Leveraging Immutability to Validate Hazard Pointers for Optimistic Traversals

JANGGUN LEE, KAIST, Korea
JEONGHYEON KIM, KAIST, Korea
JEEHOON KANG, KAIST, Korea

Hazard pointers (HP) is one of the earliest manual memory reclamation algorithms for concurrent data structures. It is widely used for its robustness: memory overhead is bounded (*e.g.*, by the number of threads). To access a node, threads first announce the protection of *each* to-be-accessed node, which prevents its reclamation. After announcement, they validate the node's reachability from the root to ensure that no threads have missed the announcement and reclaimed it. Traversal-based data structures typically use a marking-based validation strategy. This strategy uses a node's mark to indicate whether the node is to be detached. Unmarked nodes are considered safe to traverse as both the node and its successors are still reachable, while marked nodes are considered unsafe. However, this strategy is inapplicable to the efficient *optimistic traversal* strategy that skips over marked nodes.

We propose a new validation strategy for HP that supports lock-free data structures with optimistic traversal, such as lists, trees, and skip lists. The key idea is to exploit the *immutability* of marked nodes, and validate their reachability at once by checking the reachability of the *most recent unmarked node*. To ensure correctness, we prove the safety of Harris's list protected with the new strategy in Rocq using the Iris separation logic framework. We show that the new strategy's performance is competitive with state-of-the-art reclamation algorithms when applied to data structures with optimistic traversal, while remaining simple and robust.

CCS Concepts: • **Computing methodologies** → **Concurrent algorithms**; • **Software and its engineering** → **Garbage collection**; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: concurrent memory reclamation, hazard pointers, concurrent data structures

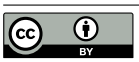
ACM Reference Format:
Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2025. Leveraging Immutability to Validate Hazard Pointers for Optimistic Traversals. *Proc. ACM Program. Lang.* 9, PLDI, Article 148 (June 2025), 37 pages. https://doi.org/10.1145/3729247

1 Introduction

Concurrent data structures allow multiple clients to access shared memory simultaneously, providing high performance. For these data structures, memory management is challenging because it is difficult to determine when memory is no longer accessed by all threads and thus safe to free.

To aid memory management for concurrent data structures, several manual reclamation algorithms have been proposed [4, 23, 26, 28, 33, 36, 41–44, 46, 48, 51, 54]. When using such algorithms, memory nodes go through the following life cycle. First, a node is *allocated*, initialized, and placed in a shared data structure. Second, it is *detached* by making it unreachable from the data structure root. Third, it is *retired* by calling the algorithm-provided retire function, scheduling its reclamation.

Authors' Contact Information: Janggun Lee, KAIST, Daejeon, Korea, janggun.lee@kaist.ac.kr; Jeonghyeon Kim, KAIST, Daejeon, Korea, jeonghyeon.kim@kaist.ac.kr; Jeehoon Kang, KAIST, Daejeon, Korea, jeehoon.kang@kaist.ac.kr.



ACM 2475-1421/2025/6-ART148
https://doi.org/10.1145/3729247

Finally, it is *reclaimed* by the reclamation algorithm by giving it back to the memory allocator. For safety, a node must not be reclaimed while accessible. Clients ensure this by *protecting* a node before accessing it, preventing reclamation by other threads.

Hazard pointers (HP). HP [35, 36] is one of the earliest reclamation algorithms. HP is widely used for its *robustness*: the number of retired but unreclaimed nodes is bounded (*e.g.*, by the number of threads or a fixed constant).[1] To ensure robustness, a thread announces the protection of *each* to-be-accessed node to prevent its reclamation so that other threads will not reclaim the node until the announcement is revoked. In contrast to HP with such per-pointer and fine-grained protection, some reclamation algorithms such as read-copy-update (RCU) [33] employ coarse-grained protection mechanisms for efficient synchronization, trading robustness for performance.

An announcement of protection, however, is insufficient to ensure safe access to a node. To see why, suppose a thread **(1)** obtains a node p to access, and **(2)** announces the protection of p. As **(1)** and **(2)** are not atomic, a reclaiming thread may retire and reclaim p between **(1)** and **(2)**. Therefore, after **(2)**, the protecting thread should *validate* that p has not been retired before accessing it.

For real-world concurrent data structures, it is often difficult to determine whether a node has been retired for validation [23]. Thus, it is common to conservatively check whether the node is still reachable from the data structure root. If the node is reachable, it has not yet been retired, as nodes should only be retired after being detached from the data structure.

For traversal-based data structures, it is often more efficient to further over-approximate unreachability for validation, because precise reachability analysis may require a potentially long traversal from the root. The standard over-approximation strategy for unreachability, which we call Val_{mark}, is based on a node's *mark*: a node is first marked as to be detached from the data structure [16]. For example, when moving from node n to the next node m, the protecting thread checks if n is marked. If n is unmarked, it is reachable, and so must be m, which validates the protection of m. If n is marked, it might have been detached, and so might m, failing to validate the protection of m. Even without reclamation, marking is essential for the correctness of data structures to ensure that a detached node cannot create a new link to a live node, preventing the live node from getting lost.

Limitations of prior approaches to applying HP to optimistic traversals. The Val_{mark} strategy is inapplicable to the efficient *optimistic traversal* strategy common in traversal-based data structures such as linked lists and trees [3, 10, 16, 20, 39, 45, 47, 52]. In optimistic traversals, when a thread encounters a marked (hence to-be-detached) node, it skips over to the next node for performance. In contrast, Val_{mark} requires the current node to be unmarked or it will restart.

As a result, data structures protected using HP with Val_{mark} are often outperformed by those using optimistic traversal [6]. For example, Harris [16]'s list is one of the earliest concurrent data structures that employs optimistic traversal. Michael [35] argues that Val_{mark} is incompatible with Harris's list, leading Michael [34] to adapt it to Val_{mark} by *sacrificing* optimistic traversal.

While prior research has explored the application of HP to optimistic traversals, these approaches have either exhibited inefficiency [4] or been restricted to specific data structures [20, 36, 37]. For example, Michael [35] presents a version of Michael and Scott [38]'s lock-free queue that uses HP, but its optimistic traversal consists of a single step, while optimistic traversal in Harris's list and other data structures in general require unbounded number of steps (see §2.4 for other examples).

Immutability-based validation for optimistic traversals. To efficiently apply HP to optimistic traversal, we propose an *immutability based validation strategy*, which we call Val_{immut}. In particular, we exploit the fact that in traversal-based data structures, a marked node's outgoing pointer fields

[1] For a full definition, see [50].

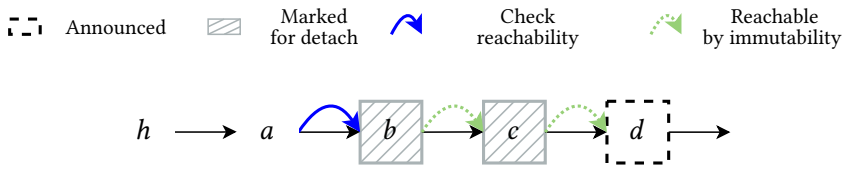


Fig. 1. Successful validation of d by reading from most recent unmarked node a. b is reachable as a is unmarked and points to b. b's next node is reachable, and as it is immutable, the next node is c; and d is reachable by the same reasoning.

never change. Our strategy is broadly applicable as immutability is essential for the correctness of traversal-based data structures [16], *i.e.*, to prove linearizability [18] (see §3.4 for more details).

By leveraging immutability, a thread can validate from a marked node by checking the reachability of the *most recent unmarked node*, allowing traversal to continue, as illustrated in Fig. 1. A thread has traversed unmarked node a and marked nodes b and c, and wishes to step from c to d. To validate the protection of d, the thread checks that a is still unmarked and points to b. If so, since a is unmarked, it is reachable, and b is reachable because a points to it. Next, c is reachable because b is immutable and its next node was c, and d is reachable for the same reason. Thus, the protection of d is validated.

Outline. Throughout this paper, we elaborate on our key contribution, namely utilizing immutability to apply HP to concurrent data structures with optimistic traversal. Specifically:

- In §2, we review the technical background of HP and the marking-based validation strategy Val_{mark} using Harris [16]'s and Michael [34]'s lock-free lists as running examples.
- In §3, we apply HP with the immutability based validation strategy Val_{immut} to representative concurrent data structures with optimistic traversal, in particular to Harris [16]'s linked list, Natarajan and Mittal [39]'s binary tree, and Shavit et al. [47]'s skiplist, and discuss the broad applicability of Val_{immut}.
- In §4, we formally verify the safety of Harris's list using HP with Val_{immut} in Rocq (formerly Coq) using the Iris separation logic framework [24, 25, 29].
- In §5, we evaluate the performance of HP with Val_{immut} against various state-of-the-art reclamation algorithms [2, 21, 23, 26, 28, 48] on various workloads. We observe that HP with Val_{immut} benefits greatly from optimistic traversal, and is competitive with other reclamation algorithms.
- In §6, we conclude with related and future work.
- In the supplementary material [30], we present all of our proofs and experimental results.

2 Background and Motivation

We review HP (§2.1) and its application using Val_{mark} to Michael's list[2] (§2.2); explain why HP with Val_{mark} is inapplicable to Harris's list with optimistic traversal (§2.3); and discuss prior approaches to applying HP to optimistic traversal (§2.4).

2.1 Hazard Pointers

Algorithm 1 outlines the high-level structure of HP. When a thread wants to access a pointer, say p, it first announces the protection of p by storing it in an *HP slot* (line 2). To validate the protection, the thread checks that p has not been retired at the time of announcement (line 3). Conversely,

[2]We discuss Michael's list before Harris's list despite the historical order for presentation purposes.

Algorithm 1 Hazard pointers algorithm sketch.

```
1: fun Protect(p)
2:     Set p to a thread-local hazard pointer slot.
3:     Check if p is not retired. If retired, fail.
4: fun Reclaim(p)
5:     Announce retirement of p.
6:     Check if p is protected. If protected, retry later.
```

when a thread wants to reclaim p, it first announces the retirement of p (line 5), then checks if any thread is protecting the node (line 6), and if not, frees p.

The safety of HP is proven through a simple case analysis of the execution order. If line 5 happens before line 3, the protection will fail, and no access will occur. If line 2 happens before line 6, and no reclamation will occur. In either case, no use-after-free will occur.

As discussed in §1, it is more efficient to over-approximate the check for retirement at line 3 with unreachability, and for traversal-based data structures, even further with marking.

2.2 Applying Hazard Pointers using Val_{mark} to Michael's List

We elaborate on the over-approximation for validation in Val_{mark} using Michael [34]'s list as an example, a lock-free singly linked list that implements a set data structure and supports the insertion and removal of nodes at any position.[3]

Fig. 2 illustrates the traversal of Michael's list. Code related to HP (shaded in purple) will be explained below. Each Node consists of an integer key and an atomic pointer next containing a pointer to the next node. Micheal's list is represented with an atomic pointer to Node, initialized to NULL. Operations on atomic pointers are given as explicit methods (*e.g.*, n.load() to atomically read from n). A key feature of this list is marking [16]. When a thread wants to detach a node, it first marks the node by tagging the least significant bit (LSB) of the next pointer field as 1.[4] Marking removes a node from the abstract set[5], but the node remains reachable from the head node. In the diagrams, outgoing edges represent the next pointer field of each node, with dashed edges indicating a pointer field with tag value 1.

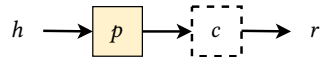
The main interface for the list is Get, Insert, and Remove operations. Each of these functions takes a list and a key as input, and returns a boolean that indicates the oepration's success. The implementation of each operation first calls Search to locate the relevant position within the list, and may modify the list depending on whether the key was found or not. For Get, it simply returns whether the key was found or not. For Insert, when key is not found, it inserts a new node with key between prev and curr with a compare-and-swap (CAS) (line 33). For Remove, when key is found, it removes curr in two CASs. First, it marks curr by changing next's LSB (line 42) from 0 to 1, with the help of decomp function that splits a pointer into an aligned pointer and its tag ensure next is aligned. Second, it detaches curr by changing prev's next from curr to next (line 43).

The goal of Search(list,key) is to traverse list and return a triple (prev, curr, found), where **(1)** prev is a reference to a next field of a Node that has a key less than the target key; **(2)** curr is a pointer to the node obtained from prev; and **(3)** found is a boolean that indicates if the key in curr equals key or is greater than key. The traversing thread begins by initializing prev to list and curr to the first node (line 7). It then enters a loop with the invariant that curr was loaded from prev in the last iteration and that its tag is 0 (line 9). Inside the loop, it first sets next to curr's next node and tag to

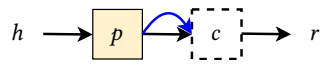
[3]We implement a set for simplicity. It is straightforward to extend the list to a map.
[4]We assume pointers are 8-byte aligned and the last 3 bits are available for tagging.
[5]In other words, the removal operation is *linearized* [18] when the node is marked.

Announced Protected Tagged Check reachability

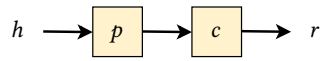
```
1:  struct Node
2:      key: int | next: Atomic<Node*>

3:  thread-local variables
4:      hp_prev, hp_curr: HSlot

5:  fun Search(list: &Atomic<Node*>, key: int)
        → (&Atomic<Node*>, Node*, bool)
6:  restart:
7:      prev ← list; curr ← prev.load()
8:      found ← false
9:      while curr ≠ null do
10:         hp_curr.set(curr)
11:         if prev.load() ≠ curr then goto restart
12:         (next, tag) ← decomp((*curr).next.load())
13:         if tag = 1 then
14:             detach curr; goto restart
15:         else
16:             if (*curr).key ≥ key then
17:                 found ← (*curr).key = key; break
18:             prev ← &(*curr).next
19:             hp_swap(&hp_curr, &hp_prev)
20:             curr ← next
21:     end while
22:     return (prev, curr, found)

23: fun Get(list: &Atomic<Node*>, key: int) → bool
24:     return Search(list, key).2

25: fun Insert(list: &Atomic<Node*>, key: int)
        → bool
26:     new ← new_node(key)
27:     loop
28:         (prev, curr, found) ← Search(list, key)
29:         if found then
30:             reclaim_node(new)
31:             return false
32:         (*new).next.store(curr)
33:         if prev.cas(curr, new) then
34:             return true
35:     end loop

36: fun Remove(list: &Atomic<Node*>, key: int)
        → bool
37:     loop
38:         (prev, curr, found) ← Search(list, key)
39:         if !found then
40:             return false
41:         (next, _) ← decomp((*curr).next.load())
42:         if (*curr).next.cas(next, next | 0b1) then
43:             if prev.cas(curr, next) then
44:                 retire(curr)
45:             return true
46:     end loop

47: fun decomp(ptr: Node*) → (Node*, int)
48:     return (ptr & ~0b111, ptr & 0b111)
```



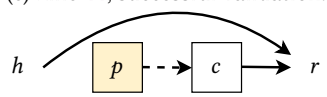
(a) Line 10, announce protection.



(b) Line 11, validation.



(c) Line 11, successful validation.

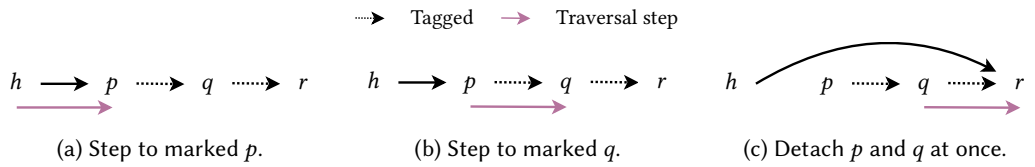


(d) Line 11, failed validation.

Fig. 2. Traversal of Michael's list with hazard pointers.

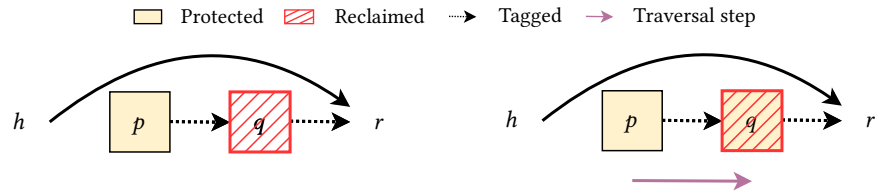
its tag (line 12). If tag is 1, indicating that curr is marked, it attempts to detach curr and continue with next as the new curr (lines 13 and 14). If tag is 0 and the search key is found, it returns (line 17). Otherwise, it updates prev and curr to curr and next for the next iteration (lines 18 to 20).

The traversing thread must protect curr before accessing it. It announces the protection of curr by storing curr to hp_curr (line 10, Fig. 2a) and validates it by checking that prev points to curr (line 11, Fig. 2b). A successful check ensures two conditions: **(1)** prev is unmarked and thus reachable; and **(2)** prev still points to curr, making curr also reachable. Together, these conditions validate curr



(a) Step to marked p. (b) Step to marked q. (c) Detach p and q at once.

Fig. 3. Traversal of marked nodes in Harris's list.



(a) Detach p and q at once, then reclaim q. (b) Incorrectly protect and access q.

Fig. 4. Unsafe traversal of Harris's list using HP with Val_{mark}.

(Fig. 2c). If validation fails (Fig. 2d), the traversal restarts. Val_{mark} over-approximates unreachability, potentially failing even if curr is reachable, *e.g.*, when prev is marked but not yet detached.

HP with Val_{mark} requires two HP slots used in a hand-over-hand fashion: **(1)** hp_curr, protecting curr; and **(2)** hp_prev, protecting prev. At the start of the loop, it first tries to protect curr with hp_curr (line 10); and when a link is followed, it swaps hp_curr and hp_prev (line 19).

2.3 Inapplicability of Hazard Pointers using Val_{mark} to Harris's List

HP with Val_{mark} is inapplicable to data structures that use optimistic traversal, where threads traverse marked nodes for performance, as validation requires nodes to be unmarked.

Harris's list. One of the most representative concurrent data structures with optimistic traversal is Harris [16]'s list. It shares the same physical structure and marking mechanism as Michael's list, but differs in its traversal strategy.

Fig. 3 illustrates a traversal of Harris's list with marked nodes p and q: **(1)** a traversing thread first steps from h to p (Fig. 3a); **(2)** instead of detaching p, it steps to q (Fig. 3b); and **(3)** steps to r and detaches both p and q at once (Fig. 3c).

Harris's list outperforms Michael's list due to fewer detach operations and less restarts: Harris's list can detach multiple marked nodes at once, whereas Michael's list detaches only one node at a time. This performance advantage becomes particularly significant under heavy contention (see Fig. 14a for details).

Incorrect application of HP with Val_{mark}. The aforementioned traversal of Harris's list is incompatible with the hand-over-hand protection of Michael's list. Fig. 4 illustrates an execution where a thread uses hand-over-hand protection to traverse Harris's list but ignores tags to maintain optimistic traversal, leading to use-after-free. **(1)** The traversing thread protects p and is about to access q. **(2)** Another thread detaches p and q at once, retires them, sees that q is not protected, and frees q (Fig. 4a). **(3)** The traversing thread announces the protection of q and successfully validates it, as p still points to q (ignoring p's tag), thus steps to and accesses the already freed q (Fig. 4b).

2.4 Prior Approahces to Applying Hazard Pointers to Optimistic Traversals

Applying HP impractically. There are two contrived approaches to apply HP to data structures with optimistic traversal [4], but neither is practical. The first approach protects *all* nodes met during traversal and validates that all remain reachable when moving to the next node. This approach is inefficient because it requires re-checking all intermediate nodes, and it is not robust, as it demands an unbounded number of HP slots. The second approach restarts unconditionally upon encountering a marked node. This not only disables optimistic traversal but also breaks lock-freedom [17][6], as traversal can repeatedly restart if the marked node is not detached, which can occur in Harris's list. Retrospectively, Michael's list was adapted from Harris's list to enable HP while preserving lock-freedom.

Applying HP to specific data structures. Several prior studies have efficiently applied HP with optimistic traversal [20, 35, 37], but they are tailored to specific data structures and do not explore its application to general data structures. In §1, we discussed Michael [35]'s adaptation of Michael and Scott [38]'s queue. We now explain the others.

Howley and Jones [20] present a lock-free binary search tree with optimistic traversal. They claim that HP can be applied to their tree by exploiting "the highest unmarked node in the tree, which can be checked for updates." While this suggests a way to protect nodes during optimistic traversal, it does not fully explain how to apply this concept concretely with HP. Notably, it overlooks the critical role of immutability and how it can be used to ensure that checking the head node confirms the reachability of marked nodes. Howley [19] (the first author's Ph.D. thesis) presents an implementation of the tree using HP, but it does not perform optimistic traversal.

Michael [37] presents a modification of HP that enables unconditional traversal for acyclic data structures with immutable links [38, 53]. However, they require the data structure's links to be updated at most once from intialization, which makes it inapplicable to data structures with mutable links [16, 39, 47].

Extending HP for optimistic traversal. HP++ [23] is an extension of HP. While data structures using HP with Val_{mark} must restart when encountering a marked node during validation, those using HP++ ignore markings and traverse over them to enable optimistic traversal. For safety, threads detaching a node protect nodes pointed by the detached node. However, this additional protection during detachment is costly (see §5 for details). Moreover, HP++ cannot be applied to data structures with complex marking strategies, such as the elimination (a,b) tree [52], because the tree does not support the *invalidation* mechanism in HP++ for tracking detached nodes. Invalidation requires data structures to mark a node by tagging its next pointer field, but the tree marks nodes using a separate flag.

3 Immutability-Based Validation

We apply our immutability-based validation strategy, Val_{immut}, to representative lock-free data structures with optimistic traversal to show its general applicability, in particular to Harris's list (§3.1); the Natarajan-Mittal binary tree (§3.2); and the Shavit-Lev-Herlihy skiplist (§3.3). We then argue that Val_{immut} is broadly applicable to traversal-based data structures that utilized marking (§3.4).

[6] A data structure is lock-free if, under any scheduling, at least one operation completes successfully.

3.1 Harris's List Revisited

We explained the optimistic traversal of Harris's list, where multiple marked nodes are detached at once (§2.3); and briefly introduced the application of HP with Val_{immut} to Harris's list (§1). We now detail this application, emphasizing previously unaddressed subtleties.

Algorithm 2 Implementation of Harris's list using HP with Val_{immut}.

```
1:  thread-local variables
2:      hp_prev, hp_curr, hp_anchor, hp_a_next: HSlot
3:  fun Search(list: &Atomic<Node*>, key: int)
        → (&Atomic<Node*>, Node*, bool)
4:  restart:
5:      prev ← list; curr ← prev.load()
6:      anchor ← NULL; a_next ← NULL
7:      found ← false; prev_clear ← true
8:      while curr ≠ NULL do
9:          hp_curr.set(curr)
10:         if prev_clear then
11:             if prev.load() ≠ curr then
12:                 goto restart
13:         else
14:             if anchor.load() ≠ a_next then
15:                 goto restart
16:         (next, tag) ← decomp((*curr).next.load())
17:         if tag = 0 then
18:             if (*curr).key ≥ key then
19:                 found ← (*curr).key = key; break
20:             else
21:                 prev_clear ← true
22:                 prev ← &(*curr).next; curr ← next
23:                 anchor ← NULL; a_next ← NULL
24:                 hp_swap(&hp_curr, &hp_prev)
25:         else
26:             prev_clear ← false
27:             if anchor = NULL then
28:                 anchor ← prev; a_next ← curr
29:                 hp_swap(&hp_anchor, &hp_prev)
30:             else if &(*a_next).next = prev then
31:                 hp_swap(&hp_a_next, &hp_prev)
32:             prev ← &(*curr).next; curr ← next
33:             hp_swap(&hp_prev, &hp_curr)
34:     end while
35:     if anchor ≠ NULL then
36:         if anchor.cas(a_next, curr) then
37:             retire from a_next to node before curr
38:             prev ← &(*anchor).next
39:         else goto restart
40:     if curr ≠ NULL &&
            decomp((*curr).next.load()).1 = 1 then
41:         goto restart
42:     else
43:         return (prev, curr, found)
```

Algorithm. Algorithm 2 presents the implementation of Harris's list using HP with Val_{immut}. The interface and implementation of Harris's list closely mirrors Michael's list (Fig. 2), with the only differences being the implementation of Search and the number of HP slots used. Thus, we focus on the differences. Code related to HP (shaded in purple) will be explained below.

A Search(list, key) invocation traverses the list to find a node with key and returns whether the node was found. The traversing thread uses four local variables: curr, the node it aims to check to in this iteration; prev, the reference to the next field of the node before curr; anchor, the last unmarked node; and a_next, the node after anchor.

At the start, the traversing thread initializes prev to head and curr to the first node, similar to Michael's list. It then enters the loop (from line 8 to line 34) with the invariant that curr is loaded from prev during the last iteration, and that anchor and a_next are non-null if prev is marked.

Inside the loop, if curr is null, the traversing thread has reached the end of the list and breaks from the loop, noting it failed to find the target node (line 8). Otherwise, it checks if curr's next pointer field is tagged (line 17). If the tag is 0, curr is unmarked, and the thread checks if curr's key is greater than or equal to the target key (line 18). If so, curr is the target node and breaks out

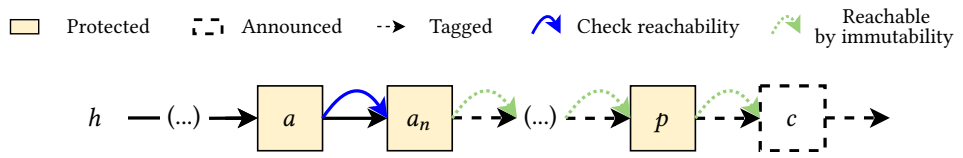


Fig. 5. Validating c by checking a in Harris's list. If successful, a is reachable as it is unmarked, and so is a_n as a points to a_n. As the chain $a_n \rightarrow \cdots \rightarrow p \rightarrow c$ is immutable, c is also reachable and is validated.

of the loop (line 19). Otherwise, it advances prev and curr to the next node, resetting anchor to NULL (line 21 to line 24). If the tag is not 0, curr is marked, and the thread traverses to the next node, updating anchor and a_next accordingly (line 26 to line 33).

After the loop, the traversing thread detaches the chain of marked nodes from anchor to curr (line 36, as in Fig. 3c) and checks that curr is still unmarked and returns the result (line 40).

A key aspect of this algorithm is maintaining the invariant that all marked node's next field is immutable. This is achieved by **(1)** performing all updates with a CAS (line 36, and line 33, line 42, line 43 of Fig. 2); **(2)** requiring the "current pointer" argument to have tag value of 0. This ensures that pointer field is updated only when it has tag 0, *i.e.*, is unmarked.

Protection. We now explain the HP-related code in detail. The traversing thread uses four HP slots to protect the nodes prev, curr, and two anchor nodes in a hand-over-hand fashion during traversal. prev is always protected, while anchor and a_next are protected if they are not null. curr will be protected in the loop. It also tracks whether prev was marked in the previous iteration using a boolean flag prev_clear, initialized to true.

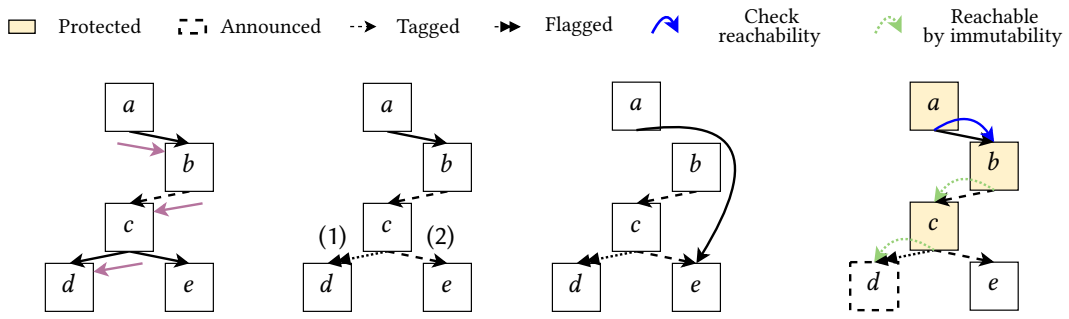
At the start, if curr is not null, the traversing thread must protect it for later access. It first sets curr to an HP slot (line 9) and then validates curr. If prev is unmarked, it validates by checking that prev still points to curr (line 11), in the same way as Michael's list. Otherwise, it validates by checking that anchor still points to a_next (line 14).

Fig. 5 shows why checking anchor in line 14 validates curr. The list has head h, and the traversing thread's anchor nodes are a and a_n, is currently at p, and wants to validate c. The nodes between a_n and p are all marked hence immutable. To validate, the thread checks that a's next pointer equals a_n, which also implies that a's next pointer field's tag is 0. If so, a is reachable since it is unmarked, and so is a_n as a points to it. As the chain $a_n \rightarrow \cdots \rightarrow p \rightarrow c$ is immutable, the thread knows that a_n (transitively) points to c, so c is reachable and validated.

After successful validation, the traversing thread updates prev_clear accordingly (lines 21 and 26), does hand-over-hand protection for anchor and a_next (lines 29 and 31), and upon successful detachment of the marked chain (line 36) retires the nodes from a_next to curr (line 37).

The protection of a_next is necessary to prevent the *ABA problem*, which leads to use-after-free. The ABA problem occurs when a node is detached, reallocated, and then re-inserted at the same position with the same physical address. Although the allocations are different, they cannot be distinguished solely by their physical addresses. For instance, if we do not protect a_n in Fig. 5, the following execution with use-after-free may occur: **(1)** The chain of marked nodes from a_n to c is detached and freed. **(2)** a_n is reallocated and re-inserted as the next pointer of a. **(3)** Protection of c is announced and successfully validated, as a still points to a_n. **(4)** However, accessing c is unsafe since it has already been freed. To prevent such errors, the traversing thread should protect a_n.

Harris's list using HP with Val_{immut} is robust and lock-free. It is robust because the number of per-thread HP slots is fixed at four. It is lock-free because a restarted thread always sees evidence of progress made by another thread. If a thread restarts at line 11, either prev points to a new node



(a) Traversal to leaf d. (b) Marking of parent c. (c) Detach marked nodes.

Fig. 6. Traversal during removal of node d in Natarajan-Mittal tree.

Fig. 7. Exploiting immutability to validate d by checking from a.

or has been marked, both indicating progress. The same applies to anchor at lines 14 and 39. If a thread restarts at line 40, curr has been marked, again signaling progress by another thread.

3.2 Natarajan-Mittal Tree

Natarajan and Mittal [39] designed a lock-free external binary search tree. We apply HP with Val_{immut} to the Natarajan-Mittal tree, similar to Harris's list.

It is worth noting the history of applying HP to the Natarajan-Mittal tree. In the original paper, the authors claim that HP can be applied [39, §3.2], but they do not provide an algorithm or implementation. The journal version includes an implementation with memory reclamation [40, §5.1], but it uses DEBRA [4] instead of HP. While an implementation using HP exists in the artifact for IBR [54], it has been identified as *incorrect* by Anderson et al. [1, §8], having memory leaks and use-after-free. To our knowledge, we present the first *correct* application of HP to the Natarajan-Mittal tree.

Algorithm. For simplicity, we focus on the parts related to HP, specifically traversal and marking. A node's pointer field to child nodes are marked before removal, making them immutable, and traversal through the tree is optimistic.[7] There are two ways to mark a pointer field: **(1)** *tagging*, indicating that it points to an internal node, and the marked node will be detached. **(2)** *flagging*, indicating that it points to a leaf node, and both the leaf node and the marked node will be detached.

Fig. 6 illustrates the process of removing node d from a tree. A thread begins by traversing to the leaf node d (Fig. 6a). Next, it prepares to detach node d by marking the parent node c in two steps (Fig. 6b); it **(1)** flags the pointer $c \to d$; then **(2)** tags the pointer $c \to e$. After updating both pointers, the thread detaches marked nodes b, c, and d using a single CAS (Fig. 6c).

Protection. Fig. 7 shows a validation example using Val_{immut} where a traversing thread wants to protect d. The thread protects the most recent unmarked node a, its next node b, and the last node of the marked chain, c. To validate d, it checks that a still points to b and is unmarked, and if so, both a and b are reachable. Since the chain $b \to c \to d$ is immutable, d is also reachable, completing validation.

[7] Unlike Harris's list, marking a node does not remove it from the abstract set; *i.e.*, it is not the linearization point.

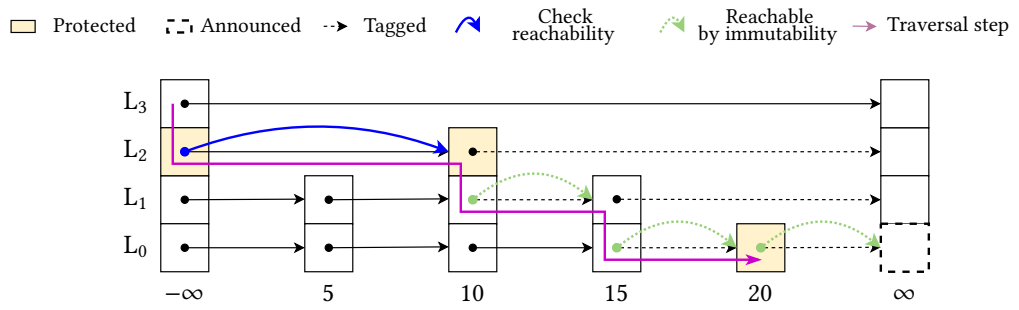


Fig. 8. Applying HP to skiplist's optimistic traversal.

3.3 Shavit-Lev-Herlihy Skiplist

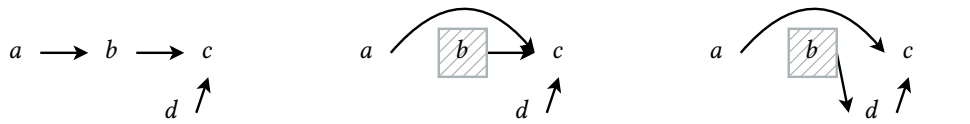
Shavit et al. [47] designed a lock-free skiplist that implements a set. We apply HP with Val_{immut} to the Shavit-Lev-Herlihy skiplist, again similarly to Harris's list.

It is worth noting the history of applying HP and reclamation algorithms in general to the Shavit-Lev-Herlihy skiplist. The original implementation with optimistic traversal was in Java, which does not require manual memory management. In unmanaged languages, it is necessary to track multiple incoming pointers to a node [6, §A.2]. A common solution is to use reference counting alongside the reclamation algorithm [23, 27, 28]. But these implementations apply HP to versions that use *non-optimistic traversal* for all operations. To our knowledge, we are the first to apply HP correctly to the Shavit-Lev-Herlihy skiplist with optimistic traversal.

Algorithm. Fig. 8 depicts a skiplist (we explain HP-related parts later). A skiplist of *level* H (indicating the maximum node height, $H = 4$ in Fig. 8), consists of a sorted singly linked list of nodes at each level from 0 to $H - 1$, with each upper level forming a sublist of the lower level. We refer to the N-th level as L_N. Each node contains a key-value pair, a height h ($\leq H$), and a list of next pointers for levels 0 to $h - 1$. The skiplist is initialized with two sentinel nodes whose height is H, $-\infty$ at the head and ∞ at the tail. Nodes can be marked multiple times, once for each level, by tagging their next pointer. Tagged next pointer fields are immutable. Nodes are marked from the top level down to the bottom, and a node is considered removed from the abstract set only if its L_0 pointer is marked. For instance, node 10 is marked at L_2 and L_1 but remains in the abstract set because L_0 is unmarked, while node 15 is marked at L_1 and L_0, removing it from the abstract set.

The skiplist uses two traversal strategies: a non-optimistic one for insertion and removal, and an optimistic one for lookup. In optimistic traversal, the thread starts at the top level and follows the highest-level pointers that do not overshoot the target key. For instance, to find key 20 in Fig. 8, a thread starts at $-\infty$ and follows the next pointer at L_3, arriving at ∞. Since this overshoots the target, the thread moves down to L_2 and proceeds to node 10, "skipping" over the node with key 5. The thread then ignores the mark on node 10, continues down to L_1 to reach node 15, and finally goes down to L_0 to reach node 20, completing the traversal.

Unlike Harris's list, the skiplist does not detach multiple marked nodes at once because detaching a node requires updates on multiple pointers. Marked nodes are detached one by one during non-optimistic traversal, in a similar manner to Michael's list.



(a) Prepare to insert d. (b) Detach b. (c) Insert d but unreachable.

Fig. 9. Example where mutability of a marked node breaks linearizability.

Protection. Fig. 8 also illustrates a validation example using Val_{immut} where a thread aims to move from 20 to ∞.[8] The skiplist's immutability and validation pattern are similar to Harris's list, with the marked chain vertically spanning multiple levels. The traversing thread protects the last unmarked node $-\infty$ and its next node 10 at L_2, and the end of the marked chain 20 at L_0. To validate ∞, the thread checks if $-\infty$ still points to 10 and remains unmarked. If so, both $-\infty$ and 10 are reachable. Since the chain $10 \longrightarrow 15 \longrightarrow 20 \longrightarrow \infty$ is immutable across levels, ∞ is still reachable and thus successfully validated.

3.4 Applicability of HP with Val_{immut}

The previous three examples illustrated the use of HP with Val_{immut}, showing that the immutability of the data structures was sufficient for validation. We now argue that Val_{immut} is broadly applicable to traversal-based data structures, as immutability is essential for their correctness. Specifically, immutability is essential for linearizability [18], the standard correctness criterion for concurrent data structures, which requires that every execution trace match a sequential execution.

To see why, consider the scenario depicted in Fig. 9, which illustrates how allowing marked nodes (pictured as dashed nodes) to remain mutable breaks linearizability in a general traversal-based data structure. Initially, there are three linked nodes, a, b and c, and a thread is attempting to insert a node d in between b and c (Fig. 9a). Then another thread intervenes, and marks and detaches b (Fig. 9b). As marked nodes can be mutated, the first thread successfully inserts d (Fig. 9c). However, node d becomes unreachable, breaking linearizability: a later removal of d will fail, a scenario impossible in a sequential execution.

4 Formal Verification

We formally verify the safety (*i.e.*, no memory errors) of Harris's list protected using HP with Val_{immut} (Algorithm 2), building upon Jung et al. [22]'s verification of HP. We first review the necessary background (§4.1), then present the key ideas in the verification of validation in Harris's list (§4.2). The verification of other parts largely follows that of Harris's list with RCU in Jung et al. [22] and is omitted. Notably, we were able to utilize Jung et al. [22]'s specifications for HP without any modifications, even though Val_{immut} is a new validation strategy. For the full proof, see our Rocq development [30].

4.1 Background

Separation logic basics. We build our proofs in the Iris separation logic framework [24, 25], and write our target programs in an untyped call-by-value λ-calculus with mutable state and concurrency, similar to Iris's HeapLang. In this following, *iProp* is the type of Iris proposition, $P_1 * P_2$ denotes the separating conjunction of two *iProp*s P_1 and P_2, and $P_1 \vdash P_2$ denotes logical entailment. Logical entailments do not allow changes to *ghost states*, which are crucial in modern

[8] Although ∞ is a sentinel node that is always reachable, its key value must still be read for confirmation, which requires protection.

Predicates

$NodeRes \triangleq Loc \to Val \to NodeId \to iProp$ $\quad$ $\mathsf{Managed}(\ell : Loc, i : NodeId, P : NodeRes) : iProp$

$\mathsf{Protected}(sid : \mathsf{HSlot}, \ell : Loc, i : NodeId, P : NodeRes) : iProp$ $\quad$ $\mathsf{HPSlot}(sid : \mathsf{HSlot}, \ell : Loc) : iProp$

Rules

(Managed-New)
$\ell \mapsto v * (\forall i.\, i \text{ fresh} \Rrightarrow P(\ell, v, i)) \Rrightarrow \exists i.\, i \text{ fresh} * \mathsf{Managed}(\ell, i, P)$

(HP-Retire)
$\{\mathsf{Managed}(\ell, \_, \_)\}\ \texttt{retire}(\ell)\ \{\mathsf{True}\}$

(HPSlot-Set)
$\{\mathsf{HPSlot}(sid, \_)\}\ sid.\mathsf{set}(\ell)\ \{\mathsf{HPSlot}(sid, \ell)\}$

(Protected-Managed-Agree)
$\mathsf{Protected}(\_, \ell, i, \_) * \mathsf{Managed}(\ell, i', \_) \vdash i = i'$

(HPSlot-Validate)
$\mathsf{Managed}(\ell, i, P) * \mathsf{HPSlot}(sid, \ell) \Rrightarrow \mathsf{Managed}(\ell, i, P) * \mathsf{Protected}(sid, \ell, i, P)$

(Protected-Access)
$$\frac{\{\exists v.\, \ell \mapsto v * P(\ell, v, i) * P_1\}\ e\ \{\ell \mapsto v' * P(\ell, v', i) * P_2\} \qquad e \text{ physically atomic}}{\{\mathsf{Protected}(sid, \ell, i, P) * P_1\}\ e\ \{\mathsf{Protected}(sid, \ell, i, P) * P_2\}}$$

Fig. 10. A specification of HP.

separation logic for describing various protocols. To support this, Iris supports *view shifts*, $P_1 \Rrightarrow P_2$, which denotes that starting from P_1, we can modify ghost states and transition to P_2. We specify an expression e as a Hoare triple $\{P\}\ e\ \{Q\}$, where P is the precondition and Q is the postcondition.

Modular specification of HP. Fig. 10 shows the modular specification of HP by Jung et al. [22]. At the center of the specifications are the Managed, HPSlot, and Protected predicates.

$\mathsf{Managed}(\ell, i, P)$ denotes ownership of a pointer ℓ with *allocation id* i and a *node resource* P. Each allocation has a unique allocation id, so if a pointer is freed and reallocated, the ids will differ. A node resource $P(\ell, v, i)$ denotes the state of a node at address ℓ with value v and allocation id i, such as immutability. Managed-New creates a Managed from a *points-to* predicate $l \mapsto v$ and a view shift to P. A points-to predicate is a primitive resource used for reading and writing. HP-Retire destroys a Managed. Thus, ownership of Managed for a node implies that it has not been retired.

$\mathsf{HPSlot}(s, \ell)$ denotes that the pointer ℓ is announced at the HP slot s; and $\mathsf{Protected}(s, \ell, i, P)$ denotes that the pointer ℓ with allocation ID i and node resource P is protected by an HP slot s. To obtain a Protected, one announces a pointer to a HPSlot using HPSlot-Set; and uses HPSlot-Validate with a Managed to validate the protection. These two steps mirror those of an HP client to protect a pointer. Protected-Access states that Protected allows access to the underlying points-to-predicate and node resource to perform e, provided that e is *atomic*: it executes in a single step without any interleaving from the scheduler. Protected-Managed-Agree says that the allocation IDs of Protected and Managed predicates for the same pointer should coincide, ensuring that a protected pointer is not deallocated and thus free from the ABA problem.

4.2 Proof of Validation in Harris's List

We sketch the proof of the validation using the anchor nodes in the Search function of Harris's list (line 14 of Algorithm 2) in three parts: the *shape invariant* that describes the invariants of memory nodes in shared memory; the *loop invariant* that describes the invariants of thread-local variables during the traversal; and the concrete *proof steps*. The shape invariant is similar to that in Jung et al. [22]'s proof of Michael's list using HP with Val_{mark}, but the loop invariant and proof steps are new in this paper.

$$\mathsf{HLNode}(\ell, v, i) \triangleq \ldots * (\forall A.\, \mathsf{IsHL}(A, \_) \vdash A(i) = (l, v)) * \bigvee \begin{cases} \mathsf{LSB}(v.\mathtt{next}) = 0 * \ell \text{ is mutable} \\ \mathsf{LSB}(v.\mathtt{next}) = 1 * \ell \text{ is immutable} \end{cases}$$

$$\mathsf{IsHL}(A, L) \triangleq \mathsf{AllNodes}(A, L) * \mathsf{ReachableNodes}(A, L) * \ldots$$

$$\mathsf{AllNodes}(A, L) \triangleq \mathop{\Large *}_{i \mapsto (\ell, v) \in A} \bigvee \begin{cases} \mathsf{LSB}(v.\mathtt{next}) = 0 * (i, \ell, v) \in L * \cdots \\ \mathsf{LSB}(v.\mathtt{next}) = 1 * \ell \text{ is immutable} \cdots \end{cases}$$

$$\mathsf{ReachableNodes}(A, L) \triangleq \mathop{\Large *}_{k \mapsto (i, \ell, v) \in L[..:-1]} \mathsf{Managed}(\ell, i, \mathsf{HLNode}) * L(k+1) = (\_, v.\mathtt{next}, \_) * A(i) = (l, v)$$

Fig. 11. Shape invariant for Harris's list using HP with Val_{immut}.

$$\mathsf{LoopInvHL} \triangleq \mathsf{ProtInv} * \mathsf{RetireChain} * \cdots$$

$$\mathsf{ProtInv} \triangleq \mathsf{Protected}(\mathsf{hp\_anchor}, \mathsf{anchor}, i_{\mathsf{anchor}}, \mathsf{HLNode}) * \mathsf{Protected}(\mathsf{hp\_prev}, \mathsf{prev}, i_{\mathsf{prev}}, \mathsf{HLNode}) * \mathsf{Protected}(\mathsf{hp\_a\_next}, \mathsf{a\_next}, i_{\mathsf{a\_next}}, \mathsf{HLNode}) * \mathsf{HPSlot}(\mathsf{hp\_curr}, \_)$$

$$\mathsf{RetireChain}(\mathsf{anchor}, i_{\mathsf{anchor}}, \mathsf{curr}, i_{\mathsf{curr}}, L') \triangleq L(0) = (i_{\mathsf{anchor}}, \mathsf{anchor}, \_) * L(|L|-1) = (i_{\mathsf{curr}}, \mathsf{curr}, \_) * \mathop{\Large *}_{k \mapsto (i, \ell, v) \in L[..:-1]} v.\mathtt{next} \text{ is immutable} * L(k+1) = (\_, v.\mathtt{next}, \_)$$

(RetireChain-curr)

$$\frac{(\mathsf{anchor}, i_{\mathsf{anchor}}, \_) \in L}{\mathsf{RetireChain}(\mathsf{anchor}, i_{\mathsf{anchor}}, \mathsf{curr}, i_{\mathsf{curr}}, L') * \mathsf{IsHL}(A, L) \vdash (\mathsf{curr}, i_{\mathsf{curr}}, \_) \in L}$$

Fig. 12. Loop invariant of Search.

Shape invariant. Fig. 11 shows the shape invariant of Harris's list using HP with Val_{immut}. For simplicity, we only show the parts relevant to the validation. The node resource $\mathsf{HLNode}(\ell, v, i)$ denotes the two possible states of a node: marked or unmarked, using the LSB of v.next. If the LSB is 0, ℓ is unmarked and still mutable. Otherwise, the LSB is 1, ℓ is marked and immutable. The first part ensures that the node is contained in the map A, explained below.

The main invariant is $\mathsf{IsHL}(A, L)$, where A is an insert-only finite map (*not* related to the abstract set Harris's list implements) from allocation id to nodes that have been added to the list, and L is the list of currently reachable nodes. The first component AllNodes states that for each node $(\ell, \_)$ in the range of A, if ℓ is unmarked, ℓ is in L (i.e., reachable), and if not, ℓ is immutable. The second component ReachableNodes stores for each node p in L (except for the last node, which is null), the Managed for p, and the facts that p is in A and p's next node is in L. During validation, one will look up information for the target node inside the two parts in the invariants.

Loop invariant of Search. Fig. 12 shows the key parts of the loop invariant LoopInvHL of Search (variables are omitted for brevity).[9] The first component ProtInv stores the fact that the anchor, a_next, and prev pointers are protected by their respective HP slots. curr is not protected, as the first thing we will do in the loop is protect it. The second component RetireChain stores the marked nodes seen during the traversal. Specifically, $\mathsf{RetireChain}(\mathsf{anchor}, \_, \mathsf{curr}, \_, L')$ states that every node in L', except for the last node curr, is immutable, and its next node is also in L'.

[9]The full invariant does case analysis on whether anchor is null, and whether a_next equals prev. We focus on the most complicated case and omit the others.

For validation, the key usage of the loop invariant is using RetireChain to show that its end pointer is reachable. Formally, RetireChain-curr states that if one has a RetireChain from anchor to curr, and the current list state is IsHL(A, L), then anchor being in L implies curr being in L. Intuitively, RetireChain-curr is sound as RetireChain is an immutable chain, so the reachability of the first element implies reachability of the entire chain, including the last element.

Proof steps. We sketch the proof steps one will perform when validating using anchor nodes. The current separation logic context will roughly be as follows:

$$\text{Protected}(\text{hp\_a\_next}, \text{a\_next}, i_{\text{a\_next}}, \text{HLNode}) * \text{HPSlot}(\text{hp\_curr}, \text{curr}) *$$
$$\text{RetireChain}(\text{a\_next}, i_{\text{a\_next}}, \text{curr}, i_{\text{curr}}, L') * \text{IsHL}(A, L) * \cdots$$

From here, we prove the validation of HPSlot(hp_curr, curr). Specifically, we proceed as follows.

- By HLNode, we have $A(i_{\text{anchor}}) = (\text{anchor}, \_)$.
- By AllNodes, for anchor, since it is unmarked, we have $(\_, \text{anchor}, v) \in L$ where $v.\text{next} = \text{a\_next}$.
- By ReachableNodes for anchor, we have $(i', \text{a\_next}, \_) \in L$ for some allocation id i'.
- By ReachableNodes for a_next, we have a Managed$(i', \text{a_next}, _)$.
- By Protected-Managed-Agree with Managed$(i', \text{a_next}, _)$ and Protected$(_, \text{a_next}, i_{\text{a_next}}, _)$, we have that $i' = i_{\text{a\_next}}$, and so $(i_{\text{a\_next}}, \text{a\_next}, \_) \in L$.
- By RetireChain-curr, we have that $(i_{\text{curr}}, \text{curr}, \_) \in L$.
- By ReachableNodes for curr, we have a Managed for it, and by HPSlot-Validate using HPSlot(hp_curr, currr), we obtain a Protected predicate for curr, finishing the proof.

5 Performance Evaluation

We compare HP with Val$_{immut}$ against representative reclamation algorithms supporting optimistic traversal. The comparison is conducted across various combinations of reclamation algorithms and data structures, summarized in Fig. 13.

Building upon prior research [7, 9, 15], we optimize HP, HP++, PEBR, CIRC0-HP, and CDRC-HP implementations by employing *asymmetric fences* [8, §4] to minimize memory fence overhead. Specifically, we replace the protection-side fence with a *compiler fence*, which prevents compiler from reordering instructions without runtime cost; and the reclamation-side fence with a *process-wide fence* [5, 31], supported by Linux and Windows, ensuring a full memory barrier across all threads. This hot-path optimization for protection substantially reduces the overall memory fence costs while preserving correctness.

All algorithms trigger reclamation once per 1,024 retirements for a fair comparison. We evaluate the performance of two non-optimistic data structures, HMList and EFRBTree, against their optimistic counterparts, HHSList and NMTree, respectively, to demonstrate the performance benefits of optimistic traversal.

We implement the benchmark driver as a Rust library[11] and compile it with Rust nightly-2025-03-03 with default and link-time optimizations. We used `jemalloc` [13] 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. Both machines exhibit similar results, so we discuss only those for AMD64T. For the full experimental results, please see the supplementary material [30, §A and §B].

[10] For HP with Val$_{immut}$, HP++, HP-BRCU, VBR, and PEBR, get() is only lock-free due to rollback/recovery.
[11] Included in the artifact [30].

(a) List of evaluated reclamation algorithms.

reclamation	description	robust?
NR	no reclamation, as a rough upper bound on performance	✗
RCU	epoch-based RCU [14, 16], as an efficient reclamation algorithm	✗
PEBR	a hybrid of pointer- and epoch-based RCU [26]	✓
VBR	version-based reclamation [48]	✓
HP with Val$_{immut}$	the original HP [35, 36] **with our immutability-based validation**	✓
HP++	an extension of HP with frontier protections and invalidation [23]	✓
HP-BRCU	an extension of HP with BRCU-expedited traversal [28]	✓
CIRC0-HP	an HP flavor of CIRC without immediate recursive destruction [21]	✗
CDRC-HP	an HP flavor of CDRC [1, 2]	✗

(b) List of evaluated data structures. ✓in "*optimistic?*" means that the underlying traversals are optimistic.

data structure	description	optimistic?	lock-free?
HashMap	a chaining hash table [34]	✓	✓
SkipList	a skip list [47] with wait-free get()[10]	✓	✓
NMTree	an external binary tree [39]	✓	✓
ElimAbTree	an (a,b) tree with elimination [52]	✓	✗
HHSList	Harris's list [16] with wait-free get() [47][10]	✓	✓
HMList	Michael list [34]	✗	✓
EFRBTree	Ellen et al. [12]'s external binary tree	✗	✓

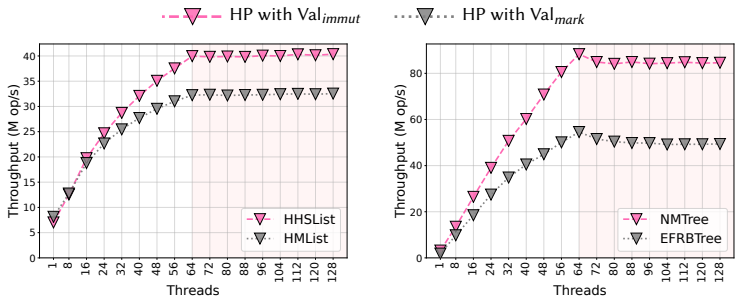
Fig. 13. Summary of evaluated reclamation algorithms and data structures.

Methodology. We measure the throughput (operations per second) and the peak memory usage for **(1)** varying numbers of threads: up to twice the number of hardware threads, 128 for AMD64T and 192 for INTEL96T; **(2)** three types of workloads: read-intensive (90% reads and 10% writes), read-write (50% reads and 50% writes), and write-only (50% inserts and 50% deletes); and **(3)** a fixed time: 10 seconds. Each thread repeatedly calls get(), insert(), and remove() randomly. The key ranges are 1K or 10K for lists, and 100K or 100M for others. Data structures are pre-filled to 50%.

Results. We discuss the results in a per-reclamation-algorithm basis, as most data structures exhibit similar trends. An exception is ElimAbTree (Fig. 15d), for which all applicable reclamation algorithms perform similarly because performance is dominated by lock contention, rendering protection overhead insignificant. This significantly degrades performance in oversubscribed scenarios, represented by red-shaded areas, far more than in other data structures.

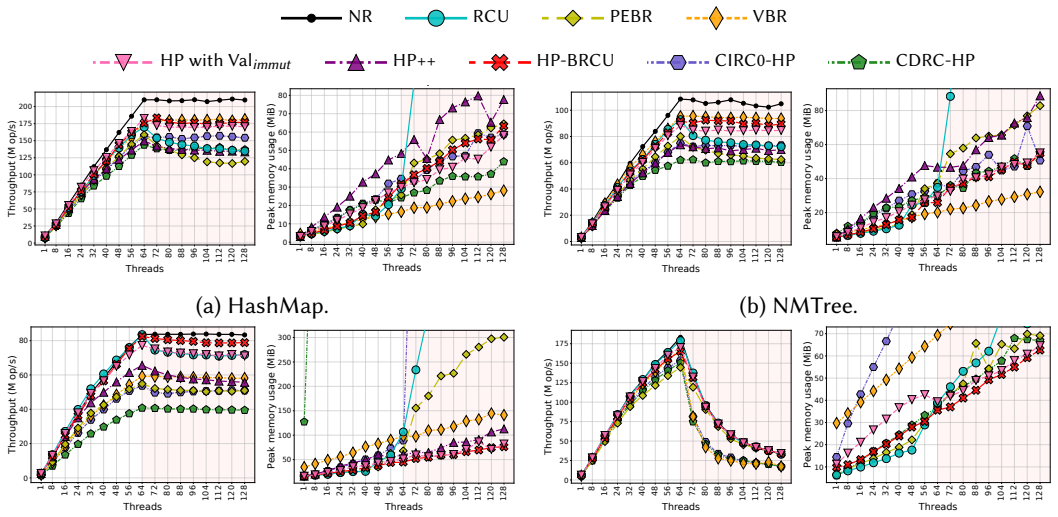
HP with Val$_{immut}$ greatly benefits from optimistic traversal enabled. Fig. 14 compares the maximum throughput that can be achieved in lists and trees, from those using HP with Val$_{mark}$ (HMList and EFRBTree) to those using HP with Val$_{immut}$ (HHSList and NMTree). For data structures with optimistic traversal outperform those without optimistic traversal by a large margin, by up to 24% for lists and 61% for trees.

Overall, HP with Val$_{immut}$ outperforms HP++, PEBR, CIRC0-HP and CDRC-HP and is competitive with HP-BRCU and VBR. Fig. 15 presents the throughput and memory usage of reclamation algorithms on read-write workloads for various data structures. Throughput steadily increases with the number of threads until oversubscription causes performance degradation. Memory usage remains linear with the number of threads for robust algorithms, while it increases dramatically for non-robust ones, especially during oversubscription.



(a) Lists with key range 100. (b) Trees with key range 100K.

Fig. 14. Throughput for a varying number of threads of read-write workloads for list and tree.



(a) HashMap. (b) NMTree.

(c) SkipList. (d) ElimAbTree.

Fig. 15. Throughput and peak memory usage for a varying number of threads on read-write workloads with key range 100K.

We now explain noteworthy points about other reclamation algorithms. The baseline NR and RCU generally outperform other reclamation algorithms, as NR simply does not reclaim at all and RCU uses coarse-grained protection to amortize overhead. In HashMap, HP with Val$_{immut}$ outperforms RCU in (near-)oversubscribed scenarios because **(1)** the traversal of HashMap is short, so the per-node protection overhead of HP is insignificant, and **(2)** RCU has more unreclaimed nodes to attempt reclamation on, significantly increasing its overhead.

HP-BRCU generally performs well, as its per-node protection overhead is amortized by BRCU-expedited traversal. Unlike RCU, it bounds unreclaimed nodes by restarting stalled threads, making it fast in HashMap. However, HP-BRCU is difficult to apply to data structures due to its complex requirements (see §6).

VBR generally performs well, as it uses a memory allocator that minimizes allocation overhead by not returning memory to the OS. However, it underperforms in SkipList due to the high frequency of updates, which is expensive in VBR as it uses multi-word CAS. Memory usage grows strictly linear across both under- and oversubscription scenarios as VBR preallocates memory based on the number of threads and the size of a node in the data structure, and never actually frees them.

HP++, PEBR, CIRC0-HP and CDRC-HP exhibit low throughout due to the cost of additional mechanisms to support optimistic traversal. Those mechanisms include: **(1)** frontier protections and invalidation for HP++, **(2)** epoch management and ejection algorithm for PEBR, and **(3)** reference counting for CIRC0-HP and CDRC-HP. HP++ is not applicable to ElimAbTree because it requires marking to use a node's next pointer field's LSB, but ElimAbTree's marking happens in a separate field (§2.4). PEBR is robust but has a memory footprint similar to non-robust RCU. PEBR's robustness relies on timely restarting stalled threads, but in practice, restarts are uncommon as frequent restarts reduce throughput. CIRC0-HP and CDRC-HP are not robust because they rely on deferred reference counting, which slows down the reclamation of long unreachable chains[12], which is especially prevalent in SkipList.

6 Related and Future Work

In §2.4, we discussed prior approaches to applying HP to optimistic traversal. We now discuss other related work.

The ERA theorem for reclamation algorithms. Sheffi and Petrank [49, 50] proved the *ERA theorem*: reclamation algorithms cannot simultaneously achieve ease of integration, robustness, and applicability. A reclamation algorithm is considered easy to integrate if it provides an API that only needs to replace existing memory accesses or be placed at the start and end of operations, is considered robust if it bounds the number of retired but unreclaimed nodes, and applicable if it can be used with "plain implementations", which are data structures without reclamation but with proper calls to the retire function, as if a reclamation algorithm were in use. The key idea of the proof is that Harris's list with reclamation algorithms is either not robust or not easy to integrate.

The authors argue that HP satisfies ease of integration and robustness, making it inapplicable to Harris's list. However, we applied HP to Harris's list (§3.1). The discrepancy arises from how HP is used. The version of HP that is easy to integrate performs protection using a dedicated protect function, which executes a load-validation loop on a single pointer. protect is restricted to Val_{mark}, rendering it incompatible with Harris's list (§2.3). On the other hand, HP with Val_{immut} requires additional memory accesses (*e.g.*, line 11 of Algorithm 2), thus does not satisfy ease of integration.

Robust and applicable reclamation algorithms. PEBR [26], DEBRA+ [4], NBR+ [51], and HP-BRCU [28] are hybrids of HP's per-pointer protection and RCU's coarse-grained protection. For robustness, they *neutralize* threads that block reclamation, *i.e.*, forcefully restart them. Once neutralized, threads must halt their operations and execute custom recovery code. DEBRA+, NBR+, and HP-BRCU use POSIX signals [32] for neutralization, while PEBR marks the thread, allowing it to neutralize itself when attempting to protect a node. These algorithms can optimistically traverse data structures because they neutralize threads when further traversal becomes unsafe.

However, PEBR, DEBRA+, and NBR+ employ a coarse-grained neutralization strategy, significantly degrading performance during long-running operations [28].

[12]This limitation was previously identified by [21], who proposed *immediate recursive destruction* to address the slow reclamation issue of deferred reference counting. However, they applied their technique only to CIRC-EBR, an EBR-based variant of their reference counting approach, and it remains unclear how to effectively extend it to their HP-based variant.

In contrast, HP-BRCU uses an efficient fine-grained signaling approach that neutralizes only the threads causing stalls. However, HP-BRCU requires data structures to be "abort-rollback-safe" and requires significant expertise to ensure correctness. A traversal is abort-rollback-safe if repeated attempts do not alter its semantics and if aborting mid-execution does not introduce safety or liveness issues. For example, traversals must not (de)allocate memory, acquire locks, perform updates (except for helping), or retire nodes. To apply HP-BRCU to data structures that perform updates during traversal [11, 34, 47], one uses *abort masks*, which partition a traversal into abort-rollback-safe sections to guarantee protection. Abort masking and its recovery process must be implemented carefully to prevent introducing bugs.

VBR [48] is an optimistic algorithm that permits access to retired nodes, with a subsequent verification step to ensure the validity of the access. To enable optimistic writes, VBR assigns a version number to each mutable node, atomically updating both the version number and the node's other fields using a wide CAS. If an outdated version number is detected, the operation fails and requires custom recovery. This approach allows VBR to support optimistic traversal. However, VBR requires a type-safe memory allocator that does not return memory to the OS; and the data that can be stored is restricted to types supporting optimistic access, such as volatile or atomic types.

Reference counting algorithms. Reference counting algorithms such as CDRC [2] and CIRC [21] aim to provide a safe API for concurrent memory management at the cost of performance. Compared to manual reclamation algorithms where clients need to manually protect and retire nodes, reference counting algorithms allow unrestricted access to nodes and automatically reclaims memory when it is no longer in use, making them much less error-prone. However, reference counting algorithms are typically slower than manual reclamation algorithms due to their updates on reference counters, which are internally used to track when to free nodes (§5).

Theoretical comparison of optimism level. Reclamation algorithms that support optimistic traversal differ in the amount of restarts required during traversal.

The most optimistic algorithms include RCU [33], reference counting, and tracing-based garbage collection, which can traverse detached nodes and do not restart during traversal. These algorithms use coarse-grained protection, protecting all nodes at once.

The intermediate algorithms include neutralization-based methods [4, 26, 28, 51], VBR [48], and HP++ [23], which can also traverse detached nodes but may require restarts. HP++ may fail to protect detached nodes if the target node is invalidated. Neutralization-based methods require a restart if the thread is neutralized. VBR's protection fails if the node's version number is outdated.

The least optimistic algorithm is HP with Val_{immut}, which can traverse marked but reachable nodes but cannot traverse detached nodes. However, this does not imply that HP is slower. We observed that HP with Val_{immut} outperforms HP++, PEBR, and reference counting, and is competitive with RCU and VBR because the protection overhead is more critical than the level of optimism (§5).

Future work: applying immutability-based validation to other reclamation algorithms. We plan to apply immutability-based validation to prior work that has reduced HP's per-node protection overhead but did not increase applicability. Hazard Eras [46], IBR [54], and Hyaline [42] use an internal epoch for validation to amortize costs across multiple protections. Wait-free Eras [41] and Crystalline [43] extend these algorithms with better progress guarantees using multi-word CAS. None of these algorithms have been thought to be efficiently applicable to optimistic traversal because their core protection mechanism is similar to HP. We believe this limitation can be addressed by leveraging immutability.

Acknowledgments

We thank the PLDI 2025 reviewers for their valuable feedback, which greatly improved the presentation of the paper. This work is supported by Institute for Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) partly under the project (No. RS-2024-00459026, Energy-Aware Operating System for Disaggregated System, 80%), partly under the Graduate School of Artificial Intelligence Semiconductor (No. IITP-2025-RS-2023-00256472, 10%), and partly under the Information Technology Research Center(ITRC) support program (No. IITP-2025-RS-2020-II201795, 10%)

Data Availability Statement

The implementation, proofs, and appendix for this paper can be found open-sourced at [30].

References

[1] 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. doi:10.1145/3453483.3454060

[2] 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. doi:10.1145/3519939.3523730

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

[4] 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. doi:10.1145/2767386.2767436

[5] Windows Dev Center. 2025. FlushProcessWriteBuffers function. https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-flushprocesswritebuffers

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

[7] M. Desnoyers, P. E. McKenney, A. S. Stern, M. R. Dagenais, and J. Walpole. 2012. User-Level Implementations of Read-Copy Update. *IEEE Transactions on Parallel and Distributed Systems* 23, 2 (2012), 375–382. doi:10.1109/TPDS.2011.159

[8] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management* (Santa Barbara, CA, USA) *(ISMM 2016)*. Association for Computing Machinery, New York, NY, USA, 36–45. doi:10.1145/2926697.2926699

[9] 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

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

[11] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. 2014. The Amortized Complexity of Non-Blocking Binary Search Trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (Paris, France) *(PODC '14)*. Association for Computing Machinery, New York, NY, USA, 332–340. doi:10.1145/2611462.2611486

[12] 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. doi:10.1145/1835698.1835736

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

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

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

[16] 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.

[17] Maurice Herlihy and Nir Shavit. 2011. On the Nature of Progress. In *Principles of Distributed Systems*, Antonio Fernàndez Anta, Giuseppe Lipari, and Matthieu Roy (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–328.

[18] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. doi:10.1145/78969.78972

[19] Shane V. Howley. 2012. *Lock-free internal binary search trees with memory management.* Ph. D. Dissertation. Trinity College Dublin, Ireland. https://hdl.handle.net/2262/77623

[20] 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. doi:10.1145/2312005.2312036

[21] 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. doi:10.1145/3656383

[22] Jaehwang Jung, Janggun Lee, Jaemin Choi, Jaewoo Kim, Sunho Park, and Jeehoon Kang. 2023. Modular Verification of Safe Memory Reclamation in Concurrent Separation Logic. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 251 (oct 2023), 29 pages. doi:10.1145/3622827

[23] 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. doi:10.1145/3558481.3591102

[24] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. doi:10.1017/S0956796818000151

[25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM, 637–650. doi:10.1145/2676726.2676980

[26] 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. doi:10.1145/3385412.3385978

[27] Max Khizhinsky. 2024. CDS C++ library. https://github.com/khizmax/libcds.

[28] 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. doi:10.1145/3626183.3659941

[29] Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. *SIGPLAN Not.* 52, 1 (Jan. 2017), 205–217. doi:10.1145/3093333.3009855

[30] Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2025. Leveraging Immutability to Validate Hazard Pointers for Optimistic Traversals (artifact and appendix). doi:10.5281/zenodo.15183251 Project webpage: https://cp.kaist.ac.kr/gc.

[31] Linux Programmer's Manual. 2025. membarrier(2) - Linux manual page. http://man7.org/linux/man-pages/man2/membarrier.2.html

[32] Linux Programmer's Manual. 2025. signal(7) — Linux manual page. https://man7.org/linux/man-pages/man7/signal.7.html

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

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

[35] 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. doi:10.1145/571825.571829

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

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

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

[39] 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. doi:10.1145/2555243.2555256

[40] Aravind Natarajan, Arunmoezhi Ramachandran, and Neeraj Mittal. 2020. FEAST: A Lightweight Lock-free Concurrent Binary Search Tree. *ACM Trans. Parallel Comput.* 7, 2, Article 10 (May 2020), 64 pages. doi:10.1145/3391438

[41] 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

[42] 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. doi:10.1145/3453483.3454090

[43] 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. doi:10.1145/3658851

[44] 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. doi:10.1145/3141879

[45] 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. doi:10.1145/2684464.2684472

[46] 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. doi:10.1145/3087556.3087588

[47] 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.

[48] 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. doi:10.4230/LIPIcs.DISC.2021.35

[49] Gali Sheffi and Erez Petrank. 2022. The ERA Theorem for Safe Memory Reclamation. arXiv:2211.04351 [cs.DC] https://arxiv.org/abs/2211.04351

[50] 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. doi:10.1145/3583668.3594564

[51] 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. doi:10.1145/3437801.3441625

[52] Anubhav Srivastava and Trevor Brown. 2022. Elimination (a,b)-trees with fast, durable updates. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 416–430. doi:10.1145/3503221.3508441

[53] R.K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.co.kr/books?id=YQg3HAAACAAJ

[54] 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. doi:10.1145/3178487.3178488

A AMD64T Full Experimental Results

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

A.1.1 Write-only Workloads.

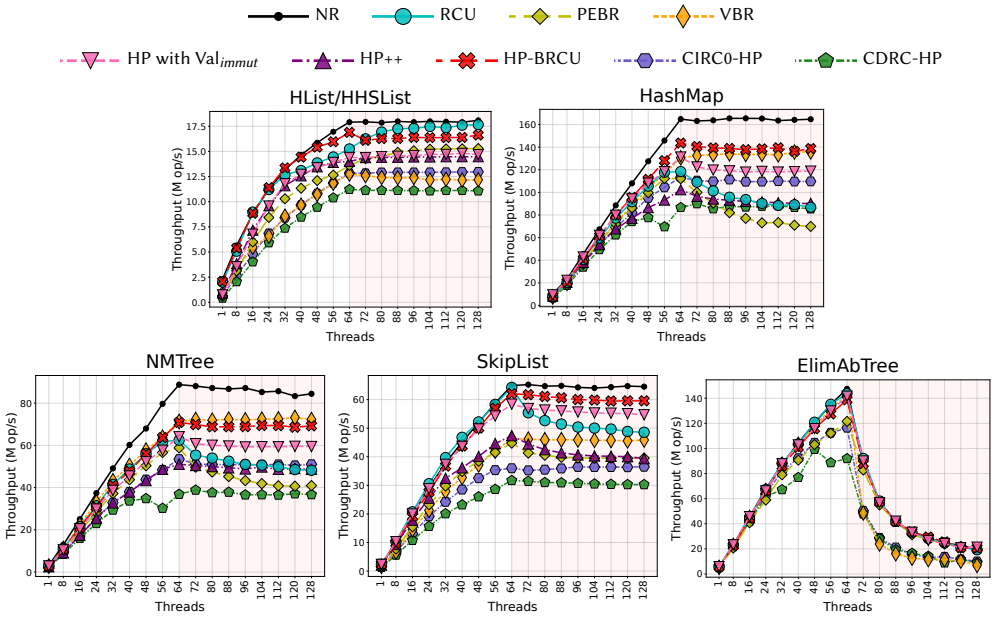


Fig. 16. Throughput (million operations per second) of write-only workloads for a varying number of threads.

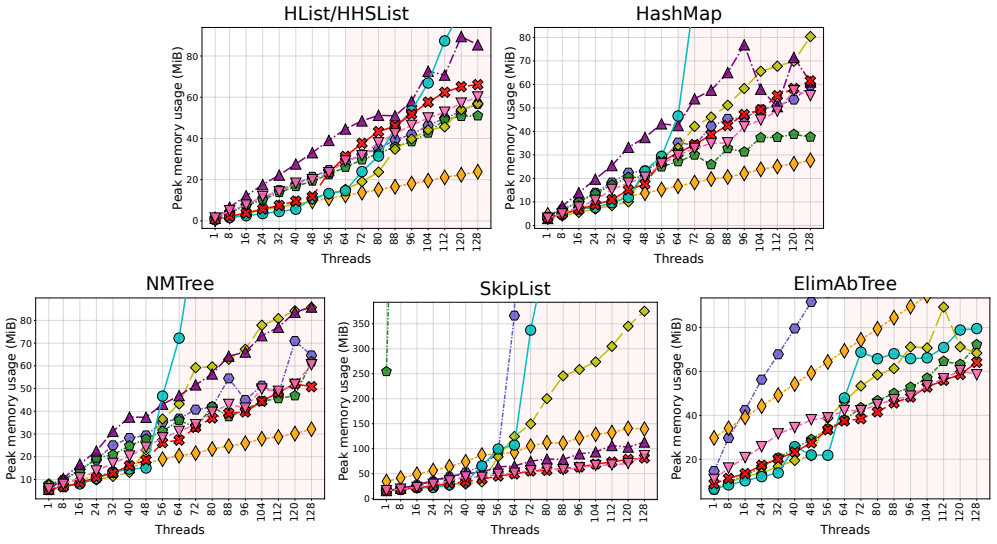


Fig. 17. Peak memory usage of write-only workloads for a varying number of threads.

A.1.2 Read-write Workloads.

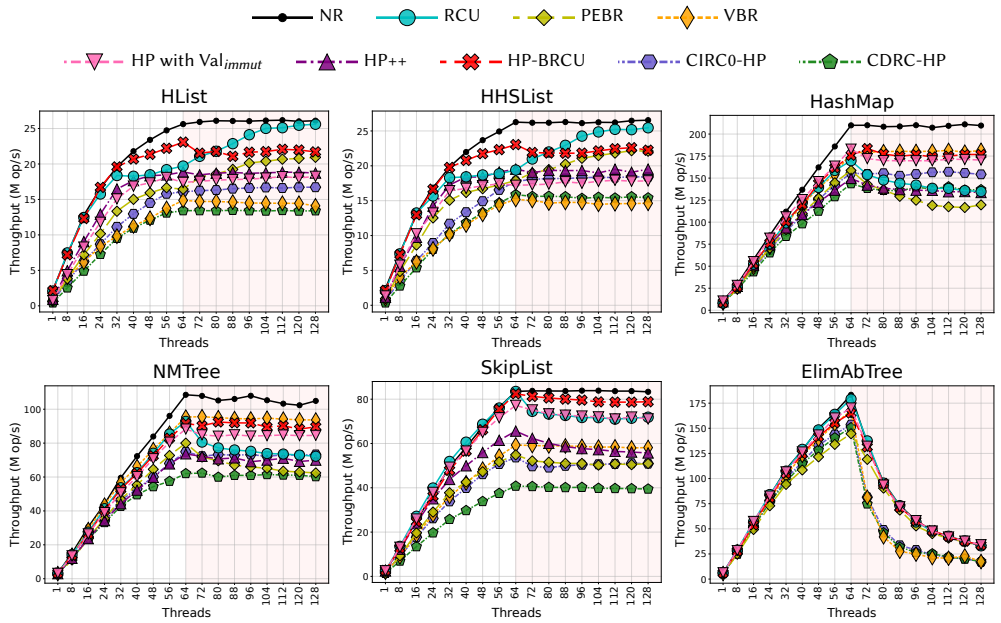


Fig. 18. Throughput (million operations per second) of read-write workloads for a varying number of threads.

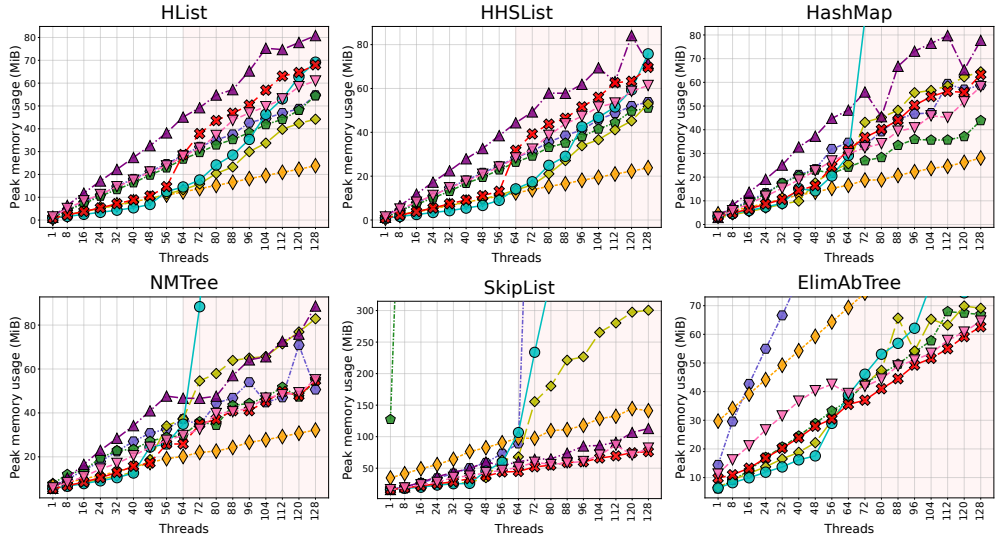


Fig. 19. Peak memory usage of read-write workloads for a varying number of threads.

A.1.3 Read-intensive Workloads.

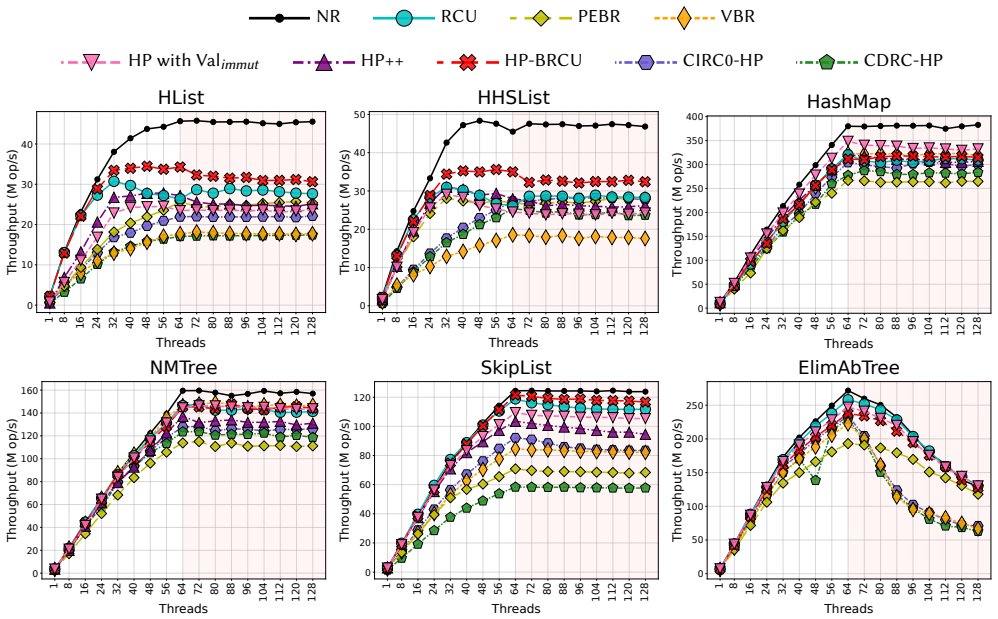


Fig. 20. Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

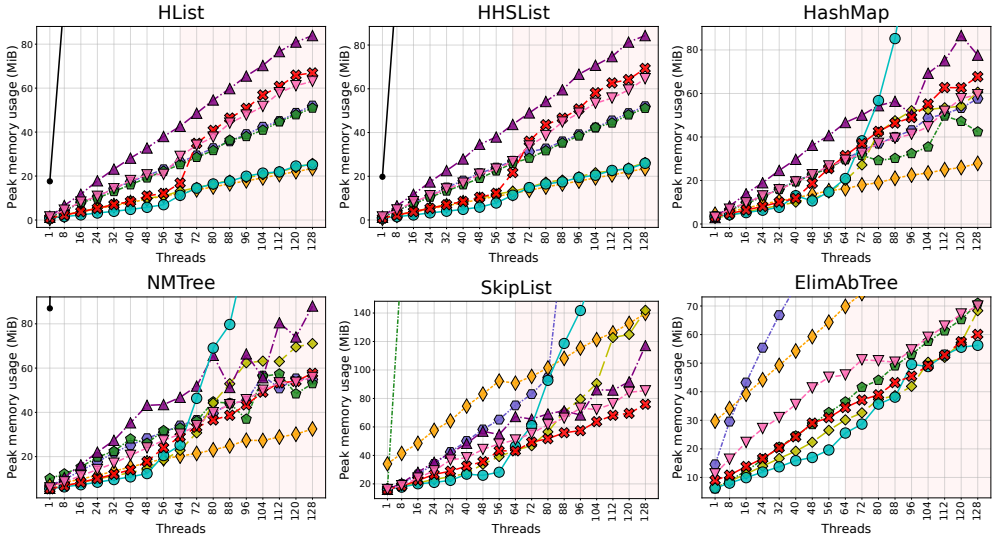


Fig. 21. Peak memory usage of read-intensive workloads for a varying number of threads.

A.2 Large Key Ranges (10K for Lists and 100M for Others)

A.2.1 Write-only Workloads.

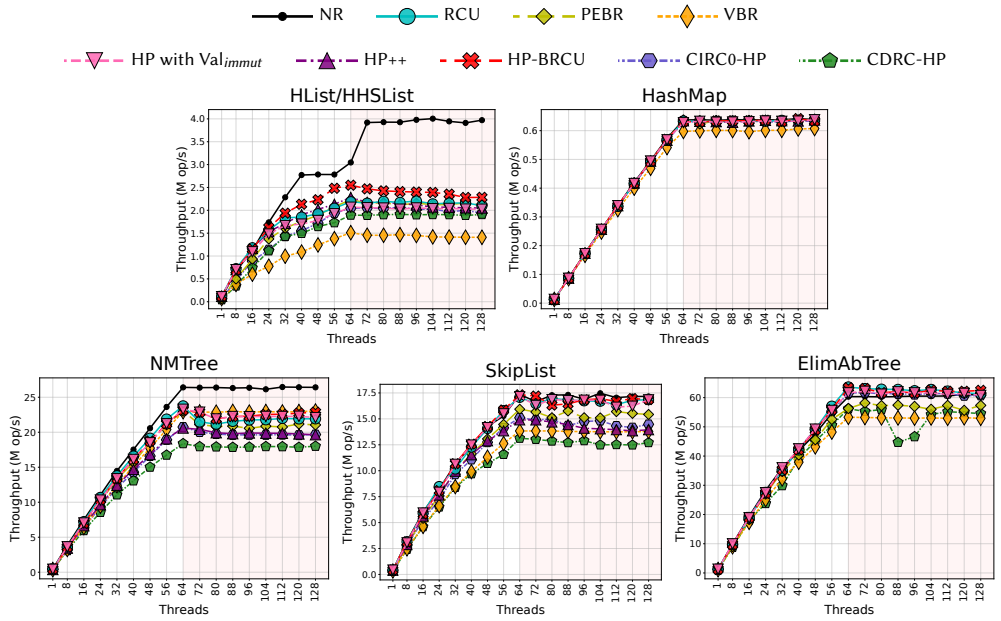


Fig. 22. Throughput (million operations per second) of write-only workloads for a varying number of threads.

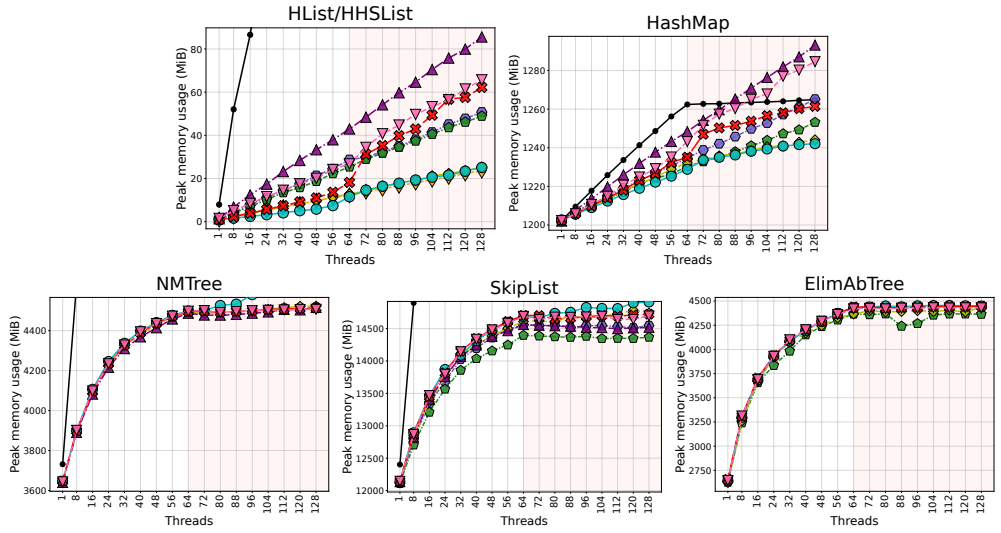


Fig. 23. Peak memory usage of write-only workloads for a varying number of threads.

A.2.2 Read-write Workloads.

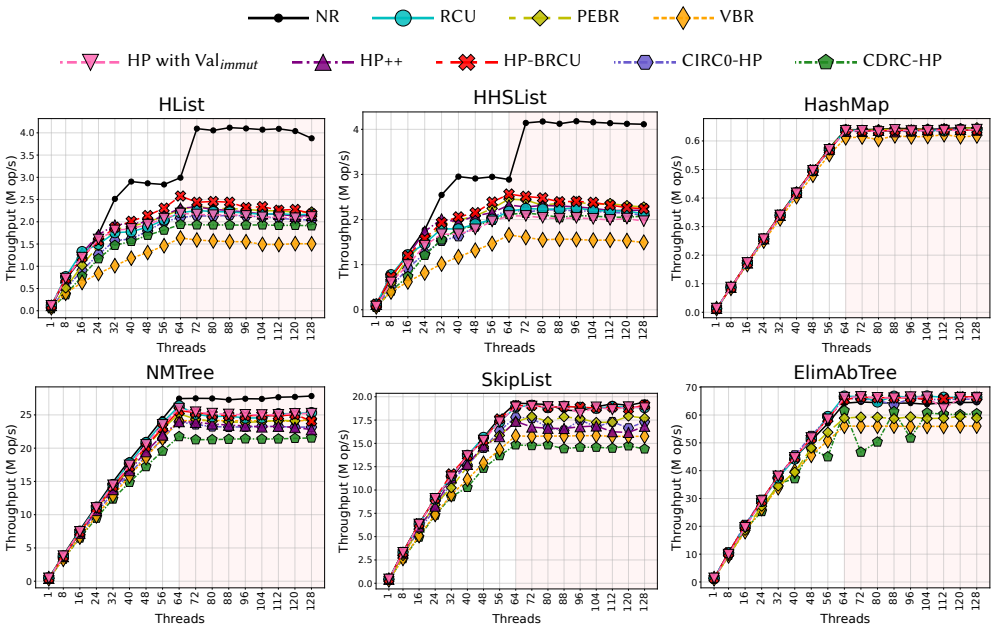


Fig. 24. Throughput (million operations per second) of read-write workloads for a varying number of threads.

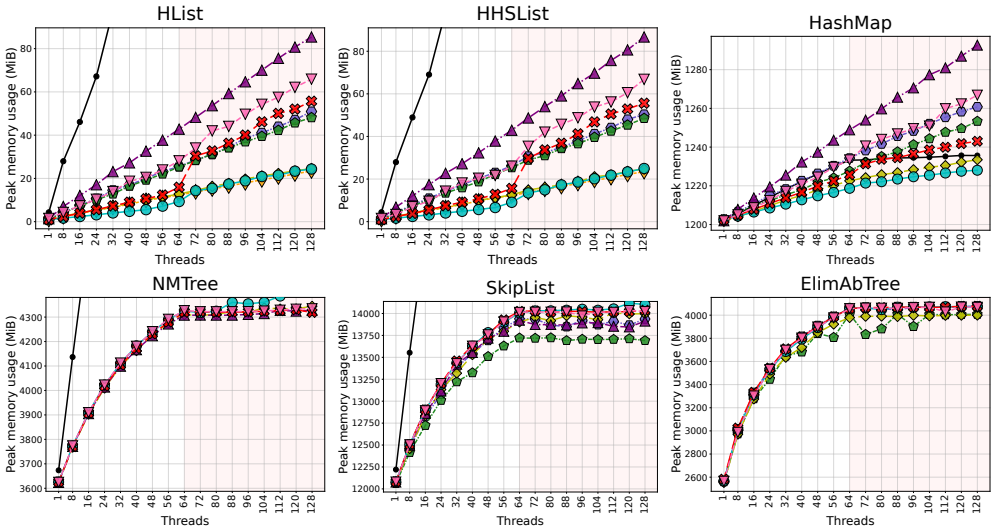


Fig. 25. Peak memory usage of read-write workloads for a varying number of threads.

A.2.3 Read-intensive Workloads.

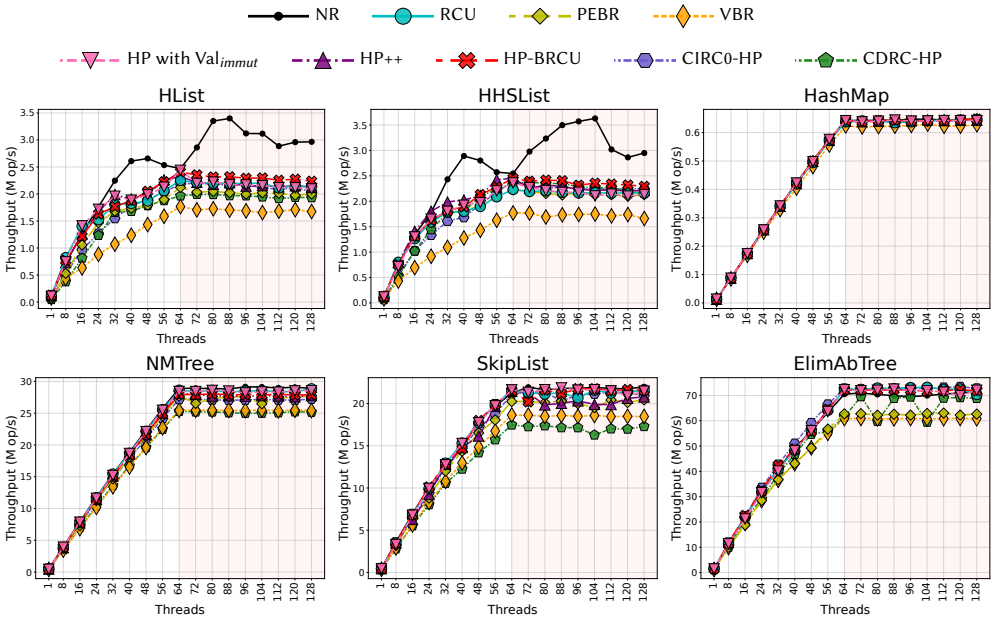


Fig. 26. Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

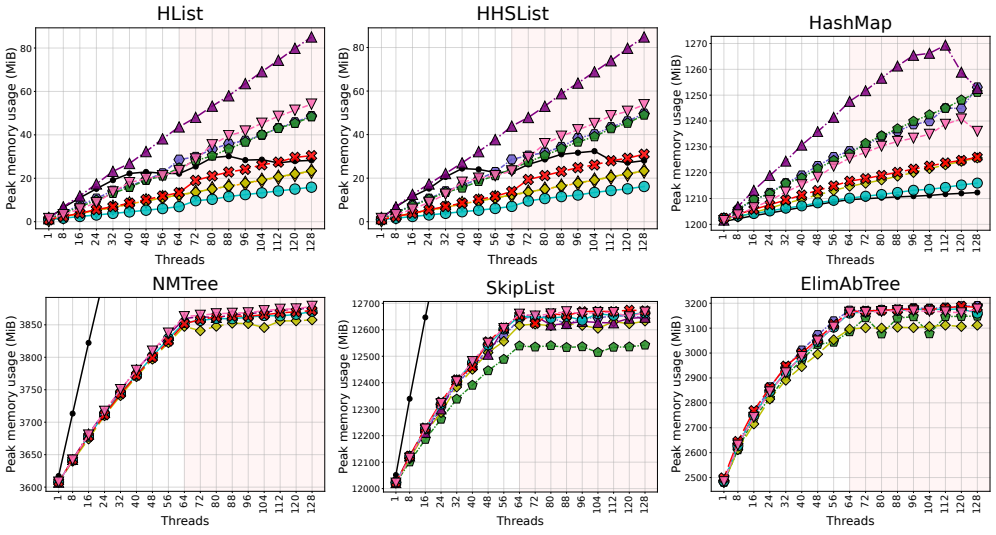
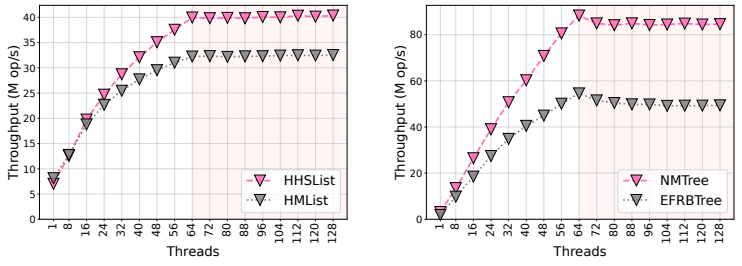


Fig. 27. Peak memory usage of read-intensive workloads for a varying number of threads.

A.3 Comparison of Traversals with and without Optimism



(a) Lists with key range 100. (b) Trees with key range 100K.

Fig. 28. Throughput (million operations per second) for a varying number of threads of read-write workloads for list and tree.

B Intel96T Full Experimental Results

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

B.1.1 Write-only Workloads.

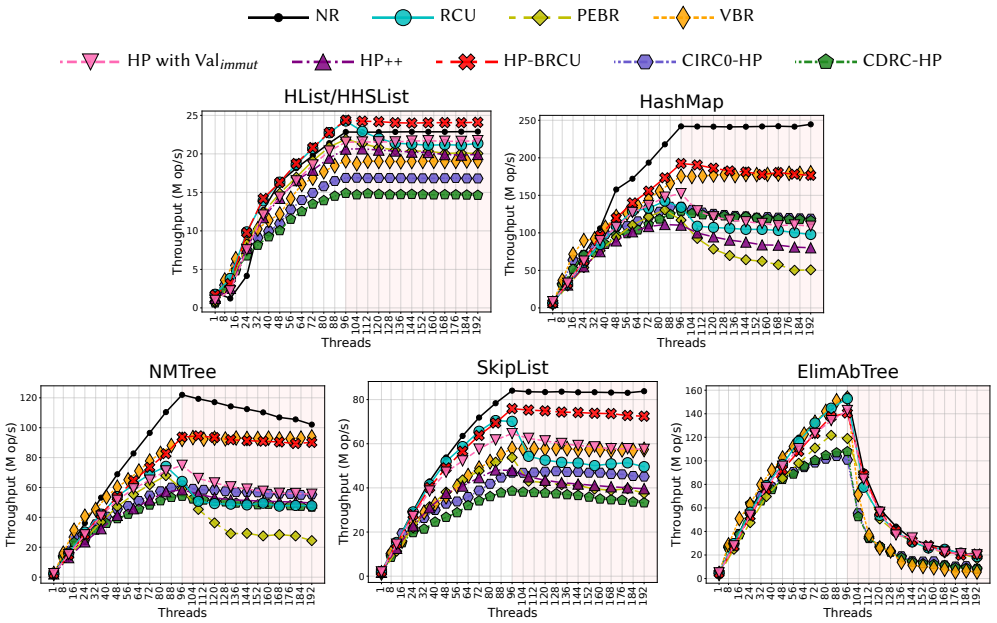


Fig. 29. Throughput (million operations per second) of write-only workloads for a varying number of threads.

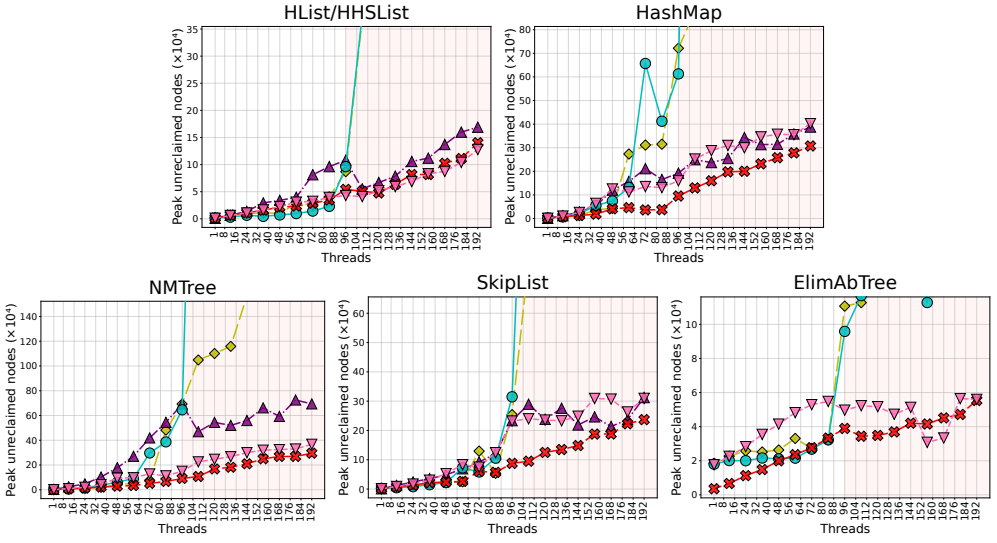


Fig. 30. Peak memory usage of write-only workloads for a varying number of threads.

B.1.2 Read-write Workloads.

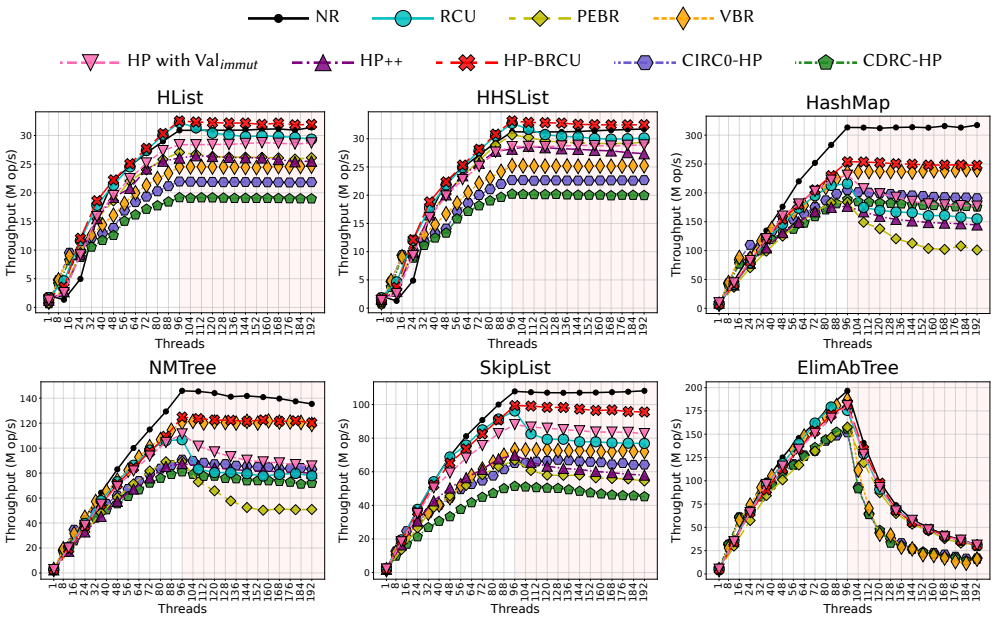


Fig. 31. Throughput (million operations per second) of read-write workloads for a varying number of threads.

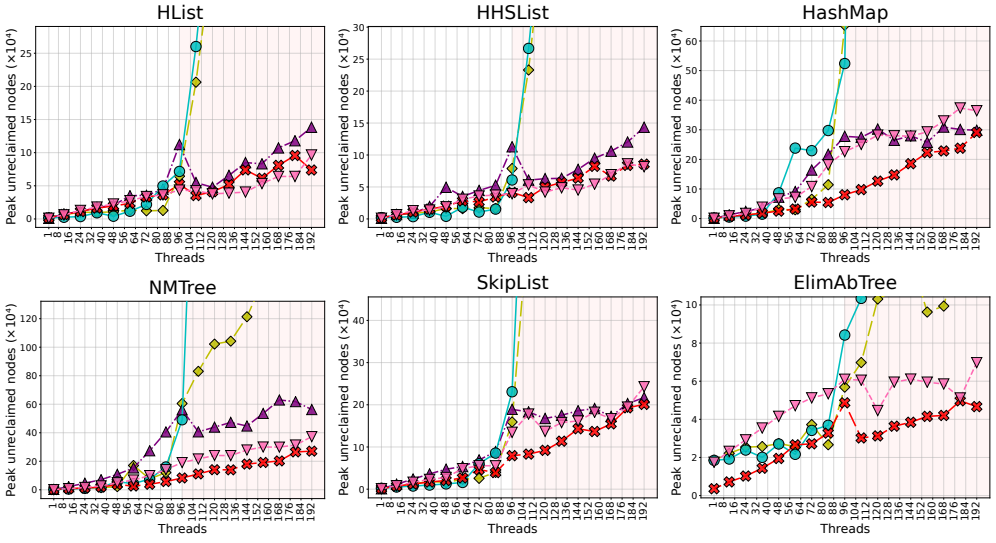


Fig. 32. Peak memory usage of read-write workloads for a varying number of threads.

B.1.3 Read-intensive Workloads.

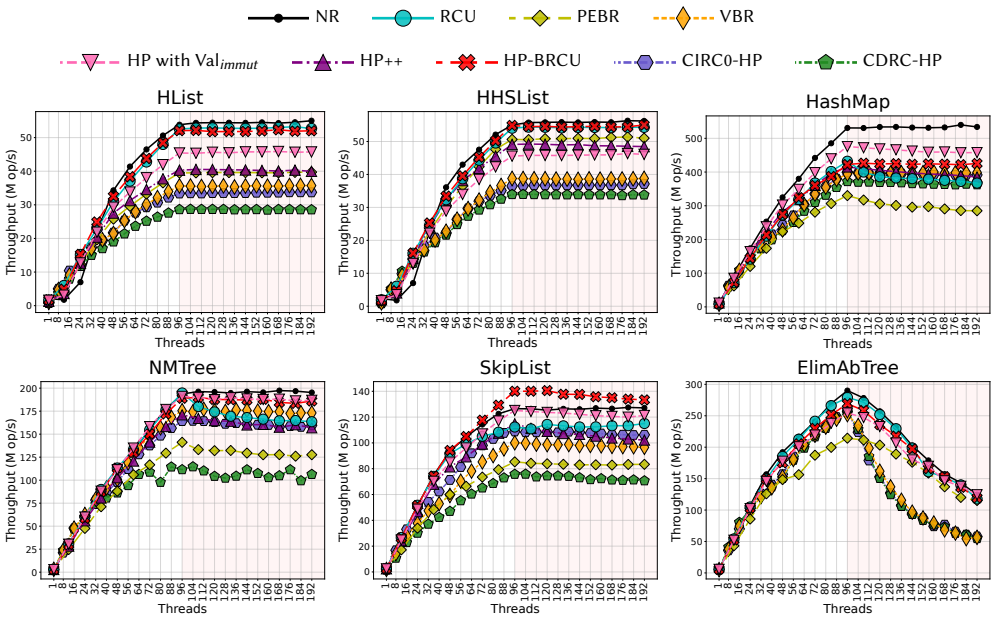


Fig. 33. Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

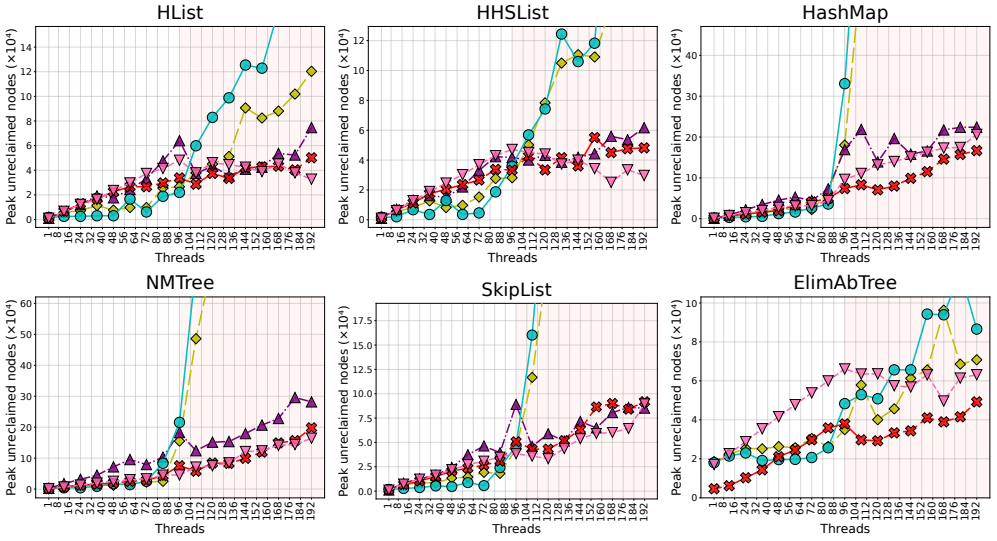


Fig. 34. Peak memory usage of read-intensive workloads for a varying number of threads.

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

B.2.1 Write-only Workloads.

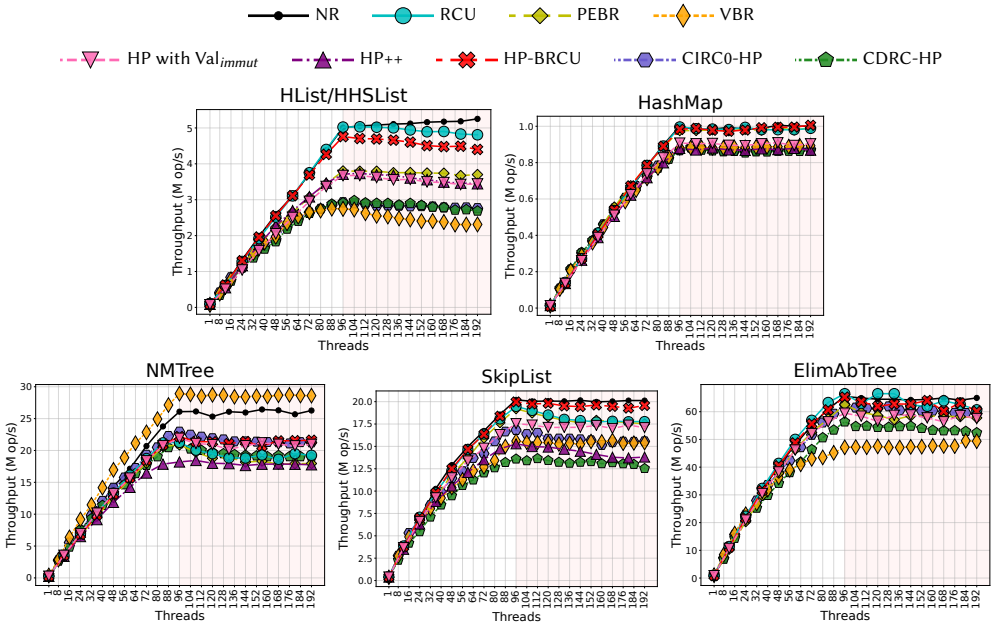


Fig. 35. Throughput (million operations per second) of write-only workloads for a varying number of threads.

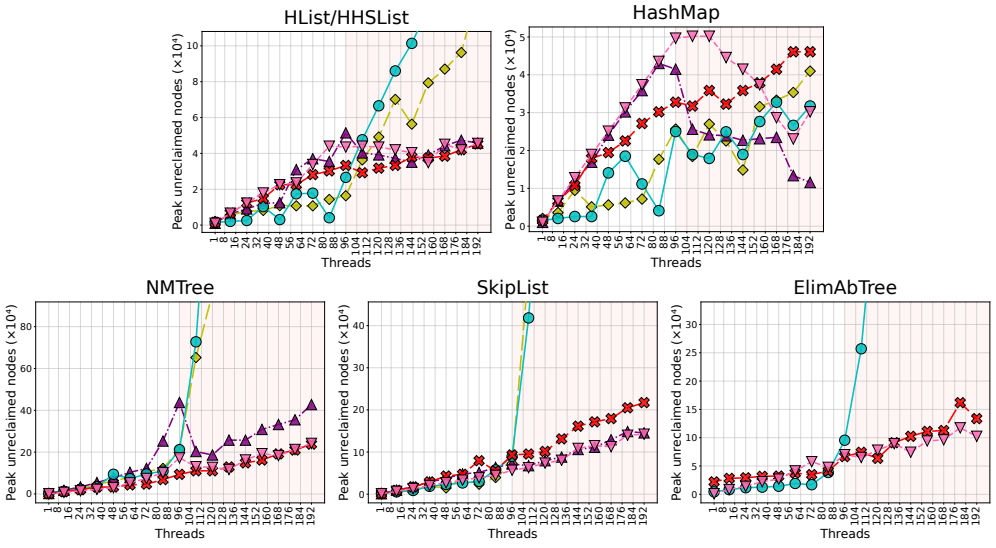


Fig. 36. Peak memory usage of write-only workloads for a varying number of threads.

B.2.2 Read-write Workloads.

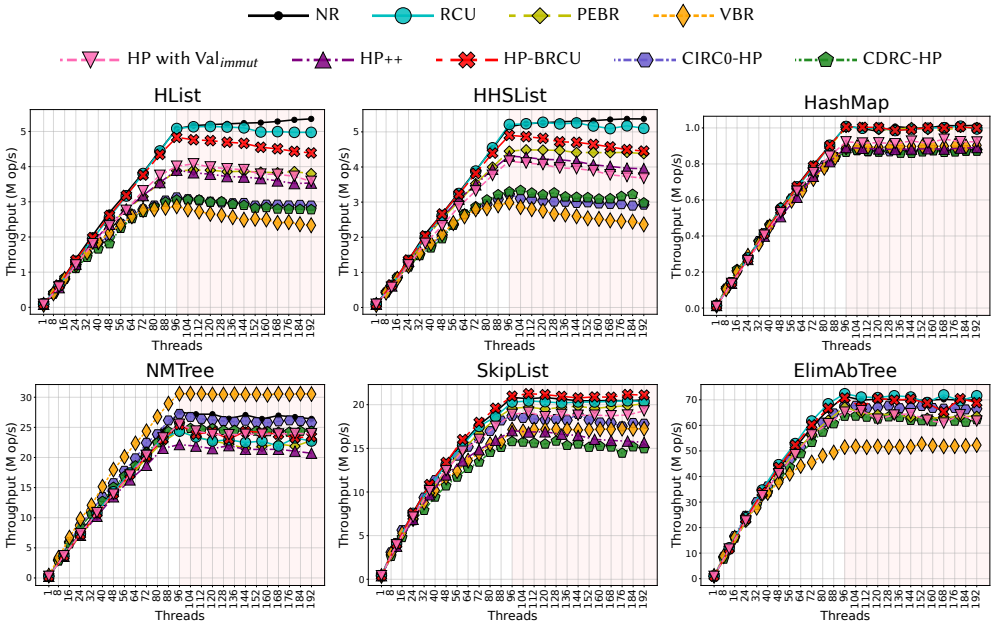


Fig. 37. Throughput (million operations per second) of read-write workloads for a varying number of threads.

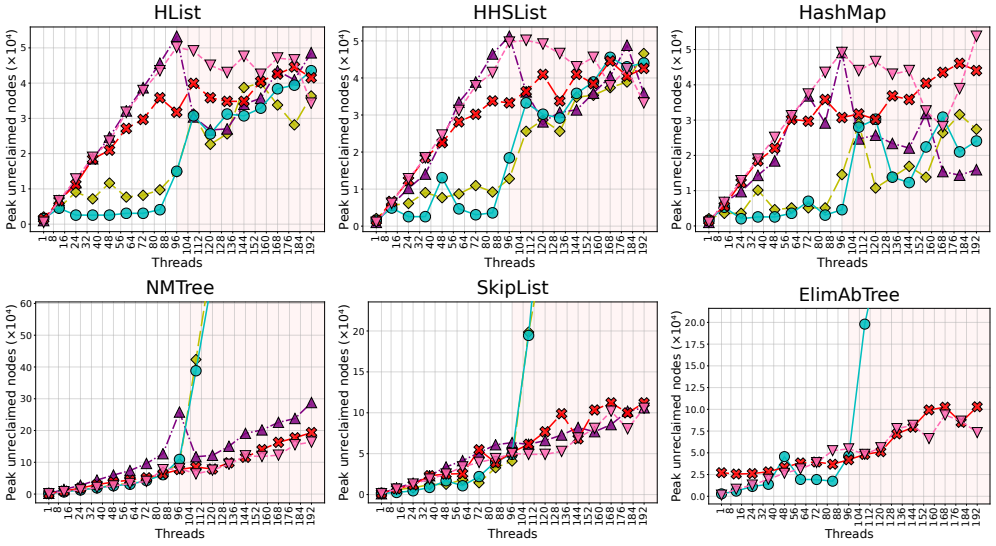


Fig. 38. Peak memory usage of read-write workloads for a varying number of threads.

B.2.3 Read-intensive Workloads.

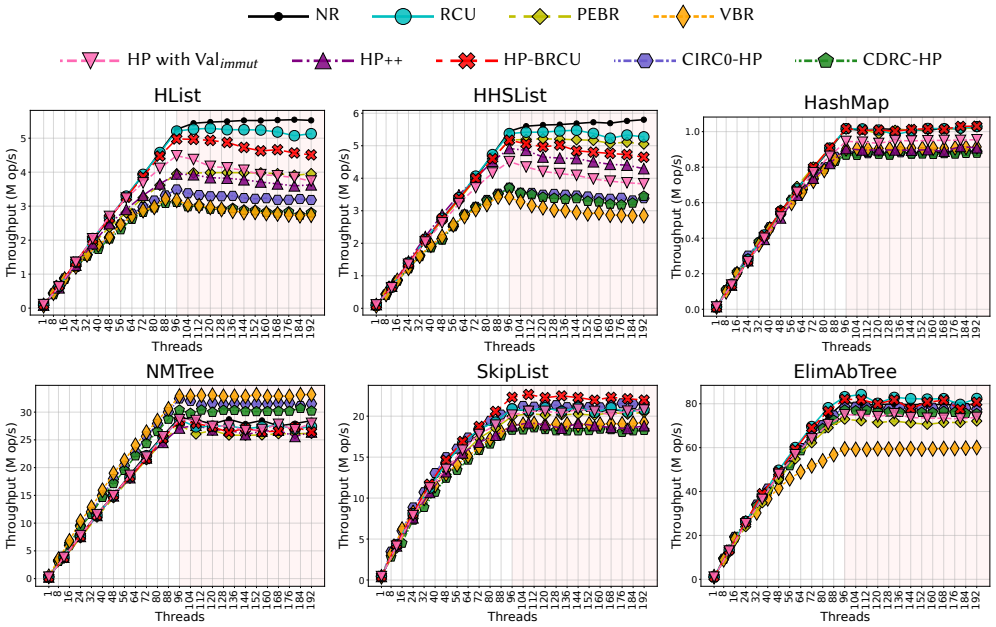


Fig. 39. Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

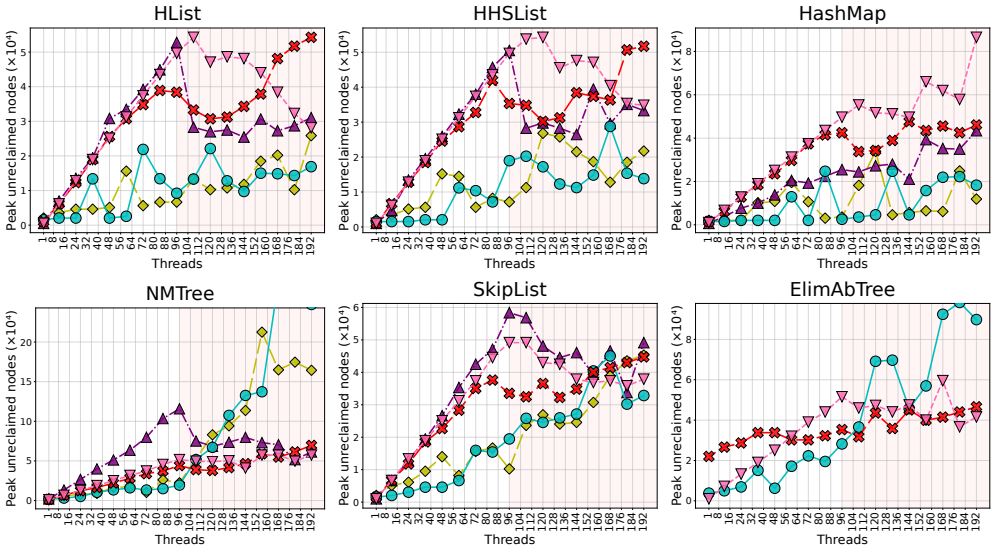
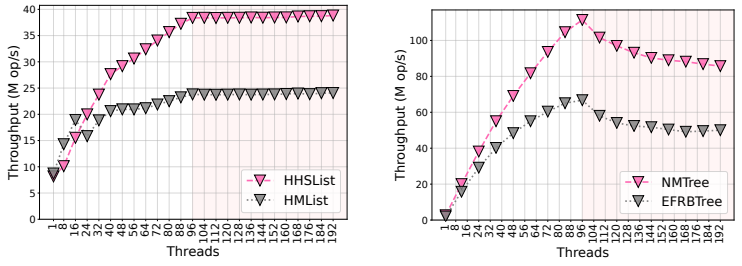


Fig. 40. Peak memory usage of read-intensive workloads for a varying number of threads.

B.3 Comparison of Traversals with and without Optimism



(a) Lists with key range 100. (b) Trees with key range 100K.

Fig. 41. Throughput (million operations per second) for a varying number of threads of read-write workloads for list and tree.

Received 2024-11-13; accepted 2025-03-06