Applying Hazard Pointers to More Concurrent Data Structures

Jaehwang Jung
jaehwang.jung@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Janggun Lee
janggun.lee@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Jeonghyeon Kim
jeonghyeon.kim@cp.kaist.ac.kr
KAIST
Daejeon, Republic of Korea

Jeehoon Kang
jeehoon.kang@kaist.ac.kr
KAIST
Daejeon, Republic of Korea

ABSTRACT

Hazard pointers is a popular semi-manual memory reclamation scheme for concurrent data structures, where each accessing thread announces the protection of each object to access and validates that the pointer is not already freed. Validation is typically done by *over-approximating* unreachability: if an object seems to be unreachable from the root of the data structure, the protecting thread decides not to access the object as it might have been freed. However, many efficient data structures are incompatible with validation by over-approximation as their *optimistic traversal* strategy intentionally ignores the warning of unreachability to achieve better performance.

We design HP++, an extension to hazard pointers that supports optimistic traversal. The key idea is *under-approximating* unreachability during validation and *patching up* the potentially unsafe accesses arising from false-negatives. Thanks to optimistic traversal, data structures with HP++ outperform the same-purpose data structures with HP under contention while consuming a similar amount of memory.

CCS CONCEPTS

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

KEYWORDS

concurrency, memory management, hazard pointers

ACM Reference Format:
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 (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 20 pages. https://doi.org/10.1145/3558481.3591102

1 INTRODUCTION

Designing concurrent data structures is notoriously hard, and memory reclamation makes it even worse: programmers should synchronize memory accesses with not only other accesses but also reclamations of memory blocks. Without proper synchronization, data structures would incur safety errors such as use-after-free and more subtle ABA problems [57, 59].

To synchronize memory accesses and reclamations without significant overhead in performance and programmability, various semi-manual memory reclamation schemes for concurrent data structures have been proposed [1, 2, 11, 28, 30, 39, 42, 43, 45, 46, 54, 56]. The clients of these schemes *retire* no-longer-used nodes instead of immediately reclaiming them, and the scheme deallocates retired nodes only when it can prove that no threads will access the nodes.

Hazard pointers (HP) [45, 46], a popular and one of the earliest reclamation schemes, requires each thread to explicitly announce *protection* of each object before accessing. But announcement alone does not guarantee the safety of access, because another thread may have concurrently retired and reclaimed the same object. Therefore, the thread must *validate* that the protected pointer is not already freed. Since it is impossible to precisely decide whether a pointer is freed, validation is done in a conservative manner that regards the pointer as unsafe to access if it *might* have been freed. That is, validation *over-approximates* the freed pointer set. Specifically, if an object is detached (made unreachable) from the data structure, it might have been retired and freed. In practice, the unreachable pointer set is further over-approximated, because it is still inefficient to precisely check the reachability of objects. For example, Harris-Michael list [44] uses the logical deletion mark of a node to over-approximate the unreachability of the next node (see §2.2).

Problem. HP is not applicable to many high-performance data structures [9, 24, 30, 34, 36, 48, 50] that *optimistically* traverse the nodes. Their traversal follows the link to the next node even when the next node might have been detached from the data structure. This is fundamentally incompatible with HP's validation by over-approximating unreachability. Therefore, to use HP, the data structure must forgo optimistic traversal. As a result, HP-compatible data structures are significantly outperformed by the same-purpose data structures utilizing optimistic traversal [16] (see §5 for our experimental evaluation).

The reclamation schemes supporting optimistic traversal, however, exhibit different trade-offs. RCU [42, 43] and EBR [28, 30] are not *robust* in that a single non-cooperative thread renders the number of retired and yet unreclaimed pointers unbounded. NBR [56] relies on a non-local jump that mandates a specific structure of data structure operations. NBR and PEBR [39] do not properly support

SPAA '23, June 17–19, 2023, Orlando, FL, USA

ACM ISBN 978-1-4503-9545-8/23/06.
https://doi.org/10.1145/3558481.3591102

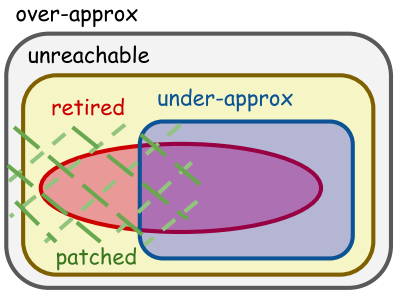


Figure 1: HP++ allows accessing the objects that are not in an under-approximation of unreachable object set, and patches up the unsafe accesses to objects that are retired but not in the under-approximation.

Algorithm 1 Abstract hazard pointers.

```
1: function Protect(p)
2:     Announce protection of p.
3:     Check if p is not retired. If retired, fail.
4: function Reclaim(p)
5:     Announce retirement of p.
6:     Check if p is protected. If protected, retry later.
```

long-running operations because they are likely to be forcefully terminated. VBR [54] requires a custom allocator that preserves the type of memory blocks and maintains per-block metadata for fully optimistic access to possibly reclaimed objects.

Solution. We design HP++, a backward-compatible extension for HP to improve applicability by enabling optimistic traversal. The key idea is *under-approximating* the unreachability in validation to allow optimistic traversal by letting the deleter mark the node *after* detaching, and *patching up* the potentially unsafe accesses arising from false-negatives by letting the deleter protect such pointers (Figure 1). Thanks to optimistic traversal, data structures with HP++ outperform the same-purpose data structures with HP under contention while consuming a similar amount of memory.

Outline. We discuss background and motivation (§2), present our design (§3), analyze its safety, applicability, progress, and robustness (§4), evaluate its performance (§5), and discuss related work (§6).

2 BACKGROUND AND MOTIVATION

We review hazard pointers (§2.1), its protection validation strategy based on the over-approximation of unreachability (§2.2), and its drawback that it does not support efficient concurrent data structures using optimistic traversal (§2.3). We also review the drawbacks of other reclamation schemes supporting optimistic traversal (§2.4).

2.1 Hazard Pointers

Hazard pointers (HP) [45, 46], a popular and one of the earliest reclamation schemes, guarantee safety by avoiding the reclamation of pointers that are explicitly announced to be in use. Algorithm 1 outlines the high-level architecture of HP.

```
1: loop
2:     h ← head.load()
3:     hp.set(h)
4:     if h ≠ head.load() then continue
5:     n ← (*h).next.load()
6:     ...
```

Figure 2: Pop function of Treiber's stack with HP.

On the one hand, a thread attempting to access a pointer, say p, announces that p is a *hazard pointer*. But announcement alone does not guarantee the safety of access because another thread may have concurrently retired and reclaimed the same pointer beforehand. To ensure that p is safe to access, the thread must *validate* the protection by checking that p is not already retired. If the check fails, the thread should recover from the failure. On the other hand, when a thread wants to reclaim p, it first announces that p is retired and then checks if anyone has announced the protection of p. If no one has announced, then it is safe to free p.

The safety of HP follows from a case analysis on the execution order.[1] **(1)** If line 2 happens before line 6, the reclaimer will not free p. **(2)** If line 5 happens before line 3, the accessor will not access p. In either case, use-after-free does not occur.

Algorithm 2 presents an implementation of HP.[2] To implement *protection notification* (lines 2 and 6 of Algorithm 1), the algorithm maintains hazards, a global list of single-writer-multi-reader slots that store hazard pointers. When a thread wants to protect a pointer p (TryProtect), it first acquires a slot in hazards (MakeHazptr), and then writes p to the obtained slot (line 6). When a thread wants to free a pointer (Retire), it first adds the pointer to retired, the bag of retired pointers, and periodically triggers reclamation (Reclaim) that takes pointers from retireds and scans hazards to free only the unprotected retired pointers (line 16).

Implementing *retirement notification* (lines 3 and 5 of Algorithm 1) is more difficult. Naively applying the same approach as the protection notification—querying retireds—is neither correct nor efficient, because retired pointers might have been already removed from retireds and freed, and looking up retireds whenever protecting a pointer incurs non-negligible cost. For correct and efficient retirement notifications, HP exploits the requirement of retiring a node: it must have been made unreachable from the entry point of the data structure. If the protector detects that the pointer to protect is unreachable, it decides that it could have been already retired. In other words, the protector over-approximates the retired pointer set as the unreachable pointer set.

2.2 Over-approximating Unreachability

However, precisely checking the reachability of an arbitrary node would require many link traversals, rendering it still impractical. As such, concurrent data structures using HP usually perform a further over-approximation as follows.

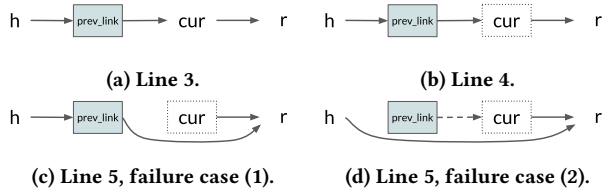
[1] For this case analysis to be sound in a weakly consistent memory model, store-load memory barriers, or sequential consistency (SC) fences are required between line 2 (resp. line 5) and line 6 (resp. line 3). See Algorithm 2.

[2] We assume C/C++'s memory model, where fence(SC) corresponds to atomic_thread_fence(memory_order_seq_cst). To keep the presentation concise, we omit the release-acquire synchronizations, which are more straightforward.

Algorithm 2 Implementation of hazard pointers.

```
   global variables:
1: retireds: ConcurrentStack<void*>
2: hazards: ConcurrentList<HazptrRecord>
3: function MakeHazptr() → Hazptr
4:     Acquire a HazptrRecord in hazards and return its reference
5: function TryProtect(hp: &Hazptr, ptr: T*) → bool
6:     hp.set(ptr)
7:     fence(SC)
8:     return whether ptr is reachable from the entry point of data structure
9: function Retire(ptr: void*)
10:    retireds.push(ptr)
11:    Call Reclaim according to configuration
12: function Reclaim()
13:    rs ← retireds.pop_all()
14:    fence(SC)
15:    for r ∈ rs do
16:        if r ∈ hazards then retireds.add(r)
17:        else free(r)
```

```
1: prev_link ← head
2: cur ← head.load()
3: while cur ≠ null do
4:     hp_cur.set(cur)
5:     if prev_link.load() ≠ cur then retry from the beginning
6:     (next, tag) ← decompose_tagged_ptr((*cur).next.load())
7:     if tag = 1 then
8:         try physically deleting cur and continue
9:     else
10:        if cur is at the desired position then return
11:        prev_link ← &(*cur).next
12:        swap(&hp_cur, &hp_prev)
13:    cur ← next
```



(a) Line 3. (b) Line 4.

(c) Line 5, failure case (1). (d) Line 5, failure case (2).

Figure 3: Traversal of Harris-Michael list with HP.

Treiber's stack. Treiber's stack [57] is a singly linked list with its head being the stack's top. Figure 2 illustrates its Pop function, which dereferences head to get the first node's pointer, say p (line 2); dereferences p to get the second node's pointer, say q (line 5); and performs a compare-and-swap (CAS) on head from p to q. As highlighted in purple, to safely dereference p, Pop first announces the protection of p (line 3) and then validates that p is still reachable. For validation, Pop checks that head still contains p (line 4). The condition is a proper over-approximation of reachability, *e.g.*, when a value is pushed, head points to a new node that points to p.

Harris-Michael list. Harris-Michael list [44] is a singly linked list that allows the insertion and deletion of a node at arbitrary positions. Deletion of a node first marks the node as *logically deleted* by tagging the least significant bit (LSB) of its next node pointer field, and then tries physically unlinking the node. Logical deletion prevents creating a link from a removed node to a live node, which makes the live node get lost.

Figure 3 illustrates the traversal strategy of Harris-Michael list. Dashed edges represent tagged pointers, blue-boxed nodes are validated protection by the traversing thread, and dash-boxed nodes have been protected by the traversing thread. (We will discuss protection shortly.) Traversal starts by initializing prev_link to head and cur to the pointer to the first node. Then it enters the loop with the invariant that cur is loaded from prev_link from the last iteration and its tag is clean (line 3). It initializes next and tag to the pointer to cur's next node and its tag (line 6). If the pointer is tagged, cur is logically deleted so it tries its physical deletion and continues the loop with a new cur (lines 7-8). If the search key is found, it returns (line 10). Otherwise, updates prev_link and cur to cur and next, respectively, for the next iteration (lines 11-13).

The traversal validates the protection of each node without precisely determining the target node's unreachability, which would require checking all the links from the head to the target node. Instead, it over-approximates the target node's unreachability by whether the target node's previous node is logically deleted, exploiting the property that only logically deleted nodes are unlinked. Specifically, the validation compares cur against the current value of prev_link (line 5). This effectively ensures two conditions at once: **(1)** that prev_link still points to cur (similar to validation of Treiber's stack), ruling out the case in which the previous node is still reachable but cur is unlinked from it (Figure 3c); and **(2)** that the previous node is not logically deleted, ruling out the case in which cur is still linked from prev_link but detached from the list as the previous node is unlinked (Figure 3d). If the validation fails, the traversal restarts from head. It is worth noting that in the failure case (1), it is safe to retry protection with the new value of prev_link without restarting from the head (not shown in the code).

This protection method requires only two hazard pointers acquired in a hand-over-hand fashion: **(1)** hp_cur that protects the current target node cur; and **(2)** hp_prev that protects the previous node containing prev_link (unless it is head). At the start of the loop, it first tries protecting cur with hp_cur (line 4); and when a link is followed, it swaps hp_cur and hp_prev (line 12).

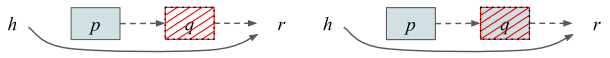
2.3 Inapplicability to Optimistic Traversal

Over-approximation of the retired pointer set, however, renders HP inapplicable to data structures that *optimistically* traverse nodes that might have been deleted to achieve better performance. That is, they intentionally ignore the warning that the nodes might have been retired already. Therefore, applying HP's pessimistic protection failure defeats the purpose of optimistic traversal.

Harris's list. Harris's list [30] is a well-known example of optimistic traversal. It shares the same structure as the Harris-Michael list but has a different traversal strategy. While the traversal in Harris-Michael list proceeds carefully, cleaning up the logically deleted nodes one by one, that in Harris's list identifies the chain of logically deleted nodes that directly precedes the destination node and unlinks the chain at once. That is, Harris's list allows traversing a chain of logically deleted nodes. Figure 4 illustrates an example

(a) Traverse to logically deleted p. (b) Proceed to logically deleted q (c) Unlink p and q at once.

Figure 4: Traversal of logically deleted nodes in Harris's list.



(a) Protection of a freed node q. (b) Validation of a freed node q.

Figure 5: Unsafe traversal of logically deleted nodes with HP.

execution: **(1)** a traversal to r sees that p is logically deleted (Figure 4a); **(2)** instead of deleting p as in Harris-Michael list, proceeds to q and sees that q is logically deleted (Figure 4b); and **(3)** proceeds to find the target r, and unlinks p and q (Figure 4c).

Optimistic traversal has two performance advantages. **(1)** Fewer CAS attempts: since the thread that logically deleted a node will also try unlinking it, the other traversing threads do not need to eagerly unlink the nodes. **(2)** Fewer successful CASes: since it can delete multiple nodes with a single CAS, the total number of successful deletion CASes decreases. Thanks to these characteristics, Harris's list outperforms Harris-Michael's list under heavy contention.

However, optimistic traversal is inherently incompatible with the hand-over-hand protection method of Harris-Michael list. Figure 5 illustrates an execution where a thread uses hand-over-hand protection but ignores logical deletion, which leads to use-after-free. **(1)** Suppose the traversing thread has validated the protection of p and is about to protect q. **(2)** Another thread detaches the chain of p and q at once, retires them, notices that q is not protected, and frees q (Figure 5a). **(3)** The traversing thread protects q and successfully validates it as p still points to q (ignoring the fact that p is logically deleted), hence accessing already freed q (Figure 5b).

There are two possible approaches to applying HP to Harris's list, but both of them are not satisfactory. **(1)** Precise validation: Validate the protection by precisely determining the reachability of the node to be protected. However, this is not practical because it requires re-checking all the links in the traversed path. **(2)** Naive restarting: On encountering a logically deleted node, restart the traversal. However, this not only disables optimistic traversal but also breaks lock-freedom,[3] because the traversal will keep restarting if a logically deleted node is not cleaned up by the thread that logically deleted it. In fact, Harris-Michael list was derived from Harris's list to make it compatible with HP while maintaining lock-freedom.

Other data structures using optimistic traversal. Optimistic traversal has been applied to many other search data structures to improve performance (see §5 for an experimental evaluation). For example, Herlihy and Shavit [34] added to Harris-Michael list a wait-free search method that simply ignores logical deletion during traversal, and Brown et al. [9], Drachsler et al. [24], Howley and Jones [36], Natarajan and Mittal [48], Ramachandran and Mittal [50] designed concurrent search trees based on logical deletion and optimistic traversal. Unfortunately, all these data structures cannot use HP for the same reason as Harris's list: there have been no practical methods to validate protection other than restarting when logical deletion is detected.[4]

2.4 Other Reclamation Schemes and Trade-Offs

Other memory reclamation schemes support optimistic traversal, but they exhibit other drawbacks.

Read-copy-update (RCU) [42, 43] and epoch-based reclamation (EBR) [28, 30] are seminal reclamation schemes at the opposite of the spectrum.[5] With RCU/EBR, clients only need to announce that they are starting and finishing an operation (dubbed *critical section*) instead of announcing the protection of each pointer as in HP. An RCU/EBR critical section protects *all* memory blocks that were not retired before the beginning of the critical section, which include all blocks that are reachable by a traversal from the entry point of a data structure in the critical section.

RCU/EBR's critical-section-based coarse-grained protection has several advantages. **(1)** Performance: RCU/EBR makes fewer announcements (per critical section) than HP (per pointer). **(2)** Simplicity: RCU/EBR does not require users to manually validate the protection of each pointer. **(3)** Applicability: RCU/EBR never fails protection, so it can be applied to a wide range of data structures where validation is practically infeasible.

However, RCU/EBR is not *robust* against non-cooperative threads: the number of retired and yet unreclaimed pointers is unbounded. If a thread does not exit its critical section, no newly retired pointers are reclaimed because the reclaimer assumes that the thread may access such pointers. On the other hand, HP is robust, because the number of hazard pointers is bounded.

DEBRA+ [11], NBR [56], and PEBR [39] are critical-section-based schemes that resolve the robustness problem. They ensure robustness by detecting non-cooperative threads and forcefully terminating (or *neutralizing* or *ejecting*) their critical sections. To notify neutralization to an offending thread, DEBRA+ and NBR send a signal to the offending thread, and PEBR marks the thread as neutralized so that its further pointer protection attempts (as in HP) fail. While the neutralized thread has to stop the traversal and handle the failure, neutralization does not prohibit optimistic traversal because it serves as the "precise validation" mechanism that notifies the thread of the imminent danger.

But these neutralization-based schemes have the following drawbacks. **(1)** DEBRA+ requires data structure-specific recovery code that is challenging to design [56]. **(2)** NBR requires data structure

[3] Roughly speaking, a data structure is lock-free if, in any circumstances, *at least one* of concurrent operations—if exist—finishes successfully.

[4] Natarajan and Mittal [48] claim that HP supports their tree, but we argue that it is not the case due to optimistic traversal as we discuss here.

[5] EBR is a class of lock-free algorithms that implements RCU's interface.

operations to be split into read and write phases, where the former does not write to the shared memory and the latter does not traverse to more nodes. But this assumption is not widely satisfied, *e.g.*, by Harris-Michael list. **(3)** All of them do not properly support long-running operations because they are likely to be neutralized to ensure the progress of reclamation. For example, they are not compatible with online analytical processing (OLAP) queries [14] in relational database management systems that may run for several hours and even days.

OA [17], AOA [16], FA [15], and VBR [54] are *optimistic* methods that allow the threads to speculatively access possibly reclaimed objects and verify its validity afterward. Thanks to optimism, they have low overhead and support optimistic traversal. However, to allow accessing reclaimed objects, they require either a custom allocator that does not return memory to the OS or a custom segmentation fault handler, which may result in difficulties when integrating into existing systems in practice.

Reference counting schemes are automatic methods that attach a counter to each object to count the incoming references and reclaim the object when the count becomes zero. Since objects are not reclaimed as long as there is an incoming link, reference counting supports optimistic traversal and does not require failure handling. While the earlier algorithms [20, 33] were considerably slower due to high contention on the reference counter, the recent CDRC [3, 4] algorithm has achieved performance comparable to semi-manual methods by deferring the updates of reference counts using the semi-manual methods underneath. However, reference counting methods require special treatment for reference cycles using weak pointers.

Formalizing the trade-offs, Sheffi and Petrank [55] recently proved that a reclamation scheme cannot achieve wide applicability, robustness, and simplicity ("ease of integration") at the same time. Under this constraint, we aim to improve the applicability of HP while retaining its robustness and relative simplicity.

3 DESIGN

We prsesent HP++'s key idea (§3.1), interface (§3.2), and implementation (§3.3). Then we present an optimization to accelerate protection while still guaranteeing robustness (§3.4).

3.1 Key Idea

As discussed in §2, HP is incompatible with optimistic traversal as its over-approximation of unreachability induces protection failures for logically deleted nodes. To support optimistic traversal while retaining robustness, we propose a drastically different approach of *under-approximating* unreachability and *patching up* the potentially unsafe accesses arising from false-negatives.

Specifically, we make unlinkers first perform the physical deletion of nodes to be deleted and then *invalidate* them. As such, invalidation is an under-approximation of unreachability. Conversely, when a traversing thread notices that a node is invalidated, then it abstains from taking further steps from that node. Invalidation plays a similar role to logical deletion in Harris-Michael list in forbidding further steps. But only the under-approximating invalidation allows optimistic traversal from logically deleted nodes in Harris's list and other efficient data structures.

But is it safe in the case of a false-negative, *i.e.*, what if the node is unlinked and retired but the traversing thread misses the invalidation notification? We observe that executions leading to use-after-free can be classified into the following two scenarios illustrated in Figure 6. Here, the three threads T1, T2, and T3 are traversing a Harris's from the node h to the node r. Initially, all threads have validated protection for h, and the nodes p and q are logically deleted, T1 has validated protection of p and is about to announce protection of q, and T2 has reached r first, and unlinked p and q by making h point to r (Figure 6a).

In the first scenario, **(1)** T2 invalidates (✗) and frees q (since it is not protected yet), and is about to invalidate p (Figure 6b); and **(2)** T1 protects q and validates the protection (Figure 6c), because p still points to q and p is not invalidated yet. As a result, T1 runs into use-after-free with q. In the second scenario, **(1)** T1 protects q and validates the protection (Figure 6d), because p still points to q and p is not invalidated yet; **(2)** T3 reaches r and logically deletes, unlinks, and frees r (Figure 6e); and **(3)** T1 protects r and validates the protection (Figure 6f), because q still points to r and q is not invalidated yet. As a result, T1 runs into use-after-free with r.

To prevent such use-after-free, we observe that the impact of false-negative is quite limited and can be easily patched up by *letting the unlinker* take some responsibility to protect the traversing thread's accesses. For the above example, it is sufficient for T2—the unlinker of p and q—to provide the following guarantees:

(1) *Invalidating both p and q (Figure 7a) before freeing any of them (Figure 7b).* Then in the first scenario, if T1 did not see invalidation of p, q must not have been freed by T2 at that point.
(2) *Protecting r before unlinking p and q (Figure 7c and 7d).* Then in the second scenario, by the time r is retired by T3, r must have been already protected by T2, because T3's physical deletion of r happens after T2's physical deletion of p and q (otherwise, T3 would have tried physically deleting p, q and r at once).

We formalize this under-approximation and patch-up idea in HP++.

3.2 Interface

We first present the interface of HP++ (Algorithm 3) with Harris's list (Algorithm 4) as a running example. Traversing threads use TryProtect to protect a pointer loaded from a *source* object. It takes arguments **(1)** hp, the hazard pointer to protect with; **(2)** ptr, the pointer to protect (corresponds to cur in Figure 3); **(3)** src and src_link, the reference of the source object and its field from which ptr was loaded (corresponds to prev_link in Figure 3); **(4)** is_invalid, the predicate to check whether src is invalidated. If src is invalidated, TryProtect returns false meaning that it is unsafe to create new protection from src. Otherwise, it returns true, but if src_link has changed from ptr, then the new value is written to ptr.

Unlinking threads use TryUnlink to physically delete and retire node(s) while protecting the traversing threads. The protection will persist until the retired nodes are invalidated by reclaimers. TryUnlink takes arguments **(1)** frontier, the pointers that the unlinker has to protect for the traversing threads; **(2)** do_unlink, the function that performs unlinking and returns the unlinked nodes; and **(3)** invalidate, the function that invalidates each unlinked node. TryUnlink returns whether the unlink was successful.

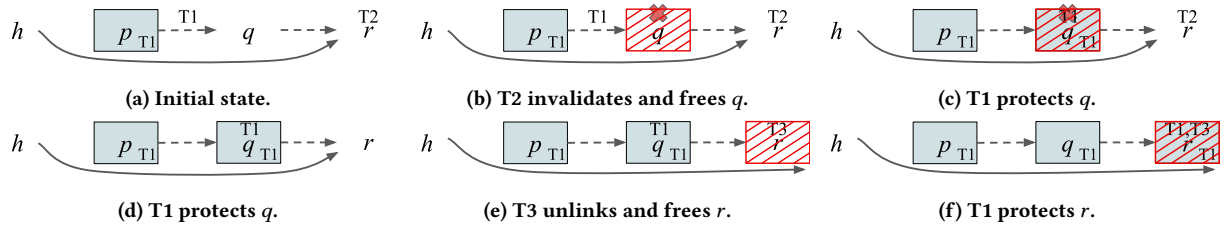


Figure 6: Problems arising from under-approximating unreachable pointer set.

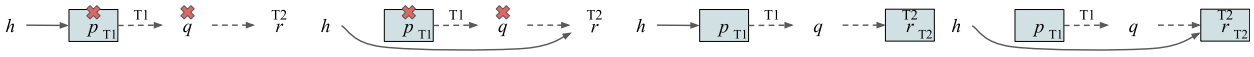


Figure 7: Patching up the problems of under-approximation.

Frontier. The *unlinking frontier* is the set of pointers that are reachable by following a single link from the to-be-unlinked objects but are not themselves to-be-unlinked. In Harris's list, the frontier is a single pointer to the next node of the last node in the unlinked chain. The frontier argument to TryUnlink must be decided ahead of the actual unlinking, and the data structure must guarantee that the frontier does not change once it is decided: otherwise, the traversing thread's access to a frontier node may not be protected. This property holds for a wide range of concurrent data structures. For example, it is already required for the correctness of data structures with logical deletion such as Harris's list and its variants.

Invalidation. Invalidation can be implemented by adding a flag to the node. But in most cases, this can be done without extra space overhead using tagged pointers, similar to logical deletion. For example, Harris's list can use the second LSB of a node's next pointer field to represent invalidation. It is worth noting that Invalidate can use store instead of atomic read-modify-write thanks to the assumption that to-be-unlinked node's links do not change.

Case study: Harris's list. We apply HP++ to Harris's list's traversing function TrySearch presented in Figure 4. For space reasons, we gloss over the code-level details (see [30]). Though anchor and anchor_next deserve attention. They are pointers that are non-null if and only if prev is logically deleted. In that case, anchor is the last node that was not logically deleted, and anchor_next is its next node, which are used for unlinking the chain of logically deleted nodes. For instance, for the scenario illustrated in Figure 4, anchor and anchor_next are h and p, respectively, and TryUnlink at line 27 tries to update h's (anchor's) next field from p (anchor_next) to r (cur, the frontier of the unlinking operation) at line 37.

The differences from the original Harris's list without manual memory management are highlighted. Purple indicates the differences already present in Harris-Michael list (Figure 3): prev and cur pointers and hp_prev and hp_cur hazard pointers advance in hand-over-hand fashion (lines 15 and 25). On the other hand, pink indicates the new changes for HP++. TrySearch uses TryProtect to protect cur (line 10). If the protection fails because prev is invalidated, it returns None to indicate that the traversal has to restart (line 11). anchor should be protected by a hazard pointer because the unlinking operation accesses anchor. anchor_next should also be protected to avoid the ABA problem, ensuring that only one thread succeeds in unlinking the chain. anchor and anchor_next inherit the protection from hp_prev (lines 22 and 24). Lastly, TryUnlink does extra work of protecting the frontier [cur] and invalidating the unlinked nodes.

3.3 Implementation

The implementation of TryProtect is self-explanatory from the interface. Though it is worth noting that is_invalid check precedes src_link check (lines 8 and 10); an SC fence is issued between protection and validation (line 7); and tags of src_link are ignored (line 9) so that protection may succeed regardless of logical deletion.

TryUnlink first acquires hazard pointers for all frontier nodes (line 14). Unlike usual protection, they do not need to be validated. Then it performs the unlink operation (line 27). If the unlink was unsuccessful, the protection can be immediately revoked (line 21). Invalidation of the unlinked nodes is deferred to and batched in DoInvalidation to maximize the benefit of under-approximation and amortize the cost of the SC fence for ordering invalidation and revocation of frontier protection. Specifically, the unlinked pointers are added to the thread-local container unlinkeds along with the associated invalidation function and hazard pointers (line 18). DoInvalidation is periodically called to execute the deferred invalidation (line 26), issue an SC fence (line 29), and release the associated hazard pointers (line 30). Then the pointers finally are moved to retireds (line 31) and later freed by Reclaim, which is the same as the original algorithm (Algorithm 2) except that it does not need a fence itself thanks to the fence in DoInvalidation.

3.4 Optimization

Asymmetric fences. TryProtect in Algorithm 3 (as well as the original HP) incurs a significant overhead due to an SC fence at line 7. To reduce the cost, we use the well-known technique of replacing a pair of SC fences with a pair of asymmetric fences that provide a similar synchronization [19, 22, 23]. Specifically, we replace the SC fences in TryProtect and DoInvalidation with

Algorithm 3 HP++ algorithm.

```
    global variables:
1:  hazards: ConcurrentList<HazptrRecord>
2:  retireds: ConcurrentStack<void*>
    thread-local variables:
3:  unlinkeds: List<(List<void*>, fn(void*), List<Hazptr>)>
4:  function TryProtect(
        hp: &Hazptr, ptr: &(T*), src: &S, src_link: &Atomic<T*>,
        is_invalid: fn(&S) → bool,
    ) → bool
5:      loop
6:          hp.set(ptr)
7:          fence(SC)
8:          if is_invalid(src) then return false
9:          ptr_new ← untagged(src_link.load())
10:         if ptr = ptr_new then return true
11:         else ptr ← ptr_new
12: function TryUnlink(
        frontier: List<void*>,
        do_unlink: fn() → Option<List<void*>>,
        invalidate: fn(void*),
    ) → bool
13:     hps ← []
14:     for ptr ∈ frontier do
15:         hp ← MakeHazptr(); hps.add(hp); hp.set(ptr)
16:     match do_unlink()
17:         case Some(unlinked) then
18:             unlinkeds.push((unlinked, invalidate, hps))
19:             Call DoInvalidation and Reclaim according to configuration
20:             return true
21:         case None then release hps; return false
22: function DoInvalidation()
23:     invalidateds ← []
24:     hps ← []
25:     for (rs, invalidate, h) ← unlinkeds.pop_all() do
26:         for r ∈ rs do
27:             invalidate(r); invalidateds.push(r)
28:         hps.append(h)
29:     fence(SC)
30:     release hps
31:     push invalidateds to retireds
32: function Reclaim()
33:     for r ∈ retireds.pop_all() do          ▷ fence is not required here
34:         if r ∈ hazards then retireds.push(r)
35:         else free(r)
```

Algorithm 4 Search function of Harris's list with HP++.

```
    thread-local variables:
1:  prev, cur: Node*
2:  anchor, anchor_next: Node*, Node*
3:  hp_prev, hp_cur, hp_anchor, hp_anchor_next : Hazptr
4:  function TrySearch(key) → Option<bool>
5:      prev ← head; cur ← prev.load()
6:      anchor ← NULL
7:      loop
8:          if cur = NULL then
9:              found ← false; break
10:         if ¬TryProtect(hp_cur,cur, prev, &(*prev).next, IsInvalid) then
11:             return None
12:         (next, tag) ← decompose_tagged_ptr((*cur).next.load())
13:         if tag = 0 then
14:             if (*cur).key < key then
15:                 prev ← cur; cur ← next; swap(&hp_cur, &hp_prev)
16:                 anchor ← NULL
17:             else
18:                 found ← (*cur).key = key; break
19:         else
20:             if anchor = NULL then
21:                 anchor ← prev; anchor_next ← cur
22:                 swap(&hp_anchor, &hp_prev)
23:             else if anchor_next = prev then
24:                 swap(&hp_anchor_next, &hp_prev)
25:             prev ← cur; cur ← next; swap(&hp_prev, &hp_cur)
26:     if anchor ≠ NULL then
27:         if TryUnlink([cur], DoUnlink, Invalidate) then
28:             prev ← anchor
29:         else return None
30:     if get_tag((*cur).next.load()) = 1 then return None
31:     else return Some(found)
32: function Invalidate(node: &Node)
33:     node.next.store(node.next.load() | 2)
34: function IsInvalid(node: &Node) → bool
35:     return node.next.load() & 2 = 2
36: function DoUnlink() → Option<List<Node*>>
37:     if (*anchor).next.compare_exchange(anchor_next, cur) then
38:         return Some([nodes from anchor_next to the node before cur])
39:     else return None
```

"light" *compiler fence* (not appearing in binary) and "heavy" *process-wide memory fence* [12, 29, 41], respectively, so that TryProtect incurs no synchronization cost at the expense of a higher cost of DoInvalidation.

Fewer heavy fences. Heavy fence, however, still incurs a huge overhead even considering their infrequent invocations. Algorithm 5 illustrates an optimization to reduce heavy fence, where the differences from Algorithm 3 are highlighted in green.

We first move the heavy fence and the following revocation of hazard pointers at line 30 to the point in Reclaim between the retrieval of retired pointers and the hazard scan (new functions and variables will be discussed shortly). Since Reclaim is less frequently invoked, the overall number of heavy fences is reduced. The modified algorithm is correct because the new fence location still plays the two roles of the SC fence of DoInvalidation in Algorithm 3: separating unlink at line 16 and hazard pointer scan at line 34; and separating invalidation at line 26 and revoking hazard pointers at line 30. In particular, line 30 in DoInvalidation of Algorithm 3 is moved to line 14 in Reclaim of Algorithm 5.

However, this optimization has a side effect of accumulating a significant number of hazard pointers due to deferred revocation, which degrades the performance of the hazards scan in Reclaim. While the hazard pointers can be revoked by after issuing an additional heavy fence, doing so would defeat the purpose of this optimization in the first place—reducing heavy fences.

To revoke hazard pointers without additional fences, we let each thread *piggyback* on another thread's heavy fence by assigning a

Algorithm 5 HP++ algorithm with epoched heavy fence.

```
   additional global variables:
1: fence_epoch: Atomic<Epoch>
   additional thread-local variables:
2: epoched_hps: List<(Epoch, Hazptr)>
3: function DoInvalidation()
4:     prepare local_invalidateds and hps as in Algorithm 3
5:     epoch ← ReadEpoch()
6:     for (old_epoch, hp) ← epoched_hps.pop_all() do
7:         if old_epoch + 2 ≤ epoch then release hp
8:         else add back to epoched_hps
9:     add hps to epoched_hps after attaching epoch
10:    push local_invalidateds to invalidateds
11: function Reclaim()
12:    rs ← retireds.pop_all()
13:    FenceEpoch()
14:    release epoched_hps
15:    for r ∈ rs do
16:        ...
17: function FenceEpoch()
18:    epoch ← fence_epoch.load()
19:    heavy fence
20:    fence_epoch.compare_exchange(epoch, epoch+1)

21: function ReadEpoch()
22:    epoch ← fence_epoch.load()
23:    loop
24:        light fence
25:        new_epoch ← fence_epoch.load()
26:        if epoch = new_epoch then return epoch
27:        epoch ← new_epoch
```

global epoch number to each period delimited heavy fences. Specifically, each thread maintains the list epoched_hps of hazard pointers attached with an epoch. DoInvalidation gets the current epoch e with ReadEpoch (line 5), adds hazard pointers to epoched_hps after attaching e (line 9), and cleans up the old hazard pointers with epoch $\leq e-2$ (line 7). The old hazard pointers are safe to clean up because there is always a heavy fence between two ReadEpoch invocations returning $e-2$ and e. To enforce this, FenceEpoch wraps a heavy fence with a read from the global epoch counter fence_epoch and its increment with a CAS; and ReadEpoch wraps a light fence with reads from fence_epoch and ensures they read the same epoch.

4 ANALYSIS

We analyze various properties of HP++ such as safety (§4.1), applicability (§4.2), lock-freedom (§4.3), and robustness (§4.4).

4.1 Safety

Our algorithm is safe: it does not incur use-after-free. As the first step towards the proof, we formalize new assumptions made in §3.

Assumption 1 (Immutability of Unlinked Nodes). *Before invoking TryUnlink for a node, its next pointers never change.*

In other words, mutating nodes are not unlinked. To the best of our knowledge, this assumption is satisfied by *all* concurrent data structures we are aware of (see §4.2 for more discussions). Based on the assumption, we prove the safety of Algorithm 3 and 5.

Theorem 4.1 (Safety). *Suppose a concurrent data structure, S, reclaims nodes using Algorithm 3 or Algorithm 5 and satisfies Assumption 1. Then S does not incur use-after-free.*

The proof essentially formalizes the key idea presented in §3.1. For space reasons, we present the full proof in Appendix A [37].

4.2 Applicability

We argue that HP++ is widely applicable, and notably, strictly more applicable than the original HP. We discuss the additional requirements for applying HP++ to a data structure, *i.e.*, Assumption 1 and the ability to recover from protection failure, which are met by a wide range of data structures (see Appendix B [37] for more detail). We will compare the applicability of HP++ and other schemes in §6.

On Assumption 1. For lock-based data structures, unlinking of a node is usually preceded by acquiring the lock for that node (*e.g.*, to load its next node), preventing modification from other threads. For lock-free data structures, if the deletion of a node happens at the entry point of the data structure, each node's links become immutable before deletion. For example, in Treiber's stack [57], each node is immutable (once added to the stack); and in Michael-Scott queue [47], only the tail node can be mutated (only once) while the tail node cannot be unlinked. The most common approach to supporting deletion at arbitrary position is two-phase, logical and physical deletion (*e.g.*, Harris's list [30] and its variant), where the correctness of the data structure already depends on the property that the links of a logically deleted node are immutable.

On recovery from protection failure. HP++ clients should recover from the failure of TryProtect as they are not allowed to traverse further. We refer the reader to Sheffi et al. [54, §4.2] for a general account of failure handling strategy, and we here discuss two simple strategies that cover most scenarios.

For many lock-free linearizable [35, 52] data structures, recovery is straightforward. Due to the non-blocking nature, any instructions before and after the linearization point should maintain the data structure's invariant without changing its abstract state (*e.g.*, reads and helping writes). As such, if protection fails before the linearization point, it is safe to simply restart the operation from the beginning; if protection fails after the linearization point, it is safe to ignore the remaining instructions and return immediately.

For lock-based data structures, recovery is difficult in general because if the thread was in a lock critical section, naive restart on protection failure may break the data structure's invariant. However, recovery is trivial for *access-aware* [55, 56] data structures. Roughly speaking, a data structure is access-aware if **(1)** each operation can be divided into alternating read phase and write phase; **(2)** in the write phase, the thread can write to only a set of pointers *reserved* (protected in the context of HP++) at the end of the read phase; and **(3)** when transitioning from a write phase to a read phase, the thread has to restart the operation from the entry point. By the definition, protection failure does not happen during a write phase. As such, to apply HP++, it suffices to restart the operation on encountering protection failure in a read phase.

Relation to the original hazard pointers. It is worth noting that Algorithm 3 is an *extension*—not *modification*—of the original HP (hence the name HP++). The two algorithms share the same high-level architecture (Algorithm 1) but differ in how to implement retirement notification. Such a shared architecture brings about the following benefits. **(1)** Hybrid: we can use both retirement notification strategies—the original and ours—in tandem to protect different nodes of a single data structure. If you wish to avoid Algorithm 3's overhead of node invalidation (which is small, though; see §5), you can use the original algorithm's over-approximation strategy. **(2)** Backward compatibility: Algorithm 3 can seamlessly replace the original HP in the existing implementation of concurrent data structures with reclamation.

Furthermore, it is straightforward to switch the retirement notification strategy from the original to ours. Specifically, Algorithm 3 does not incur additional restarts than the original. To see why, it suffices to show that Algorithm 3 successfully protects a pointer as far as the original algorithm does, or in other words, the protection validation condition of the original HP (reachability) implies that of Algorithm 3 (node validation).

Suppose the original algorithm traverses from a node, say p, to another node, say q, and successfully protects q. Since the only thing we know about q is that p points to q, the reachability of q should be derived from the following conditions: **(1)** p is reachable from the entry point of the data structure; and **(2)** p still points to q. Since p is reachable, p should not have been invalidated so that Algorithm 3 would pass p's validity check at line 8. Since p still points to q, it would pass the p-to-q link check at line 10. As such, Algorithm 3 would successfully protect the same pointer q.

4.3 Lock-Freedom

In general, the original HP does *not* preserve lock-freedom: when it is applied to a lock-free data structure, the result is not necessarily lock-free. For instance, applying HP with the naive restart approach breaks the lock-freedom of Harris's list (§2.3). To preserve lock-freedom, the data structure should employ some helping mechanism, which may require non-trivial modification [7].

In contrast, our schemes preserve lock-freedom because they under-approximate unreachability and allow optimistic traversal.

Theorem 4.2 (Lock-freedom preservation). *If a lock-free data structure reclaims nodes using Algorithm 3 and 5 and satisfies Assumption 1, then the resulting data structure is also lock-free.*

Proof. We prove by contradiction that our algorithm's protection failures do not jeopardize the lock-freedom of data structures. Suppose otherwise and there exists an (infinite) execution history of the resulting data structure with reclamation where no operations are finished successfully after some timestamp, say t_0.

For each protection failure, consider the *unlinker* thread, say U, that invalidated the node and the *accessor* thread, say A, that tried traversing from that node. Then U cannot induce A's protection failure again because **(1)** A has already observed the node's unlinking and it will not traverse from it again; and **(2)** U's operation that unlinked the node is running forever by the assumption so that it will not invalidate another node. As such, there are at most $N \times N$ protection failures after t_0, where N is the number of threads.

Let t_1 be a timestamp after all protection failures after t_0. Then there will be no protection failures after t_1. Then by the lock-freedom of the original data structure, at least one operation should finish successfully after t_1, inducing a contradiction. □

But our algorithm does not preserve wait-freedom[6], *e.g.*, it makes Heller et al. [32], Herlihy and Shavit [34]'s wait-free search method just lock-free as traversal may be restarted indefinitely due to protection failures. We believe this is a fundamental limitation of any robust and widely applicable reclamation schemes, based on the recent result on the trade-offs of reclamation schemes [55].

4.4 Robustness

Like the original HP, our algorithm is robust: the number of garbages that are unlinked but not reclaimed yet, is bounded. In Algorithm 3, the number of the additional hazard pointers for unlinked nodes' next pointers, announced at line 14 in TryUnlink, are bounded by the maximum number of next pointer for each thread. In Algorithm 5, we can still bound the number by simply invoking Reclaim that invalidates such hazard pointers at line 14.

5 EXPERIMENTAL EVALUATION

We implemented HP++ as a Rust library[7] and evaluated it on a suite of synthetic benchmarks[8].

Other than HP++, our benchmark suite includes the following reclamation schemes: **HP**: the original hazard pointers [45, 46] with asymmetric fence optimization (§3.4); **EBR**: epoch-based RCU [28, 30]; **PEBR**: pointer- and epoch-based reclamation [39]; **RC**: the EBR flavor of CDRC [4]; and **NR**: the baseline that does not reclaim memory.[9] The implementation of HP++ triggers Reclaim per 128 TryUnlinks (for other schemes, per 128 Retires), and DoInvalidation per 32 TryUnlinks.[10] We implemented HP and RC on our own, and used the public implementation of PEBR and `crossbeam-epoch` crate [21] for EBR.

Our benchmark suite consists of the following representative data structures: **HMList**: Harris-Michael linked list [44], **HHSList**: Harris's list [30] with wait-free get() method [34] (HP not applicable)[11], **HashMap**: chaining hash table using HMList (for HP) or HHSList (for others) for each bucket [44], **SkipList**: lock-free skiplist by Herlihy and Shavit [34], with wait-free get() for schemes other than HP; **Bonsai**: a non-blocking variant of Bonsai tree [13]'s tree, **EFRBTree**: Ellen et al. [26]'s tree[12], and **NMTree**: Natarajan-Mittal tree [48] (HP not applicable).

The benchmark suite was compiled with Rust nightly-2022-12-28 with default optimization enabled. We used jemalloc [27] to reduce contention on the memory allocator. We conducted experiments on a dedicated machine with single-socket AMD EPYC 7543 (2.8GHz,

[6] Roughly speaking, a data structure is wait-free if, in any circumstances, *every* operation finishes successfully.
[7] Publicly available at https://github.com/kaist-cp/hp-plus
[8] Publicly available at https://github.com/kaist-cp/smr-benchmark
[9] We do not include evaluation for NBR and VBR, but we expect that they generally outperform EBR as reported by Singh et al. [56] and Sheffi et al. [54].
[10] We found these numbers big enough to amortize the cost of the expensive synchronizations and small enough to bound the number of unreclaimed objects.
[11] For PEBR and HP++, get() is only lock-free due to protection failure (§4.3). The same applies to SkipList.
[12] We omitted the implementation for EFRBTree with RC as it requires nontrivial design efforts: cycles involving operation descriptor should be broken with weak pointers.

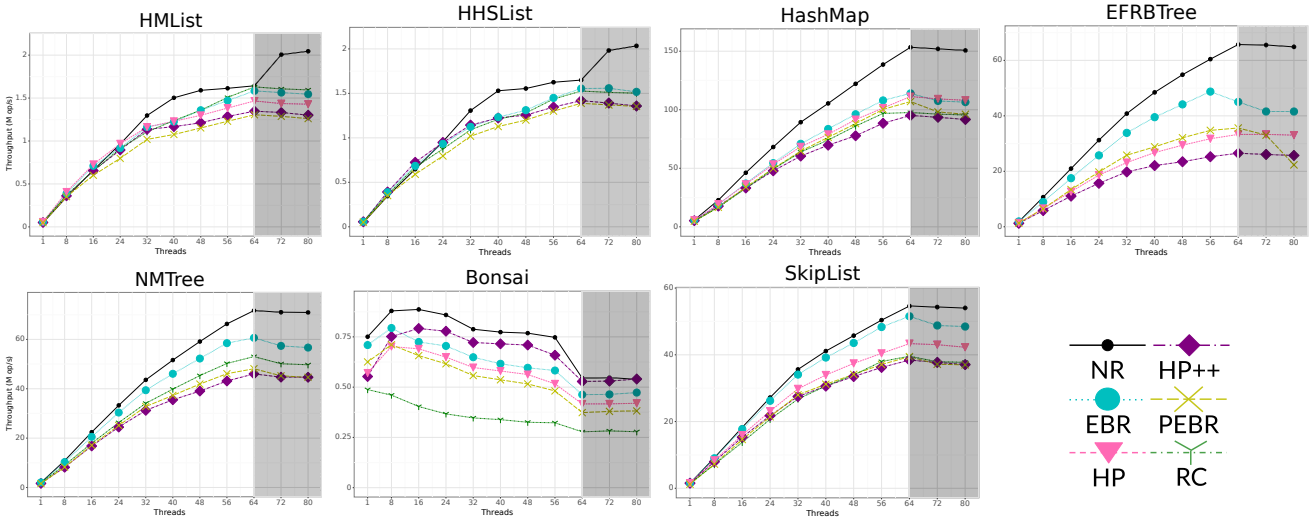
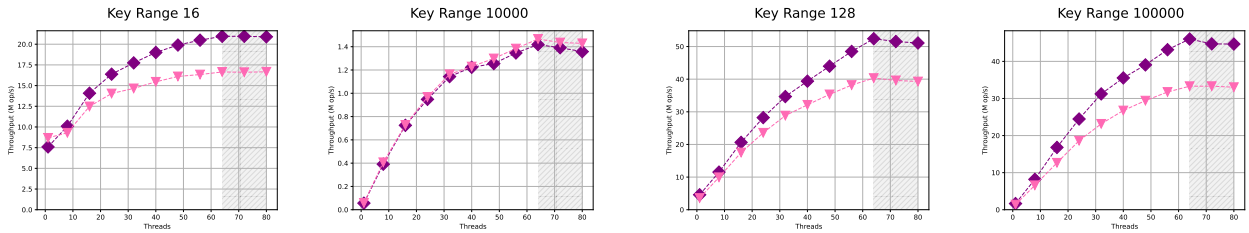


Figure 8: Throughput of read-write workloads for varying number of threads.



(a) HMList with HP vs. HHSList with HP++. **(b) EFRBTree with HP vs. NMTree with HP++.**

Figure 9: Throughput of read-write workloads for each category of data structures, varying key ranges and number of threads.

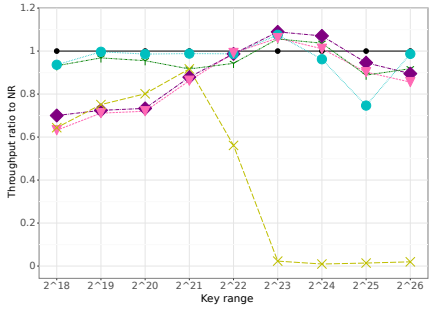


Figure 10: Throughput of long-running read operations.

32 cores, 64 threads) and eight 32GiB DDR4 DRAMs (256GiB in total). The machine runs Ubuntu 22.04 and Linux 5.15 with the default configuration.

Methodology. We measured throughput (operations per second) and the peak number of retired yet unreclaimed objects for **(1)** varying number of threads: 1, 8, 16, 24, ⋯, 80; **(2)** three types of workloads: write-only (50% inserts and 50% deletes), read-write (50% reads and 50% writes), and read-most (90% reads and 10% writes); **(3)** varying key ranges: small (16 for lists, 128 for the others) and big (10K for lists, 100K for the others); and **(4)** fixed time: 10 seconds (Figure 8, 9 and 11). The data structures are pre-filled to 50%. For each scenario, we report the average for 10 runs unless specified otherwise. The three types of workloads exhibit similar results, so we report the result for read-write workloads for space reasons. For the full experimental evaluation result, see Appendix C [37].

In addition, we evaluate the performance of long-running operations under heavy reclamation by measuring the throughput of read operations of lists (HMList for HP and HHSList for others) with big key ranges: $2^{18}, 2^{19}, \cdots, 2^{26}$ (Figure 10). Specifically, 32 threads perform get() for random elements, and the other 32 threads push and pop to the head of the list.

Throughput. We observe that HP++'s advantage of compatibility with optimistic traversal outweighs the overhead added to HP from invalidation and frontier protection.

First, we observe that HP++'s overhead of per-node invalidation and additional protection of pointers over HP is moderately small. Figure 8 presents the throughput under the big key range. The thread oversubscription ranges are highlighted grey. For lists (HMList, HHSList), HP++'s throughput is similar to HP and is around 87-93% of EBR. For high-throughput data structures (HashMap, EFRBTree, NMTree, SkipList), HP++'s throughput is about 80-90% of HP and 55-90% of EBR. Bonsai exhibits different performance

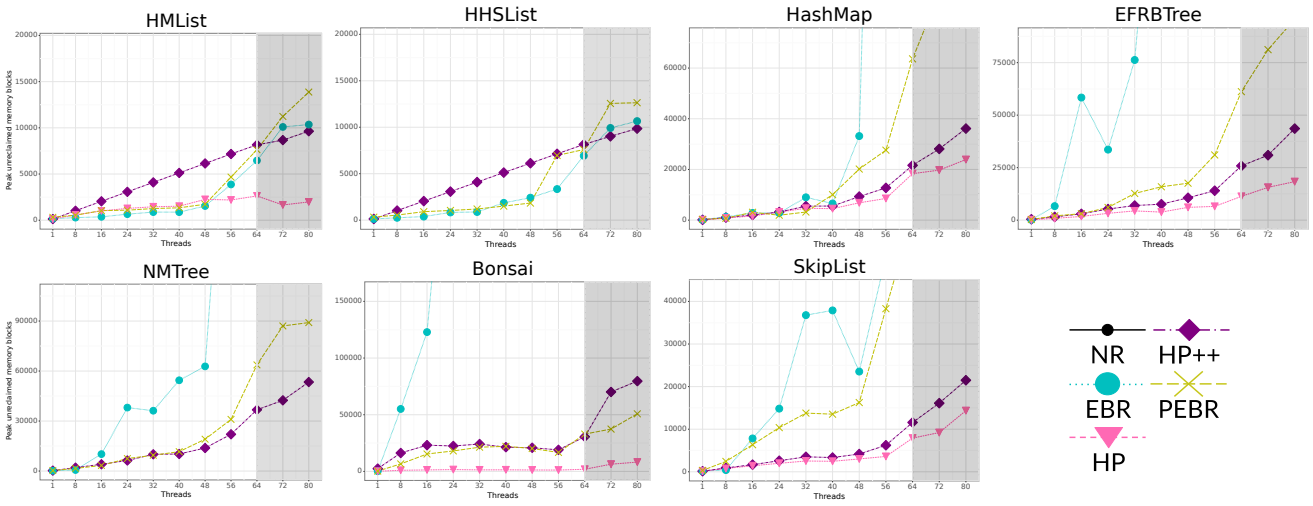


Figure 11: Peak number of unreclaimed blocks of read-write workloads for varying numbers of threads.

characteristics. Notably, RC is much slower because tree rebalancing involves large amounts of link updates which modify reference counters, and HP is less efficient because it has to validate protections wrt. the root pointer, which would fail if there were any changes to the tree. On the other hand, HP++ does not incur any overhead since Bonsai does not require frontier protection.

Second, we observe that the performance benefit of optimizations enabled by HP++'s compatibility with optimistic traversal is large. Figure 9 compares the maximum throughput that can be achieved in each category of data structure (list and tree) with the original HP (only applicable to HMList and EFRBTree) and HP++ (applicable to HHSList and NMTree as well) under varying degrees of contention. Under heavy contention (small key range) or for trees, HP++ beats HP by a large margin.

Long-running operations. We observe that HP++ is suitable for long-running operations. Figure 10 compares the throughput of long-running read operations with varying the key range. PEBR's relative throughput plunges with the key ranges $> 2^{21}$, because its coarse-grained neutralization hinders long-running operations. On the other hand, HP++ does not suffer from such a problem since its protection failure is fine-grained (more precise): TryProtect fails only when the source object is invalidated.

Memory footprint. We observe that HP++ maintains the robustness of the original HP despite unlink frontier protection.

Figure 11 presents the peak number of retired but unreclaimed memory blocks.[13] In low-throughput cases, HP++ leaves more unreclaimed memory than other schemes due to frontier protection and deferred retirement. However, under high contention or thread oversubscription, HP++ prevents the number of unreclaimed memory blocks from growing rapidly. While HP++'s number of unreclaimed blocks is larger than HP's in such cases, it still maintains the overall trend of HP's, and is smaller than PEBR's in most cases because PEBR's neutralization is coarse-grained.

[13]We do not report the result for RC, because this metric is not well-defined for reference counting schemes. Appendix C [37] reports the real memory usage, where RC has a similar trend to its underlying scheme, EBR in our benchmark.

6 RELATED WORK

HP [45, 46] and RCU/EBR [28, 30, 42, 43] are classic reclamation schemes with opposing characteristics: HP is robust but slow and not widely applicable, whereas RCU/EBR is fast and widely applicable but not robust (§2.4). To mitigate the problems of either scheme, many new methods have been proposed. For example, to reduce the performance overhead of HP, Dice et al. [22] proposed the usage of asymmetric fences [29] (§3.4), and IBR [58], Hazard Eras [51], and Hyaline [49] introduced validation by epoch.

From now on, we focus on comparing HP++ with other robust and widely applicable manual schemes. Table 1 presents a qualitative comparison of HP++ and the other state-of-the-art schemes with such characteristics, and Table 2 summarizes the applicability of reclamation schemes to various concurrent data structures.[14]

PEBR [39] and NBR [56] are hybrid schemes that utilize both the RCU/EBR's critical-section-based and HP's per-pointer protection. To resolve the robustness problem of RCU/EBR, they employ the mechanism called neutralization (or ejection) that forcefully terminates long critical sections that block the progress of reclamation. The neutralized thread should stop the operation and run a recovery procedure since it is no longer safe to resume. To help recovery, they use hazard pointers to protect the objects relevant for recovery. With these schemes, threads can optimistically traverse data structures since they know that they will be notified by neutralization when further traversal is dangerous. However, their coarse-grained neutralization may severely degrade the performance of long-running operations (Figure 10).

PEBR and NBR differ in their neutralization and notification mechanism. In PEBR, the neutralization simply marks the offending thread as neutralized, and the traversing thread should explicitly announce the protection of the next node and then validate that it is not neutralized. Upon detecting neutralization, the thread should run a data structure-specific recovery procedure using the objects protected so far, similarly to HP++.

[14]Table 2 is adapted from the analysis by Singh et al. [56] with some minor fixes. See Appendix B [37] for more detail.

criterion	PEBR	NBR	VBR	**HP++**
system requirement	heavy fence (optional)	signal, non-local jump	custom allocator, wide CAS	heavy fence (optional)
failure condition	neutralization	neutralization	outdated object/field	invalidated object
failure handling	custom handling	only applicable to access-aware DS	custom handling	custom handling
overhead	protection, validation, critical section	protection on phase change, critical section	validation	protection, validation, frontier protection, invalidation
unreclaimed objects	O(hazards + neutralization threshold)	O(hazards + neutralization threshold)	O(threads)	O(hazards + frontiers + reclamation threshold)

Table 1: Comparison of HP++ with recent reclamation schemes that are robust and widely applicable.

Data structure	HP	DEBRA+	NBR	RCU/ EBR	**HP++**, PEBR, VBR
linked list [32]	✗	✗	▲	✓	▲
linked list [30]	✗	*	✓	✓	✓
linked list [44]	✓	*	✗	✓	✓
partially ext. BST [24]	✗	✗	**	✓	✓
ext. BST [26]	✓	*	✓	✓	✓
ext. BST [48]	✗	*	✓	✓	✓
ext. BST [25]	✓	*	✗	✓	✓
ext. BST [18]	✗	✗	▲	✓	▲
int. BST [36]	✗	*	✓	✓	✓
int. BST [50]	✗	✗	✗	✓	✓
partially ext. AVL [6]	✓	✗	✗	✓	✓
partially ext. AVL [24]	✗	✗	✗	✓	✓
ext. relaxed AVL [31]	✗	✓	✓	✓	✓
ext. AVL [8]	✗	✓	✓	✓	✓
patricia trie [53]	✗	*	▲	✓	▲
ext. chromatic tree [9]	✗	✓	✓	✓	✓
ext. (a,b)-tree [8]	✗	✓	✓	✓	✓
ext. interpol. tree [10]	✗	✗	✗	✓	▲

Table 2: Applicability of reclamation schemes. ✓: supported. ✗: not supported. ▲: supported but wait-freedom not preserved (§4.3) *: requiring significant design effort for data structure-specific recovery. **: requiring code restructuring to satisfy the scheme's assumption.

On the other hand, NBR uses POSIX signal for neutralization and non-local jump in the signal handler for failure handling (§2.4). This technique was first introduced in DEBRA+ [11]. The problem with DEBRA+ is that it requires recovery code that is challenging to design. One of the reasons is that the code between a non-local jump's anchor (sigsetjmp) and jumper (siglongjmp) should follow general rules to prevent the undefined behaviors, *e.g.*, it should not modify any global variables, (de)allocate heap memory, or invoke system calls. DEBRA+ proposes a recovery strategy based on operator descriptor and helping. The strategy applies to some data structures [8, 9, 31], but recovery for the other data structures has not been explicitly discussed.

NBR can be understood as a principled methodology to improve the applicability of DEBRA+. To simplify the recovery strategy, NBR requires data structure operations to be split into read and write phases, where the former does not write to the shared memory and the latter does not traverse to more nodes. But this assumption, or more specifically, the condition (3) of access-aware data structures (§4.2), is not satisfied by several data structures because **(1)** they restart from an intermediate node after helping [25, 44] or after failing to update the terminal node [10]; or **(2)** they read and write to the shared memory in an interleaved manner during an internal cleanup [6, 24] or during delete operation [50]. However, NBR is likely to outperform HP++ and PEBR, because it does not require explicit protection and validation during the read phase thanks to the immediacy and asynchrony of signals.

VBR [54] is an optimistic method that allows accessing possibly reclaimed objects and then verifying that the access was valid. Unlike the earlier optimistic schemes OA [17], AOA [16] and FA [15] which only allow optimistic reads and use hazard pointers to pessimistically protect write accesses, VBR is fully optimistic. To allow optimistic writes, VBR attaches the version number to each mutable field of objects and updates the field and its version atomically using wide CAS. When an object or its field is outdated, the operation fails and the client should handle it manually. Thanks to maximal optimism, VBR incurs only a small validation overhead and is likely to outperform HP++ and PEBR. However, optimistic methods require a custom user-level allocator that does not return memory to the OS (or a custom segmentation fault handler) to allow accessing reclaimed objects. VBR additionally requires that the allocator preserve the type of memory blocks even when reallocated to avoid the ABA problem in updates.

ACKNOWLEDGMENTS

We thank the SPAA 2023 reviewers for their concrete, in-depth, and valuable feedback. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT2201-06.

REFERENCES

[1] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2017. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 483–498. https://doi.org/10.1145/3064176.3064214

[2] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. ThreadScan: Automatic and Scalable Memory Reclamation. *ACM Trans. Parallel Comput.* 4, 4, Article 18 (may 2018), 18 pages. https://doi.org/10.1145/3201897

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

[4] Daniel Anderson, Guy E. Blelloch, and Yuanhao Wei. 2022. Turning Manual Concurrent Memory Reclamation into Automatic Reference Counting. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 61–75. https://doi.org/10.1145/3519939.3523730

[5] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

[6] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 257–268. https://doi.org/10.1145/1693453.1693488

[7] Trevor Brown. 2017. Reclaiming memory for lock-free data structures: there has to be a better way. *CoRR* abs/1712.01044 (2017). arXiv:1712.01044 http://arxiv.org/abs/1712.01044

[8] Trevor Brown. 2017. Techniques for Constructing Efficient Lock-free Data Structures. https://doi.org/10.48550/ARXIV.1712.05406

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

[10] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. 2020. Non-Blocking Interpolation Search Trees with Doubly-Logarithmic Running Time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 276–291. https://doi.org/10.1145/3332466.3374542

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

[12] Windows Dev Center. 2023. FlushProcessWriteBuffers function. https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-flushprocesswritebuffers

[13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/2150976.2150998

[14] E.F. Codd, S.B. Codd, and C.T. Salley. 1993. *Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate.* Codd & Associates.

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

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

[17] Nachshon Cohen and Erez Petrank. 2015. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (Portland, Oregon, USA) *(SPAA '15)*. Association for Computing Machinery, New York, NY, USA, 254–263. https://doi.org/10.1145/2755573.2755579

[18] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

[19] 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. https://doi.org/10.1109/TPDS.2011.159

[20] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele Jr. 2002. Lock-free reference counting. *Distributed Computing* 15, 4 (01 Dec 2002), 255–271. https://doi.org/10.1007/s00446-002-0079-z

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

[22] 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. https://doi.org/10.1145/2926697.2926699

[23] 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

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

[25] 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. https://doi.org/10.1145/2611462.2611486

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

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

[28] Keir Fraser. 2004. *Practical lock-freedom.* Ph. D. Dissertation.

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

[30] 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.

[31] Meng He and Mengdu Li. 2017. Deletion without Rebalancing in Non-Blocking Binary Search Trees. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 70)*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 34:1–34:17. https://doi.org/10.4230/LIPIcs.OPODIS.2016.34

[32] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. 2006. A Lazy Concurrent List-Based Set Algorithm. In *Principles of Distributed Systems*, James H. Anderson, Giuseppe Prencipe, and Roger Wattenhofer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–16.

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

[34] Maurice Herlihy and Nir Shavit. 2012. *The Art of Multiprocessor Programming, Revised Reprint* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[35] 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. https://doi.org/10.1145/78969.78972

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

[37] Jaehwang Jung, Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2023. Applying Hazard Pointers to More Concurrent Data Structures. https://cp.kaist.ac.kr/gc

[38] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A Promising Semantics for Relaxed-Memory Concurrency. *SIGPLAN Not.* 52, 1 (jan 2017), 175–189. https://doi.org/10.1145/3093333.3009850

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

[40] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. 2017. Repairing Sequential Consistency in C/C++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 618–632. https://doi.org/10.1145/3062341.3062352

[41] Linux Programmer's Manual. 2023. membarrier(2) - Linux manual page. http://man7.org/linux/man-pages/man2/membarrier.2.html

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

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

[44] 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. https:

//doi.org/10.1145/564870.564881
[45] Maged M. Michael. 2002. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing* (Monterey, California) *(PODC '02)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/571825.571829
[46] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (June 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8
[47] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *PODC 1996*.
[48] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP '14)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2555243.2555256
[49] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 987–1002. https://doi.org/10.1145/3453483.3454090
[50] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *Proceedings of the 16th International Conference on Distributed Computing and Networking* (Goa, India) *(ICDCN '15)*. Association for Computing Machinery, New York, NY, USA, Article 37, 10 pages. https://doi.org/10.1145/2684464.2684472
[51] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *SPAA 2017*.
[52] Gal Sela, Maurice Herlihy, and Erez Petrank. 2021. Linearizability: a Typo. *CoRR* abs/2105.06737 (2021). arXiv:2105.06737 https://arxiv.org/abs/2105.06737
[53] Niloufar Shafiei. 2019. Non-Blocking Patricia Tries with Replace Operations. *Distrib. Comput.* 32, 5 (oct 2019), 423–442. https://doi.org/10.1007/s00446-019-00347-1
[54] Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures* (Virtual Event, USA) *(SPAA '21)*. Association for Computing Machinery, New York, NY, USA, 443–445. https://doi.org/10.1145/3409964.3461817
[55] Gali Sheffi and Erez Petrank. 2022. The ERA Theorem for Safe Memory Reclamation. *CoRR* abs/2211.04351 (2022). https://doi.org/10.48550/arXiv.2211.04351 arXiv:2211.04351
[56] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. 2021. NBR: Neutralization Based Reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 175–190. https://doi.org/10.1145/3437801.3441625
[57] R. K. Treiber. 1986. Systems programming: coping with parallelism.
[58] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based memory reclamation. In *PPoPP 2018*.
[59] Wikipedia. 2023. ABA problem. https://en.wikipedia.org/wiki/ABA_problem

A SAFETY PROOF

We restate and prove the safety of our algorithms (Theorem 4.1), first for Algorithm 3 and then for its optimized variant Algorithm 5.

Theorem A.1 (Safety of Algorithm 3). *Suppose a concurrent data structure, S, reclaims nodes using Algorithm 3 and satisfies Assumption 1. Then S does not incur use-after-free.*

Proof. To demonstrate the absence of use-after-free, we prove that for each pointer q, every dereference of q *happens before* (denoted by $\overset{hb}{\longrightarrow}$) q's reclamation.

Consider each dereference of q. By an assumption of hazard pointers, the dereference happens between the announcement and revocation of its protection on a hazard pointer slot.[15] For convenience, we label the protector thread's events as follows:

- P1: Announce protection of q (line 6)
- P2: Issue SC fence (line 7)
- P3: Check p is not invalidated yet (line 8)
- P4: Check p still points to q (line 10)
- P5: Dereference q
- P6: Revoke protection of q

Suppose q is reclaimed. Then it should have been unlinked at line 16 and scheduled for invalidation at line 18 in TryUnlink and then freed at line 35 in Reclaim. Similarly, we label the unlinker and reclaimer threads' events in the happens-before order as follows:

- R1: Unlink q (line 16)
- R2: Invalidate q (line 26)
- R3: Issue SC fence (line 29)
- R4: Add q to the retired set (line 31)
- R5: Remove q from the retired set (line 33)
- R6: Check q is not protected by any hazard pointer slots (line 34)
- R7: Free q (line 35)

Here, R1-R4 belongs to the unlinker thread and R5-R7 to the reclaimer thread, but the whole events are strictly ordered due to the synchronization between R4 and R5 on the retired set.

Now we reformulate our proof goal as P5 $\overset{hb}{\longrightarrow}$ R7. We prove the goal by case analysis on the order of P2 and R3. Such a case analysis is safe even in relaxed-memory concurrency models [5, 38, 40] because the two events are SC fences.

Case 1: P2 $\overset{hb}{\longrightarrow}$ R3. The proof is the same as the original hazard pointers. Since P1 $\overset{hb}{\longrightarrow}$ P2 $\overset{hb}{\longrightarrow}$ R3 $\overset{hb}{\longrightarrow}$ R6 and the announcement of protection at P1 makes the branch condition at R6 false, the protection should be revoked before R6, or in other words, we have P6 $\overset{hb}{\longrightarrow}$ R6. As such, we have P5 $\overset{hb}{\longrightarrow}$ P6 $\overset{hb}{\longrightarrow}$ R6 $\overset{hb}{\longrightarrow}$ R7.

Case 2: R3 $\overset{hb}{\longrightarrow}$ P2. In this case, q is unlinked before being freshly protected, which is forbidden in the original hazard pointers. As discussed in §3.1, we rely on p's unlinker to invalidate p and to forbid the traversal from p to q. But invalidation is not enough because there is a gap between p's unlink at line 16 and invalidation at line 26. In other words, invalidation is not a sound over-approximation but an under-approximation of the unreachable pointer set. To patch up the problems arising from this under-approximation, p's unlinker protects q between p's unlink and invalidation.

To see how it guarantees safety, we first label the events of the thread that unlinks p in the happens-before order as follows:

- U1: Announce protection of p's next pointers (line 14)
- U2: Unlink p (line 16)
- U3: Invalidate p (line 26)
- U4: SC fence (line 29)
- U5: Revoke protection of p's next pointers (line 30)

By the strict ordering of SC fences, we have either P2 $\overset{hb}{\longrightarrow}$ U4 or U4 $\overset{hb}{\longrightarrow}$ P2. But the latter is impossible because then we have U3 $\overset{hb}{\longrightarrow}$ U4 $\overset{hb}{\longrightarrow}$ P2 $\overset{hb}{\longrightarrow}$ P3, contradicting the assumption that P3's check succeeds. As such, we have P2 $\overset{hb}{\longrightarrow}$ U4.

Since R1 $\overset{hb}{\longrightarrow}$ R3 $\overset{hb}{\longrightarrow}$ P2 $\overset{hb}{\longrightarrow}$ P4, we have that p still points to q even after q is unlinked. Then by the assumption of hazard pointers on unlinking, p is also unlinked *not after* so is q. In other words, we have either U2 $\overset{hb}{\longrightarrow}$ R1 (p is unlinked before so is q) or U2 = R1 (p and q are unlinked together). But the latter is impossible because then we have U3 = R2 $\overset{hb}{\longrightarrow}$ R3 $\overset{hb}{\longrightarrow}$ P2 $\overset{hb}{\longrightarrow}$ P3, contradicting the assumption that the check at P3 succeeds. As such, we have U2 $\overset{hb}{\longrightarrow}$ R1.

[15] The hazard pointer slot may not be invalidated indefinitely, but then q will not be reclaimed and thus never incurs use-after-free.

By Assumption 1, U1 announces the protection of q. Since U1 $\overset{hb}{\to}$ U2 $\overset{hb}{\to}$ R1 $\overset{hb}{\to}$ R6 and the announcement of q's protection at U1 makes the branch condition at R6 false, we have U5 $\overset{hb}{\to}$ R6.

Putting it all together, we have P1 $\overset{hb}{\to}$ P2 $\overset{hb}{\to}$ U4 $\overset{hb}{\to}$ U5 $\overset{hb}{\to}$ R6. Since the announcement of q's protection at P1 makes the branch condition at R6 false, we have P6 $\overset{hb}{\to}$ R6. As such, we have P5 $\overset{hb}{\to}$ P6 $\overset{hb}{\to}$ R6 $\overset{hb}{\to}$ R7, concluding the proof. □

Lemma A.2 (Piggyback of ReadEpoch). *Suppose e is an epoch and L1 and L2 are invocation events of light fence in ReadEpoch that return e and $e+2$, respectively. If L1 $\overset{hb}{\to}$ L2, then there exists an invocation event of heavy fence, say H, such that L1 $\overset{hb}{\to}$ H $\overset{hb}{\to}$ L2.*

Proof. Consider the invocation of FenceEpoch that increases fence_epoch to $e+2$. By the strict ordering of asymmetric fences, we have either L1 $\overset{hb}{\to}$ H or H $\overset{hb}{\to}$ L1. But the latter is impossible because then $e+1$ read at line 18 before H should be visible to the read of e at line 25 after L1, contradicting the coherence. On the other hand, we have H $\overset{hb}{\to}$ L2 because $e+2$ written at line 20 after H is read at line 22 before L2. As such, we have L1 $\overset{hb}{\to}$ H $\overset{hb}{\to}$ L2. □

Theorem A.3 (Safety of Algorithm 5). *Suppose a concurrent data structure, S, reclaims nodes using Algorithm 5 and satisfies Assumption 1. Then S does not incur use-after-free.*

Proof. In the proof of Theorem 4.1, SC fence is used at R3 and U4. A heavy fence is still issued in the place of R3 at line 19 in Algorithm 5. For U4, we instead prove that there exists a heavy fence event, say H, that satisfies U3 $\overset{hb}{\to}$ H $\overset{hb}{\to}$ U5. Then the rest of the proof remains unaffected.

In Algorithm 5, U5 is executed either at line 14 in Reclaim or at line 7 in DoInvalidation. In the former, a heavy fence is executed in U5's previous line; and in the latter, we can apply Lemma A.2 from the branch condition. In either case, we have U3 $\overset{hb}{\to}$ H $\overset{hb}{\to}$ U5. □

B IN DEPTH DISCUSSION OF HP++'S APPLICABILITY

We discuss how to apply HP++ to the following concurrent data structures. Assumption 1 is satisfied for the data structures in consideration, as they all mark nodes before detaching, and mark nodes become immutable.

We first note the differences between Table 2 and the original analysis by Singh et al. [56]. We argue that HP **(1)** does not properly support Harris's list [30] (§2.3); and **(2)** supports Ellen et al. [25, 26]'s binary search trees (BSTs). In particular, their delete operation registers a *descriptor* so that the other *helper* threads can execute the operation on behalf of the deleter thread. The helper threads traverse from the descriptor so that it can validate the protections required for traversal by reading the same descriptor to prove the reachability of the protected nodes.

HP++ supports many lists [32, 44] for the same reason as Harris's list [30]; those data structures supported by NBR because they are access-aware (§4.2); and Ellen et al. [25]'s and Bronson et al. [6]'s trees because they are already supported by the original HP (§4.2). We can easily recover from protection failures in these data structures because **(1)** a node is protected before writing to the shared memory so that the operation is safe to restart; **(2)** a node is protected while holding its lock so that it cannot be invalidated and the protection never fails; or **(3)** a node is protected for helping the other threads so that the operation is safe to terminate without helping.

Ramachandran and Mittal [50] presents a lock-free internal BST that removes a node by promoting the largest key of its subtree. An operation first starts in a read-only traversal phase, where protection failure can simply restart. The interesting points are deletion, especially when an internal node with two children is deleted. When a node to delete, say n is found, we **(1)** traverse the right-subtree to find the smallest successor node say s; **(2)** mark s and update the key of n with the key of s; **(3)** remove s; **(4)** replace n with an unmarked version of n; and **(5)** we may perform helping and additional traversals from n as needed. Our actions on protection differ depending on the execution of the linearization point in (2), when the update on n succeeds. If protection from n fails before the update, then another thread has deleted this node first, so the operation restarts. If protection from n fails after the update then another thread has done helping, so the operation can simply return. If protection from s fails, it must be after the linearization point, so we can simply return. All the other protection failures must have been during a traversal from n, so we can simply restart the traversal from n.

Drachsler et al. [24] presents an external BST with logical ordering in the form of a sorted doubly linked list. In an update, a search will first traverse down the tree, and then traverse the list to find the desired node. Then, the thread will acquire all locks related to the desired node and perform updates. The operation can easily be recovered from protection failures as follows. **(1)** Before acquiring the first lock, the traversal is read-only, so we can simply restart the operation. **(2)** After acquiring a lock for a node edge, protection against outgoing edges is guaranteed to succeed as modifying it would require locking the node. As such, no restarts can happen while holding a lock.

Brown et al. [10] is an interpolation tree that performs rebuilding when too many nodes become empty, by replacing it with an optimized subtree. Similar to other trees, the nodes first start in a read-only traversal phase, where protection failure can simply restart. If the terminal node is marked for rebuilding, an update is performed, and the subtrees found during the traversal paths are rebuilt if necessary. Otherwise, helping is performed on the found node. During rebuilding, if protection fails, then the node of interest has already been unlinked, hence rebuilding is done so the helping can simply terminate.

C THE FULL EXPERIMENTAL EVALUATION RESULTS

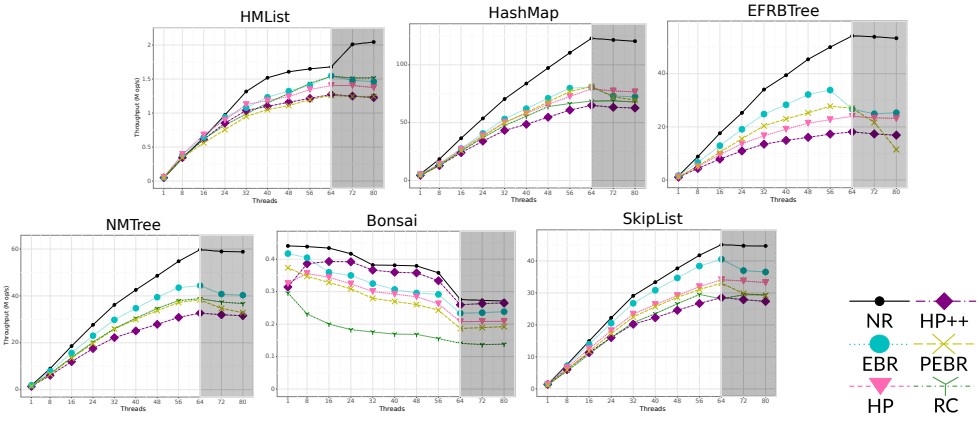


Figure 12: Throughput (million operations per second) of write-only workloads for a varying number of threads.

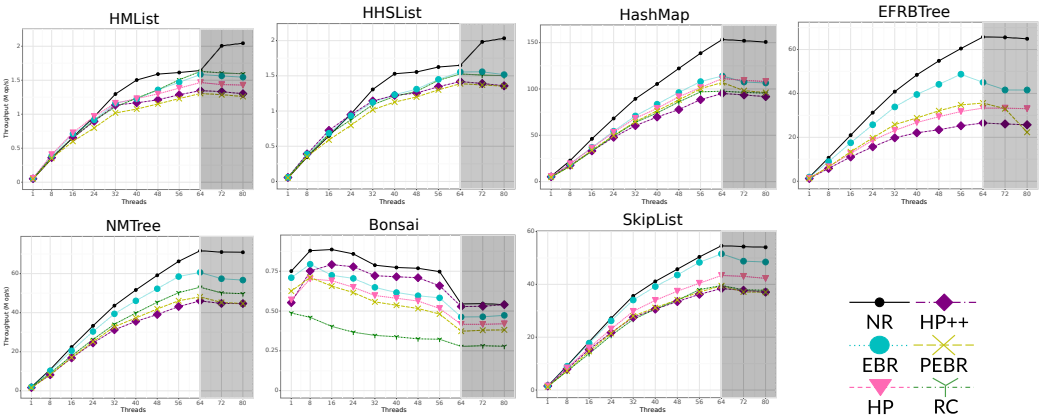


Figure 13: Throughput (million operations per second) of read-write workloads for a varying number of threads.

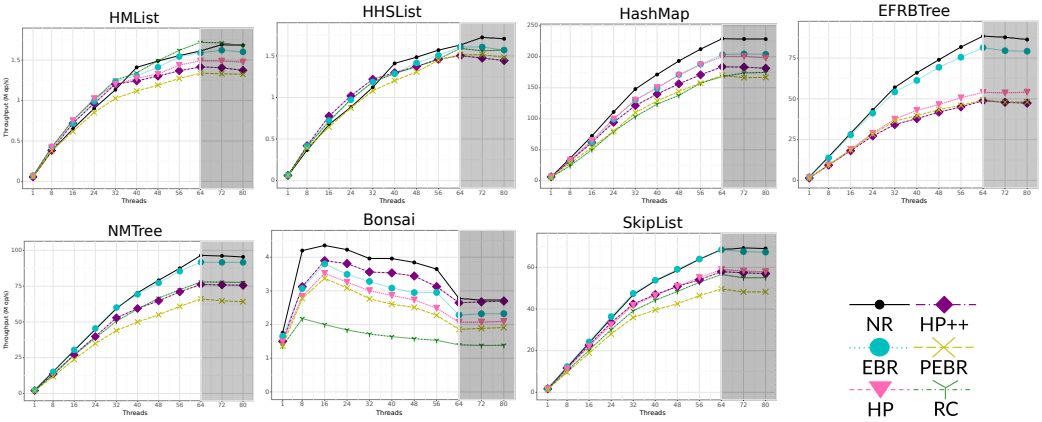


Figure 14: Throughput (million operations per second) of read-most workloads for a varying number of threads.

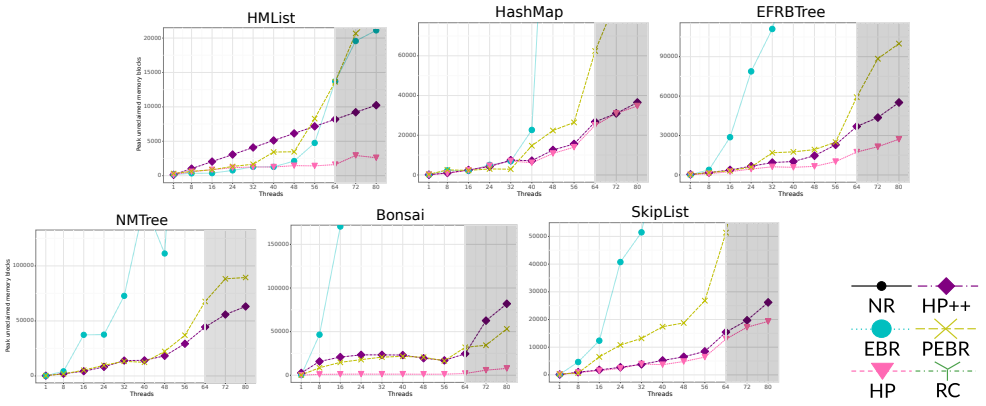


Figure 15: Peak number of unreclaimed blocks of write-only workloads for a varying number of threads.

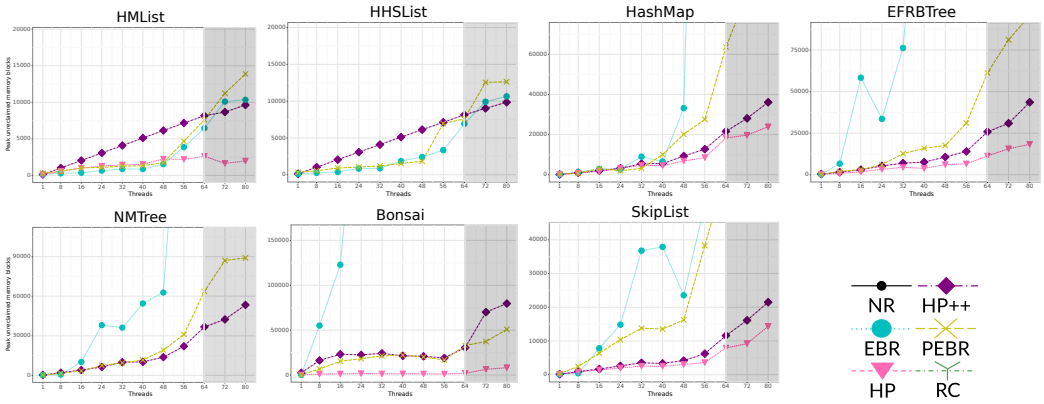


Figure 16: Peak number of unreclaimed blocks of read-write workloads for a varying number of threads.

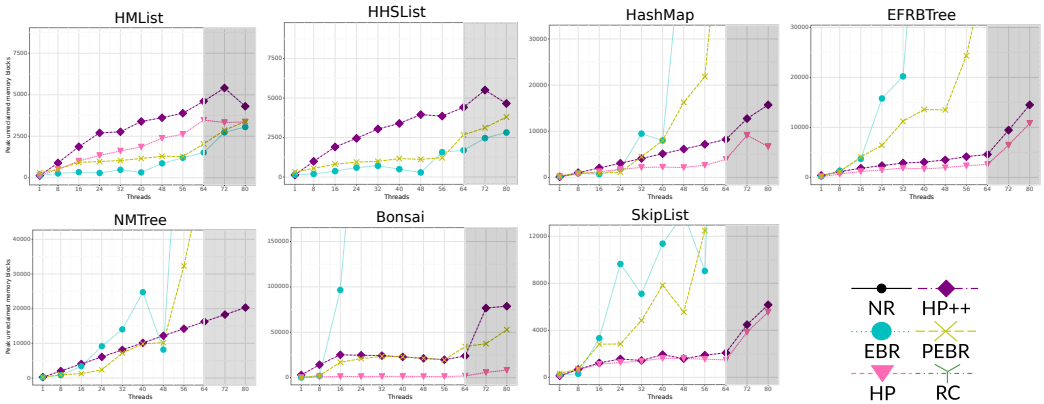


Figure 17: Peak number of unreclaimed blocks of read-most workloads for a varying number of threads.

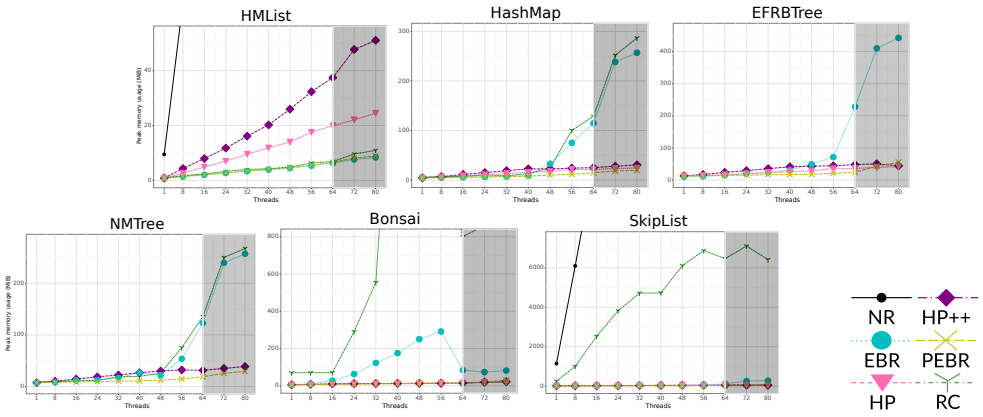


Figure 18: Peak number of memory usage of write-only workloads for a varying number of threads.

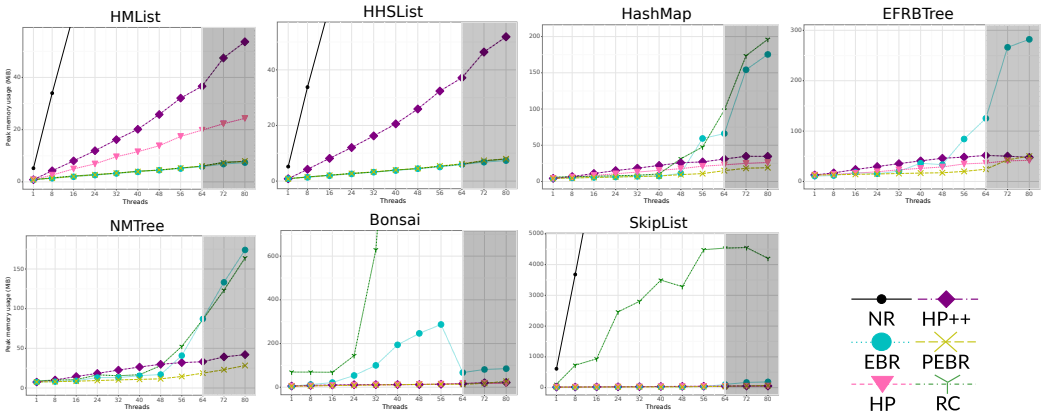


Figure 19: Peak number of memory usage of read-write workloads for a varying number of threads.

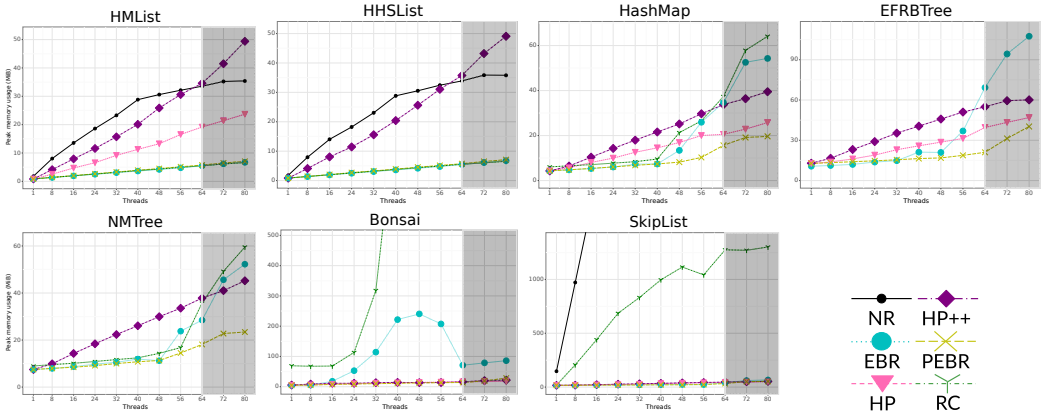


Figure 20: Peak number of memory usage of read-most workloads for a varying number of threads.

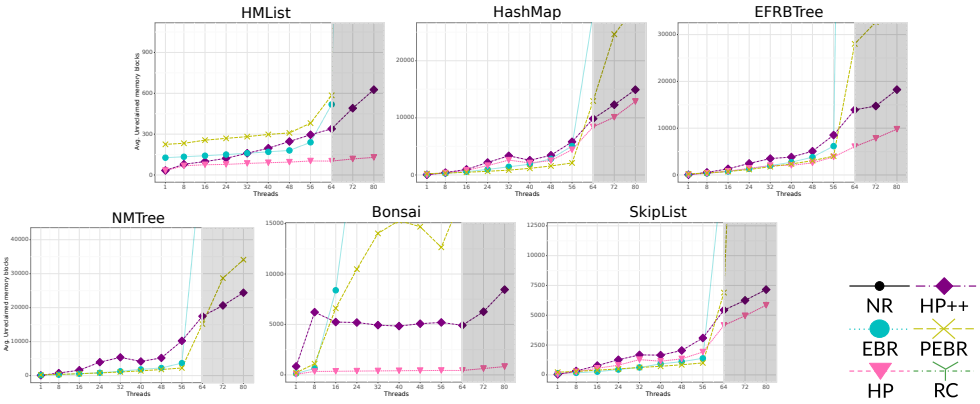


Figure 21: Average number of unreclaimed blocks of write-only workloads for a varying number of threads.

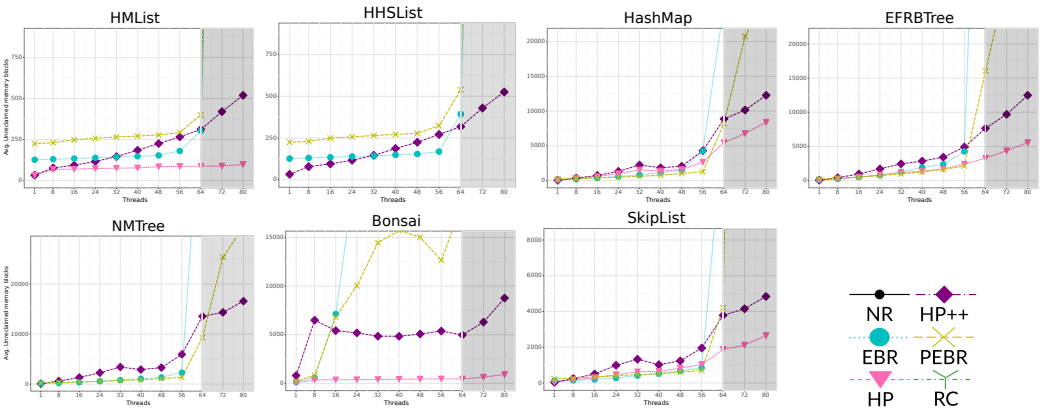


Figure 22: Average number of unreclaimed blocks of read-write workloads for a varying number of threads.

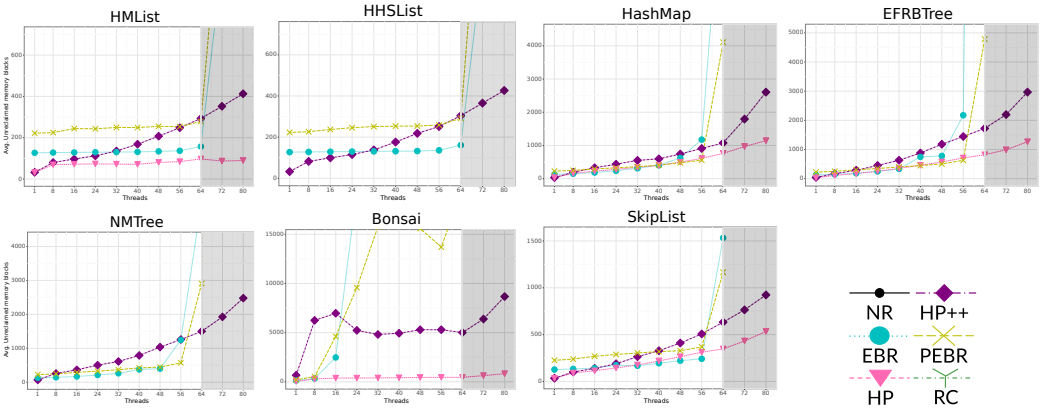


Figure 23: Average number of unreclaimed blocks of read-most workloads for a varying number of threads.