roots (HP).

Fig. 10. Missing roots in Fig. 9.

5 Concurrent Deferred Partial Tracing with Local Root Optimization

Although CPT (§4) eliminates most RC updates during object-graph traversal, it still suffers from a bottleneck caused by high root mutation rates. We address this bottleneck with our complete solution, *concurrent deferred PT* (CDPT), which replaces RC-based local-root protection with HP. We present the key idea of integrating HP into our design (§5.1) and then detail the *phase barrier* mechanism required for correct HP scanning (§5.2).

5.1 Key Idea

The bottleneck stems from the frequent creation and destruction of local roots [6]: each such operation requires an atomic read-modify-write (RMW) on the target object's root count.[13] In read-intensive workloads, multiple threads frequently access the same popular objects, so their root count cache lines are constantly invalidated, saturating the memory bus (*e.g.*, Figs. 12e and 12f).

As discussed in §2.2, HP protects individual pointers using a per-mutator SWMR array at the cost of a single atomic store, making it well suited for local-root protection.

Modifications. The collector only needs to perform one additional step in the RT phase (§4.3): scan the mutators' HP slots alongside the existing RC scan. Mutators can now create roots using either HP or RC protection. HP-roots are ideal for mutator-local pointers, while RC-roots remain necessary for globally shared pointers. The rule that all new root creations must occur within a phase-critical section is unchanged (Algorithm 5). Destroying an HP-root does not trigger Yuasa's deletion barrier, unlike RC-root destruction. Algorithm 8 also illustrates a protect method for this purpose, highlighted in yellow. This method allocates a new HP slot and protects a pointer acquired within a critical section (line 128), effectively converting Local<'g, T> (whose lifetime is tied to the Guard) into Local<'h, T>, where 'h denotes the mutator's lifetime (line 35).

Notably, our design eliminates the memory fence required by the original HP technique after protection. This is because the *Release-Acquire* synchronization (§4.1) between the mutator's local epoch unpinning (line 48) and the collector's local epoch read (line 65) already ensures that all HP protections are visible to the collector once it observes the mutator's unpinning.

5.2 Phase Barrier

Problem. Because mutators may update their HP slots while the collector concurrently scans them, the collector's view can be inconsistent under a relaxed memory model, *missing root pointers*.

Fig. 9 illustrates an example scenario in which the collector may miss both pointers p and q: **(a)** the mutator (denoted as T_1) initially holds a Local pointer p (*i.e.*, HP-protected) to a; **(b)** it then follows the next link to create another Local pointer q to b; and **(c)** finally drops p. Fig. 10 illustrates a possible timeline in which the collector misses roots in the previous scenario. Suppose the RT

[13] RC destruction not only decrements the count, but also triggers Yuasa's deletion barrier if tracing is ongoing (line 95).

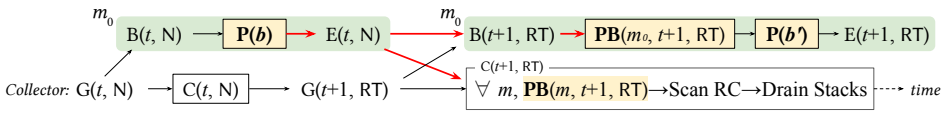


Fig. 11. Revised phase consensus diagram for the N and RT phases with the phase barrier.

Algorithm 9 Phase barrier (modified Algorithm 4).

```
131: function Guard::new() → Guard
132:     curr_e ← EPOCH.load(Acquire)
133:     loop { ... }
134:     if curr_e.phase() = RT
             && handle.last_obsrv ≠ curr_e then
135:         for hp ∈ handle.hps do shade(hp.load())
136:     handle.last_obsrv ← curr_e
```

```
137: struct Handle
138:     epoch: Atomic<Epoch>
139:     // SWMR HP slots
140:     hps: ConcurrentVec<Atomic<void*>>
141:     last_obsrv: Epoch
```

phase begins between (a) and (b), and the collector immediately starts scanning HP slots due to the absence of active critical sections. During (b), the mutator creates q (*i.e.*, writes to an HP slot) and drops p (*i.e.*, clears an HP slot). But the collector observes the *later* protection state for p and the *earlier* one for q. Specifically, the collector reads q's slot *before* the mutator writes to it and reads p's slot *after* the mutator clears it, thereby missing both roots and reclaiming the objects prematurely.

Solution. The primary cause is that the collector cannot *atomically* scan all HP slots of a single mutator. Conventionally, on-the-fly GCs [7, 30, 31] solve this by suspending one mutator at a time to scan its local roots. We draw inspiration from this strategy with a lock-free alternative: a *phase barrier* that shades local HP slots upon the first initiation of a phase-critical section in the RT phase. Specifically, when a mutator first enters a critical section under a new timestamp t in RT, it scans its own HP slots and shades all unmarked objects (*i.e.*, transitions white objects to grey by pushing them onto the local mark stack). This ensures that for each mutator, either itself or the collector scans its HP slots: for an *inactive* mutator that remains unpinned, the collector eventually scans its HP slots directly. For an *active* mutator, on the other hand, either the mutator or the collector is guaranteed to scan at least the pointers that were present at the end of its previous critical section.

Fig. 11 revisits the phase consensus diagram from Fig. 6, now augmented with phase barriers. In this diagram, an epoch e is represented as a pair consisting of a timestamp t and a phase identifier p; for example, we write B(t, RT) to denote the beginning of a critical section in the RT phase with timestamp t. The phase barrier for a mutator m during epoch (t, p) is denoted by PB(m, t, p). Using this notation, the HP-scanning work of the RT phase (§4.3) can be reformulated as executing a phase barrier for every mutator m. Suppose a mutator m_0 in the N phase protects a pointer b using HP and then ends its critical section. After advancing to the RT phase, the protection of b is guaranteed to be scanned before any subsequent mutations to its HP slots, denoted as P(b'). Revisiting the example in Fig. 9, the phase barrier in (b) shades p before acquiring q and eventually dropping p.

Algorithm. Algorithm 9 shows the modifications, highlighted in yellow. Each Handle stores its HP-protected local roots in a SWMR array (line 140) and tracks the last observed epoch (line 141). On entering a critical section, if the phase is RT and the epoch has not yet been observed (line 134), the mutator executes the phase barrier and updates the last observed epoch.

5.3 Correctness

The three properties from §4.4 hold for CDPT, as implied by the following lemma:

Lemma 5.1 (CDPT Preserves Correctness). *HP-based local roots and the phase barrier preserve the weak tricolor invariant.*

Proof sketch. HP acquisition (line 127) duplicates an existing heap edge into an HP slot without modifying the object graph; if the original heap edge is later removed, the Yuasa barrier on that deletion ensures the target remains protected. HP destruction (clearing an HP slot) is safe without Yuasa's deletion barrier because the target is either already shaded by the phase barrier on first RT entry (line 134), shaded by Yuasa's barrier on the CAS if the original heap edge was removed (lines 83 and 95), or it remains reachable via a persistent heap edge that the collector will trace. The full proof appears in the appendix [5].

6 Language-Agnostic Safety Rules

Our algorithm (§4 and 5) is language-agnostic and applies equally to Rust and C/C++. Its correctness is governed by four safety rules described below. In Rust, the ownership type system enforces these rules at compile time. In C/C++, they can be enforced through a combination of API design, compiler-assisted static analysis, and established coding conventions.

Rule 1: Thread-safety of user data. The user-defined type T must be safe to access from multiple threads (§3); if it contains mutable state, it must be protected by internal synchronization (*e.g.*, locks). This is necessary because the collector and mutators access objects concurrently.

In Rust, the Send + Sync trait bounds on T enforce this at compile time. In C/C++, mutable fields can be protected with std::mutex or std::atomic; Clang's thread safety analysis [44] further provides compile-time verification, serving as the closest C++ analog to Rust's Send + Sync bounds.

Rule 2: No unboxing of managed objects. A managed object must never be moved from the heap onto a mutator's stack. Moving it would bring its unrooted child pointers, whose root counts were decremented by unroot_outgoings (§3), out of the collector's reach, causing premature reclamation.

In Rust, Shared implements Deref returning &T (a non-owning immutable reference), physically preventing the move. In C/C++, the managed pointer API provides no method to extract the owned object. Moving from the dereferenced pointer, *e.g.*, std::move(*ptr), remains syntactically possible but constitutes a usage contract violation, as with other C++ conventions such as iterator validity. Alternatively, the C++ API could strictly prevent unboxing at compile time by returning a const T&. However, this introduces a structural trade-off: because standard synchronization primitives like std::atomic::store or std::mutex::lock are not const member functions, programmers would be forced to explicitly declare such fields with the mutable keyword, increasing the integration burden.

Rule 3: Pointer-extraction safety. unroot_outgoings and shade_outgoings must enumerate only fields from which a mutator cannot extract the pointer (*e.g.*, Shared, AtomicShared). Fields behind wrappers that grant mutable access to the pointer itself (*e.g.*, Mutex<Shared<T>>) must retain their root counts; otherwise, a mutator could extract the unrooted pointer to its stack.

In Rust, the procedural macro (§3) enforces this by enumerating only Shared and AtomicShared fields and skipping fields behind mutable wrappers. In C/C++, the user must manually implement unroot_outgoings and shade_outgoings; C++26 static reflection [19] could automate this enumeration in the future.

Rule 4: Scope-limited local pointers. Uncounted local pointers derived from a phase-critical section must not be used after the section ends, nor passed to another thread, unless first promoted to an RC-root (Shared) or protected by an HP (protect; line 127).

In Rust, the lifetime system enforces this: Local<'g, T> is tied to the lifetime 'g of the Guard, so the compiler rejects any use after the guard is dropped. In C/C++, RAII ensures cleanup, and making the local pointer type non-copyable prevents accidental stashing. The remaining gap, use-after-scope of derived references, is a known class of C++ bugs analogous to iterator invalidation. This can be effectively mitigated at compile time by annotating the API with the [[clang::lifetimebound]] attribute, anticipating the guarantees of broader C++ lifetime safety proposals [75].

7 Evaluation

We implement CPT and CDPT as a Rust library and evaluate them using a synthetic micro-benchmark suite of concurrent data structures (§7.1), showing that our library outperforms automatic GCs such as BDWGC and CIRC while achieving performance comparable to manual schemes like epoch-based RCU and HP, and a macro-benchmark based on a production cache library (§7.2), demonstrating that CDPT can be integrated with negligible overhead. Our implementation and benchmarks are available in the supplementary material [5].

All benchmarks were compiled with Rust nightly-2025-10-01 with default and link-time optimization. We used `jemalloc` [35] to reduce contention on the memory allocator. We conducted experiments on two dedicated machines: **AMD64T**: single-socket AMD EPYC 7543 (2.8GHz, 32 cores, 64 threads) with eight 32GiB DDR4 DRAMs (256GiB in total), and **INTEL96T**: dual-socket Intel Xeon Gold 6248R (3.0GHz, 48 cores, 96 threads) with twelve 32GiB DDR4 DRAMs (384GiB in total). The machines run Ubuntu 24.04 and Linux 6.8 with the default configuration. The results from the two machines exhibit similar trends, so we primarily focus on the AMD64T results here. For the full results, see the appendix [5].

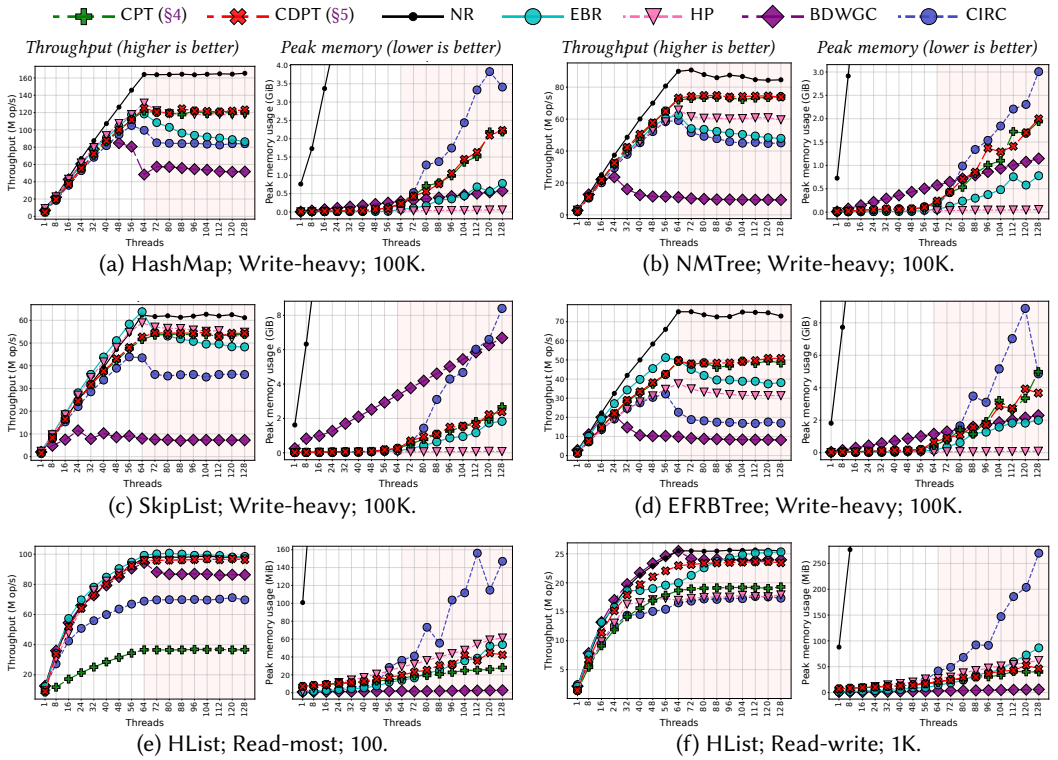
7.1 Micro-benchmark

The micro-benchmark compares the following reclamation schemes: **CPT**: our partial solution, *concurrent PT* (§4); **CDPT**: our full solution, *concurrent deferred PT* (§5); **NR**: the baseline that does not reclaim memory; **EBR**: epoch-based RCU [37, 39], as an efficient manual scheme; **HP**: hazard pointers with asymmetric fence optimization [29, 38] and immutability-based validation [54]; **BDWGC**: Boehm–Demers–Weiser collector (BDWGC) [14, 77]; and **CIRC**: EBR-based CIRC with immediate recursive destruction [47].

It evaluates the following lock-free map data structures, which we believe capture representative use cases of concurrent GC libraries. **HList**: Harris's linked list [39], representing workloads with long sequences of read-intensive operations; **HashMap**: a chaining hash table built on HList, representing workloads with many short link chains; **NMTree**: the Natarajan–Mittal tree [60], representing high-performance data structures; **SkipList**: a skip list [69], representing high-indegree DAG-structured data; and **EFRBTree**: Ellen's tree [33], representing cyclic graph-structured data.

We observe that the default BDWGC triggers collections far too frequently, which severely degrades performance. For a fair comparison with the other baselines, we configure BDWGC with a larger heap size comparable to theirs: at the start of each benchmark, we explicitly set the heap size to 2× (*maximum data structure size in bytes*) × (*number of threads*). Additionally, we disable generational and incremental collection, as enabling either mode consistently degraded both throughput and latency by roughly 2× across all workloads. This is because BDWGC's incremental marking (GC_collect_a_little_inner()) executes while holding the global allocation lock, serializing all concurrent allocators during collection.

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)** a varying number of threads: 1, 8, 16, 24, ⋯, 128 (up to twice the number of hardware



(a) HashMap; Write-heavy; 100K.
(b) NMTree; Write-heavy; 100K.
(c) SkipList; Write-heavy; 100K.
(d) EFRBTree; Write-heavy; 100K.
(e) HList; Read-most; 100.
(f) HList; Read-write; 1K.

Fig. 12. Throughput and peak memory usage for a varying number of threads.

Scheme	# RC updates
CDPT	0 (0/op)
CPT	267,743,133 (0.5/op)
CIRC	934,918,046 (2.1/op)

(a) Number of RC updates.

Scheme	P90	P99	P99.9	P99.99	Avg.
CDPT	1.7	2.8	29.8	80.6	1.2
CIRC	1.7	4.7	50.5	476	1.4
BDWGC	2.7	23.0	1,290	2,580	7.4

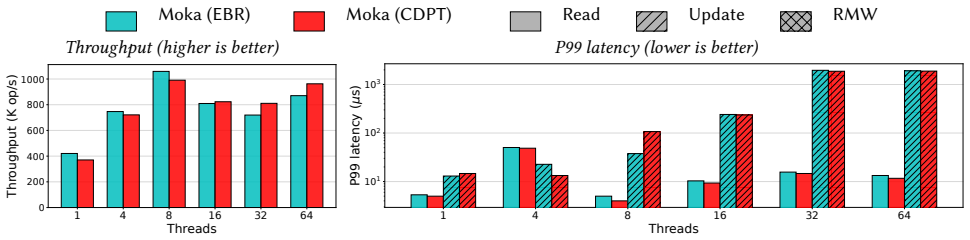
(b) Tail latencies (μs) of operations.

Fig. 13. RC update counts and operation tail latencies for SkipList (Write-heavy, 100K keys, 64 threads).

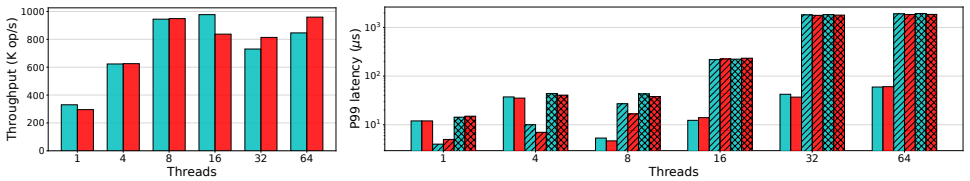
threads); **(2)** three types of workloads: **write-heavy** (50% inserts and 50% removes), **read-write** (50% reads and 50% writes), and **read-most** (90% reads and 10% writes); and **(3)** fixed time: 10 seconds. The key ranges for HList are 100 and 1K, and the key ranges for the others are 100K and 10M. The data structures are pre-filled to 50%.

Results. Across multiple workloads, we observe that CDPT **(1)** generally outperforms other automatic GCs (BDWGC and CIRC), being comparable to manual schemes (EBR and HP), while **(2)** preserving a moderate memory footprint comparable to CIRC.

Figures 12a to 12d present the performance results of various high-performance data structures, each with distinctive characteristics. As the number of mutators increases, BDWGC suffers from severe contention, performing 89% worse than EBR in the SkipList with 64 threads (Fig. 12c). CIRC alleviates this contention but still degrades significantly on complex data structures due to the



(a) Workload A (50% read, 50% update).



(b) Workload F (50% read, 50% RMW).

Fig. 14. Moka+YCSB macro-benchmark: throughput and per-operation P99 latency. Hatching distinguishes operation types; color distinguishes reclamation scheme.

overhead of RC updates and weak references. For example, compared with EBR, its performance drops by 32% in SkipList (Fig. 12c) and by 55% in EFRBTree (Fig. 12d) with 64 threads. This degradation arises not only from maintaining numerous references but also from the two-level deferral in its SMR backend: one for user data and another for counters. Building on this trend, CPT and CDPT incur moderate overheads on complex data structures because of tracing. For instance, each SkipList node maintains up to 32 next pointers, increasing the per-object scanning cost; as a result, CDPT performs 17% worse than EBR with 64 threads. Nevertheless, CPT and CDPT effectively mitigate the RC update overhead that dominates CIRC (Fig. 13a). CDPT also achieves stable and moderate tail latencies (Fig. 13b), unlike CIRC, whose RC updates and immediate recursive destruction [47] cause latency spikes, and BDWGC, which suffers from STW pauses. CPT and CDPT outperform CIRC on these complex data structures, indicating that PT-based garbage collection provides a more practical and general solution for various data structures.

For linked lists (Figs. 12e and 12f), which represent read-dominated workloads, CDPT achieves performance comparable to read-optimized schemes such as EBR and BDWGC, while CPT performs notably worse, *e.g.*, falling behind CIRC in Fig. 12e. This gap primarily stems from differences in their RC-protection strategies. All three schemes, CPT, CDPT, and CIRC, traverse data structures within a critical section; however, CPT and CDPT terminate the section after protecting the found node with RC or HP, whereas CIRC keeps it open across the caller's subsequent operations (*e.g.*, reading a value). This design entails a clear trade-off: while keeping the critical section open reduces read-modify-write (RMW) overhead during reads, it also delays the collector's progress. Conversely, CPT's shorter critical sections incur additional RMW operations, explaining its performance penalty on read-most workloads. CDPT breaks this trade-off through local root optimization: by protecting found nodes with HP instead of RC, it avoids the overhead while keeping critical sections short.

7.2 Macro-benchmark

To evaluate CDPT in a realistic application setting, we integrate it into Moka [28], a production-grade concurrent cache library for Rust. Moka internally relies on epoch-based RCU (crossbeam) [25] for safe memory reclamation of evicted cache entries. We replace this EBR backend with CDPT and

benchmark both variants using the Yahoo! Cloud Serving Benchmark (YCSB) [23]. The integration required modifying 1.3K lines out of Moka's 34K lines (under 4% of the codebase) and the result is a net reduction of 120 lines, as CDPT's automatic reclamation eliminates all manual defer_destroy calls, unsafe blocks for shared memory access, and the associated bookkeeping. The changes are almost entirely mechanical: replacing crossbeam pointer types with their CDPT equivalents (*e.g.*, Atomic<T> → AtomicShared<T>; Shared<'g, T> → Local<'g, T>), adding TraceObj trait implementations to two data types, and removing deferred-destruction code. No algorithmic changes were made to Moka's hash table or eviction logic, and all existing tests pass after the port.

Methodology. We measured throughput (operations per second), per-operation P99 latency, and peak memory usage across YCSB workloads A–D and F[14] for **(1)** a varying number of threads: 1, 4, 8, 16, 32, 64 (up to the number of hardware threads); and **(2)** fixed parameters: 2M records, a cache capacity of 200K entries (10%), Zipfian distribution (θ=0.99), and 2M operations per run.

Results. Figure 14 presents throughput and per-operation P99 latency for workloads A and F, the two most write-intensive workloads; workloads B–D (read-dominant) show no notable differences between the two variants.

Across all workloads (A–D and F), CDPT's throughput is comparable to EBR (geometric mean +0.26%; per-workload: A +0.1%, B +0.3%, C +1.8%, D −0.5%, F −0.4%). In Workload A (Fig. 14a), CDPT is 10.6% faster at 64 threads. In Workload F (Fig. 14b), the most write-intensive workload with RMW operations, the gain is even larger at high concurrency: CDPT is 13.4% faster at 64 threads. These improvements at high thread counts stem from CDPT's dedicated collector thread, which offloads reclamation work from mutators; in contrast, Moka's EBR backend requires each mutator to periodically perform reclamation, leading to increased contention.

P99 latencies in Workloads A and F are largely comparable between the two variants. In Workload A, read P99 remains stable across thread counts, while update P99 shows high variance at 32+ threads as both variants contend on cache eviction; neither consistently dominates. In Workload F, CDPT reduces RMW P99 latency (geometric mean −2.7%), as the dedicated collector reduces mutator-side reclamation pauses during multi-step RMW operations. Overall, P99 latencies are nearly identical (geometric mean −0.8%), with read P99 slightly improved under CDPT (−3.8%).

Peak memory under CDPT is approximately 1.5× higher (geometric mean +47%). This ratio is stable across thread counts: the geometric mean per thread count ranges from +44% at 1 thread to +49% at 64. The overhead stems from two sources: **(1)** CDPT wraps every allocated object with an 8-byte root count header (line 22), whereas EBR imposes no per-object metadata; and **(2)** under EBR an unlinked object is reclaimed once all concurrent readers advance past the current epoch, whereas CDPT must complete a full tracing cycle to confirm unreachability, so garbage accumulates longer. This is a cost of automatic tracing over manual reclamation, but in return CDPT eliminates all manual deferred-destruction and unsafe blocks required by EBR, as its type-safe API guarantees safe shared memory access.

8 Related and Future Work

Safe memory reclamation. Beyond the schemes discussed in §2.2, many SMR variants have been proposed: era-based interval tracking [61, 66, 81], RC-based reclamation [62, 63], optimistic version-based access [70], compiler-assisted local-root gathering [22], and epoch-based multiversion GC [8]. All require the programmer to explicitly retire unreachable objects and provide no automatic cycle collection. FreeAccess [22] uses a compiler plugin to transparently gather local roots; our approach achieves a similar effect at the library level through Rust's type system.

[14]Workload E (short-range scans) is excluded because Moka's hash-based cache does not support ordered key iteration.

Partial tracing. As discussed in §1, Bacon et al. defined partial tracing but deemed it impractical; even the standard GC textbook [46] omits the topic.[15] The only known implementation is the rust-gc library [68], which is thread-local: pointers and objects cannot be shared across threads. An experimental concurrent extension [76] aimed to support thread-safe pointers but was never completed; the effort has since been discontinued. We therefore do not include rust-gc as a baseline.

Techniques for reducing tracing costs. Many techniques reduce the managed heap size to limit tracing work, notably generational collection [55, 79] and escape analysis [2, 9, 20]. A complementary direction is *hybrid heaps* [15, 45, 52, 64, 73], which integrate GC-managed and manually managed regions. Snowflake [64] embeds a manually managed, SMR-governed region within .NET's managed runtime, enabling selective non-blocking reclamation alongside GC. Our work takes the opposite approach: embedding a managed heap within an unmanaged environment. This approach lets programmers choose per-resource strategies while the type system verifies safe usage at compile time.

Reference counting. As discussed in §2.3, modern concurrent DRC schemes, inspired by the classic zero-count table [27], eliminate STW pauses by integrating SMR: *deferred decrement schemes* such as Isolde [82] and CDRC [3, 4] buffer decrements until an SMR mechanism (*e.g.*, RCU [56, 57] or HP [58, 59]) confirms safety, while *deferred reclamation schemes* such as OrcGC [24] and CIRC [47] apply decrements immediately and use SMR to guard pending objects. All retain per-object reference counts and thus cannot automatically reclaim cycles; Jung et al. further observe that deferral can impede reclamation of linked structures. Perceus [67] takes a compiler-driven approach, using precise RC insertion and reuse analysis to achieve garbage-free execution in Koka, though it targets a primarily single-threaded, functional setting.

GC with finalizers in Rust. Alloy [43] is a conservative GC for Rust built on BDWGC that solves the long-standing challenge of safely reusing Rust destructors as GC finalizers. It introduces three compiler analyses implemented as modifications to rustc: *finalizer safety analysis* (rejecting unsafe destructors), *finalizer elision* (skipping redundant finalizers), and *premature finalizer prevention* (inserting fences to keep values alive). Alloy and our work are complementary: Alloy addresses safe finalization through compiler support atop a conservative, STW collector, whereas CDPT provides concurrent, precise tracing at the library level but does not yet support finalizers.

Future work. We plan to extend our GC library with *generational collection* and *concurrent compaction* to better support practical workloads. In particular, we aim to revisit generational collection under partial tracing [6], a combination that, to our knowledge, remains unexplored. For compaction, Alaska [80] shows that object movement in unmanaged languages can be enabled via *handles*, a form of double indirection that the compactor updates during defragmentation. This mechanism is structurally similar to our *hazard pointer* slots, where each slot acts as a mutable reference the collector may update. We believe this connection suggests a promising path toward a practical, relocatable GC for unmanaged languages.

References

[1] Santosh G. Abraham and Janak H. Patel. 1987. Parallel Garbage Collection on a Virtual Memory System. In *International Conference on Parallel Processing, ICPP'87, University Park, PA, USA, August 1987*. Pennsylvania State University Press, 243–246.

[2] Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. 2024. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. *Proc. ACM Program. Lang.* 8, PLDI, Article 159 (June 2024), 24 pages. https://doi.org/10.1145/3656389

[15] The textbook instead uses the term "partial tracing" to refer to the trial-deletion algorithm.

[3] 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

[4] 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

[5] Anonymous Author(s). 2025. Supplementary Material for the Submission.

[6] David F. Bacon, Perry Cheng, and V. T. Rajan. 2004. A Unified Theory of Garbage Collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Vancouver, BC, Canada) *(OOPSLA '04)*. Association for Computing Machinery, New York, NY, USA, 50–68. https://doi.org/10.1145/1028976.1028982

[7] Mordechai Ben-Ari. 1984. Algorithms for on-the-fly garbage collection. *ACM Trans. Program. Lang. Syst.* 6, 3 (July 1984), 333–344. https://doi.org/10.1145/579.587

[8] Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, Yihan Sun, and Yuanhao Wei. 2021. Space and Time Bounded Multiversion Garbage Collection. In *35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 12:1–12:20. https://doi.org/10.4230/LIPIcs.DISC.2021.12

[9] Bruno Blanchet. 2003. Escape analysis for JavaTM: Theory and practice. *ACM Trans. Program. Lang. Syst.* 25, 6 (Nov. 2003), 713–775. https://doi.org/10.1145/945885.945886

[10] Hans-J. Boehm. 1996. Simple garbage-collector-safety. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 89–98. https://doi.org/10.1145/231379.231394

[11] Hans-J. Boehm. n.d.. Advantages and Disadvantages of Conservative Garbage Collection. https://www.hboehm.info/gc/issues.html. Accessed: 2025-11-09.

[12] Hans-J. Boehm. n.d.. Conservative GC Algorithmic Overview. https://www.hboehm.info/gc/gcdescr.html. Accessed: 2025-11-09.

[13] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly parallel garbage collection. *SIGPLAN Not.* 26, 6 (May 1991), 157–164. https://doi.org/10.1145/113446.113459

[14] 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

[15] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. 2003. Ownership types for safe region-based memory management in real-time Java. *SIGPLAN Not.* 38, 5 (May 2003), 324–337. https://doi.org/10.1145/780822.781168

[16] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-Blocking Trees. *SIGPLAN Not.* 49, 8 (feb 2014), 329–342. https://doi.org/10.1145/2692916.2555267

[17] 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

[18] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. 21–28.

[19] Wyatt Childers, Peter Dimov, Dan Katz, Barry Revzin, Andrew Sutton, Faisal Vali, and Daveed Vandevoorde. 2025. Reflection for C++26. ISO/IEC JTC1/SC22/WG21 proposal P2996R12. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p2996r12.html

[20] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape analysis for Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, USA) *(OOPSLA '99)*. Association for Computing Machinery, New York, NY, USA, 1–19. https://doi.org/10.1145/320384.320386

[21] Austin Clements and Rick Hudson. 2016. *Proposal: Eliminate STW Stack Re-scanning*. Technical Report. Google Go Project. https://go.googlesource.com/proposal/+/master/design/17503-eliminate-rescan.md Last updated: 2016-10-21.

[22] Nachshon Cohen. 2018. Every Data Structure Deserves Lock-Free Memory Reclamation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 143 (oct 2018), 24 pages. https://doi.org/10.1145/3276513

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[24] 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

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

[26] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1029873.1029879

[27] 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

[28] Moka Developers. 2026. Moka: A high performance concurrent caching library for Rust. https://github.com/moka-rs/moka. v0.12.13.

[29] 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

[30] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966–975. https://doi.org/10.1145/359642.359655

[31] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966–975. https://doi.org/10.1145/359642.359655

[32] Dana Drachsler, Martin Vechev, and Eran Yahav. 2014. Practical Concurrent Binary Search Trees via Logical Ordering. *SIGPLAN Not.* 49, 8 (feb 2014), 343–356. https://doi.org/10.1145/2692916.2555269

[33] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Zurich, Switzerland) *(PODC '10)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1835698.1835736

[34] Toshio Endo and Kenjiro Taura. 2002. Reducing pause time of conservative collectors. In *Proceedings of the 3rd International Symposium on Memory Management* (Berlin, Germany) *(ISMM '02)*. Association for Computing Machinery, New York, NY, USA, 119–131. https://doi.org/10.1145/512429.512432

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

[36] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Association for Computing Machinery, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

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

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

[39] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

[40] Laurence Hellyer, Richard Jones, and Antony L. Hosking. 2010. The locality of concurrent write barriers. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 83–92. https://doi.org/10.1145/1806651.1806666

[41] Martin Hirzel and Amer Diwan. 2000. On the type accuracy of garbage collection. In *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA) *(ISMM '00)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/362422.362428

[42] Shane V. Howley and Jeremy Jones. 2012. A Non-Blocking Internal Binary Search Tree. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Pittsburgh, Pennsylvania, USA) *(SPAA '12)*. Association for Computing Machinery, New York, NY, USA, 161–171. https://doi.org/10.1145/2312005.2312036

[43] Jacob Hughes and Laurence Tratt. 2025. Garbage Collection for Rust: The Finalizer Frontier. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 401 (Oct. 2025), 27 pages. https://doi.org/10.1145/3763179

[44] DeLesley Hutchins, Aaron Ballman, and Dean Sutherland. 2014. C/C++ Thread Safety Analysis. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 41–46. https://doi.org/10.1109/SCAM.2014.34

[45] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATEC '02)*. USENIX Association, USA, 275–288.

[46] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (second ed.). CRC Press. https://doi.org/10.1201/9781003276142

[47] 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), 24 pages. https://doi.org/10.1145/3656383

[48] Jaehwang Jung, Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2023. Applying Hazard Pointers to More Concurrent Data Structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (Orlando, FL, USA) *(SPAA '23)*. Association for Computing Machinery, New York, NY, USA, 213–226. https://doi.org/10.1145/3558481.3591102

[49] Jeehoon Kang and Jaehwang Jung. 2020. A Marriage of Pointer- and Epoch-Based Reclamation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 314–328. https://doi.org/10.1145/3385412.3385978

[50] Jeonghyeon Kim, Jaehwang Jung, and Jeehoon Kang. 2024. Expediting Hazard Pointers with Bounded RCU Critical Sections. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures* (Nantes, France) *(SPAA '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3626183.3659941

[51] Marisa Kirisame, Pranav Shenoy, and Pavel Panchekha. 2022. Optimal heap limits for reducing browser memory use. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 160 (Oct. 2022), 21 pages. https://doi.org/10.1145/3563323

[52] Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 694–709. https://doi.org/10.1145/3582016.3582045

[53] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[54] Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2025. Leveraging Immutability to Validate Hazard Pointers for Optimistic Traversals. *Proc. ACM Program. Lang.* 9, PLDI, Article 148 (June 2025), 22 pages. https://doi.org/10.1145/3729247

[55] Henry Lieberman and Carl Hewitt. 1983. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM* 26, 6 (June 1983), 419–429. https://doi.org/10.1145/358141.358147

[56] Paul E. McKenney. 2004. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels.* Ph. D. Dissertation. OGI School of Science and Engineering at Oregon Health and Sciences University.

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

[58] Maged M. Michael. 2002. 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

[59] 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

[60] 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

[61] Ruslan Nikolaev and Binoy Ravindran. 2020. *Universal Wait-Free Memory Reclamation.* Association for Computing Machinery, New York, NY, USA, 130–143. https://doi.org/10.1145/3332466.3374540

[62] 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

[63] Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock- and Wait-Free Reclamation. *Proc. ACM Program. Lang.* 8, PLDI, Article 235 (June 2024), 25 pages. https://doi.org/10.1145/3658851

[64] 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

[65] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 16th International Conference on Distributed Computing and Networking* (Goa, India) *(ICDCN '15)*. Association for Computing Machinery, New York, NY, USA, Article 37, 10 pages. https://doi.org/10.1145/2684464.2684472

[66] Pedro Ramalhete and Andreia Correia. 2017. 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

[67] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. 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, 96–111. https://doi.org/10.1145/3453483.3454032

[68] rust-gc Developers. 2025. rust-gc. https://github.com/Manishearth/rust-gc accessed 12 Nov 2025.

[69] 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.

[70] Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *35th International Symposium on Distributed Computing (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 35:1–35:18. https://doi.org/10.4230/LIPIcs.DISC.2021.35

[71] Gali Sheffi and Erez Petrank. 2023. The ERA Theorem for Safe Memory Reclamation. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing* (Orlando, FL, USA) *(PODC '23)*. Association for Computing Machinery, New York, NY, USA, 102–112. https://doi.org/10.1145/3583668.3594564

[72] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: Neutralization Based 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, 175–190. https://doi.org/10.1145/3437801.3441625

[73] Codruț Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Safe and efficient hybrid memory management for Java. In *Proceedings of the 2015 International Symposium on Memory Management* (Portland, OR, USA) *(ISMM '15)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/2754169.2754185

[74] Guy L. Steele. 1975. Multiprocessing compactifying garbage collection. *Commun. ACM* 18, 9 (Sept. 1975), 495–508. https://doi.org/10.1145/361002.361005

[75] Herb Sutter. 2019. Lifetime Safety: Preventing Common Dangling. ISO/IEC JTC1/SC22/WG21 proposal P1179R1. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf

[76] rust-gc Developers. 2015. "Implement Cgc<T> – a concurrent threadsafe garbage collector" (pull request #6). GitHub pull request, https://github.com/Manishearth/rust-gc/pull/6. accessed 12 Nov 2025.

[77] The BDWGC Project. 2025. BDWGC. https://github.com/bdwgc/bdwgc

[78] 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

[79] David Ungar. 1984. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. Association for Computing Machinery, New York, NY, USA, 157–167. https://doi.org/10.1145/800020.808261

[80] Nick Wanninger, Tommy McMichen, Simone Campanoni, and Peter Dinda. 2024. Getting a Handle on Unmanaged Memory. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 448–463. https://doi.org/10.1145/3620666.3651326

[81] 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

[82] 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

[83] Albert Mingkun Yang and Tobias Wrigstad. 2022. Deep Dive into ZGC: A Modern Garbage Collector in OpenJDK. *ACM Trans. Program. Lang. Syst.* 44, 4, Article 22 (Sept. 2022), 34 pages. https://doi.org/10.1145/3538532

[84] Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software* 11, 3 (1990), 181–198. https://doi.org/10.1016/0164-1212(90)90084-Y

[85] 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 Correctness Proofs

We provide formal proofs of the safety and liveness properties stated in §4.4 for our GC design. The proofs cover CDPT, which subsumes CPT.

A.1 System Model and Definitions

We define the system state as a tuple $S = (\mathcal{O}, \mathcal{R}, \mathcal{E}, C, \mathcal{M})$, where:

- $\mathcal{O}$ is the finite set of heap objects.
- $\mathcal{R}$ is the set of roots, partitioned into $\mathcal{R}_{RC}$ (global/shared roots protected by RC) and $\mathcal{R}_{HP}$ (local roots protected by hazard pointers).
- $\mathcal{E}$ = (phase, color_bit, timestamp) represents the global epoch. The phases are $\{N, RT, CT\}$.
- $C : \mathcal{O} \times \mathcal{E} \to \{\text{White}, \text{Grey}, \text{Black}\}$ maps objects to their marking color relative to the current epoch.
- $\mathcal{M}$ is the union of all mark stacks (per-mutator local stacks and the global stack).

Definition A.1 (Phase-Relative Color). An object o is White in epoch $\mathcal{E}$ if o.mark_bit $\neq \mathcal{E}$.color_bit. Otherwise, it is Grey if $o \in \mathcal{M}$ (present in some mark stack), or Black if $o \notin \mathcal{M}$ (already processed).

Assumption 1 (Memory Consistency). *The collector's epoch update is a Release operation, and mutator epoch loads are Acquire operations (lines 42 and 44). The deletion barrier employs a SeqCst fence (lines 81 and 93) to synchronize with the collector's SeqCst fence (line 62).*

Assumption 2 (Weak Fairness). *The collector thread is weakly fair: if the collector has work to do (tracing or sweeping), it will eventually be scheduled to execute.*

Definition A.2 (Grey-Protected). An object $o \in \mathcal{O}$ is *grey-protected* if it is reachable from a Grey object via a path consisting exclusively of White nodes.

Definition A.3 (Weak Tricolor Invariant $\mathcal{I}_{weak}$). For any two objects $a, b \in \mathcal{O}$ such that a points to b: if $C(a)$ = Black, then b must be either Grey, Black, or White and grey-protected.

A.2 Helper Lemmas

To reason about the concurrent interleaving of mutators and the collector, we establish three helper lemmas regarding synchronization and visibility.

Lemma A.4 (Global Visibility / Safety of Ragged Transitions). *Let M be a mutator with local epoch $e_M = N$. If the global epoch $\mathcal{E}$ transitions to RT, the deletion barrier in M observes RT before performing any heap deletion.*

Proof. The deletion barrier in AtomicShared::cas executes a SeqCst fence (line 81) followed by a load of the global epoch variable. The SeqCst fence prevents local reordering of the deletion store before the epoch load, while the Release/Acquire synchronization (per assumption 1) between the collector's epoch increment (line 61) and the mutator's epoch load ensures that a mutator observing the new epoch must also observe all prior collector writes establishing RT. Thus, the mutator reliably observes RT and shades the target, preventing the hiding of objects during the phase lag. □

Lemma A.5 (Phase Barrier Snapshot). *For any mutator M, immediately after M executes the phase barrier (line 134) upon its first entry into a critical section in the RT phase, all pre-existing objects in $\mathcal{R}_{HP}^{M}$ are colored Grey or Black.*

Proof. The phase barrier (Algorithm 9) mandates that upon observing the transition to RT locally, M iterates its existing HP slots and shades all unmarked objects. This establishes a snapshot-at-the-beginning property for pre-existing local roots at the moment of acknowledgement. The

condition at line 134 ensures this scan occurs exactly once per epoch, and the Release ordering at line 48 ensures that the collector observes the completed scan when reading the mutator's epoch at line 65. Roots acquired *after* this barrier are handled by Theorem A.7 (Step 3). □

Lemma A.6 (Correctness of CT Validation). *If the collector successfully transitions from $CT \to N$, then the system is globally quiescent ($\mathcal{M} = \emptyset$).*

Proof. The validation relies on a double-checked protocol using per-mutator modification timestamps (ms_modified), as described in §4.3. Before modifying its mark stack, each mutator atomically updates its ms_modified to the current timestamp. The collector performs the following three-step check: **(1)** records all mutators' ms_modified values and verifies none equals the current timestamp, **(2)** verifies all mark stacks are empty, and **(3)** re-checks the ms_modified values and confirms they have not changed since step (1). If all three conditions hold, then no mutator could have pushed a Grey object onto its mark stack between steps (1) and (3) without updating its ms_modified, which would be detected in step (3). The SeqCst fences (lines 43 and 62) ensure that mark-stack modifications are visible to the collector during this check. Since $\mathcal{M} = \emptyset$ and no concurrent modification went undetected, the system is quiescent. □

A.3 Safety

Theorem A.7 (Soundness; §4.4). *CDPT maintains the weak tricolor invariant $\mathcal{I}_{weak}$ throughout the RT and CT phases. Consequently, no live object is reclaimed.*

Proof. We prove this by structural induction on the set of possible operations that modify the object graph or root set. Assume $\mathcal{I}_{\text{weak}}$ holds at time t. We show it holds at $t + 1$.

Base Case ($N \to RT$). The definition of White flips based on the new epoch's color bit (Fig. 8). All objects are White relative to the new epoch; hence no Black → White edge can exist. $\mathcal{I}_{\text{weak}}$ holds trivially.

Step 1: Mutator Insertion (Dijkstra Barrier). A mutator writes pointer p into a Black object o via AtomicShared::cas. The insertion barrier (line 75) checks whether the container's color equals the current black color and, if so, shades p Grey. The edge becomes Black → Grey, satisfying $\mathcal{I}_{\text{weak}}$.

Step 2: Mutator Deletion (Yuasa Barrier). A mutator removes edge $o \to p$ via AtomicShared::cas. The deletion barrier (line 95) shades the old target p Grey. If $o \to p$ was the sole path from a Grey ancestor to White object p, p becomes a Grey root, preserving reachability. By Lemma A.4, this holds even when the mutator is still in the N phase locally (ragged transition): the SeqCst fence at line 81 ensures the mutator observes the RT epoch before any deletion takes effect.

Step 3: Local Root Acquisition (HP Protection). A mutator reads pointer p from heap object o into a local HP slot (line 127).

- *Edge duplication:* This operation does not create a new edge in the heap graph; it duplicates an existing edge $o \to p$ into the root set $\mathcal{R}_{HP}$.
- Since $\mathcal{I}_{\text{weak}}$ holds for the existing edge $o \to p$, the target p is already protected (either Grey, Black, or grey-protected via o).
- If the original heap edge $o \to p$ is subsequently removed by another mutator, the Yuasa deletion barrier on that CAS (lines 71 and 88) shades p Grey.
- Therefore, creating a local root reference to an already-protected object cannot violate the invariant.

Step 4: Local Root Destruction (HP Clearing). A mutator clears an HP slot, removing a local root to object p. Unlike RC root destruction, this does *not* trigger a Yuasa deletion barrier. We show this is safe by case analysis:

(a) *HP established before RT:* The phase barrier (line 134) shaded p Grey or Black on the mutator's first *RT*-phase entry, per Lemma A.5. Since colors are monotonic (White → Grey → Black), p remains non-White when the HP is cleared.
(b) *HP established during RT/CT:* The pointer p was loaded from a heap edge within a phase-critical section. If that heap edge is later removed by another mutator, the Yuasa barrier on that CAS (lines 71 and 88) already shades p Grey. If the heap edge persists, the collector will trace through it. In both cases, p is Grey or Black when the HP is cleared.

In either case, removing the HP root does not create a new Black → White edge and thus preserves $\mathcal{I}_{\text{weak}}$.

Step 5: Allocation. Newly allocated objects are colored Black relative to the current epoch (Fig. 8). The child pointers of the new object originate from Shared or Local values held by the allocating mutator; these are either RC-roots (root count > 0) or CS-protected references. By the inductive hypothesis ($\mathcal{I}_{\text{weak}}$ holds at time t), each such target w is either: **(1)** already Grey or Black (if it was previously shaded), or **(2)** White but grey-protected (reachable from some Grey object via White nodes). After allocation, unroot_outgoings (lines 102 and 105) decrements the root counts of these child pointers, but does not modify the heap graph topology. The edges from the new Black object to its children therefore satisfy $\mathcal{I}_{\text{weak}}$: each target is Grey, Black, or grey-protected. Thus $\mathcal{I}_{\text{weak}}$ is preserved.

Conclusion. All operations preserve $\mathcal{I}_{\text{weak}}$. Reclamation only occurs for White objects after the cycle transitions to N (Fig. 8), at which point no Black → White edges exist in the heap. Therefore, no live (reachable) object is reclaimed. □

A.4 Liveness

Theorem A.8 (Termination; §4.4). *The tracing cycle (RT ∪ CT) completes in a finite number of steps, provided each mutator's phase-critical section terminates in finite time.*

Proof. We show termination in two phases: (1) the set of White objects eventually becomes empty, after which (2) all Grey objects are drained in finite time.

(1) Monotonicity: Color transitions are monotonic (White → Grey → Black). Once an object's mark bit is set, it cannot revert to White within the same cycle.
(2) Finite State: The set O is finite. Let $W = |\{o \in O : C(o) = \text{White}\}|$.
(3) White is monotonically non-increasing: Every shading operation (by the collector or by a mutator's barrier) transitions an object from White to Grey, reducing W by one. No operation increases W. Since $W \geq 0$ and is bounded, W reaches zero in finite steps.
(4) Grey is eventually drained: Once $W = 0$, no new Grey objects can be created (there are no White objects left to shade). The collector processes Grey objects from mark stacks, turning them Black. By assumption 2, the collector eventually processes all remaining Grey objects, yielding $|\text{Grey}| = 0$.
(5) Fairness: By assumption 2, the collector eventually executes marking steps. Phase consensus (Algorithm 6) ensures the collector can advance phases once all pinned mutators have acknowledged the current epoch.
(6) Valid Termination: By Lemma A.6, the cycle only ends (transitions from CT to N) when globally quiescent: all mark stacks are empty and no Grey objects remain.

Therefore, both W and |Grey| reach zero in finite steps, and the collector correctly detects this via CT validation. □

Theorem A.9 (Reclamation; §4.4). *Any object $o \in O$ that becomes unreachable will be reclaimed within a finite number of GC cycles.*

Proof. Consider an object o that is unreachable at the start of cycle k.

(1) Inaccessibility: No mutator can obtain a reference to o (it is not in $\mathcal{R}$ and no reachable object points to it), so no barrier is triggered on o's behalf.
(2) Isolation: No root or reachable object points to o; the collector never encounters o during tracing, so o is never shaded.
(3) Reclamation: By Theorem A.8, cycle k terminates. o remains White and is reclaimed during the subsequent sweep in the N phase.

Objects becoming unreachable *during* cycle k may survive one cycle due to the Yuasa barrier (floating garbage): the barrier may shade an object Grey even as the last true reference is deleted. Such objects are not shaded in cycle $k+1$ and are reclaimed at its end, within at most two cycles. □

B AMD64T Full Experimental Results

B.1 Small Key Ranges (100 for Lists and 100K for Others)

B.1.1 Write-heavy Workloads.

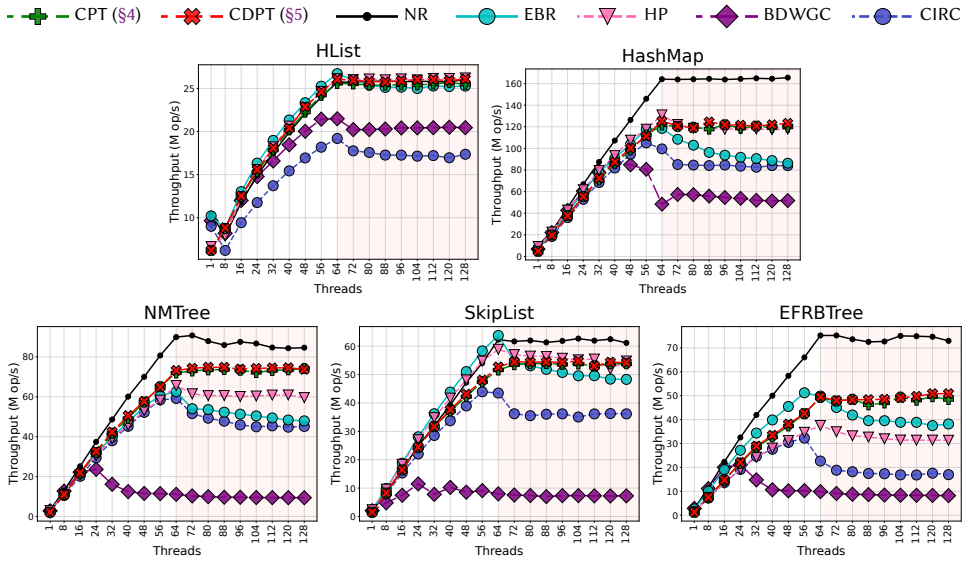


Fig. 15. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with small key ranges.

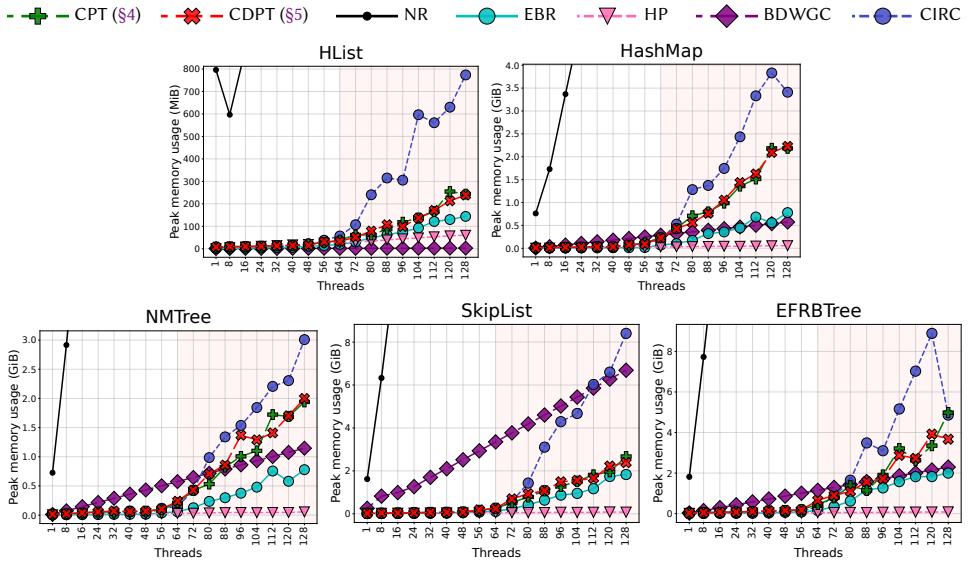


Fig. 16. Peak memory usage of write-heavy workloads for a varying number of threads with small key ranges.

B.1.2 Read-write Workloads.

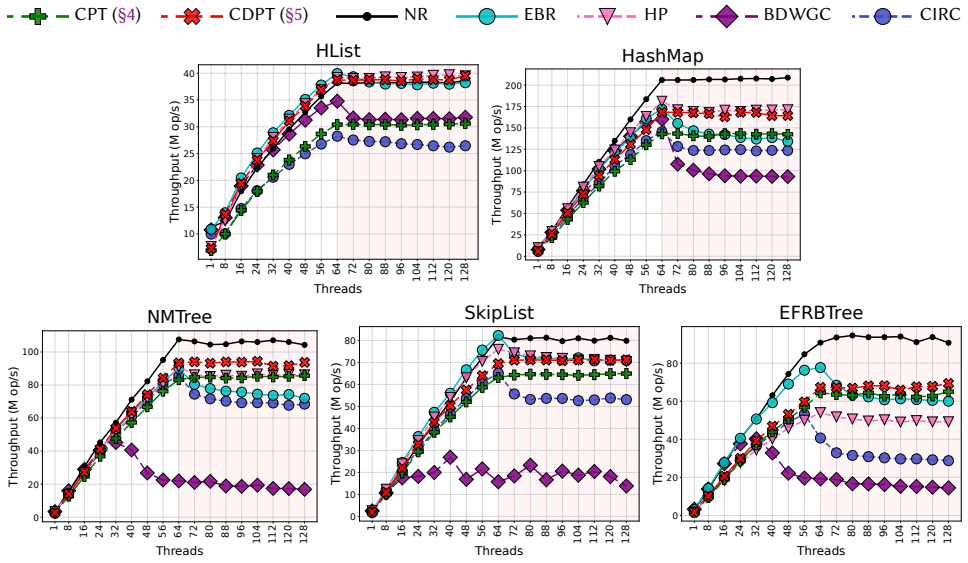


Fig. 17. Throughput (million operations per second) of read-write workloads for a varying number of threads with small key ranges.

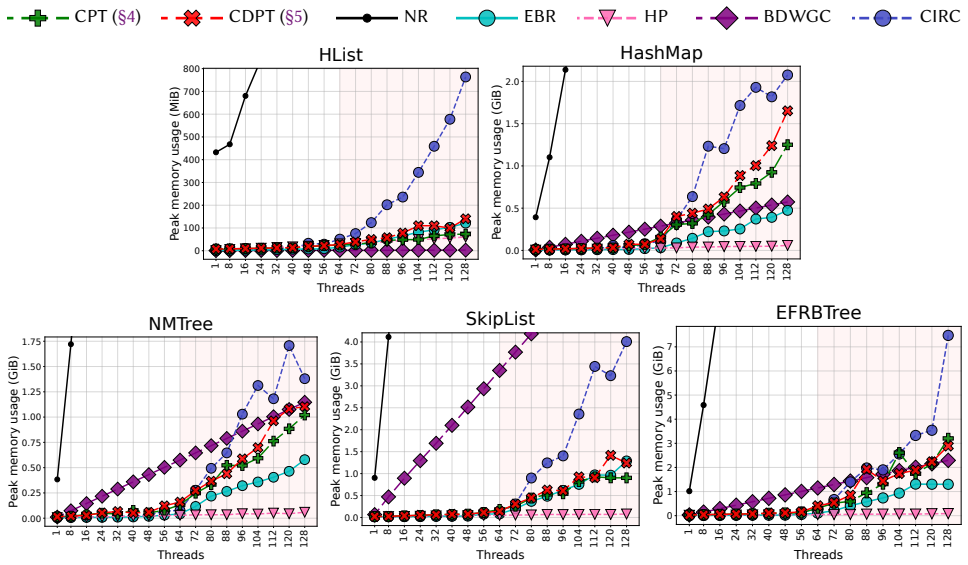


Fig. 18. Peak memory usage of read-write workloads for a varying number of threads with small key ranges.

B.1.3 Read-most Workloads.

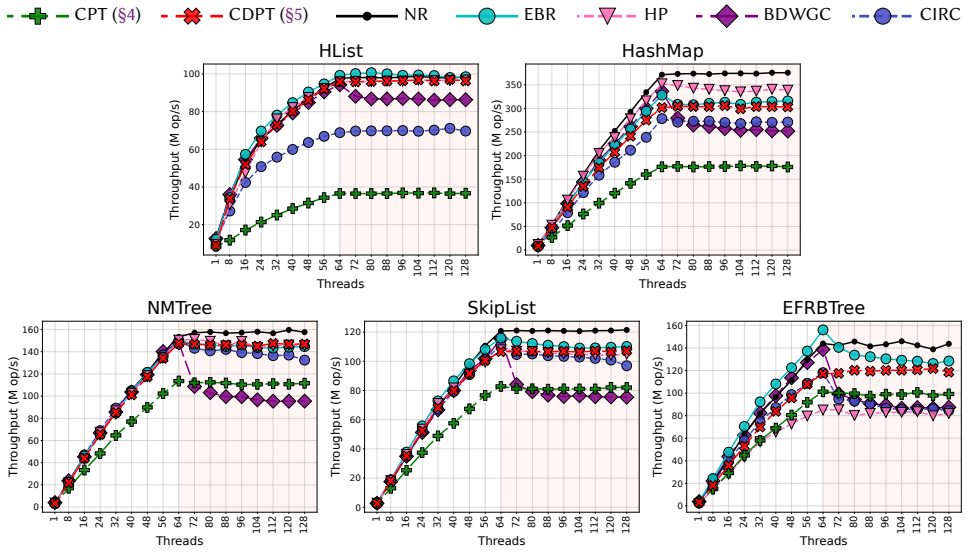


Fig. 19. Throughput (million operations per second) of read-most workloads for a varying number of threads with small key ranges.

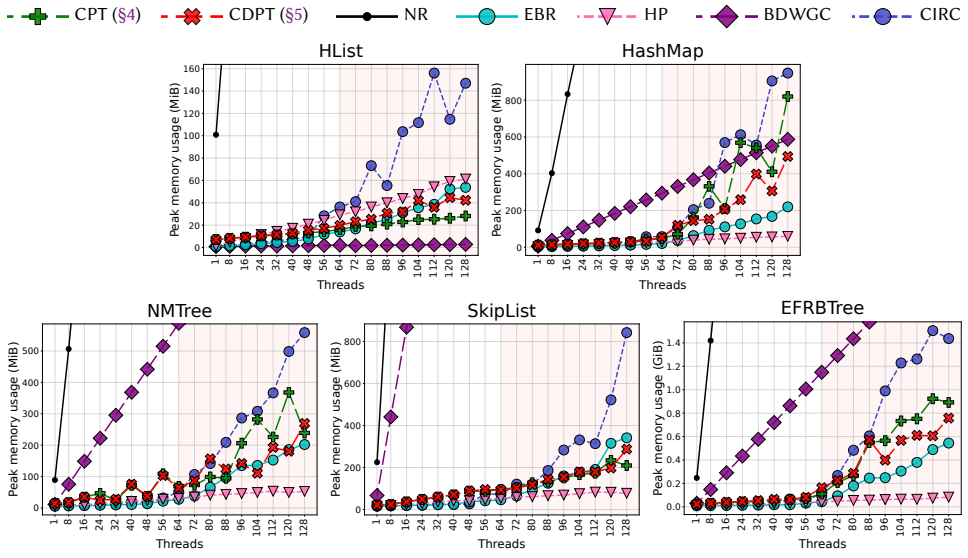


Fig. 20. Peak memory usage of read-most workloads for a varying number of threads with small key ranges.

B.2 Large Key Ranges (1K for Lists and 10M for Others)

B.2.1 Write-heavy Workloads.

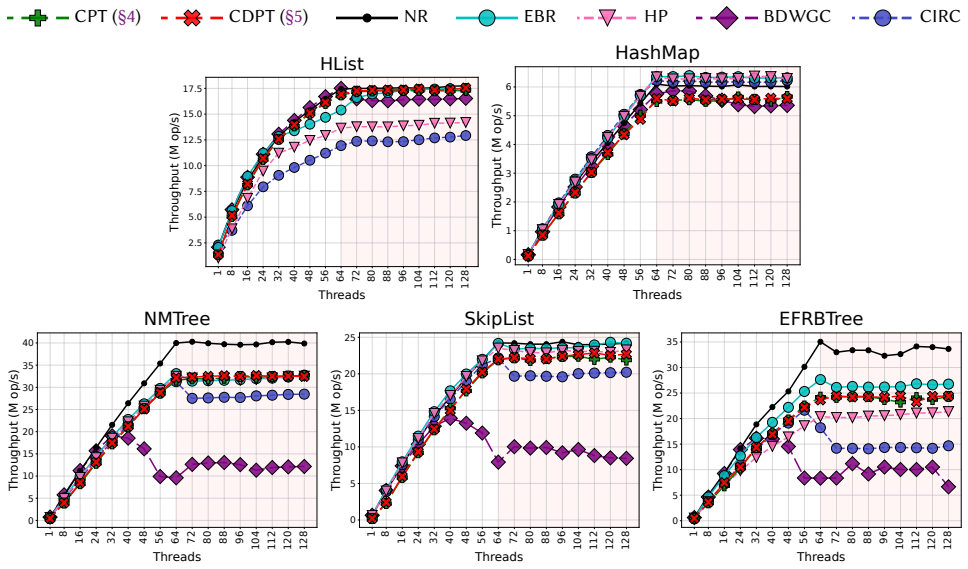


Fig. 21. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with large key ranges.

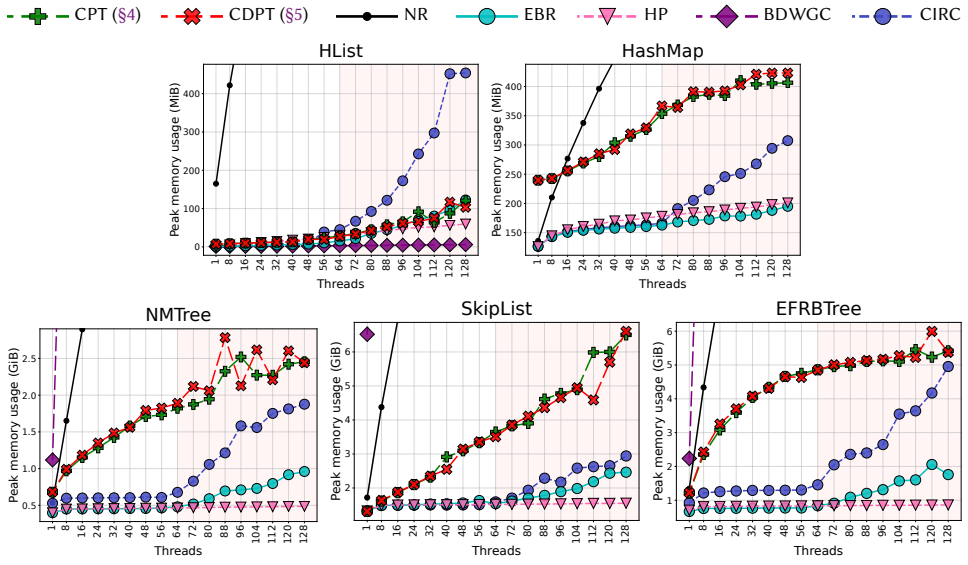


Fig. 22. Peak memory usage of write-heavy workloads for a varying number of threads with large key ranges.

B.2.2 Read-write Workloads.

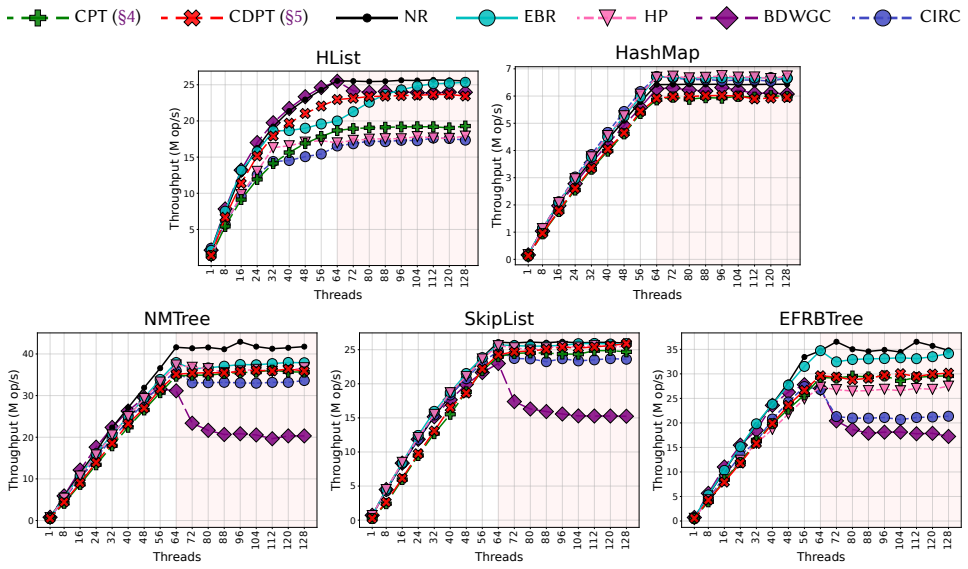


Fig. 23. Throughput (million operations per second) of read-write workloads for a varying number of threads with large key ranges.

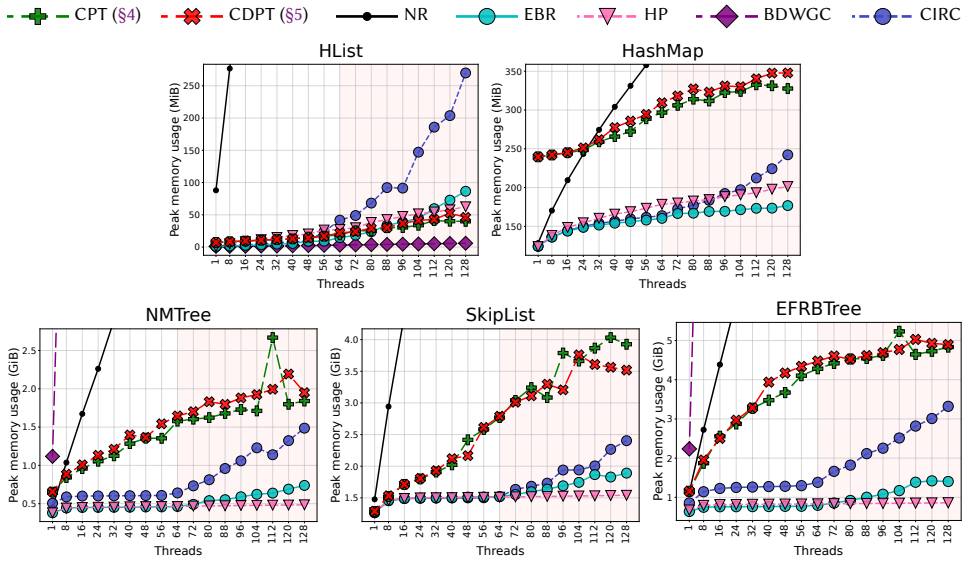


Fig. 24. Peak memory usage of read-write workloads for a varying number of threads with large key ranges.

B.2.3 Read-most Workloads.

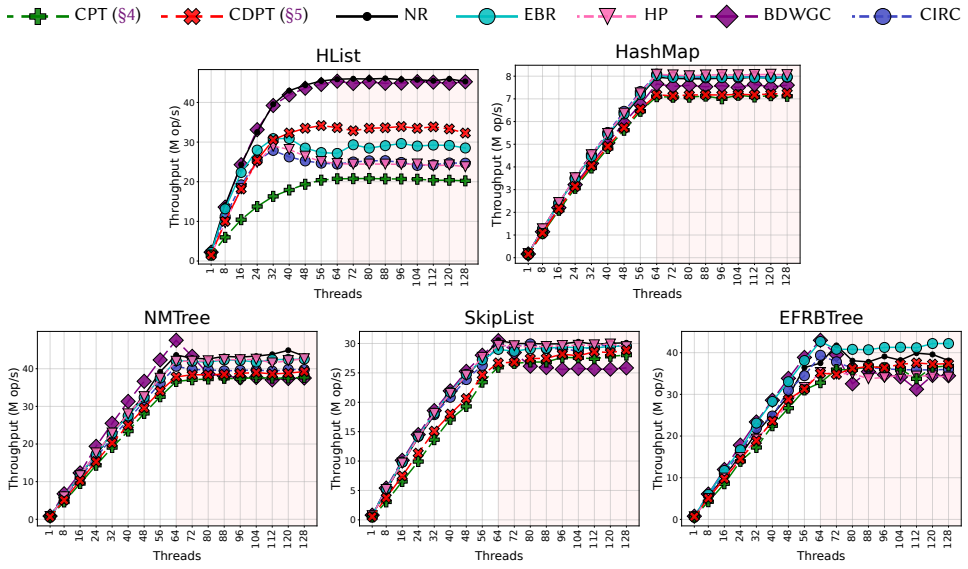


Fig. 25. Throughput (million operations per second) of read-most workloads for a varying number of threads with large key ranges.

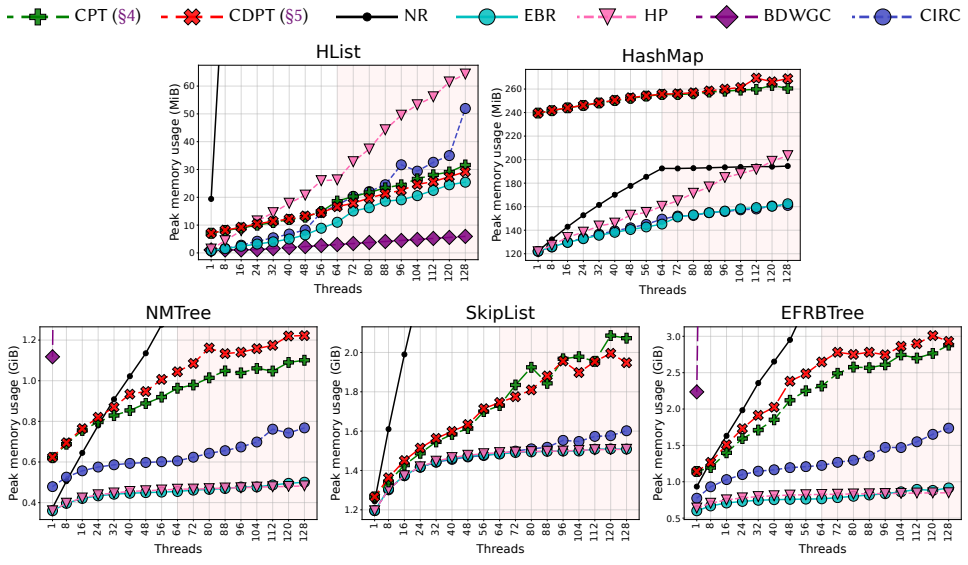
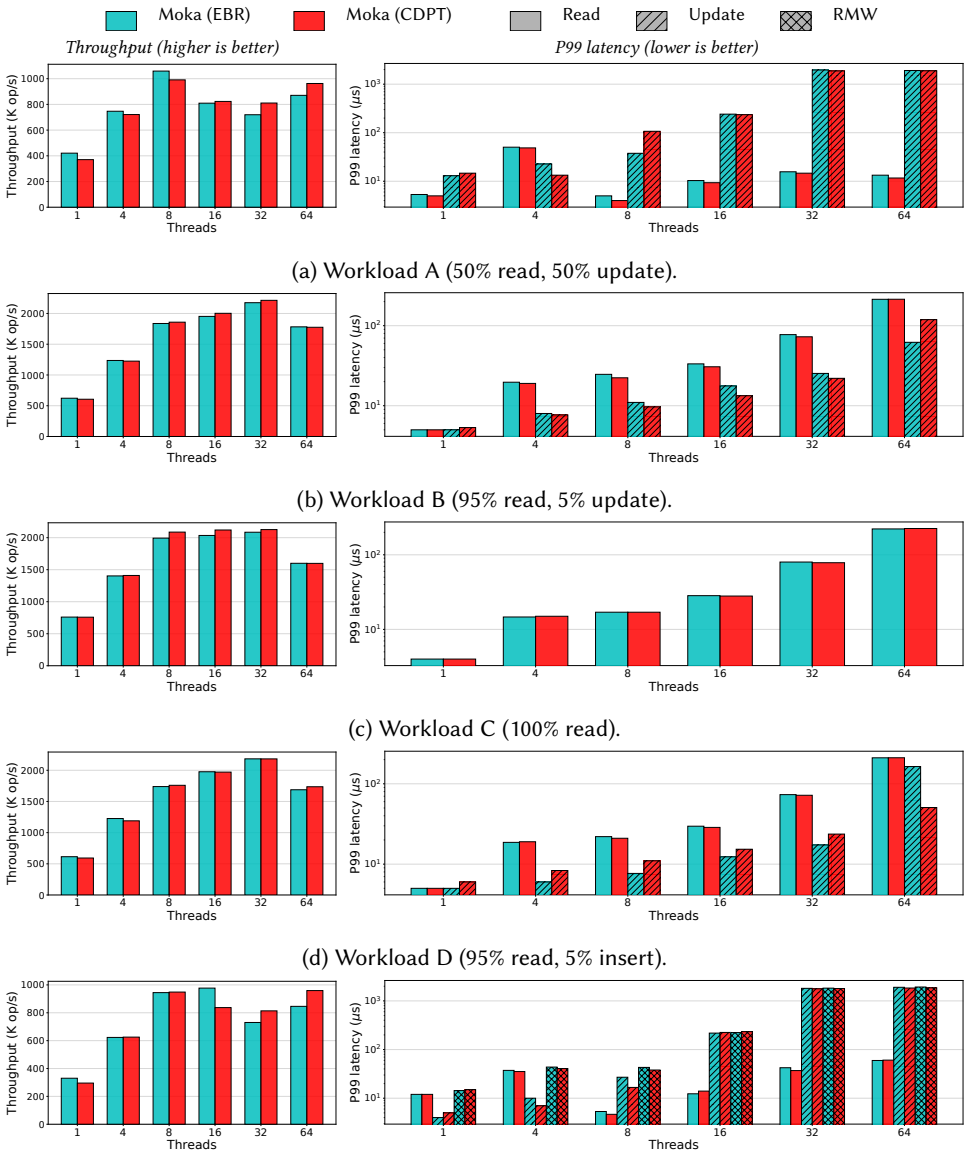


Fig. 26. Peak memory usage of read-most workloads for a varying number of threads with large key ranges.

B.3 Macro-benchmark



(a) Workload A (50% read, 50% update).

(b) Workload B (95% read, 5% update).

(c) Workload C (100% read).

(d) Workload D (95% read, 5% insert).

(e) Workload F (50% read, 50% RMW).

Fig. 27. Moka+YCSB macro-benchmark: throughput and per-operation P99 latency. Hatching distinguishes operation types; color distinguishes reclamation scheme.

C INTEL96T Full Experimental Results

C.1 Small Key Ranges (100 for Lists and 100K for Others)

C.1.1 Write-heavy Workloads.

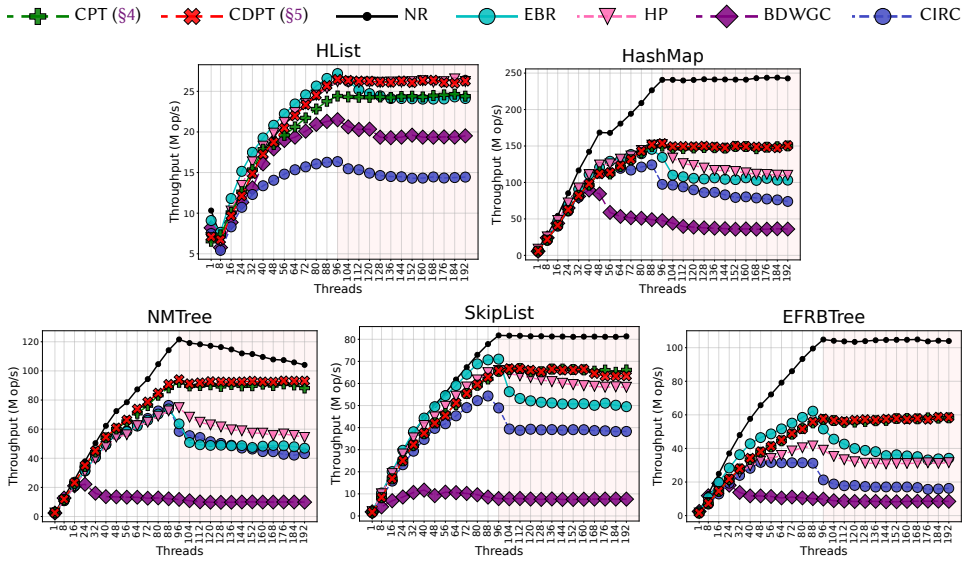


Fig. 28. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with small key ranges.

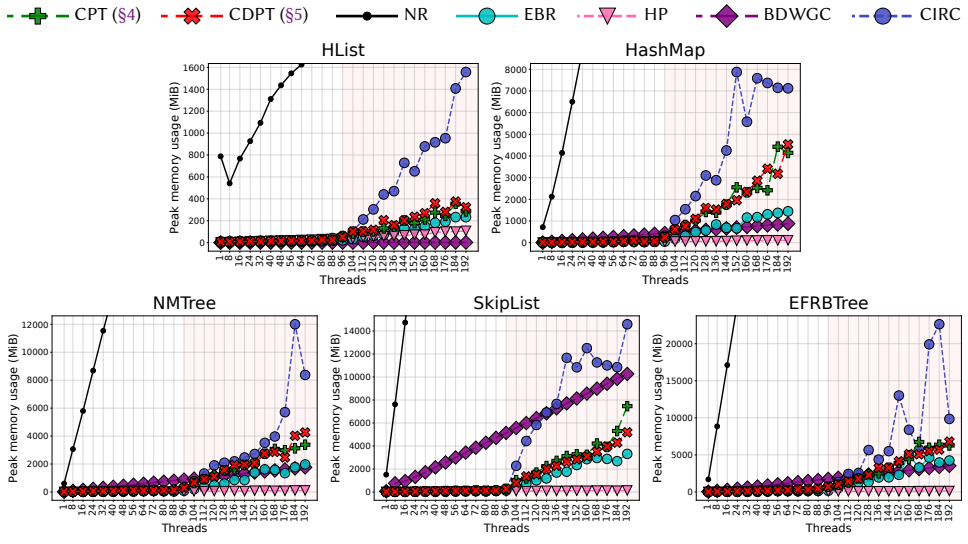


Fig. 29. Peak memory usage of write-heavy workloads for a varying number of threads with small key ranges.

C.1.2 *Read-write Workloads.*

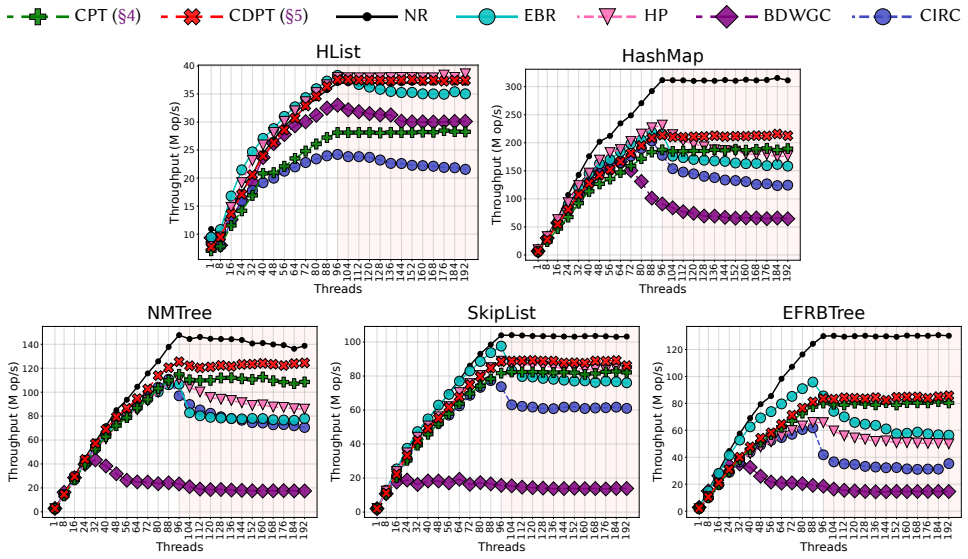


Fig. 30. Throughput (million operations per second) of read-write workloads for a varying number of threads with small key ranges.

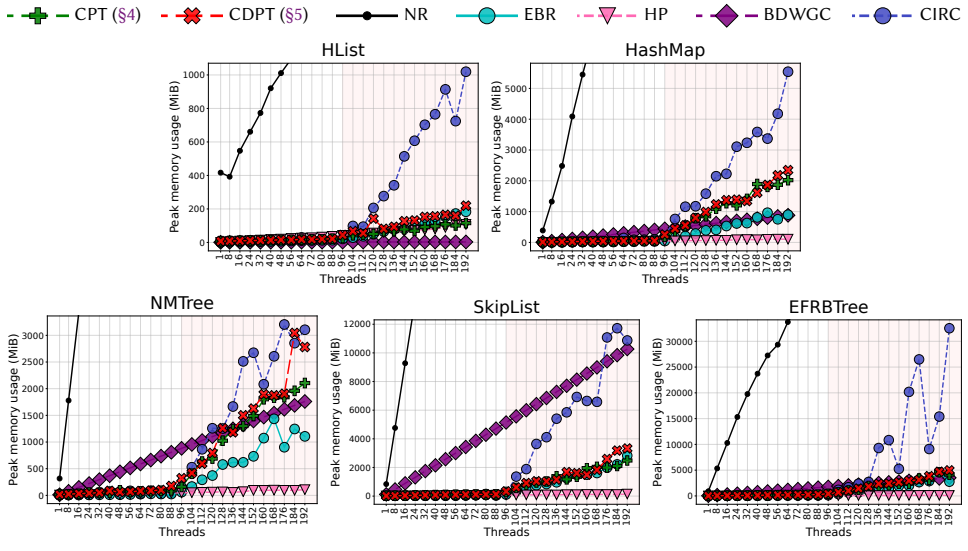


Fig. 31. Peak memory usage of read-write workloads for a varying number of threads with small key ranges.

C.1.3 Read-most Workloads.

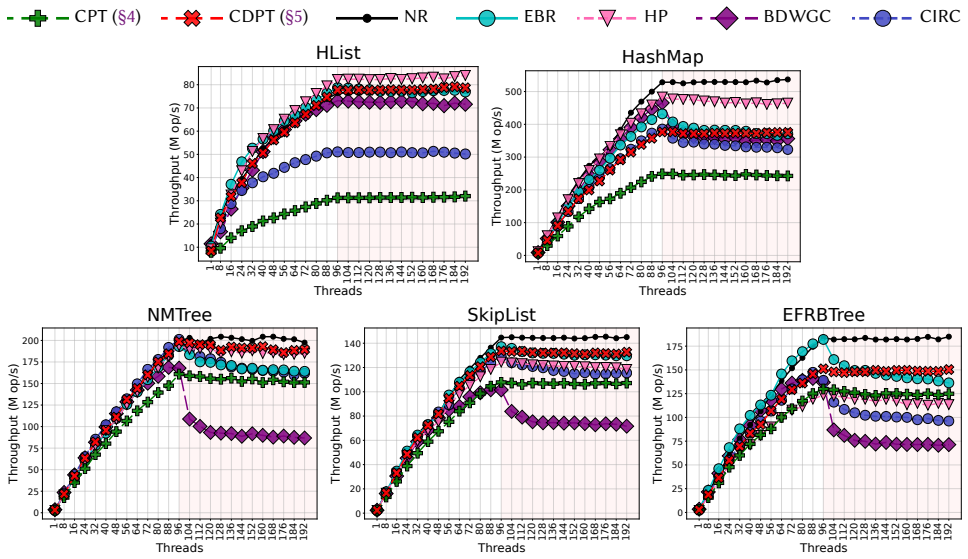


Fig. 32. Throughput (million operations per second) of read-most workloads for a varying number of threads with small key ranges.

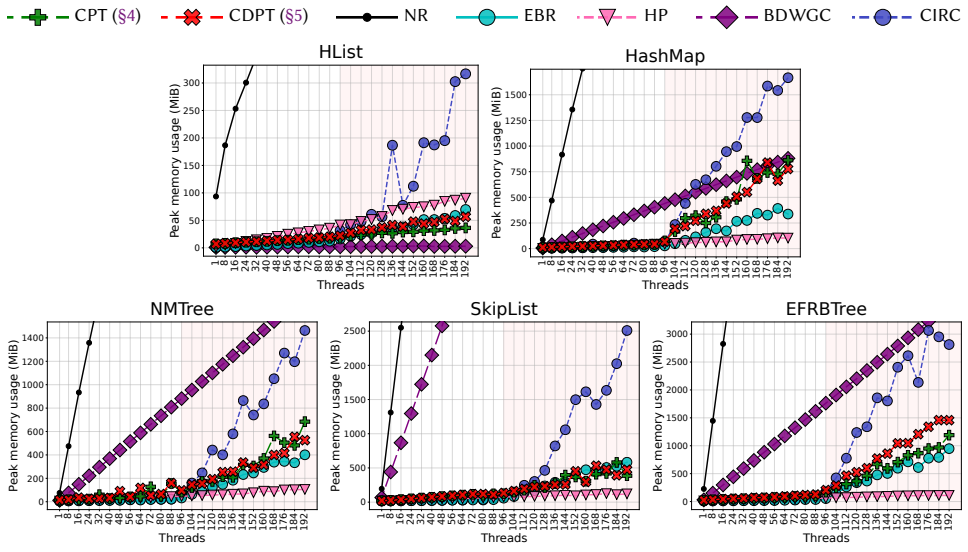


Fig. 33. Peak memory usage of read-most workloads for a varying number of threads with small key ranges.

C.2 Large Key Ranges (1K for Lists and 10M for Others)

C.2.1 Write-heavy Workloads.

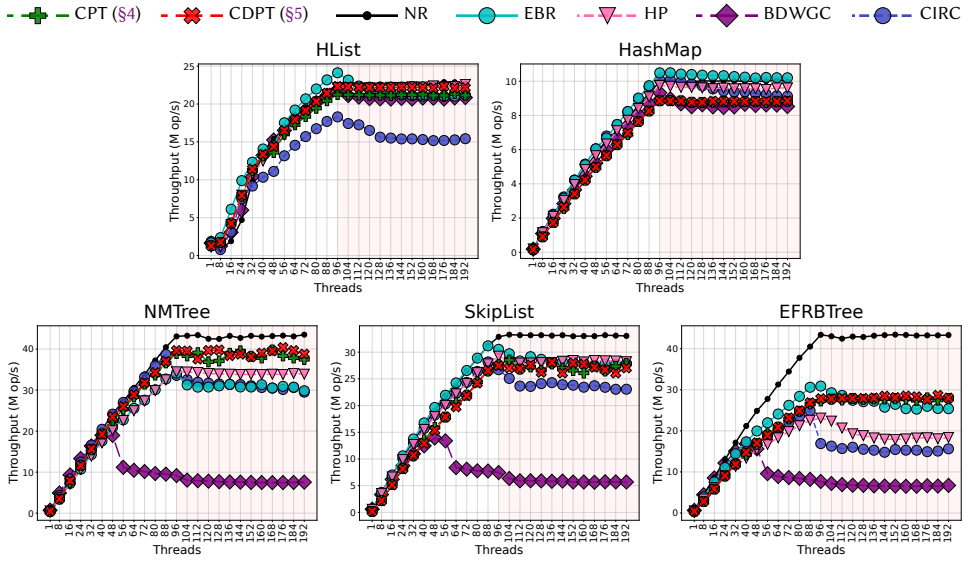


Fig. 34. Throughput (million operations per second) of write-heavy workloads for a varying number of threads with large key ranges.

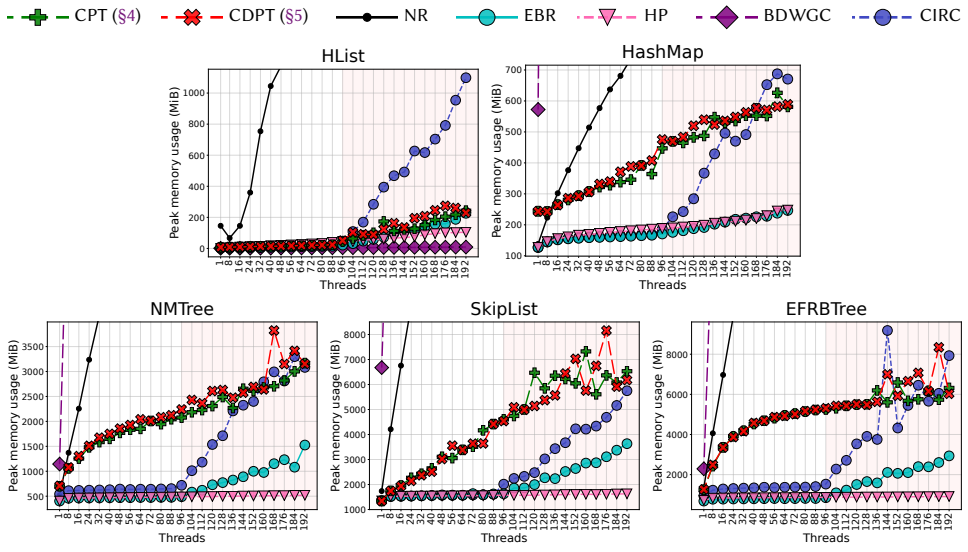


Fig. 35. Peak memory usage of write-heavy workloads for a varying number of threads with large key ranges.

C.2.2 Read-write Workloads.

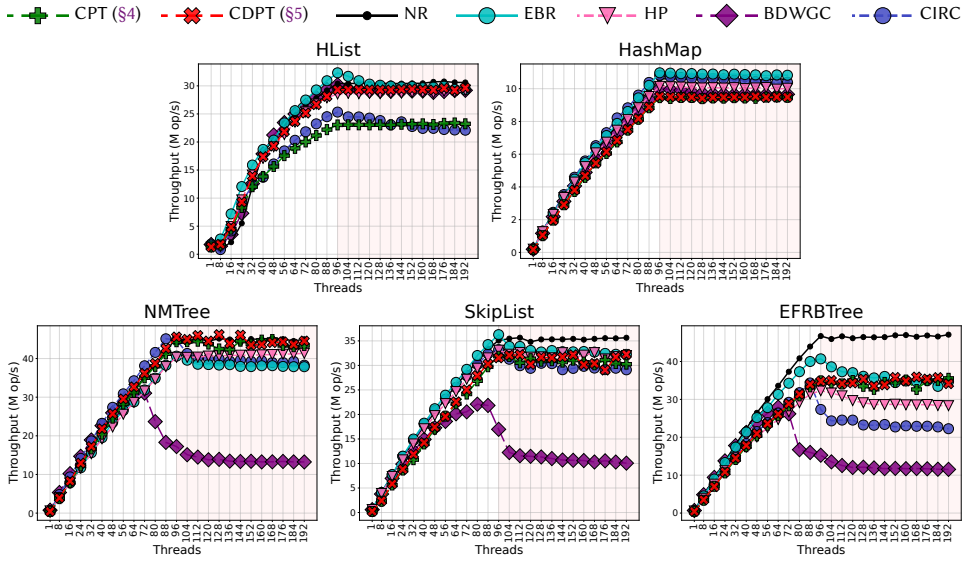


Fig. 36. Throughput (million operations per second) of read-write workloads for a varying number of threads with large key ranges.

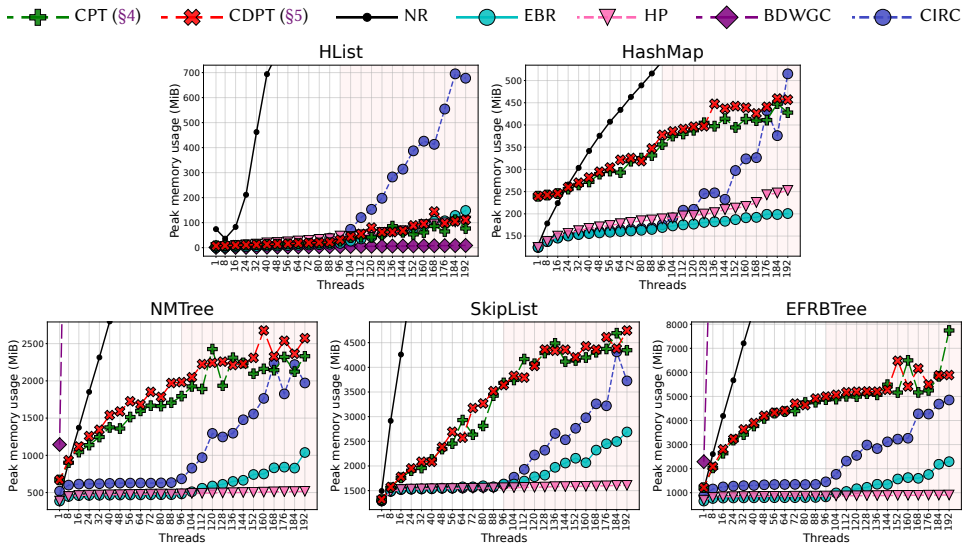


Fig. 37. Peak memory usage of read-write workloads for a varying number of threads with large key ranges.

C.2.3 Read-most Workloads.

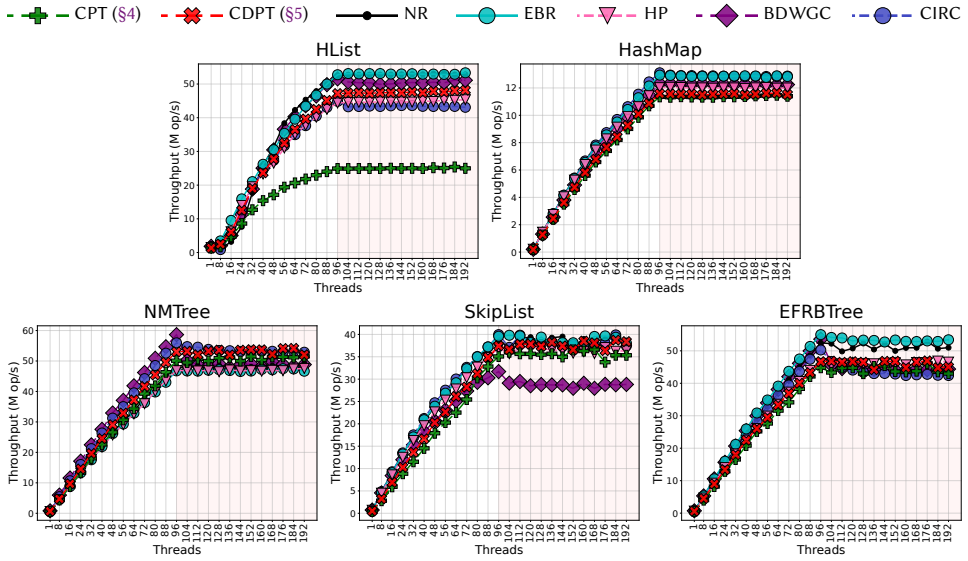


Fig. 38. Throughput (million operations per second) of read-most workloads for a varying number of threads with large key ranges.

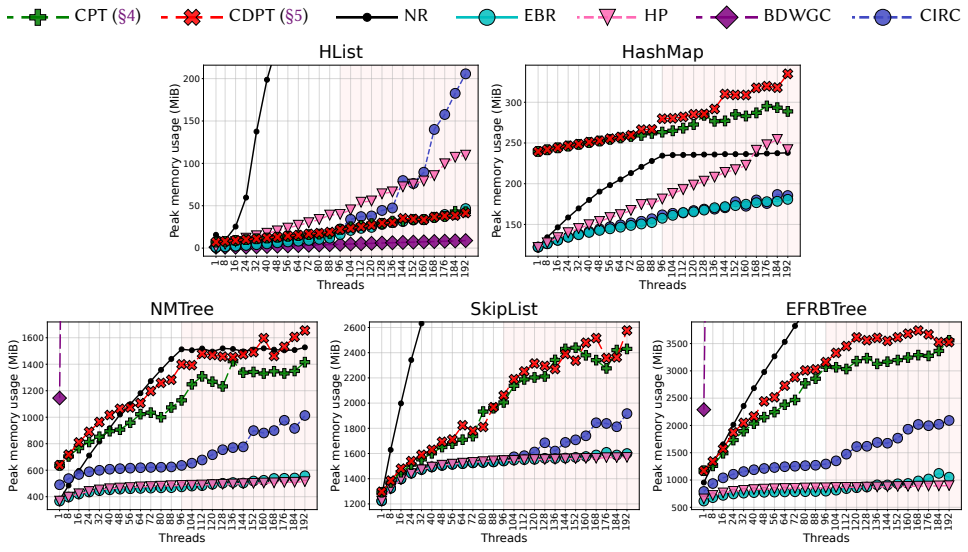


Fig. 39. Peak memory usage of read-most workloads for a varying number of threads with large key ranges.

C.3 Macro-benchmark

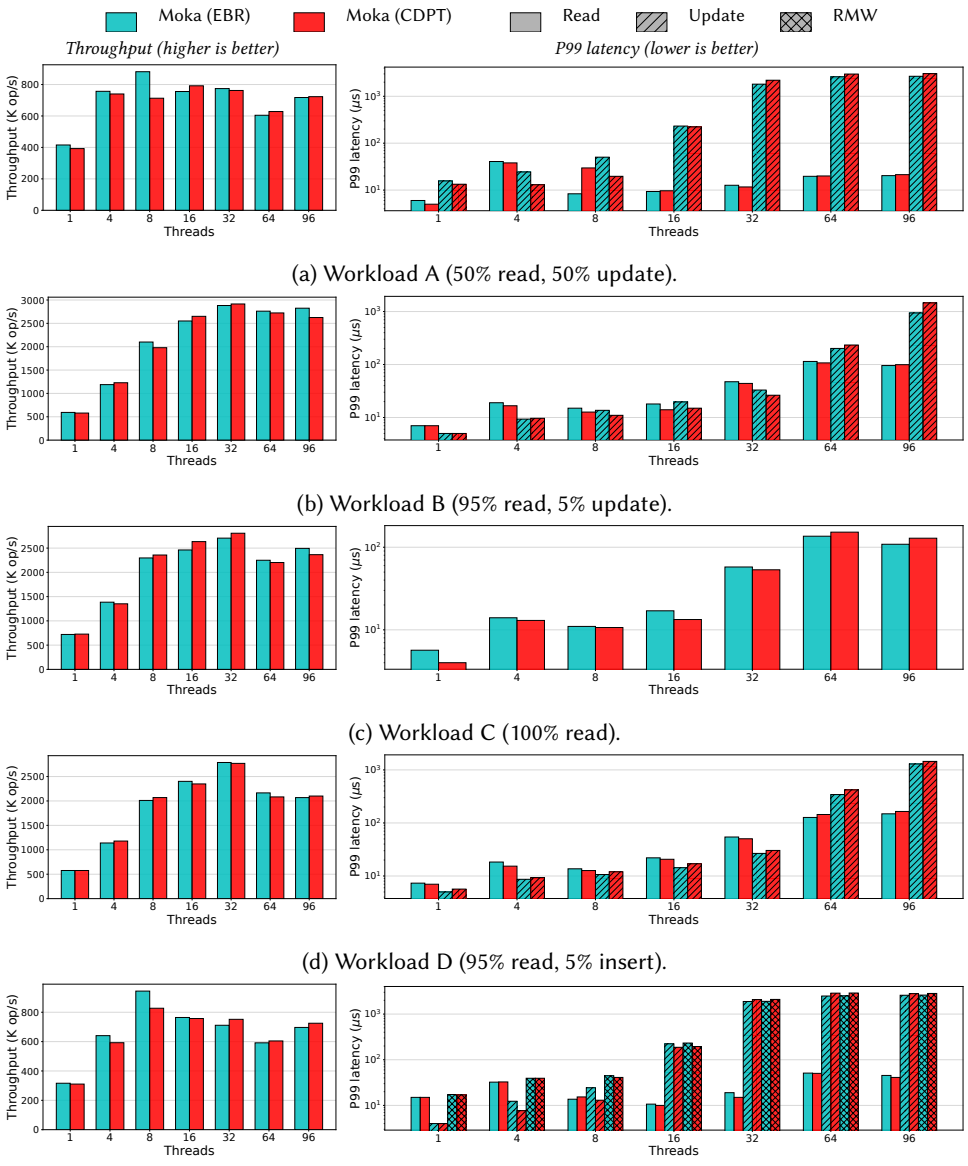


Fig. 40. Moka+YCSB macro-benchmark: throughput and per-operation P99 latency. Hatching distinguishes operation types; color distinguishes reclamation scheme.