Expediting Hazard Pointers with Bounded RCU Critical Sections

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

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

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

ABSTRACT

Reclamation schemes for concurrent data structures tackle the challenge of synchronizing memory accesses and reclamation. Early schemes faced a tradeoff between *robustness* and *efficiency*: *hazard pointers* (HP) bounds the number of unreclaimed nodes, but it is inefficient due to per-node protection; and *RCU* sacrifices robustness for efficiency as a single thread may block the entire reclamation. Recent schemes attempt to break the tradeoff by sending signals to blocking threads to abort their operations. However, they are **(1)** inefficient due to starvation in long-running operations and frequent signals, and **(2)** inapplicable to a wide class of data structures.

We design a novel reclamation scheme that overcomes the above limitations. To address the long-running operations and applicability, we propose **HP-RCU**, integrating *RCU-expedited* traversal that *alternates* between HP and RCU phases. To additionally ensure robustness against stalled threads, we develop **HP-BRCU** by modularly replacing RCU with *bounded RCU* (BRCU) that efficiently bounds the duration of RCU phases by rarely sending signals. We show that HP-BRCU is robust, widely applicable, and as efficient as RCU, outperforming robust schemes across various workloads.

CCS CONCEPTS

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

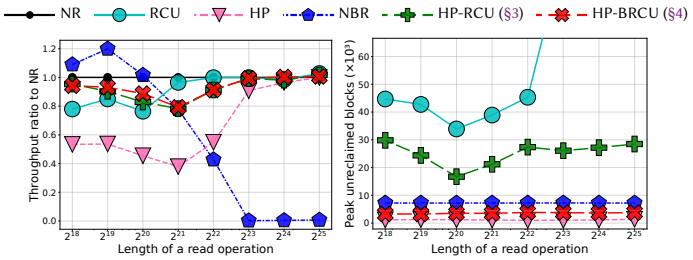
KEYWORDS

concurrency, memory management, hazard pointers, read-copy-update

ACM Reference Format:
Jeonghyeon Kim, Jaehwang Jung, and Jeehoon Kang. 2024. Expediting Hazard Pointers with Bounded RCU Critical Sections. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2024), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 34 pages. https://doi.org/10.1145/3626183.3659941

1 INTRODUCTION

Reclamation schemes [1, 2, 8, 18, 19, 25, 31, 32, 35, 36, 40, 43, 45, 48, 50] for concurrent data structures tackle the challenging task of synchronizing memory accesses and reclamation. Without proper synchronization, concurrent data structures would incur safety errors such as use-after-free and more subtle ABA problems [51]. To prevent such errors, data structures employ reclamation schemes to *retire* the unlinked nodes to schedule their reclamation, and *protect* the nodes that might be in use to defer their reclamation sufficiently.



(a) Throughput **(b) Peak # of unrecl. blocks**

Figure 1: Throughput and peak number of unreclaimed blocks of long-running read operations.

Early reclamation schemes face a tradeoff between *robustness* and *efficiency*. *Hazard pointers* [35, 36] (HP) is robust in that it bounds the number of retired yet unreclaimed nodes by the number of *individually* protected local pointers. However, HP is not efficient due to the per-node traversal overhead incurred by individual protection. On the other hand, *RCU* [31, 32] provides efficient coarse-grained protection for local pointers without per-node protections. Nevertheless, RCU is not robust because a single thread that is either **(1)** *stalled* (*e.g.*, preempted) or **(2)** executing *long-running operation* (*e.g.*, long traversal and OLAP [9]) may logically protect *all* nodes in the memory, thereby blocking their reclamation.

Several recent reclamation schemes [1, 2, 8, 48, 49] attempt to break the tradeoff between robustness and efficiency through *signal-based rollback*. For efficiency, DEBRA+ [8] and NBR(+) [48, 49] protect local pointers in a coarse-grained fashion, similar to RCU. For robustness, upon reaching the batch threshold of retired nodes, they send signals to blocking threads, *forcefully* aborting and rolling back their ongoing operations. This process ensures that old retired nodes are no longer accessed and thus safe to reclaim.

Problem. However, the coarse-grained signal strategy of DEBRA+ and NBR(+), in fact, sacrifices efficiency for robustness. **(1)** The application is made starving in long-running operations, which continuously roll back without ever reaching completion if the duration of operations exceeds the signal frequency. For example, Figure 1 depicts the throughput and peak number of unreclaimed blocks for long-running read operations under a heavy load of reclamation. While NBR(+) maintains constant memory usage, its throughput declines nearly to zero after reaching a specific length of an operation. **(2)** The frequent signals degrade the performance. For each batch, NBR sends expensive signals to *all* concurrent threads to abort potentially ongoing operations, whose overhead is not easily amortized even with a large batch threshold: while it may

SPAA 2024, June 17–21, 2024, Nantes, France.

ACM ISBN 979-8-4007-0416-1/24/06
https://doi.org/10.1145/3626183.3659941

alleviate the overhead of signaling, it increases the cache misses during traversals and reclamations due to a large memory footprint.

Additionally, DEBRA+ and NBR(+) do not apply to a wide class of concurrent data structures. DEBRA+ necessitates recovery code for cleaning up aborted operations, but it remains unclear how to design this for general data structures. In particular, DEBRA+ does not apply to those data structures that internally use locks [10, 21]. While NBR has wider applicability, it is still limited to *access-aware* [46] data structure, where after an operation performs writes, successive reads during traversal must start only from data structure entry points. As such, NBR does not apply to data structures such as the Harris-Michael list [36] and lock-free skip list [22].

Contributions. We present a robust, efficient, and widely applicable [46] reclamation scheme that resolves the above limitations.

In §3, we first build our initial solution **HP-RCU** that breaks the tradeoff between efficiency and robustness in a long-running traversal. Specifically, it *alternates* between the phases of HP and RCU during the traversal, which we refer to as *RCU-expedited* traversal. For efficiency, the majority of traversal steps (*i.e.*, following links) occur in RCU phases, circumventing most of HP's per-node overheads. To address long-running operations, the traversal periodically *checkpoints* acquired local pointers with HP and re-initiates the RCU phase, prompting the reclamation process. Moreover, it is more applicable than HP and NBR because **(1)** it supports optimistic traversal [24], and **(2)** it can resume a traversal not only from the entry points but also from an arbitrary checkpointed node. Figure 1 demonstrates that HP-RCU maintains consistent throughput while showing a reasonably bounded memory footprint.

In §4, we develop our full solution **HP-BRCU** that additionally provides robustness against stalled threads. We first design *bounded RCU* (BRCU), a separate epoch-based RCU implementation [18, 19] that captures the essence of signal-based rollback to efficiently *bound* the duration of critical sections. We then modularly replace RCU with BRCU in HP-RCU. BRCU's signaling policy is more efficient than NBR(+) because it *selectively* sends signals, targeting only the slow threads. Indeed, Figure 1 shows that HP-BRCU maintains almost the same throughput as HP-RCU, while exhibiting a comparable memory footprint to HP and NBR(+).

In §5 and §6, we theoretically and experimentally demonstrate that HP-BRCU is indeed robust; as efficient as RCU, outperforming the existing robust schemes such as HP, NBR(+), and VBR [45] for a variety of workloads; and applicable to a wide class of data structures including Harris's list [19] with optimistic traversal [24], Harris-Michael list [34] and Natarajan-Mittal tree [37] with helping, and lock-free skiplist [22] with reference counting.

In §7, we conclude the paper by discussing related works. To the best of our knowledge, HP-BRCU is the only scheme that achieves robustness against stalled threads and long-running operations while maintaining efficiency (see Table 2 for details).

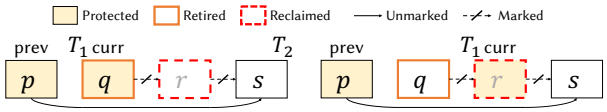
2 BACKGROUND AND MOTIVATION

2.1 Hazard Pointers

HP [35, 36] guarantees safety by preventing the reclamation of each thread's local pointers protected individually. Algorithm 1 presents

Algorithm 1 HP interface and implementation

```
1:  struct Shield
2:      function new() → Shield
3:      method protect(ptr: void*)
4:  function ProtectFrom<T>(shield: &Shield, src: &Atomic<T>)
5:      ptr ← src.load()
6:      loop
7:          shield.protect(ptr); fence(SC)
8:          ptr_validate ← src.load()
9:          if ptr = ptr_validate then return
10:         ptr ← ptr_validate
11: function Retire<T>(ptr: T*)
12: function Reclaim()
13:     take retired pointers; fence(SC)
14:     check shields and reclaim unprotected retired pointers
```



(a) T_2 **physically unlinks** q **and** r. **(b)** $T1$ **protects and accesses** r.

Figure 2: Use-after-free error of Harris's list with HP.

HP's interface and implementation.[1] A Shield is a protection slot for a local pointer to a shared object. A thread may create a new() shield and protect() a local pointer. However, even after protecting it, a pointer is still unsafe to dereference because a concurrent thread may already have Retired and Reclaimed it. To ensure safety, the thread should *validate* that the protected pointer is not retired yet. Validation strategy drastically differs among data structures, and it is one of the most challenging aspects of using HP [46].

In data structures where each node is unlinked from its predecessor before being retired, it is possible to employ a validation strategy that simply checks whether the node is still reachable from its predecessor. The ProtectFrom helper function streamlines protection validation for those cases. It first loads a local pointer from a shared source location (line 5). If the pointer remains the same (line 9) after publishing its protection (line 7), it is reachable from the source and thus not retired yet. Then even after another thread Retires that pointer, a subsequent call of Reclaim will observe the announced protection (line 14), delaying the reclamation.

Tradeoff. HP is robust against stalled threads and long-running operations: retired nodes remain unreclaimed only if they are individually protected by a shield, and the number of shields is bounded by client data structures. However, HP is not efficient: it degrades the performance of the original data structure because it incurs *per-node overhead*. Specifically, each traversed node needs to be written to a shield and re-read for validation. We observe that HP underperforms RCU for read-heavy workloads (see §6).

Moreover, validation makes it challenging to apply HP to highly efficient *optimistic traversal* strategy [6, 14, 19, 23, 24, 37, 42] that follows a link from possibly retired nodes. For instance, Figure 2

[1] We present all algorithms in Rust-style pseudo codes. We also assume C/C++ memory model, but omit the release and acquire orderings for concision; and write fence(SC) for atomic_thread_fence(memory_order_seq_cst).

Algorithm 2 RCU interface

```
1: function CriticalSection<R>(body: fn() → R) → R
2: function Defer(task: fn())
```

illustrates a potential problem of applying HP to Harris's list [19]. Figure 2a: Suppose a thread, say T_1, traverses and protects p and q and another thread, say T_2, unlinks and retires q and r from the list at once. Since r is not protected, it can be reclaimed. Figure 2b: But then T_1 may follow the link from the retired node q to r, leading to use-after-free error without validation. HP++ [24] generalizes HP to support optimistic traversal by protecting more nodes (*e.g.*, r by T_2), but its additional protection incurs performance overhead.

2.2 Epoch-Based RCU

RCU [31, 32] is a general task scheduler that applies also to reclamation. RCU overcomes HP's drawback on per-node overhead by logically protecting multiple nodes in a coarse-grained fashion. Algorithm 2 presents RCU's interface. The CriticalSection function delimits a *critical section* inside which a thread executes an operation. A thread may Defer the execution of a task until all concurrent threads exit their ongoing critical sections. When using RCU for safe reclamation, a critical section logically protects all nodes that are reachable from data structure entry points at its beginning. Once a node becomes unreachable, a thread Defers its reclamation. It is safe because deferred reclamation will not be executed until all ongoing critical sections conclude, and any critical sections occurring after the deferring cannot reach that node.

Epoch-based RCU implementations [18, 19] ensure Defer's correctness with an *epoch*, a monotonically increasing integer. For example, Fraser [18]'s implementation works as follows: **(1)** a *global epoch* is shared across all threads; **(2)** each critical section starts with assigning the global epoch to its *local epoch*; and **(3)** the epochs of any concurrent critical sections must differ by at most one. If a task is deferred at the global epoch e, then it is safe to execute it at the epoch $e + 2$ because all concurrent critical sections with the local epoch e or $e - 1$ must have exited beforehand.

Tradeoff. RCU is efficient and widely applicable: it does not incur per-node overhead and its coarse-grained protection allows optimistic traversal. However, RCU is not robust against stalled threads or long-running operations because a single critical section may logically protect *all* nodes, thereby blocking their reclamation.

2.3 Signal-Based Rollback

Several recent reclamation schemes overcome the tradeoff between robustness and efficiency [2, 8, 25, 43, 45, 48, 50], but they have their own limitations. Here, we focus on the two schemes based on *signal-based rollback* that are closely related to our solution HP-BRCU and discuss the other methods in §7.

DEBRA+ [4, 8] and NBR(+) [48, 49] use *signaling* (*e.g.*, POSIX `pthread_kill` [29]) for robust and efficient reclamation. Similarly to the efficient RCU scheme, threads in both schemes first enter a critical section (*non-quiescent state* in DEBRA+ and *read phase* in NBR) that does not require per-node protections. For robust reclamation of old retired nodes, reclaiming threads may forcefully abort the other threads within critical sections by sending signals, which makes the recipient immediately abort the critical section and *roll back*, *i.e.*, exit the critical section and re-run the interrupted task via a non-local jump (*e.g.*, POSIX `sigsetjmp`).

Efficiency. As we have already discussed in §1, prior reclamation schemes with signal-based rollback sacrifice efficiency for robustness: **(1)** DEBRA+, NBR indeed sacrifice efficiency for robustness suffering from *starvation* in long-running operations, *e.g.*, long traversal or OLAP [9]. This limitation is also applied to other schemes [25, 45] that involve a strategy of restarting an ongoing operation. **(2)** NBR(+) degrades its performance due to its frequent signals, which are sent to *all* threads for each batch. While NBR+ alleviates this problem by timestamping signals to piggyback on other threads' signals, its overhead of reclamation is not amortized easily even with a large batch threshold due to the increased cache misses from a large memory footprint. (see §6 for detail).

Applicability. Since the critical section can be aborted or re-run, it must not contain any important shared memory modifications including linearization points to avoid semantics violations.

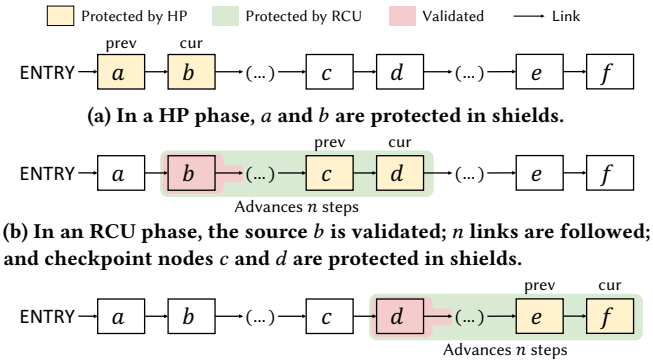
DEBRA+ requires *recovery code* to clean up the aborted operation: upon receiving a signal, the thread quits the critical section, runs the recovery code, and then restarts the operation. Brown [8] proposes a recovery strategy for *helping*-based lock-free data structures whose operations can be structured as follows: **(1)** the body of a critical section publishes an operation descriptor; **(2)** body's execution can be helped by the recovery code or the other threads; and **(3)** the rest of the operation is run outside the critical section. Since the helping procedure may run in recovery code outside critical sections, the nodes associated with the descriptor are protected in HP shields before publishing it. However, it is unclear how to apply such a helping-based recovery strategy to other data structures.

NBR(+) admits a simple recovery strategy by limiting its scope to so-called *access-aware* [46] data structures, whose operations are strictly divided into alternating sequences of *read phase* and *write phase*. The read phase functions as a critical section where only reads from the entry points are allowed. When transitioning to a write phase, the necessary nodes are protected in HP shields as in DEBRA+. This systematic interface allows application to more lock-free data structures and some lock-based data structures. However, this requirement makes NBR inapplicable to algorithms that perform helping (*e.g.*, physical deletion) during traversal, *e.g.*, Harris-Michael list [34] and lock-free skip list [22] (see Table 1 for detail). Specifically, after conducting helping writes, it must restart the entire traversal because it cannot resume from the protected nodes.

3 HP-RCU: OPTIMIZING HAZARD POINTERS WITH RCU CRITICAL SECTIONS

3.1 RCU-Expedited Traversal

HP-RCU is a backward-compatible *extension* of HP that supports *RCU-expedited traversals*. It *alternates* between the phases of HP (fine-grained protection of individual pointers) and RCU (coarse-grained protection of any reachable pointers) during traversals. Figure 3 illustrates an example traversal on a linked list. Figure 3a: Initially in an HP phase, the nodes a, b at the entry point of the linked list are protected by shields. Figure 3b: Entering an RCU phase, the traversal follows a certain number of links from



(a) In a HP phase, a and b are protected in shields.

(b) In an RCU phase, the source b is validated; n links are followed; and checkpoint nodes c and d are protected in shields.

(c) Repeats the step (3b) until reaching the destination f.

Figure 3: Example RCU-expedited traversal on a linked list.

a, b in a critical section. Similar to HP, it validates that b is still reachable from the entry point before following a link from it; but in contrast to HP, pointers acquired in RCU phases are safe to dereference without individual protection or validation. Before exiting the RCU critical section and phase, the traversal *checkpoints* ultimately acquired local pointers c, d to use in the next phase, by individually protecting them with HP shields. However, unlike HP, the protections do not need to be validated because they cannot be reclaimed thanks to the RCU critical section (see §3.2 for how it works). Figure 3c: The traversal continues by alternating between HP and RCU phases until reaching the destination f.

By following the majority of links in RCU phases, HP-RCU avoids most of HP's per-node overhead and validation efforts. At the same time, despite its use of RCU critical sections, HP-RCU enjoys HP's robustness against long-running operations when a traversal is split into multiple critical sections with a bounded number of instructions. (See §5 for a more detailed analysis.)

Example. Algorithm 3 presents an RCU-expedited search algorithm TrySearch (line 7) for Harris-Michael list [34], which searches and protects the position of a key. All RCU critical sections are highlighted in green, and HP protections are highlighted in yellow. The first critical section, triggered in InitCursor (line 14)[2], loads prev from ENTRY, initializes cur (line 16), and protects them with dedicated shields (line 17). Subsequently, the entire traversal involves iteratively executing Steps (line 18), which advances the local pointers a bounded number of times. The repetition condition and validation of cur are highlighted in red. At its beginning, Steps validates that cur, the source of the ongoing traversal, is not marked (line 24).[3] By observing that cur is not marked, the traversal ensures that cur is not retired yet and traversal is safe to resume. Throughout the traversal, when following a link from a local pointer created within the critical section, the target node, such as next (line 23), is logically protected by RCU. Upon completing a traversal with a bounded length, it protects the acquired local pointers which the client will dereference (line 32), and checks whether it has reached a destination node (line 33). If so, the traversal concludes by returning whether the key was found.

[2]While InitCursor is in an RCU phase, it can also be implemented in a HP phase.
[3]Note that this validation stems from the original data structure implementation.

Algorithm 3 RCU-expedited Harris-Michael list search

```
1:  global variable
2:      ENTRY: Atomic<Node*>                ▷ The entry point of the linked list.
3:  thread-local variables
4:      prev, cur: Node*; prev_s, cur_s: Shield
5:  enum StepResult<R>
6:      Finish(R), Continue, Fail
7:  function TrySearch(key: Key) → Option<bool>
8:      InitCursor()                        ▷ Initialize prev and cur by following ENTRY.
9:      loop
10:         match Steps(key)                ▷ Advance multiple steps at once.
11:             case Finish(found) then return Some(found)
12:             case Continue then continue
13:             case Fail then return None
14: function InitCursor()
15:     CriticalSection(λ.
16:         prev ← ENTRY.load(); cur ← (*prev).next.load()
17:         prev_s.protect(prev); cur_s.protect(cur))   ▷ Protect within CS (R2)
18: function Steps(key: Key) → StepResult<bool>
19:     return CriticalSection(λ.
20:         found ← None
21:         repeat MaxSteps times
22:             if cur = ⊥ then break
23:             next ← (*cur).next.load()
24:             if next.tag() ≠ 0 then                  ▷ Validation (R1)
25:                 next ← next.with_tag(0)
26:                 match (*prev).next.cas(cur, next)   ▷ Physical deletion
27:                     case Ok then Retire(cur); cur ← next; continue
28:                     case Err then return Fail
29:             if (*cur).key ≥ key then
30:                 found ← Some((*cur).key = key); break
31:             prev ← cur; cur ← next
32:         prev_s.protect(prev); cur_s.protect(cur)    ▷ Protect within CS (R2)
33:         if let Some(f) ← found then return Finish(f)
34:         return Continue)
```

Algorithm 4 Two-step retirement

```
1: function Retire<T>(ptr: T*)
2:     RCU-Defer(λ. HP-Retire(ptr))   ▷ From Algorithm 1 and Algorithm 2
```

3.2 Two-Step Retirement

HP-RCU is directly built on top of HP and RCU. In fact, HP-RCU reuses the original implementations of HP's Shield, Reclaim and RCU's CriticalSection without modifications. Only the Retire function is reimplemented with HP-Retire[4] (Algorithm 1) and RCU-Defer[4] (Algorithm 2), as illustrated in Algorithm 4.

Recall from §2.1 that HP-Retire schedules the reclamation of an unlinked node until it is no longer protected by any Shields. However, invoking HP-Retire for unlinked nodes in the presence of RCU phases leads to *use-after-free* errors because a traversal within a critical section follows links without any protections with HP. To ensure safety, we employ *two-step retirement*: an unlinked node's HP-Retire is RCU-Defered (line 2). Then a Retired pointer

[4]For clarity, we prepend SMR scheme names to those functions.

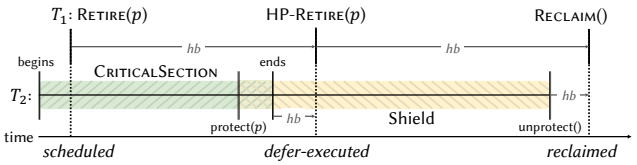


Figure 4: Timeline of retiring (by T_1) and dereferencing (by T_2) pointer p. *hb* stands for *happens-before* relation.

is eventually Reclaimed after **(1)** all ongoing critical sections conclude (coarse-grained), and then **(2)** it is no longer protected by any Shields (fine-grained). As such, a pointer acquired within a critical section is also safe to dereference without protection. Additionally, it eliminates the need for validation after protecting a pointer with a Shield if acquired within a critical section because a critical section prevents any reachable pointers from being HP-Retired.

Figure 4 presents a timeline example where a reclaimer thread T_1 and a reader thread T_2 interact with each other under the two-step retirement. Thread T_1 Retires a pointer p while T_2 is in a critical section. Due to the ongoing critical section, the execution of HP-Retire is delayed. Concurrently, T_2 protects p within its critical section, without requiring validation. Once T_2 exits its critical section, HP-Retire may be executed. But the reclamation of p is delayed again because T_2 has a shield protecting p. As a result, T_2 can safely dereference p even after a critical section concludes. The pointer p is finally reclaimed after T_2 resets the protection.

3.3 Revalidation

We delve into the safety of RCU-expedited traversals. Resuming a traversal with RCU phase from arbitrary previously protected nodes may result in a *use-after-free* error: if a previously protected node has already been unlinked from the data structure, the next links might point to reclaimed memory blocks. As such, at the beginning of the subsequent critical section, a thread must *revalidate* that the previously protected node, from which the traversal is to be resumed, is not retired yet. Recall from §2.1 that the original HP's ProtectFrom also imposes the same requirement of ensuring the source is not retired.

Notably, in lock-free data structures that incorporate the well-known technique of *logical deletion* [19, 22, 34, 37], checking whether a node is logically deleted is sufficient for revalidation. This is because a logical deletion *happens before* its physical deletion and retirement. In the earlier example of the Harris-Michael list (Algorithm 3), the original data structure implementation seamlessly streamlines revalidation (line 24). It is safe because the traversal is aborted if it **(1)** observes a logical deletion status of cur, and **(2)** fails to resolve its invalid status through physical deletion. Revalidation for other data structures can be implemented similarly by checking logical deletion status. We further formalize the safety requirements for the RCU-expedited traversal in the latter section (§5).

4 HP-BRCU: BOUNDING CRITICAL SECTIONS

We design HP-BRCU by replacing RCU in HP-RCU with BRCU: a *bounded RCU* that captures the essence of signal-based rollback (§4.1); and improve its efficiency with *abort-masking* (§4.2) and ease-of-integration with *double buffering* (§4.3).

Algorithm 5 Bounded RCU (BRCU)

```
1:  struct Local
2:      epoch: Atomic<Epoch>
3:      thread: RawThread                          ▷ used for sending signal
4:  global variables
5:      EPOCH: Atomic<Epoch>
6:      TASKS: ConcurrentQueue<(fn(), Epoch)>
7:      LOCALS: ConcurrentList<Local>
8:  thread-local variables
9:      local: Local*
10:     chkpt: Checkpoint
11:     status: Atomic<Status>                     ▷ Out (default) | INCs
12:     tasks: ArrayVec<fn(), MaxLocalTasks>       ▷ fixed-capacity vector
13:     push_cnt: Integer
14: function CriticalSection<R: Copy>(body: fn() → R) → R
15:     Checkpoint(chkpt)                          ▷ e.g., sigsetjmp
16:     status.store(INCs); local.epoch.store(EPOCH.load()); fence(SC)
17:     result ← body()
18:     local.epoch.store(⊥); status.store(Out)
19:     return result
20: function SignalHandler()
21:     if status.load() = INCs then Rollback(chkpt)   ▷ e.g., siglongjmp
22: function Defer(task: fn())
23:     tasks.push(task)
24:     if ¬ tasks.is_full() then return
25:     fence(SC); E_g ← EPOCH.load(); fence(SC)
26:     TASKS.push(tasks.pop_all(), E_g)           ▷ push tasks with tagging epoch
27:     push_cnt++
28:     for other ∈ LOCALS do
29:         E_o ← other.epoch.load()
30:         if E_o ≠ ⊥ && E_o < E_g then
31:             if push_cnt < ForceThreshold then return
32:             SendSignal(other.thread)           ▷ e.g., pthread_kill
33:     push_cnt ← 0; EPOCH.cas(E_g, E_g + 1)
34:     execute all tasks from TASKS.pop_all_expired_within(E_g − 1)
```

4.1 Bounded RCU

The BRCU algorithm inherits the concept of signal-based rollback from prior schemes, especially NBR(+) [48, 49]. Recall from §2.3 that the objective of signal-based rollback is to immediately abort ongoing critical sections, allowing others to reclaim nodes. However, prior schemes suffer from performance degradation due to frequent signals. We introduce BRCU as an optimized solution that sends signals *selectively* to slow threads, providing a significant performance advantage over NBR(+) (see §6 for performance evaluation).

Requirements. BRCU provides nearly the same methods as the original RCU (§2.2): CriticalSection and Defer. However, the body of critical sections can now be aborted midway and rolled back to the beginning upon receiving a signal. In other words, the body must not execute any transient shared memory writes such as lock acquisition or logical modification to a data structure.

Formally, we require that the body of the critical section is *abort-rollback-safe*. A *region* (*i.e.*, a sequence of operations) is *rollback-safe* [45] if executing it to completion zero or more times does not affect the operation's semantics. In addition, a region is *abort-rollback-safe* if **(1)** it is rollback-safe, and **(2)** aborting it in the middle does not affect the operation's semantics or introduce liveness

problems, such as memory leak and deadlock. For a notable example, a read-only traversal is abort-rollback-safe: restarting the traversal does not compromise the operation's semantics (*i.e.*, it eventually reaches the destination again), and aborting it will not alter the shared memory. However, Defer is not even rollback-safe as it modifies the global task registry in a rollback-unsafe manner. On the other hand, the physical deletion during a traversal (*e.g.*, line 26 in Algorithm 3) is rollback-safe but not abort-rollback-safe: attempting the same physical deletion again would simply fail, not modifying the shared memory again, but a memory leak occurs if the operation is aborted between successful deletion and Deferring a reclamation. For another example, a lock acquisition (*e.g.*, accessing the standard I/O) is also abort-rollback-unsafe, potentially leading to deadlock if aborted before releasing the lock.

Rollback-safety implies *idempotence*, but the converse does not hold because a rollback-safe operation does not alter the shared memory (modulo helping), while an idempotent operation may do.

Algorithm. BRCU maintains three internal global variables: the global epoch, the deferred task set, and the list of Local data for participating threads, each comprising a local epoch and the system's thread object (`pthread_t` in POSIX). Each thread maintains five thread-local variables: the pointer to its Local, the rollback checkpoint (`sigjmp_buf` in POSIX), the status flag indicating whether the thread is in a critical section (InCs), the deferred task set, and the number of tasks it flushed to the global task sets.

Algorithm 5 implements BRCU. All interaction steps related to the signal-based rollback policy are highlighted in cyan. The CriticalSection method (line 14) starts by creating a checkpoint (line 15) where the execution will resume upon receiving a signal. It then sets the status to InCs, indicating that the thread is within a critical section, highlighted in green where the execution can be aborted and rolled back at any point. Then it sets the local epoch, executes the given function, resets the epoch and status, and finally returns the function's result (which must not involve heap memory allocation). When a thread receives a signal, the SignalHandler (line 20) checks if the current thread is in a critical section, and if so, rolls back to the latest checkpoint (line 15).[5]

The Defer method (line 22) accumulates the given task to the local batch (line 23), and if full, it migrates the batch to the global task set after tagging it with the current global epoch (line 26). It then attempts to advance the global epoch. If all threads inside critical sections are with the latest epoch, it increments the global epoch (line 33) without signals. If there are violating threads, it gives up advancing until the threshold is reached. Once the threshold is reached, it *forcefully* advances the global epoch by aborting the violating threads' critical sections with signals (line 32). This ensures the number of the deferred tasks scheduled by a thread in an epoch does not exceed the constant MaxLocalTasks × ForceThreshold, which will be utilized in proving HP-BRCU's robustness (§5). Finally, it executes the old deferred tasks (line 34).

Our implementation utilizes the `SIGUSR1` POSIX signal [30], and assumes that signals are delivered immediately: the signaled thread is interrupted *before* the signaling thread returns from the system call (see the appendix [27] for details), which holds for Linux [26] and FreeBSD [49]. For systems without such a guarantee, we may further defer the task execution by interacting with the scheduler [4].

[5]Every access to status should be wrapped with *compiler fences* (`atomic_signal_fence` in C/C++) to prevent instruction reordering.

4.2 Abort-Masking

We design HP-BRCU by modularly replacing RCU with BRCU in HP-RCU. BRCU-expedited traversal ensures robustness against stalled threads while still preserving all the benefits of RCU-expedited one, but additionally requiring abort-rollback-safety (§4.1).

Motivation. Generally, a region becomes abort-rollback-unsafe because it comprises a write operation such as lock acquisition or retirement. For instance, the traversal of the Harris-Michael list in the original form is abort-rollback-unsafe due to physical deletion (§4.1). For this reason, such an algorithm is not supported by DEBRA+ and NBR(+). In contrast, it can be implemented with HP-BRCU across multiple BRCU critical sections as follows: upon observing a logical deletion of cur, the traversing thread **(1)** exits the critical section after protecting prev and cur; **(2)** unlinks and retires cur; and then **(3)** resumes traversal from prev in a new critical section that revalidates prev. However, this strategy involves overheads due to revalidations and critical section re-establishments.

Solution. We further overcome this drawback by equipping BRCU with *abort-masked* region that transforms a rollback-safe region into an abort-rollback-safe one. In an abort-masked region, upon receiving a signal, the thread does not immediately roll back; instead, it rolls back at the end of the masked region. Deferring the rollback is similar to running body outside a critical section. As such, body should be safe to execute without the protection of a critical section, *e.g.*, the nodes used in an abort-masked region should be protected in HP shields before entering the region (see §4.3 for example).

It is straightforward to apply the abort-masking to lock-free data structures. **(1)** The user should first identify a sub-region within the traversal that performs write operations (*e.g.*, physical deletion), and wrap that sub-region with an abort-masked region (*e.g.*, line 24 in Algorithm 8). Here, it is likely that this sub-region is already rollback-safe thanks to the concurrent nature of the data structure. Specifically, the data structure must ensure safety during concurrent executions, which implies rollback safety. **(2)** Then the user must identify the pointers used in the masked region and protect them with outliving HP shields before entering the region (*e.g.*, line 23 in Algorithm 8). This protection step mirrors the protections implemented upon exiting the critical section (*e.g.*, line 32 in Algorithm 3).

Algorithm. Algorithm 6 illustrates modifications for abort-masking, highlighted in purple. The Mask function (line 8) creates an abort-masked region executing body that defers the rollback until it exits. It first sets the thread's status to InRm, representing an abort-masked region, and executes the given body. If the thread receives a signal in this state, SignalHandler sets the status to RbReq indicating that rollback was requested, and returns to the original context. Then Mask tries switching the status from InRm back to InCs. This should be an atomic CAS because it may race with SignalHandler. If it fails, the status must be in RbReq, so it Rollbacks to the last checkpoint. Otherwise, it returns the result of body.

Algorithm 6 Abort-masking for BRCU

```
1:  thread-local variables
2:      status: Atomic<Status>          ▷ Out (default) | InCs | InRm | RbReq
3:      ...
4:  function SignalHandler()
5:      current ← status.load()
6:      if current = InCs then Rollback(chkpt)
7:      if current = InRm then status.store(RbReq)
8:  function Mask<R: Copy>(body: fn() → R) → R
9:      ▷ Precondition: called from the body of CriticalSection. ◁
10:     status.store(InRm)
11:     result ← body()
12:     if status.cas(InRm, InCs).is_err() then Rollback(chkpt)
13:     return result
```

4.3 Double Buffering

To facilitate the applicability of HP-BRCU, we present a high-level API called Traverse that enables users to seamlessly integrate HP-BRCU into the original traversal algorithm.

In contrast to HP-RCU, HP-BRCU maintains robustness against long-running operations without explicitly exiting the critical section after each checkpoint. This is because a lengthy critical section will be aborted by other threads, obviating the need for explicit alternation as in Algorithm 3. But still, periodic checkpointing with HP in critical sections remains essential to prevent starvation under long-running operations, where a thread never finishes its operation indefinitely rolling back to the entry point of the data structure. However, checkpointing becomes more challenging in that strategy because the protection of multiple pointers may be aborted in the middle. If the execution is interrupted during checkpointing, the collection of shields (*i.e.*, *protector*) falls to an *incomplete* state (*e.g.*, prev_s protects the latest one, while cur_s does not).

We address the illustrated challenge on checkpointing by employing *double buffering* [52]: rather than utilizing only one protector for checkpointing, we now incorporate another one for backup and dynamically switch between the two at each checkpoint. Then at least one is protecting a complete collection of pointers even if a traversal is rolled back during checkpointing. As such, the traversal can be resumed by starting from the complete protector.

Algorithm 7 presents the Traverse method that makes intermediate backup protections from which the traversal can resume. The entire traversal is conducted within this function with three arguments: **(1)** prot, the Protector for the ultimately acquired Cursor, **(2)** an init closure that creates the initial cursor from the entry point, and **(3)** a step closure that traverses the data structure one step from the given cursor. Protector generalizes a collection of shields (*e.g.*, the pair of prev_s and cur_s) to protect the traversal result of type Cursor (*e.g.*, the pair of prev and cur). The two functions are called within critical sections so that BRCU's requirements apply (§3.3 and §4.1). These requirements can typically be met in most lock-free data structures because the following links and reading keys are already abort-rollback-safe and occasional abort-rollback-unsafe steps can be conducted within masked regions.

The source of the traversal, src, is initialized by protecting the Cursor returned from init within a critical section (line 12). Then prots, curs, and the current index of the synchronized pair comp_idx

Algorithm 7 The Traverse API for HP-BRCU

```
1:  trait Validatable                    ▷ e.g., a set of dereferenceable pointers.
2:      method validate() → bool
3:  trait Protector                      ▷ e.g., a set of shields.
4:      type Cursor: Validatable + Copy  ▷ a target cursor for this protector.
5:      function new() → Self
6:      method protect(cur: &Cursor)
7:  enum StepResult<C, R>
8:      Finish(C, R), Continue(C), Fail
9:  function Traverse<P: Protector, R: Copy>(prot: &P,
        init: fn() → P::Cursor
        step: fn(P::Cursor) → StepResult<P::Cursor, R>
    ) → Option<(P::Cursor, R)>
10:     extra ← Protector::new()
11:     src ← CriticalSection(λ.
12:         s ← init(); extra.protect(s); return s)   ▷ Initialize the first cursor.
13:     prots ← [&extra, prot]; curs ← [src, ⊥]
14:     comp_idx ← 0                     ▷ Always points to a complete buffer.
15:     result ← CriticalSection(λ.
16:         cur ← curs[comp_idx mod 2]
17:         if ¬ cur.validate() then return None
18:         for i = 1, ... do
19:             s_res ← step(cur)        ▷ Advance a step forward.
20:             if let Finish(c, _) = s_res | Continue(c) = s_res then cur ← c
21:             if s_res.is_finish() || i mod BackupPeriod = 0 then
22:                 prots[(comp_idx + 1) mod 2].protect(cur)
23:                 curs[(comp_idx + 1) mod 2] ← cur
24:                 comp_idx++           ▷ The next buffer is now complete.
25:             if let Finish(c, r) = s_res then return Some((c, r)))
26:     if comp_idx mod 2 = 0 then
27:         swap(prots[0], prots[1])     ▷ Move the latest protection to prot.
28:     return result
```

are initialized (lines 13 and 14). These implement the double buffering policy and should be declared outside the critical section as they are shared across multiple executions of the critical section body. Note that comp_idx always holds the index of pairs that have complete states so that subsequent critical section can always select an appropriately protected cursor and resume by validating it.

When the second critical section begins, it first revalidates the target of the current backup protection (line 17), and upon success, resumes traversal from that point. This revalidation is done by calling the user-defined validate method of the Validatable trait. As such, the revalidation steps are handled automatically by the Traverse method once the user implements the Validatable trait. The step function is called repeatedly in a loop protected by a BRCU critical section (line 19). Following each step, Traverse determines whether to create a new backup protection (line 21), by protecting the current cursor using the next protector (line 22). After successfully creating a checkpoint, it increments comp_idx, permitting the use of the next protector (line 24). The critical section concludes when the step function returns a Finish value (line 25). With the protection in prot and returned value from Finish, the caller may subsequently perform other higher-level operations, *e.g.*, insertion and deletion, by dereferencing the acquired cursor.

Algorithm 8 Harris-Michael list search with Traverse

```
1:  struct ListCursor : Validatable + Copy
2:      prev, cur: Node*
3:      method validate() → bool
4:          return (*cur).next.load().tag() = 0
5:  struct ListCursorProtector : Protector
6:      prev, cur: Shield
7:      type Cursor = ListCursor
8:      function new()
9:          prev = Shield::new(); cur = Shield::new()
10:     method protect(cursor: &Cursor)
11:         prev.protect(cursor.prev); cur.protect(cursor.cur)
12: thread-local variable
13:     prot: ListCursorProtector      ▷ Two shields that protect prev and cur.
14: function TrySearch(key: Key) → Option<(ListCursor, bool)>
15:     (mask_prev_s, mask_cur_s) ← (Shield::new(), Shield::new())
16:     return Traverse(prot,
17:         (λ. prev ← ENTRY.load(); return { prev, (*prev).next.load() })
18:         (λ { prev, cur }.                          ▷ step: advances one step.
19:             if cur = ⊥ then return Finish({ prev, cur }, false)
20:             next ← (*cur).next.load()
21:             if next.tag() ≠ 0 then
22:                 next ← next.with_tag(0)
23:                 mask_prev_s.protect(prev); mask_cur_s.protect(cur)
24:                 succ ← Mask(λ. ▷ Abort-unsafe physical deletion
25:                     match (*prev).next.cas(cur, next)
26:                         case Ok then Retire(cur); return true
27:                         case Err then return false)
28:                 if succ then return Continue({ prev, next })
29:                 else return Fail
30:             if (*cur).key ≥ key then
31:                 return Finish({ prev, cur }, (*cur).key = key)
32:             return Continue({ cur, next })))
```

Example. Algorithm 8 re-implements the TrySearch function from the prior example of Harris-Michael list (Algorithm 3) in HP-BRCU. The interface of TrySearch remains almost the same, additionally returning the final ListCursor (*i.e.*, prev and cur) (line 14). The two closure arguments init and step correspond to InitCursor and Steps in Algorithm 3, respectively. In addition, it provides a validate method (line 3) as a trait Validatable implementation for the ListCursor. The validation simply checks whether cur, the pointer to be dereferenced in the step is marked. This method enables Traverse to automatically validate the protected cursor after rolling back by calling the validate method of the cursor (line 17 in Algorithm 7). With these arguments and trait implementation, the Traverse function manages the entire traversal and periodic checkpointing, ultimately returning the acquired and protected cursor. It fails only if it observes that the cursor is invalidated after rolling back. In case of failure, the client may need to retry the traversal by recalling TrySearch, although such failures are rare in practical workloads.

Additionally, the algorithm demonstrates the usage of the masked region. It utilizes the Mask function to attempt physical deletion within a critical section (§4.2). This process comprises two parts. First, it protects the local pointers with outliving shields, highlighted in yellow, to use in the masked region.[6] Then, it performs the physical deletion and retirement of the unlinked node within Mask, highlighted in purple. Although the physical deletion procedure is abort-rollback-unsafe, it can be executed safely because the abort-masking guarantees the sequence of unlinking and retirement not to be interrupted midway.

5 ANALYSIS

Safety. We formalize the requirements on the body of the BRCU phase discussed so far, including revalidation, as follows:

- **R1** Suppose p is a pointer created before the body. After following a link from p to a target node, say q, it validates that p is not retired yet before dereferencing q.
- **R2** Suppose p is a pointer created in the body. Then p is safe to dereference or protect (without validation) within the body.
- **R3** The body runs only abort-rollback-safe operations.

These requirements are essential in proving the safety of HP-BRCU.

Theorem 5.1 (BRCU's Correctness). *Suppose that a task is scheduled and defer-executed. We say these events are e_s and e_x, respectively. Then for every critical section, whose start and end events are e_f and e_t, we have either $e_s \overset{hb}{\longrightarrow} e_f$ or $e_t \overset{hb}{\longrightarrow} e_x$.*

Here, we assume C/C++ relaxed memory model and $\overset{hb}{\longrightarrow}$ is *happens-before* relation. The theorem means that a task scheduled concurrently with a critical section ($e_s \overset{hb}{\nrightarrow} e_f$) is always deferred until its end ($e_t \overset{hb}{\longrightarrow} e_x$). The statement is the same as the original RCU's correctness [33] but it additionally considers rollbacks.

Theorem 5.2 (HP-BRCU's Safety). *Suppose a data structure meets the original HP's requirements and HP-BRCU's requirements **R1**, **R2**, **R3**. Then it does not incur use-after-free.*

The proofs formalize the key idea presented in §3 and §4, and the correctness of the original HP [35, 36] (see the appendix [27] for proofs).

Applicability. Table 1 summarizes the applicability of reclamation schemes to various data structures. HP-BRCU is strictly more applicable than the original HP because the requirement for revalidation (§3.3) is more lenient than the original HP's. In particular, it supports optimistic traversal (§2.1): once the source is validated, then the traversal from it is protected in the critical section even if the intermediate nodes get deleted while traversal. Furthermore, HP-BRCU surpasses DEBRA+ and NBR(+) in applicability thanks to resuming after revalidation (§3.3). Specifically, resuming enables a traversal with a critical section to start from arbitrary nodes, making HP-BRCU applicable to data structures involving abort-rollback-unsafe physical deletion during traversals. In our evaluation, HP-BRCU applies to the same set of data structures as VBR [45], HP++ [24], and PEBR [25]. They all share the same strategy of aborting ongoing operations, either by signal or by flag, and restarting the aborted operations. Consequently, they apply to a data structure as long as its operations can be recovered from abortion.

[6] Although cur is not dereferenced during a physical deletion, the protection is necessary to prevent an ABA problem.

Data structure	HP, HE, IBR	DEBRA+	NBR	RCU	**HP-RCU**, **HP-BRCU**, VBR, HP++, PEBR
linked list [21]	✗	✗	▲	✓	▲
linked list [19]	✗	*	✓	✓	✓
linked list [34]	✓	*	✗	✓	✓
partially ext. BST [14]	✗	✗	**	✓	✓
ext. BST [16]	✓	*	✓	✓	✓
ext. BST [37]	✗	*	✓	✓	✓
ext. BST [15]	✓	*	✗	✓	✓
ext. BST [10]	✗	✗	▲	✓	▲
int. BST [23]	✗	*	✓	✓	✓
int. BST [42]	✗	✗	✗	✓	✓
partially ext. AVL [3]	✓	✗	✗	✓	✓
partially ext. AVL [14]	✗	✗	✗	✓	✓
ext. relaxed AVL [20]	✗	✓	✓	✓	✓
ext. AVL [5]	✗	✓	✓	✓	✓
patricia trie [44]	✗	*	▲	✓	▲
ext. chromatic tree [6]	✗	✓	✓	✓	✓
ext. (a,b)-tree [5]	✗	✓	✓	✓	✓
ext. interpol. tree [7]	✗	✗	✗	✓	▲
skip list [22]	▲	✗	✗	✓	▲

Table 1: Applicability of reclamation schemes. ✓: supported. ✗: not supported. ▲: supported but wait-freedom not preserved (§5). *: requiring significant design effort for recovery. **: requiring code restructuring to satisfy the scheme's assumption.

The ERA theorem [46] posits that every robust and applicable scheme is inherently not *easy to integrate*. The same holds for HP-BRCU as it necessitates consideration for an operation restart due to revalidation failures. However, we want to emphasize that its impact on programmability and performance is limited because an operation restart is already considered in the majority of concurrent data structures we are aware of, particularly to handle cases where an operation fails to perform CAS at the linearization point. Moreover, for programmability, we provide a high-level Traverse function (§4.3) that does not require users to explicitly consider inter-critical-section HP protection. In terms of performance, revalidation failures are much less likely than the original HP because most traversals are performed inside BRCU.

Analysis. Our partial solution HP-RCU is already robust against non-preempted long-running operations, provided that their RCU phases have a bounded number of instructions. However, it is not yet robust against stalled threads (§2.2). Even though a long traversal is divided into multiple RCU phases, a thread can be stalled within an RCU phase, blocking the reclamation process.

On the other hand, HP-BRCU is robust against stalled threads as well, as both of its components HP and BRCU are robust. Formally, HP-BRCU bounds the number of retired and yet unreclaimed garbage objects by $2GN + GN^2 + H$, where $G = \text{MaxLocalTasks} \times \text{ForceThreshold}$, N is the number of threads, and H is the number of shields. Recall from §4.1 that the number of the deferred tasks scheduled by a thread in an epoch is bounded by G. The first term is the garbages (G) put in the two old BRCU epochs (2) by the threads (N), the second term is the garbages (G) put in the same BRCU epoch retired by all threads (N) being reclaimed by each thread (N), and the third term is the explicitly HP-protected nodes (H). While the second term is quadratic on the number of threads, it is unlikely to reach the bound in practice as it happens only in a pessimistic case that *every* thread is reclaiming nodes at the same time (see §6).

Progress Guarantee. HP-BRCU preserves the lock-freedom of concurrent data structures despite its use of BRCU that possibly hinders the progress. However, HP-BRCU does not preserve wait-freedom, *e.g.*, it makes Heller et al. [21], Herlihy and Shavit [22]'s wait-free search just lock-free as traversal may be rollbacked indefinitely.

Theorem 5.3 (Lock-Freedom Preservation). *Suppose a lock-free data structure traverses and reclaims nodes using HP-BRCU methods, and its operations allocate a bounded number of nodes. Then the resulting data structure is also lock-free.*

Proof. Suppose otherwise and there exists an infinite execution history of a data structure equipped with HP-BRCU where no operations are finished successfully after some timestamp, say t_0.

We first prove the finiteness of forced global epoch advancements (line 32 in Algorithm 5) after t_0. Suppose otherwise. Since each forced advancement is performed by a thread with MaxTasks × ForceThreshold garbages (line 31) and puts at least these garbages to an old epoch, the forced advancements collectively put an infinite garbage to an old epoch. It is possible only if there are an infinite number of allocations after t_0. But it contradicts the assumption that each operation allocates a bounded number of nodes, and thus the whole program allocates a bounded number of nodes.

Let $t_1 > t_0$ be a timestamp after all rollbacks. By the lock-freedom of the underlying data structure, at least one operation should finish successfully after t_1, contradicting the assumption that no operations are finished after t_0. □

6 EXPERIMENT

We implemented our technique as a Rust library and evaluated it on a suite of synthetic benchmarks.[7] The benchmark suite includes the following reclamation schemes: **NR**: baseline that does not reclaim memory; **RCU**: epoch-based RCU [18, 19]; **HP**: hazard pointers [35, 36] with asymmetric fence optimization [12, 13]; **HP++**: extension of HP for optimistic traversal [24]; **PEBR**: pointer- and epoch-based reclamation [25]; **NBR**: optimized neutralization-based reclamation (NBR+) [48]; **NBR-Large**: NBR+ with a high reclamation threshold; **VBR**: version-based reclamation [45]; **HP-RCU**: our partial solution with RCU-expedited traversal (§3); and **HP-BRCU**: our full solution with RRCU-expedited traversal (§4).

The implementation of HP-RCU and HP-BRCU try advancing the global epoch per 128 Retires, and HP-BRCU forces advancing it after two successive unsuccessful advancements. For a fair comparison, other schemes except for NBR-Large also trigger reclamation per 128 retirements and per 8,192 retirements for NBR-Large[8]. We implemented NBR and VBR on our own because there is no public implementation in Rust, and used public implementation of HP/HP++ [24], PEBR [25], and epoch-based RCU [11].

[7] Available at the project website [27].
[8] The original author's implementation [47] triggers reclamation per 32,768 retirements. However, we found that our configuration performs better in our benchmark.

The benchmark suite consists of the following data structures: **HList**: Harris's list [19]; **HMList**: Harris-Michael list [34]; **HHSList**: Harris's list [19] with wait-free get() [22][9]; **HashMap**: chaining hash table using HMList (for HP) or HHSList (for others) for each bucket [34]; **SkipList**: lock-free skip list [22] with wait-free get() for schemes other than HP; and **NMTree**: Natarajan-Mittal tree [37]. We implemented data structures for applicable schemes only, *e.g.*, we did not implement HMList and SkipList for NBR.

The benchmark was compiled with Rust nightly-2023-12-19 with default optimization and link-time optimization. We used jemalloc [17] to reduce contention on the memory allocator. We conducted experiments on two dedicated machines: **AMD64T**: single-socket AMD EPYC 7543 (2.8GHz, 32 cores, 64 threads) with eight 32GiB DDR4 DRAMs (256GiB in total), and **INTEL96T**: dual-sockets Intel Xeon Gold 6248R (3.0GHz, 48 cores, 96 threads) with twelve 32GiB DDR4 DRAMs (384GiB in total). The machines run Ubuntu 22.04 and Linux 5.15 with the default configuration. All machines exhibit similar results, so we discuss only those for AMD64T. For the full experimental results, see the appendix [27].

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, $\cdots$, 128 (twice the number of hardware threads); **(2)** four types of workloads: read-only (100% reads), read-intensive (90% reads and 10% writes), read-write (50% reads and 50% writes) and write-only (50% inserts and 50% deletes); and **(3)** fixed time: 10 seconds. For map data structures, each thread repeatedly calls get(), insert(), and remove() methods randomly. The key ranges for lists are 1K and 10K, and the key ranges for others are 100K and 100M. The data structures are pre-filled to 50%. Figure 5 and 7 show representative results from this map benchmark.

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: $2^{18}, 2^{19}, \cdots, 2^{29}$. Specifically, 32 threads perform get() for random elements, and the other 32 threads push and pop to the head of the list. Figure 6 shows representative results from this long-running operation benchmark.

Short-running operations. We observe that HP-BRCU has a performance advantage over other robust schemes thanks to **(1)** no per-node overhead in BRCU-expedited traversal, **(2)** optimistic traversal, and **(3)** amortized cost of expensive signals.

Figure 5 presents the throughput of read-only workloads. The thread oversubscription ranges are highlighted in red. For HHSList (Figure 5a), HP-BRCU performs comparably with RCU and NBR, outperforming HP++, PEBR, and VBR with per-node overhead of protection and/or validation. For HashMap (Figure 5b) with a generally short traversal length of 1.7. HP and HP++ (without per-operation cost) outperform RCU (with epoch management), NBR (with signal management) and HP-BRCU (with both).

Figure 7 presents the throughput of workloads involving write operations. In general, HP-BRCU performs comparably to (outperformed by 5% in the worst case) or outperforms RCU. For HList (Figure 7a), the gap among all schemes reduces compared to the read-only workloads due to the heavy overhead of write and reclamation. However, NBR shows performance degradation after reaching 56 threads due to frequent neutralization signals. For HashMap and NMTree (Figure 7b and 7c), VBR takes the lead over EBR and HP-BRCU. VBR benefits significantly from its customized memory allocator, which does not return memory blocks to the operating system. Conversely, the performance decline of NBR worsens, and NBR-Large also demonstrates lower performance than others. For SkipList (Figure 7d), HP, HP++, and PEBR's performance is degraded due to the protection of multiple local pointers. In contrast, HP-BRCU minimizes protection overheads by protecting pointers only once at the end of a critical section. VBR also underperforms in SkipList due to the overhead of double-word atomic pointers.

Across Figure 7a–7d, HP-BRCU effectively bounds the number of unreclaimed nodes, showing a memory footprint comparable to HP and HP++. The numbers are well below the theoretical bound (§5) because its quadratic term happens only for pessimistic cases.

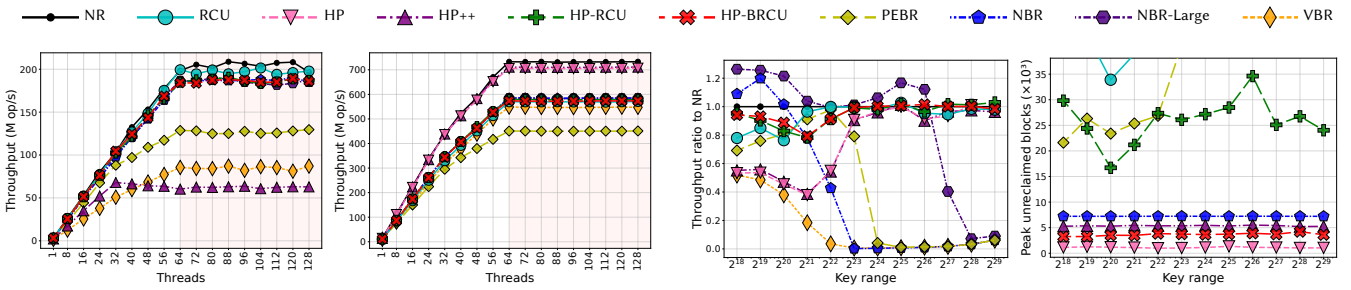
Long-running operations. Figure 6 compares the NR-normalized throughput of long-running read operations with varying key ranges. For large key ranges, NBR, VBR, and PEBR severely suffer from frequent operation restarts as their rollback condition is coarse-grained. In contrast, HP-RCU maintains both consistent throughput and moderate memory footprint by splitting the long operation into short RCU phases. Furthermore, HP-BRCU achieves consistent and bounded memory usage by rolling back stalled threads while preserving its high throughput.

7 RELATED WORK

Table 2 summarizes the robustness and efficiency of various reclamation schemes. Only HP-BRCU is robust and efficient for long-running operations at the same time. For instance, RCU [31, 32] and Stamp-it [40] are efficient but not robust because the length of a critical section is unbounded. IBR [50] is not robust against long-running operations, as a thread may occupy an indefinite range of epochs. PEBR [25], VBR [45], DEBRA+ [8] and NBR(+) [48, 49] bound their memory footprints, but they starve in long-running operations due to their *coarse-grained* rollback conditions (§2.3). On the other hand, HP [35, 36], HP++ [24], and HE [43] are robust against long-running operations, thanks to their fine-grained rollback conditions (*i.e.*, validation based on the state of individual nodes). However, they suffer from per-node overheads and HP and HE are not widely applicable (§5). HP-BRCU strikes a balance in this tradeoff: **(1)** its rollback condition is fine-grained because an operation is validated by checkpointed nodes, and **(2)** it avoids per-node overheads through BRCU-expedited traversals.

There are several variants of HP that partially overcome HP's drawbacks while retaining robustness. IBR [50], HE [43], and Hyaline-1S [38] improve efficiency by validating protections with epoch, which requires fewer fences. However, they do not support optimistic traversal in general. PEBR [25] and HP++ [24] support optimistic traversal. To do so, PEBR employs an *ejection* synchronization protocol between traversing and reclaiming threads, which may degrade the performance of long-running operations, and HP++ lets the retiring threads issue additional protections on behalf of traversing threads. All HP variants inherit the inefficiency due to per-node protection overhead.

[9]For HP-BRCU, VBR, HP++, PEBR, and NBR, get() is only lock-free due to rollback/recovery. The same applies to SkipList.

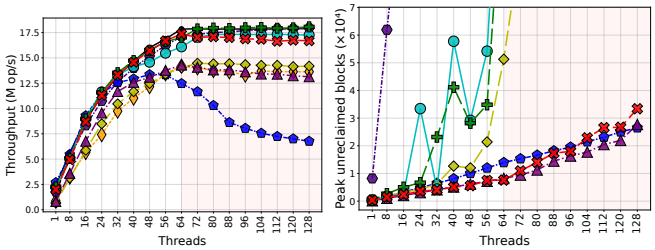


(a) HHSList. 1K. (b) HashMap. 100K.

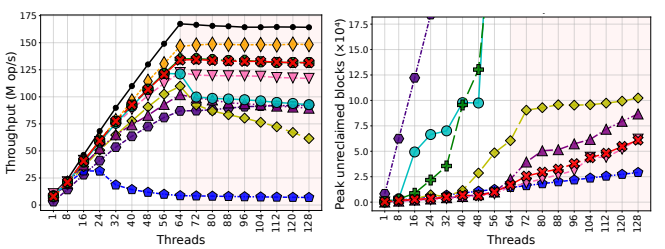
Figure 5: Throughput of read-only workloads for varying number of threads.

(a) Throughput (b) Peak # of unrecl. blocks

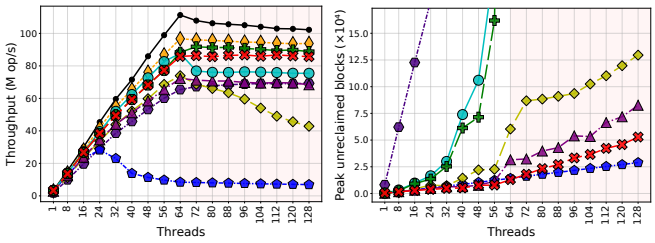
Figure 6: Throughput and peak number of unreclaimed blocks of long-running read operations.



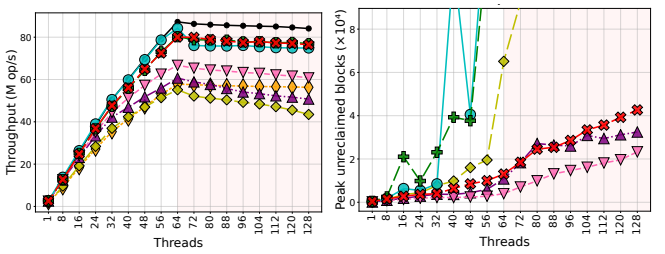
(a) HList. 1K. Write-only workloads.



(b) HashMap. 100K. Write-only workloads.



(c) NMTree. 100K. Read-write workloads.



(d) SkipList. 100K. Read-write workloads.

Figure 7: Throughput (million operations per second) and peak number of unreclaimed blocks for a varying number of threads.

criterion		RCU	HP, HP++	HE	PEBR	VBR	IBR	DEBRA+, NBR(+)	**HP-RCU**	**HP-BRCU**
robustness	stalled threads	✗	✓	✓	✓	✓	✓	✓	✗	✓
	long-running ops.	✗	✓	✓	✓	✓	✗	✓	✓	✓
efficiency	low per-node overhead	✓	✗	▲	✗	▲	▲	✓	✓	✓
	starvation-freedom in long-running ops.	✓	▲	▲	✗	✗	▲	✗	▲	▲

Table 2: Comparing the robustness and efficiency of various reclamation schemes. *low per-node cost*: ✓: none. ▲: usually validation only. ✗: protection and validation; *starvation-freedom in long-running ops.*: ✓: no starvation. ▲: less starvation (fine-grained failure). ✗: more starvation (coarse-grained failure).

ACKNOWLEDGMENTS

We thank the anonymous reviewers of PPoPP'24 and SPAA'24 for their in-depth feedback. We also thank Janggun Lee for actively providing insightful feedback throughout the project. This work was supported by: **(1)** Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) partly under the project (No. RS-2024-00396013, DRAM PIM Hardware Architecture for LLM Inference Processing with Efficient Memory Management and Parallelization Techniques, 70%), partly under the Information Technology Research Center (ITRC) support program (No. IITP-2024-2020-0-01795, Development of Dependable and Highly Usable Big Data Platform, and Analysis and Prediction Services Technology in Edge Clouds, 10%), and partly under the Graduate School of Artificial Intelligence Semiconductor (No. IITP-2024-RS-2023-00256472, 10%); and **(2)** Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. RS-2024-00397386, 10%).

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] 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

[4] 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

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

[6] 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

[7] 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

[8] 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

[9] 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.

[10] 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

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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

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

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

[19] 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.

[20] 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

[21] 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.

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

[23] 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

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

[25] 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

[26] Michael Kerrisk. 2010. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook* (1st ed.). No Starch Press, USA.

[27] Jeonghyeon Kim, Jaehwang Jung, and Jeehoon Kang. 2024. Expediting Hazard Pointers with Bounded RCU Critical Sections. https://cp.kaist.ac.kr/gc

[28] 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

[29] Linux Programmer's Manual. 2023. pthread_kill(3) — Linux manual page. https://man7.org/linux/man-pages/man3/pthread_kill.3.html

[30] Linux Programmer's Manual. 2023. signal(7) — Linux manual page. https://man7.org/linux/man-pages/man7/signal.7.html

[31] 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.

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

[33] Paul E. McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kamiński, and Jens Maurer. 2023. P2545R4: Read-Copy Update (RCU). https://wg21.link/p2545r4.

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

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

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

[37] 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

[38] 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

[39] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 391–407.

[40] Manuel Pöter and Jesper Larsson Träff. 2018. Brief Announcement: Stamp-it, a more Thread-efficient, Concurrent Memory Reclamation Scheme in the C++ Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA '18)*. Association for Computing Machinery, New York, NY, USA, 355–358. https://doi.org/10.1145/3210377.3210661

[41] Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: A Simpler and Faster Operational Concurrency Model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI*

2019). Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

[42] 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

[43] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *SPAA 2017*.

[44] 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

[45] 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

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

[47] Ajay Singh. 2023. Simple, Fast and Widely Applicable Concurrent Memory Reclamation via Neutralization. *IEEE Transactions on Parallel and Distributes Systems* (Nov. 2023). https://doi.org/10.5281/zenodo.10203082

[48] 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

[49] Ajay Singh, Trevor Alexander Brown, and Ali José Mashtizadeh. 2024. Simple, Fast and Widely Applicable Concurrent Memory Reclamation via Neutralization. *IEEE Transactions on Parallel and Distributed Systems* 35, 2 (2024), 203–220. https://doi.org/10.1109/TPDS.2023.3335671

[50] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based memory reclamation. In *PPoPP 2018*.

[51] Wikipedia contributors. 2023. ABA problem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=ABA_problem&oldid=1166964304. [Online; accessed 4-August-2023].

[52] Wikipedia contributors. 2023. Multiple buffering — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Multiple_buffering&oldid=1164687618. [Online; accessed 4-August-2023].

A SAFETY PROOF

We review the proof of the hazard pointer's safety (Appendix A.1), and prove BRCU's correctness (Appendix A.2) and HP-BRCU's safety (Appendix A.3) stated in §5. We first present our assumption on the underlying concurrency model.

Assumption 1 (Concurrency Model). *We assume the underlying concurrency model satisfies the following properties:*

- *the model has the concept of* happens-before *relation (denoted by $\overset{hb}{\longrightarrow}$) that is a partial order on events;*
- *for a matching pair of release write, say w, and acquire read, say r, we have $w \overset{hb}{\longrightarrow} r$;*
- *For a pair of SC fence invocations, say a and b, we have either $a \overset{hb}{\longrightarrow} b$ or $b \overset{hb}{\longrightarrow} a$; and*
- *If a thread, say a, sends a signal to another thread, say b, then we have:*

$$[b\text{'s suspension due to signal delivery}] \overset{hb}{\longrightarrow} [a\text{'s return from signaling system call}] .$$

Our proof in this section works as far as the underlying concurrency model satisfies Assumption 1, which is indeed satisfied by a wide class of models including the C/C++ model [28], Armv8/RISC-V model [41], and x86-TSO model [39] when combined with the POSIX model of signals.

$\overset{hb?}{\longrightarrow}$ stands for the reflexive closure of $\overset{hb}{\longrightarrow}$.

A.1 Hazard Pointers

Let p's unprotection be the event that removes p from shield, and p's shield scan be the event that the shields are scanned for p's reclamation at line 14 in Algorithm 1.

Lemma A.1. *Suppose a pointer, say p, is retired and then reclaimed in hazard pointers. If [p's protection $\overset{hb}{\longrightarrow}$ p's dereference $\overset{hb}{\longrightarrow}$ p's unprotection] and [p's protection $\overset{hb}{\longrightarrow}$ p's shield scan], then [p's dereference $\overset{hb}{\longrightarrow}$ p's reclamation]* (i.e., *p does not incur use-after-free).*

Proof. We label the events of p's protection- and reclamation-related events as follows:

- P_1: p's protection
- P_2: p's dereference
- P_3: p's unprotection
- R_1: p's shield scan (line 14)
- R_2: p's reclamation (line 14)

Here, we have $P_1 \overset{hb}{\longrightarrow} P_2 \overset{hb}{\longrightarrow} P_3$ and $P_1 \overset{hb}{\longrightarrow} R_1 \overset{hb}{\longrightarrow} R_2$. For R_2 to pass its check, we have $P_2 \overset{hb}{\longrightarrow} P_3 \overset{hb}{\longrightarrow} R_1 \overset{hb}{\longrightarrow} R_2$, from which the conclusion is immediate. □

Theorem A.2 (Hazard Pointer's Safety of Validation). *Suppose a pointer, say p, is retired and then reclaimed in hazard pointers. If [p's protection $\overset{hb}{\longrightarrow}$ SC fence $\overset{hb}{\longrightarrow}$ p's validation $\overset{hb}{\longrightarrow}$ p's dereference $\overset{hb}{\longrightarrow}$ p's unprotection] and [p's retirement $\overset{hb}{\nrightarrow}$ p's validation]* (i.e., *p's reachability is validated), then [p's dereference $\overset{hb}{\longrightarrow}$ p's reclamation].*

Proof. We label the events of p's protection- and reclamation-related events as follows:

- P_1: p's protection (*e.g.*, line 7 in Algorithm 1)
- P_2: SC fence (*e.g.*, line 7)
- P_3: p's validation (*e.g.*, line 9)
- R_1: p's retirement
- R_2: SC fence in Reclaim (line 13)
- R_3: p's shield scan (line 14)

Here, we have $P_1 \overset{hb}{\longrightarrow} P_2 \overset{hb}{\longrightarrow} P_3$ and $R_1 \overset{hb}{\longrightarrow} R_2 \overset{hb}{\longrightarrow} R_3$.

By the strict ordering of SC fences, we have either $P_2 \overset{hb}{\longrightarrow} R_2$ or $R_2 \overset{hb}{\longrightarrow} P_2$. For the latter case, we have $R_1 \overset{hb}{\longrightarrow} R_2 \overset{hb}{\longrightarrow} P_2 \overset{hb}{\longrightarrow} P_3$, contradicting the assumption. For the former case, we have $P_1 \overset{hb}{\longrightarrow} P_2 \overset{hb}{\longrightarrow} R_2 \overset{hb}{\longrightarrow} R_3$. From Lemma A.1, the conclusion is immediate. □

Theorem A.3 (Hazard Pointer's Safety of Early Protection). *Suppose a pointer, say p, is retired and then reclaimed in hazard pointers. If [p's protection $\overset{hb}{\longrightarrow}$ p's dereference $\overset{hb}{\longrightarrow}$ p's unprotection] and [p's protection $\overset{hb}{\longrightarrow}$ p's retirement], then [p's dereference $\overset{hb}{\longrightarrow}$ p's reclamation].*

Proof. By Lemma A.1. □

Theorem A.2 and Theorem A.3 collectively mean hazard pointer's safety: If every pointer is properly protected according to these theorems, then data structure does not incur use-after-free error.

A.2 Bounded RCU

Theorem A.4 (BRCU's Correctness, Restatement of Theorem 5.1). *Suppose that a task is scheduled and defer-executed. We say these events are e_s and e_x, respectively. Then for every critical section, whose start and end events are e_f and e_t, we have either $e_s \xrightarrow{hb} e_f$ or $e_t \xrightarrow{hb} e_x$.*

Proof. Let T be the thread that executes the critical section. We first label the events of the beginning of T's critical section as follows:

- F_1: Load, say $\mathcal{E}_f$, from the global epoch (line 16 in Algorithm 5)
- F_2: Store $\mathcal{E}_f$ to the local epoch (line 16)
- $F_3 (= e_f)$: Issue SC fence (line 16)

We then label the events of the scheduling event e_s of a task as follows:

- $S_1 (= e_s)$: Issue the first SC fence (line 25)
- S_2: Load, say $\mathcal{E}_s$, from the global epoch (line 25)
- S_3: Issue the second SC fence (line 25)

By the strict ordering of SC fences, we have either $F_3 \xrightarrow{hb} S_1$ or $S_1 \xrightarrow{hb} F_3$. For the latter case, we have $e_s = S_1 \xrightarrow{hb} F_3 = e_f$, from which the conclusion is immediate. For the former case, we have $F_1 \xrightarrow{hb} F_3 \xrightarrow{hb} S_1 \xrightarrow{hb} S_2$. From the coherence and the monotonicity of the global epoch (only updated by CAS), we have $\mathcal{E}_f \le \mathcal{E}_s$.

We then label the events of the Defer invocation that executes the scheduled task as follows:

- X_1: Issue the first SC fence (line 25)
- X_2: Load $\mathcal{E}_s + 1$ from the global epoch (line 25)
- X_3: Issue the second SC fence (line 25)
- X_4: Load, say $\mathcal{E}_T$, from T's local epoch (line 29)
- X_5: Send signal to T is the conditions are met (line 32)
- X_6: Perform CAS on the global epoch from $\mathcal{E}_s + 1$ to $\mathcal{E}_s + 2$ (line 33)
- $X_7 (= e_x)$: Execute those tasks deferred at $\mathcal{E}_s$ (line 34)

By the strict ordering of SC fences, we have either $S_1 \xrightarrow{hb} X_3$ or $X_3 \xrightarrow{hb} S_1$. For the latter case, we have $X_2 \xrightarrow{hb} X_3 \xrightarrow{hb} S_1 \xrightarrow{hb} S_2$. From the coherence and the monotonicity of the global epoch, we have $\mathcal{E}_s + 1 \le \mathcal{E}_s$, which is a contradiction. For the former case, we have $F_2 \xrightarrow{hb} F_3 \xrightarrow{hb} S_1 \xrightarrow{hb} X_3 \xrightarrow{hb} X_4$. From the coherence of the local epoch, we have either $\mathcal{E}_f \le \mathcal{E}_T$ or $\mathcal{E}_T$ is unpinned. We prove $e_t \xrightarrow{hb} X_7 = e_x$ by a case analysis on the value of $\mathcal{E}_T$, from which the conclusion is immediate.

- If $\mathcal{E}_T$ is unpinned, we have $e_t \xrightarrow{hb} X_4$ from release-acquire synchronization. Then we have $e_t \xrightarrow{hb} X_4 \xrightarrow{hb} X_5 = e_x$.
- If $\mathcal{E}_T \le \mathcal{E}_s$, then T must have been signaled at X_5. From Assumption 1, we have $e_t \xrightarrow{hb} X_7 = e_x$.
- If $\mathcal{E}_T > \mathcal{E}_s$, we have $\mathcal{E}_f \le \mathcal{E}_s < \mathcal{E}_T$. Then we have $e_t \xrightarrow{hb}$ [storing $\mathcal{E}_T$ to T's local epoch] $\xrightarrow{hb} X_4 \xrightarrow{hb} X_7 = e_x$.

□

A.3 HP-BRCU

Lemma A.5 (Reachability inside CriticalSection). *Suppose an HP-BRCU's CriticalSection invocation reaches a node, say q, inside its body, and the critical section's start and end events are e_f and e_t, respectively. Then q is not HP-BRCU-retired before e_f, or [q's HP-BRCU-retirement] $\xrightarrow{hb}\!\!\!\!\!\rightarrow e_f$ in short.*

Proof. We prove by induction on the length of the traversal to q inside the critical section.

For the base case, assume that q is the first node to traverse inside the critical section. If q was traversed from data structure entry points, then the conclusion is immediate. Otherwise, q was traversed from a local pointer, say p, protected before e_f. We have [p's HP-BRCU-retirement] $\xrightarrow{hb}\!\!\!\!\!\rightarrow e_f$ by the assumption on p. If [q's HP-BRCU-retirement] $\xrightarrow{hb} e_f$, from the link from p to q retrieved after e_f, we have [p's HP-BRCU-retirement] $\xrightarrow{hb?}$ [q's HP-BRCU-retirement] $\xrightarrow{hb} e_f$, contradicting the assumption.

For the inductive case, assume that the traversal to q first traverses to p and then follows a link from p to q. By induction hypothesis, we may further assume that [p's HP-BRCU-retirement] $\xrightarrow{hb}\!\!\!\!\!\rightarrow e_f$. If [$q$'s HP-BRCU-retirement] $\xrightarrow{hb} e_f$, from the link from p to q retrieved after e_f, we have [p's HP-BRCU-retirement] $\xrightarrow{hb?}$ [q's HP-BRCU-retirement] $\xrightarrow{hb} e_f$, contradicting the assumption. □

Corollary A.6 (No Retired Nodes inside CriticalSection). *Suppose an HP-BRCU's CriticalSection invocation reaches a node, say q, inside its body, and the critical section's start and end events are e_f and e_t, respectively. If q is HP-retired, then we have $e_t \xrightarrow{hb}$ [q's HP-retirement].*

Proof. From Theorem A.4, we have either ([q's HP-BRCU-retirement] $\overset{hb}{\rightarrow} e_f$) or ($e_t \overset{hb}{\rightarrow}$ [q's HP-retirement]). For the former case, from Lemma A.5, we have [q's HP-BRCU-retirement] $\overset{hb}{\twoheadrightarrow} e_f$, which is a contradiction. For the latter case, the conclusion is immediate. □

Theorem A.7 (HP-BRCU Safety of Dereference in Critical Section). *Suppose HP-BRCU's requirements **R1**, **R2**, **R3** are met; and a pointer, say p, is retired and then reclaimed in HP-BRCU. If a p' dereference in a BRCU critical section, then [p's dereference $\overset{hb}{\rightarrow}$ p's reclamation].*

Proof. Let e_f and e_t be the start and end events of the critical section, respectively. From the assumption that p is reached in the critical section and Corollary A.6, we have:

$$p\text{'s dereference} \overset{hb}{\rightarrow} e_t \overset{hb}{\rightarrow} p\text{'s HP-retirement} \overset{hb}{\rightarrow} p\text{'s reclamation}\ .$$

□

Theorem A.8 (HP-BRCU Safety of Protection in Critical Section). *Suppose HP-BRCU's requirements **R1**, **R2**, **R3** are met; and a pointer, say p, is retired and then reclaimed in HP-BRCU. If [p's protection $\overset{hb}{\rightarrow}$ p's dereference $\overset{hb}{\rightarrow}$ p's unprotection] and p's protection is in a BRCU critical section, then [p's dereference $\overset{hb}{\rightarrow}$ p's reclamation].*

Proof. Let e_f and e_t be the start and end events of the critical section, respectively. From the assumption that p is later dereferenced, p is successfully protected during critical section, so we have [p's protection] $\overset{hb}{\rightarrow} e_t$. From the assumption that p is reached in the critical section and Corollary A.6, we have $e_t \overset{hb}{\rightarrow}$ [p's HP-retirement]. From above and Theorem A.3, the conclusion is immediate. □

Theorem A.2, Theorem A.3, Theorem A.7, and Theorem A.8 collectively mean HP-BRCU's safety (Theorem 5.2): If every pointer is properly protected according to these theorems, then data structure does not incur use-after-free error.

B AMD64T FULL EXPERIMENTAL RESULTS

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

B.1.1 Write-only Workloads.

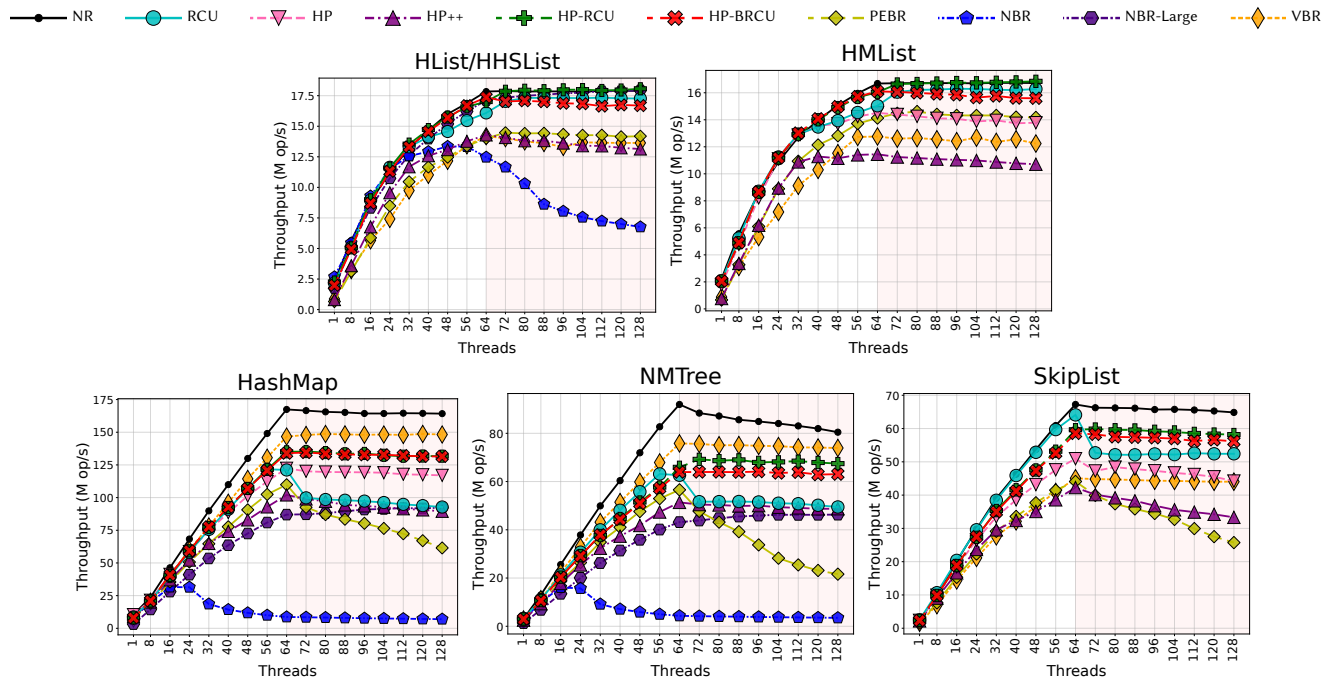


Figure 8: Throughput (million operations per second) of write-only workloads for a varying number of threads.

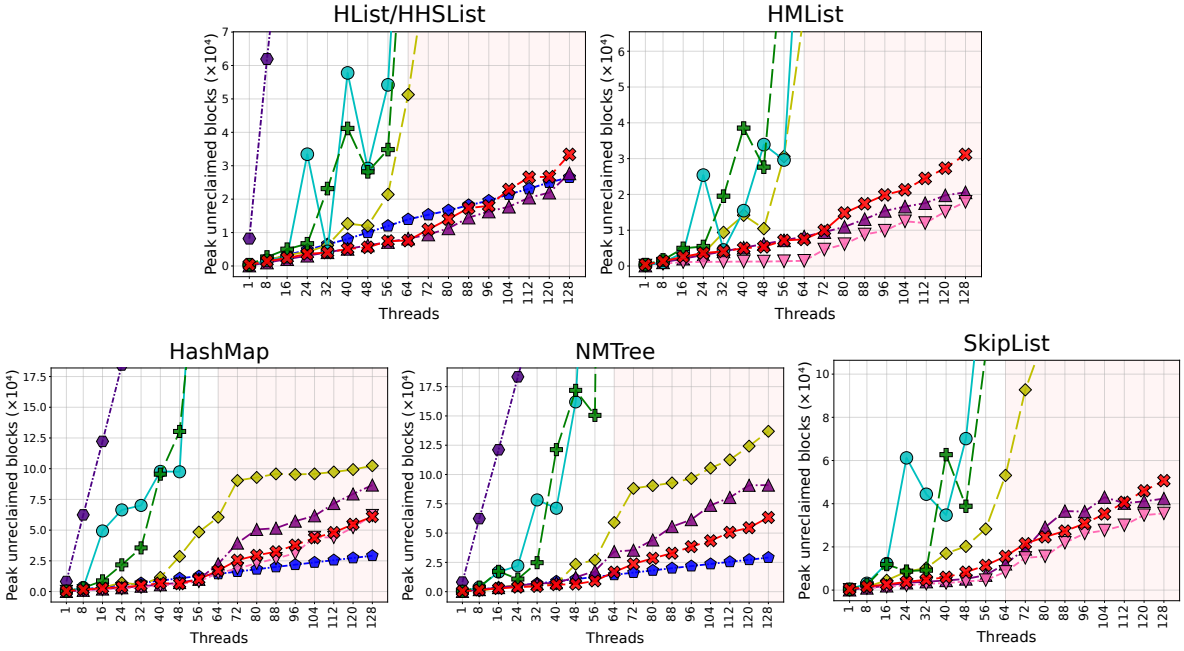


Figure 9: Peak number of unreclaimed blocks of write-only workloads for a varying number of threads.

B.1.2 Read-write Workloads.

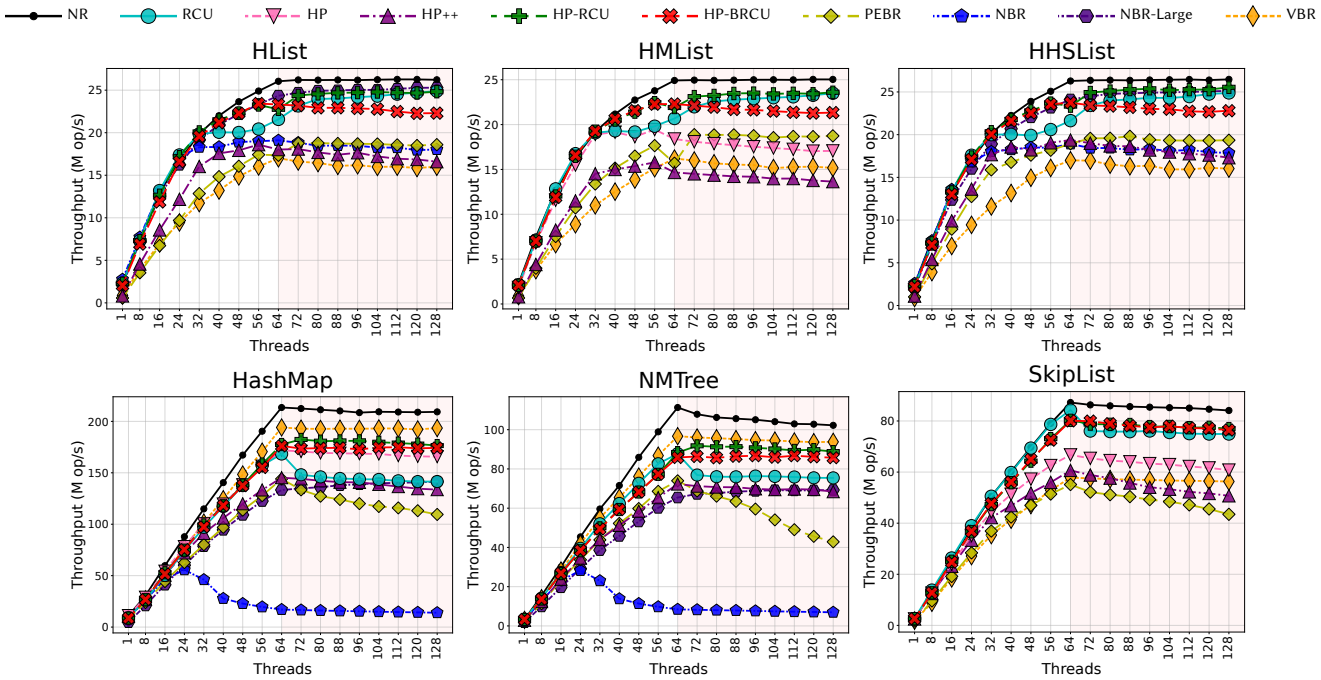


Figure 10: Throughput (million operations per second) of read-write workloads for a varying number of threads.

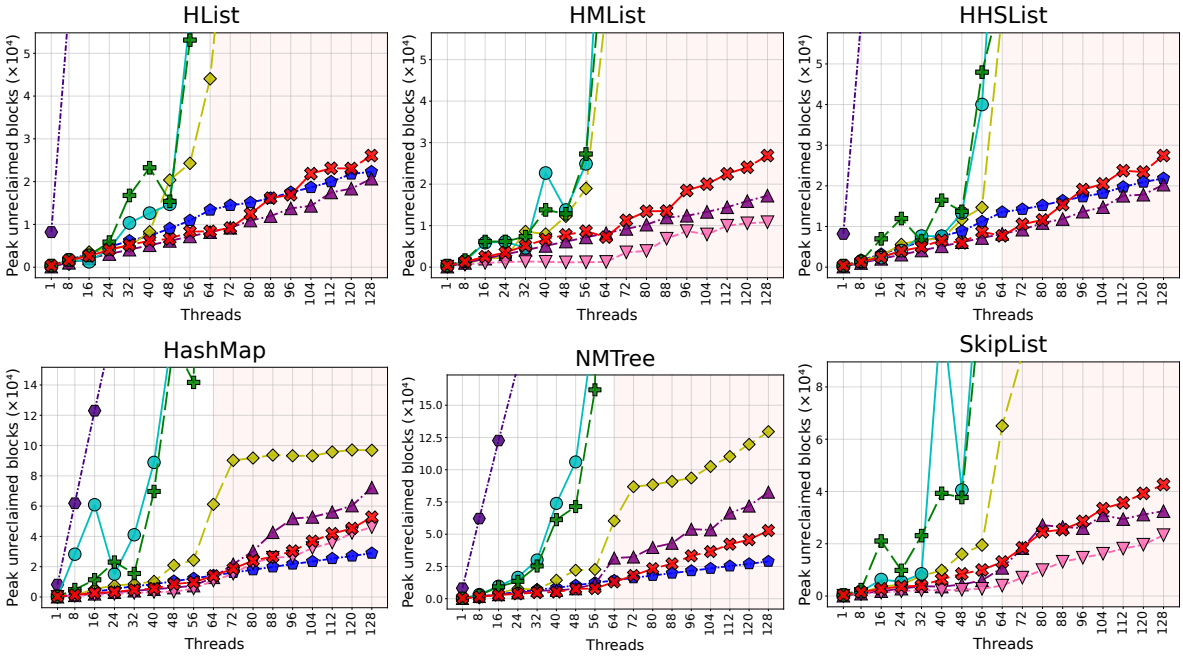


Figure 11: Peak number of unreclaimed blocks of read-write workloads for a varying number of threads.

B.1.3 *Read-intensive Workloads.*

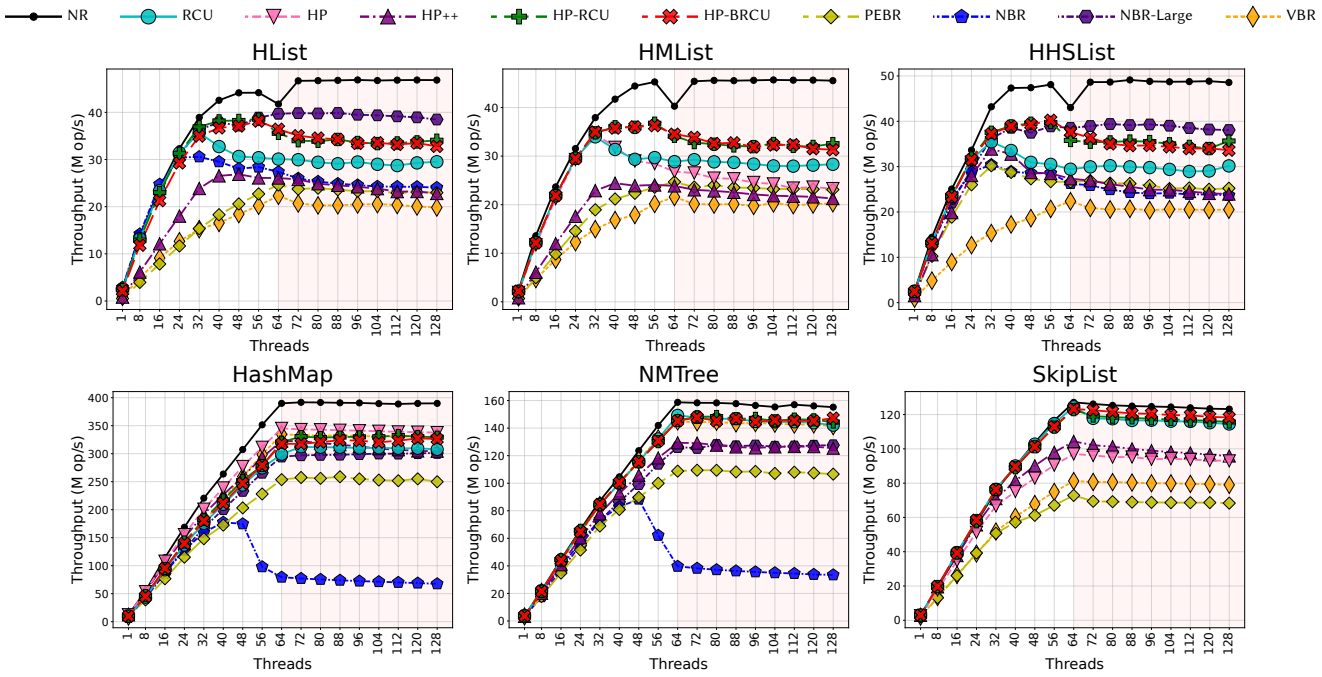


Figure 12: Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

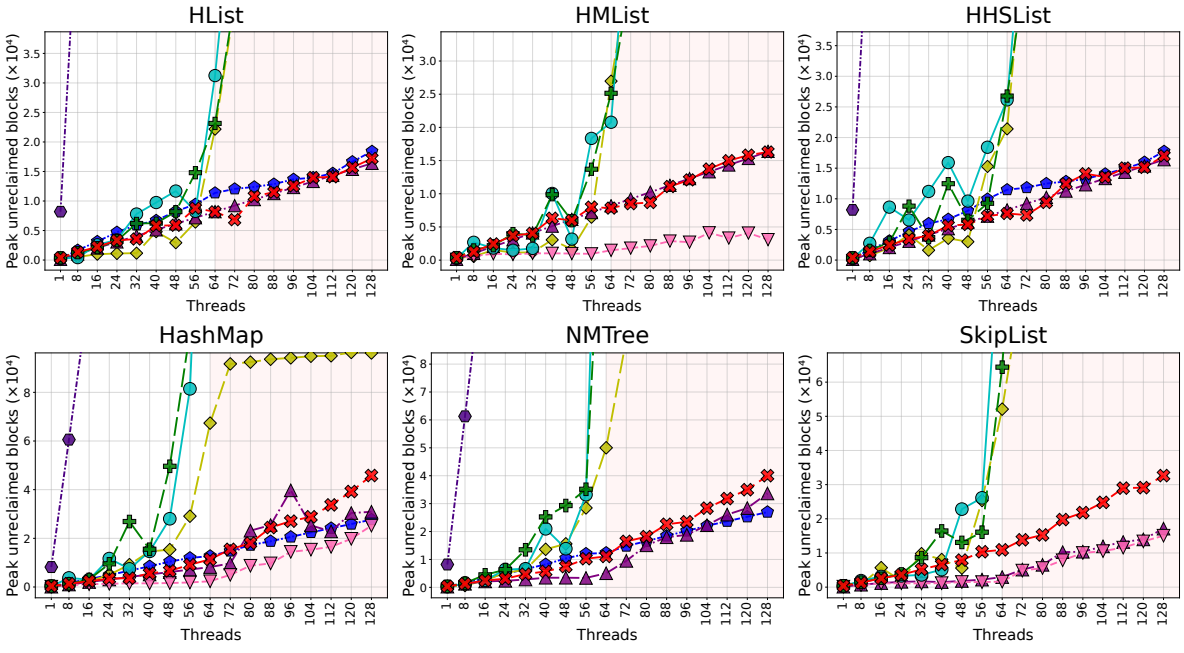


Figure 13: Peak number of unreclaimed blocks of read-intensive workloads for a varying number of threads.

B.1.4 Read-only Workloads.

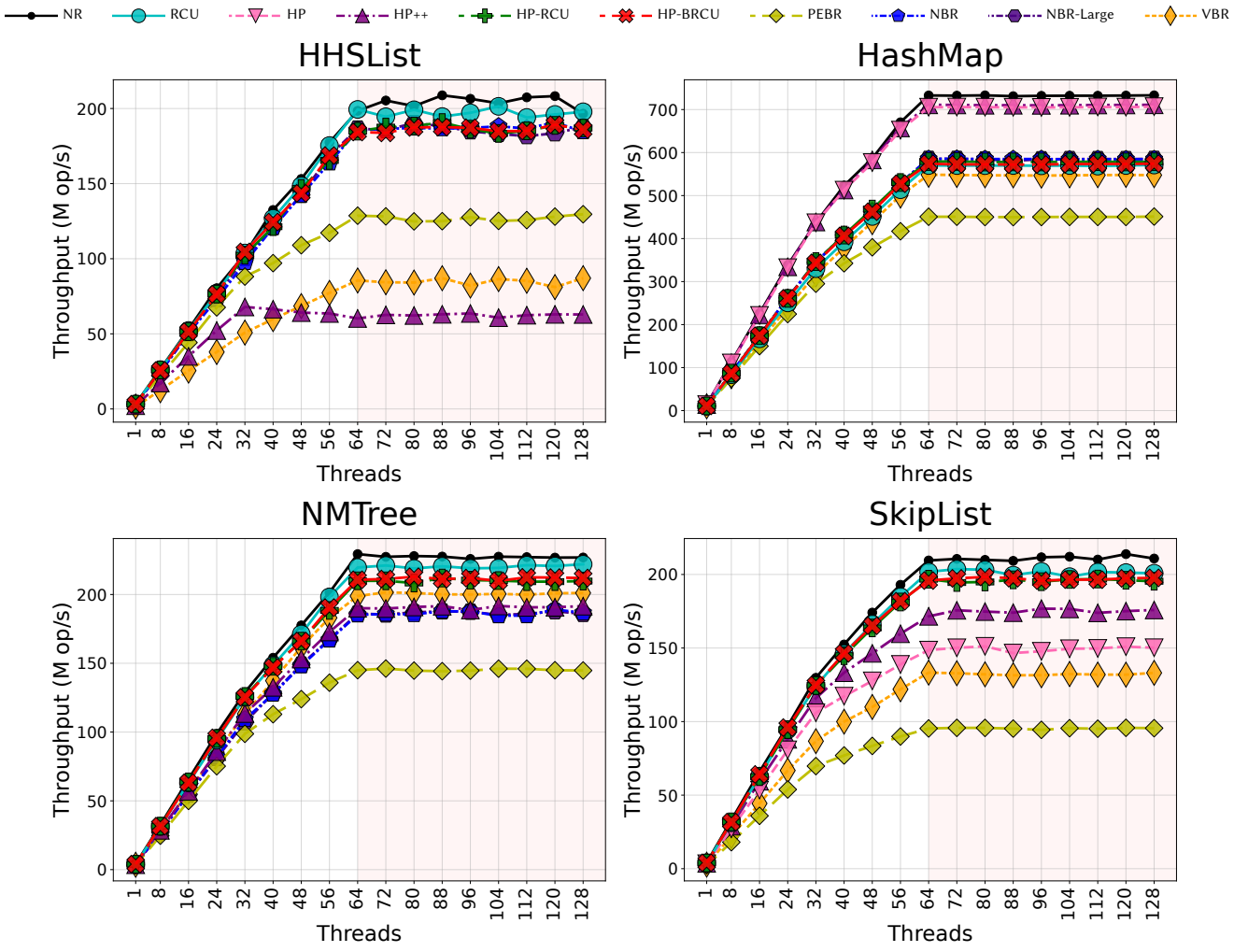


Figure 14: Throughput (million operations per second) of read-only workloads for a varying number of threads.

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

B.2.1 Write-only Workloads.

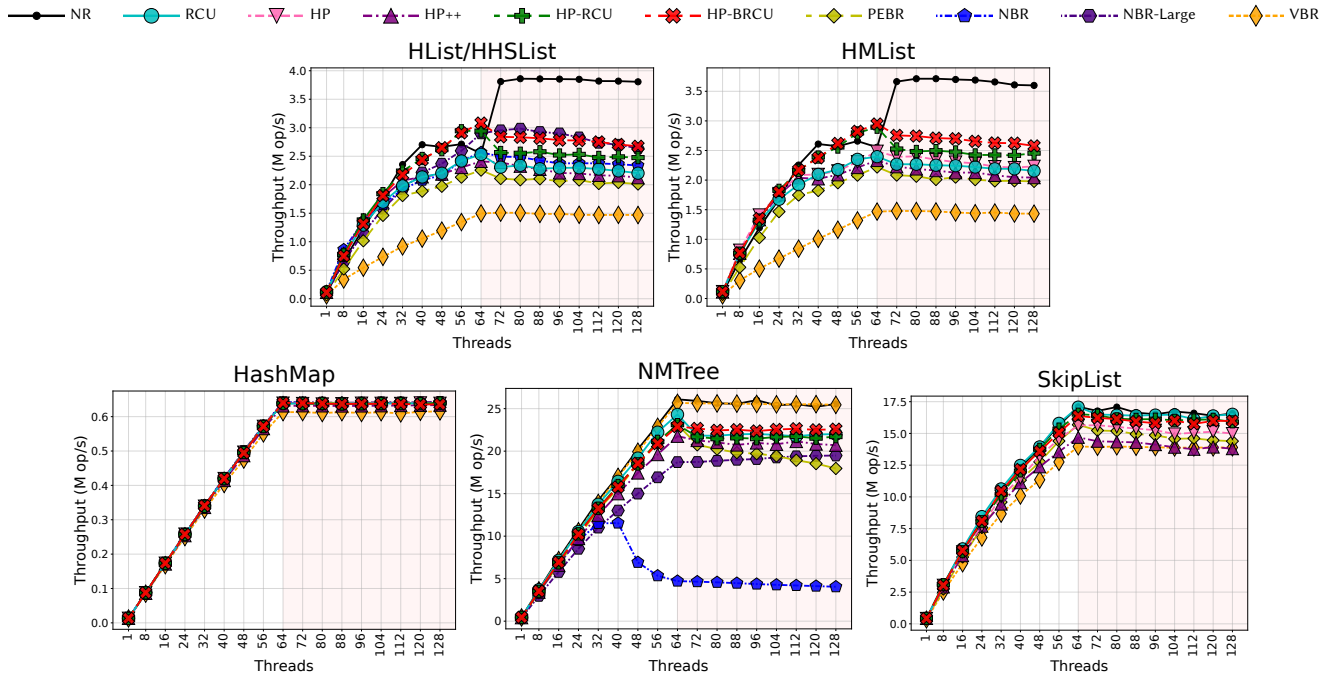


Figure 15: Throughput (million operations per second) of write-only workloads for a varying number of threads.

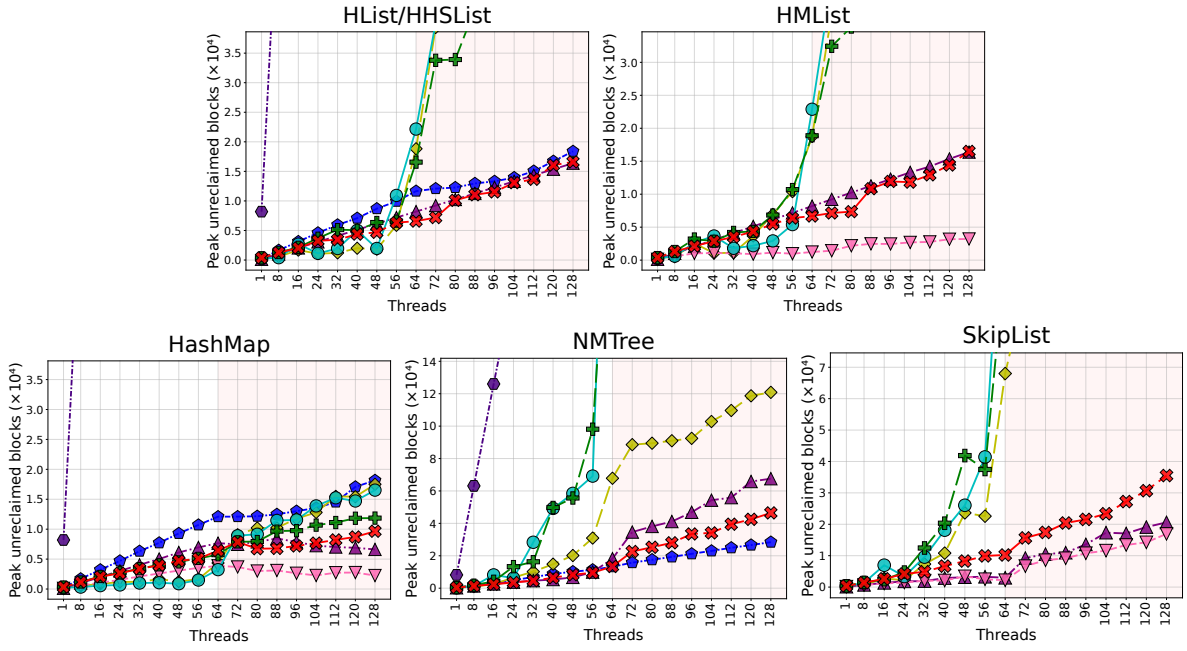


Figure 16: Peak number of unreclaimed blocks of write-only workloads for a varying number of threads.

B.2.2 Read-write Workloads.

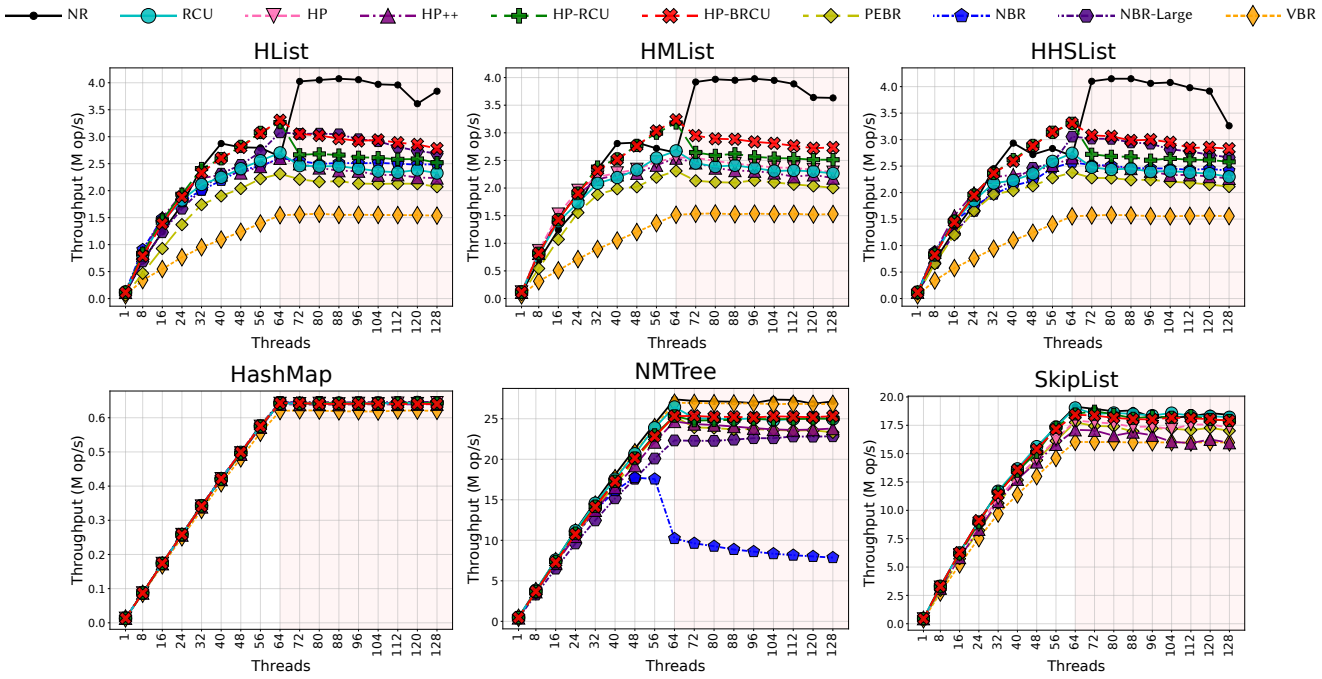


Figure 17: Throughput (million operations per second) of read-write workloads for a varying number of threads.

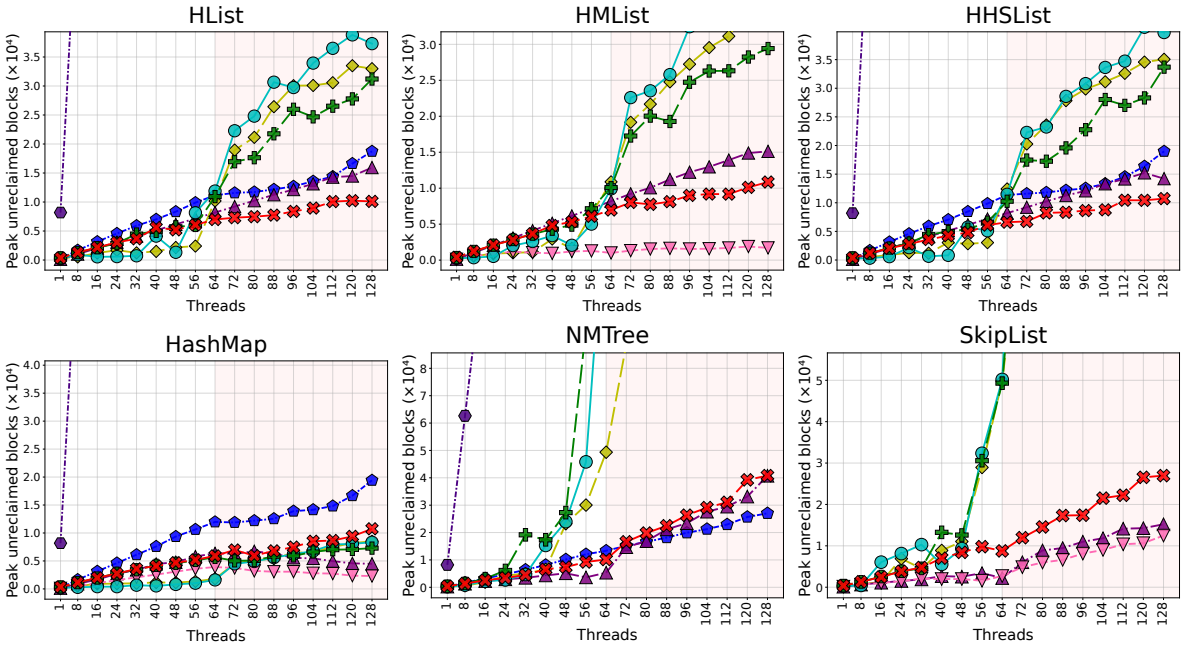


Figure 18: Peak number of unreclaimed blocks of read-write workloads for a varying number of threads.

B.2.3 Read-intensive Workloads.

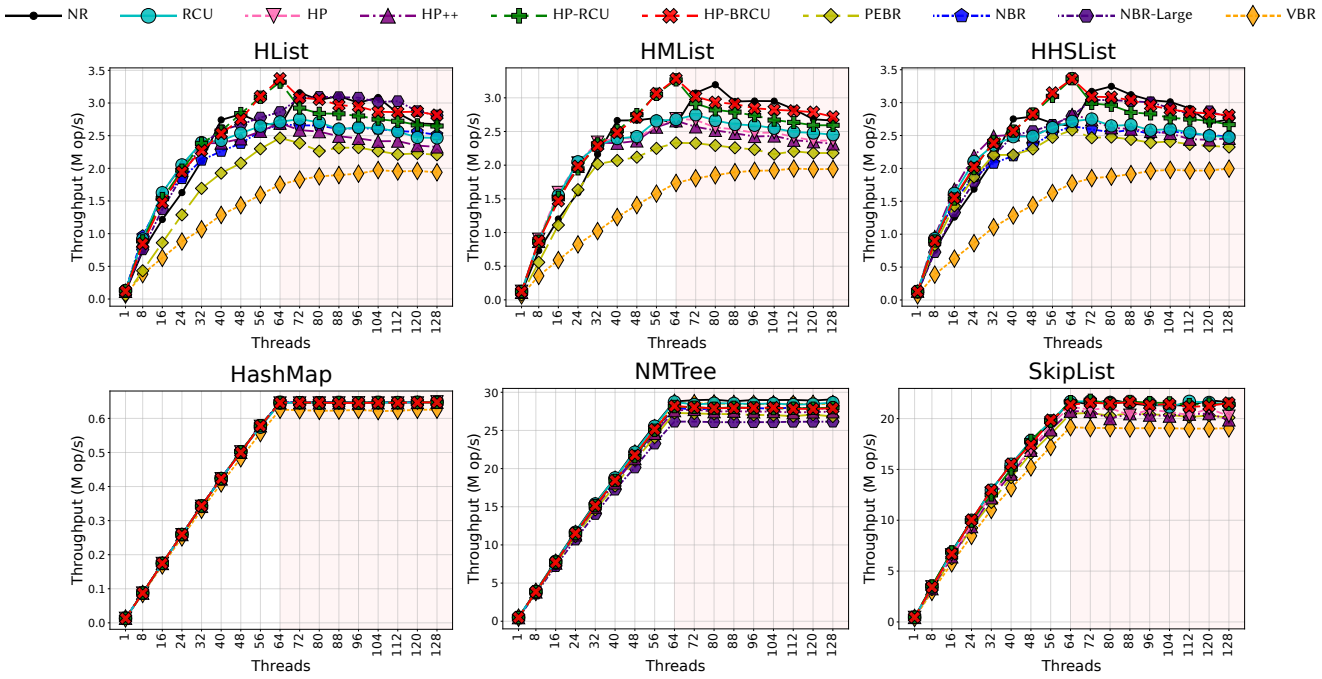


Figure 19: Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

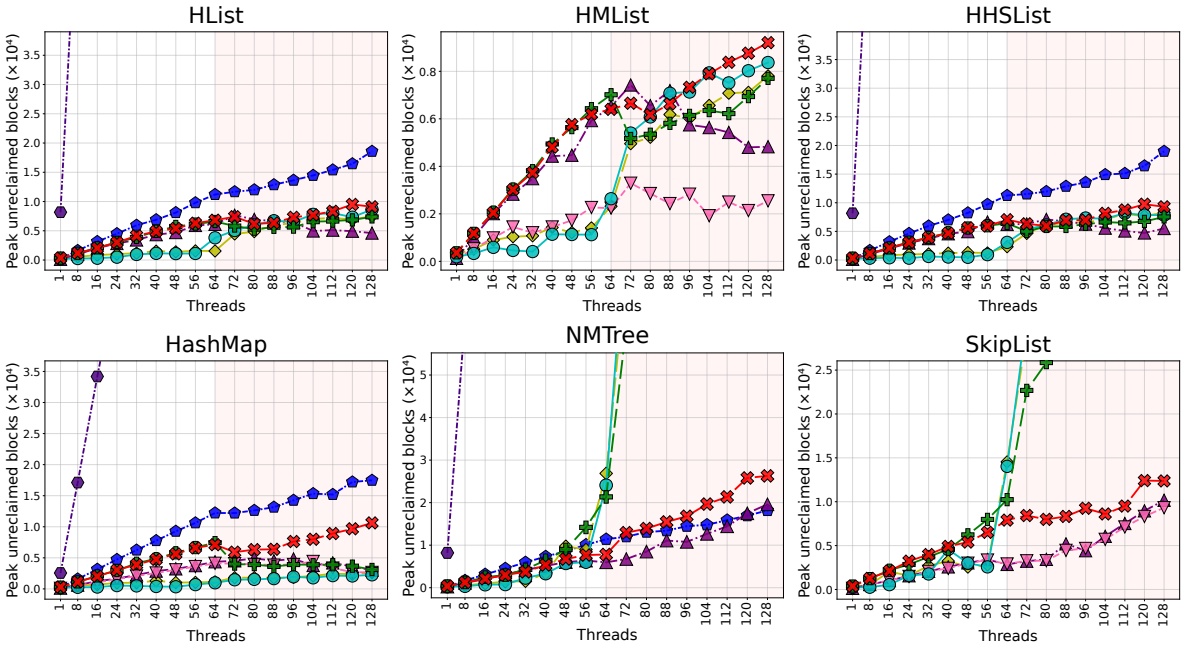


Figure 20: Peak number of unreclaimed blocks of read-intensive workloads for a varying number of threads.

B.2.4 Read-only Workloads.

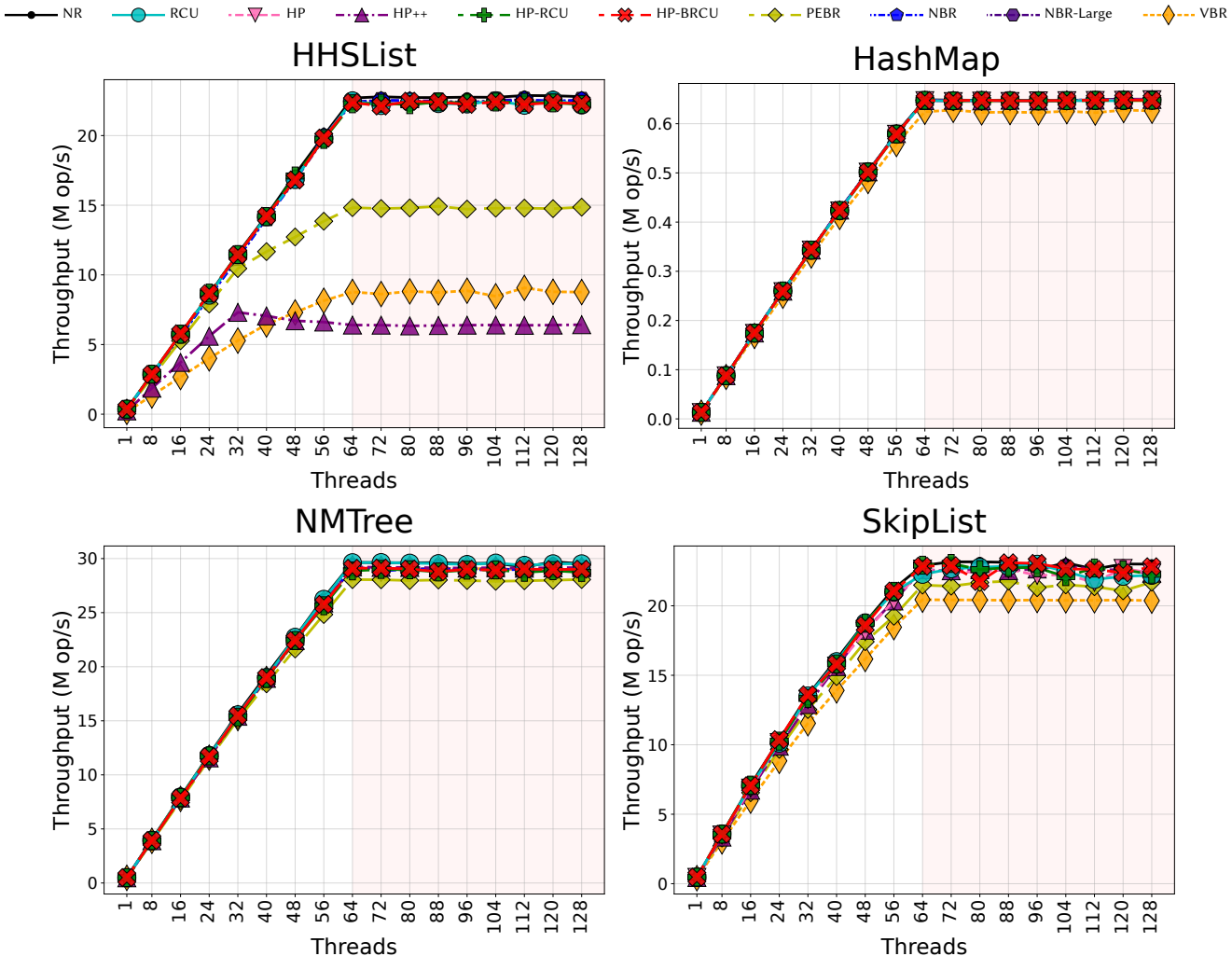


Figure 21: Throughput (million operations per second) of read-only workloads for a varying number of threads.

B.3 Long-Running Operations

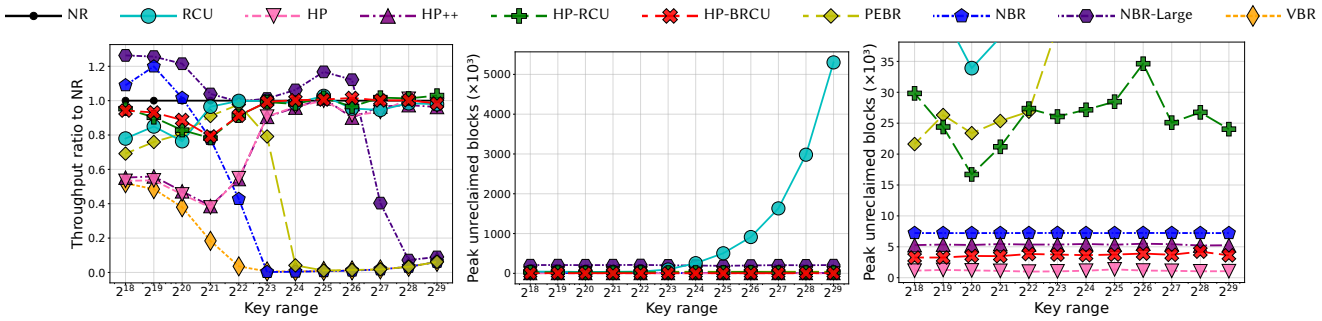


Figure 22: Throughput (million operations per second) and peak number of unreclaimed blocks of long-running read operations.

C INTEL96T FULL EXPERIMENTAL RESULTS

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

C.1.1 Write-only Workloads.

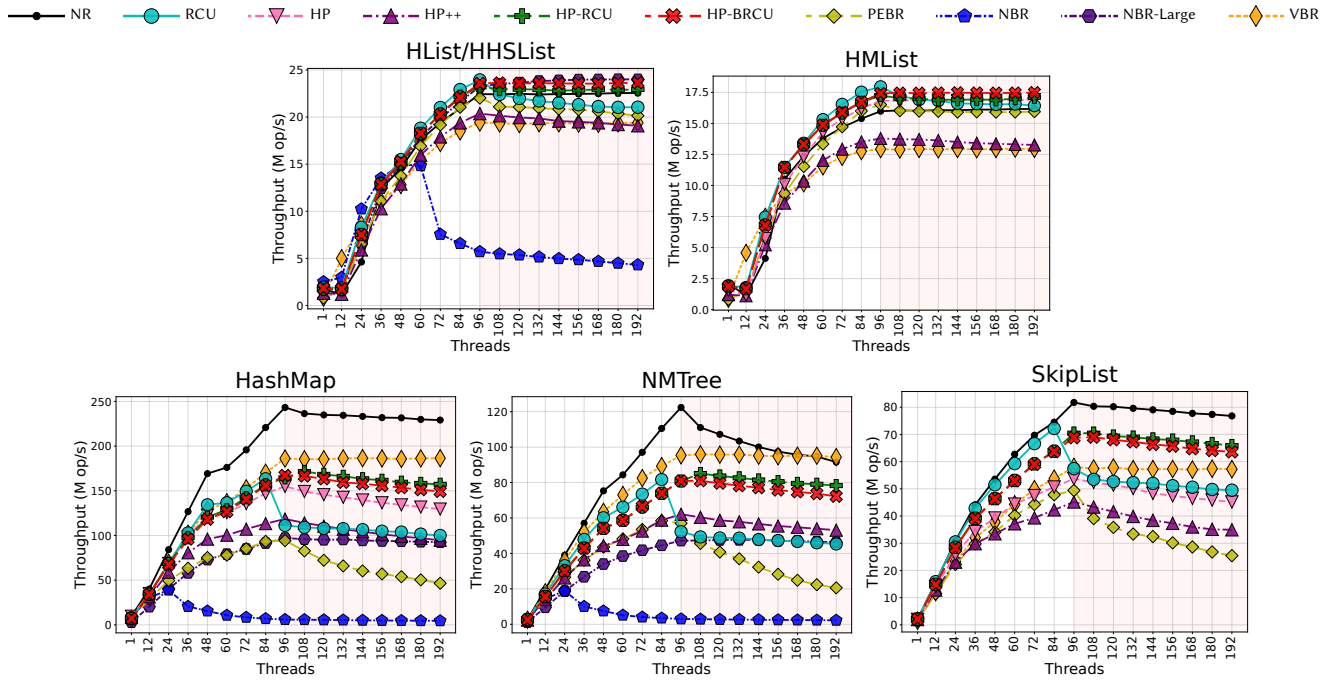


Figure 23: Throughput (million operations per second) of write-only workloads for a varying number of threads.

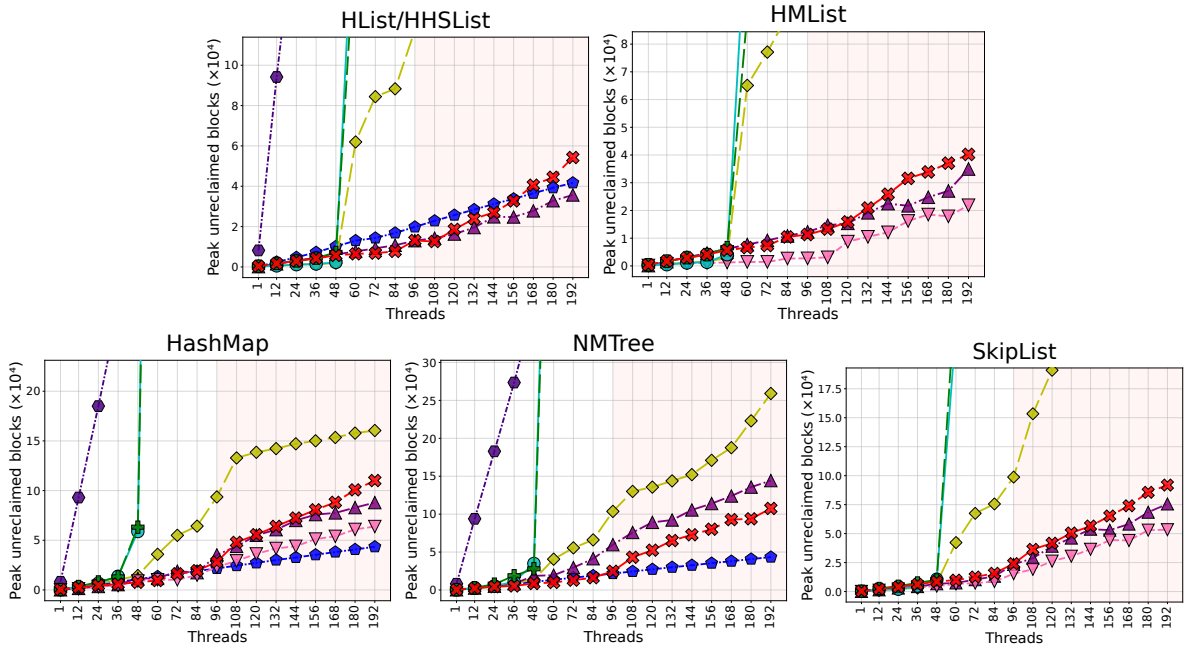


Figure 24: Peak number of unreclaimed blocks of write-only workloads for a varying number of threads.

C.1.2 *Read-write Workloads.*

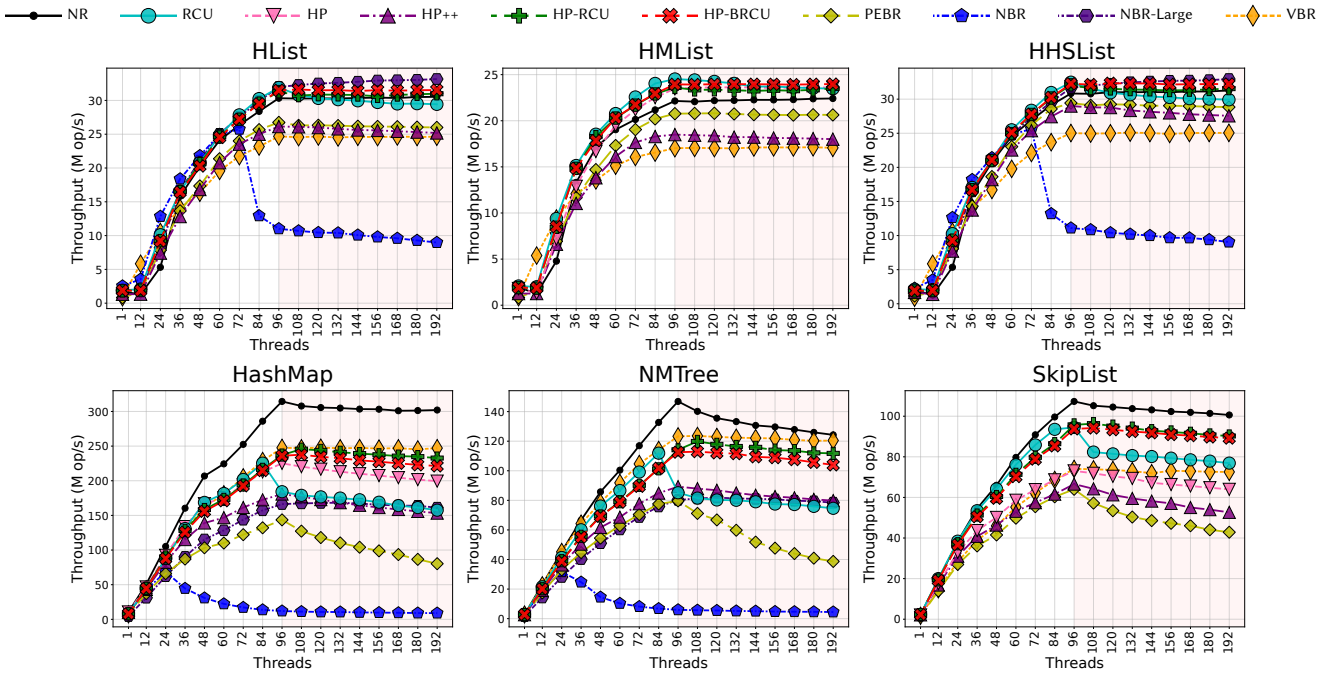


Figure 25: Throughput (million operations per second) of read-write workloads for a varying number of threads.

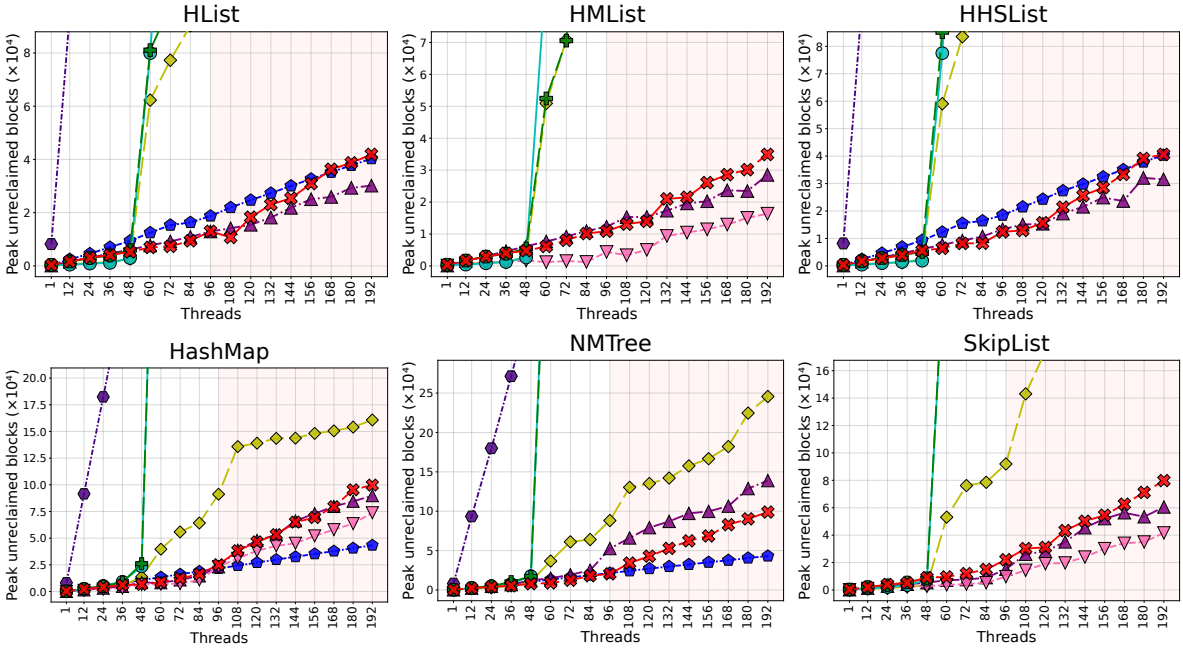


Figure 26: Peak number of unreclaimed blocks of read-write workloads for a varying number of threads.

C.1.3 *Read-intensive Workloads.*

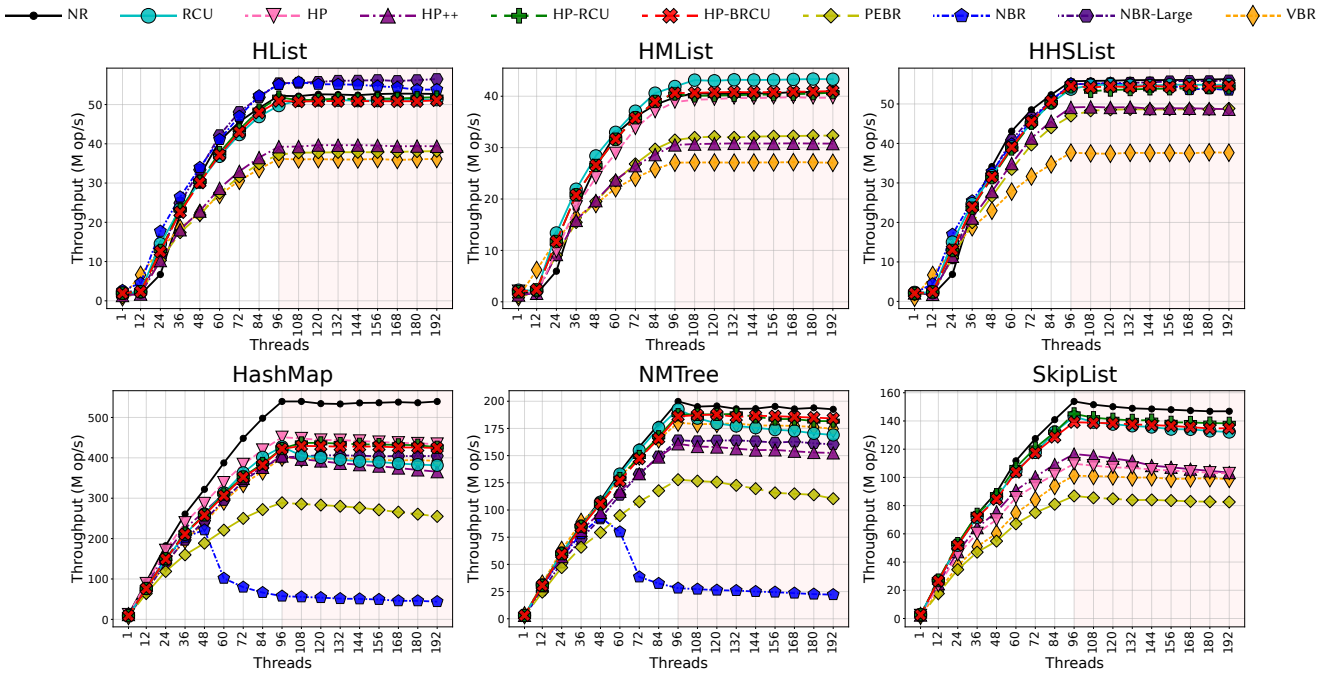


Figure 27: Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

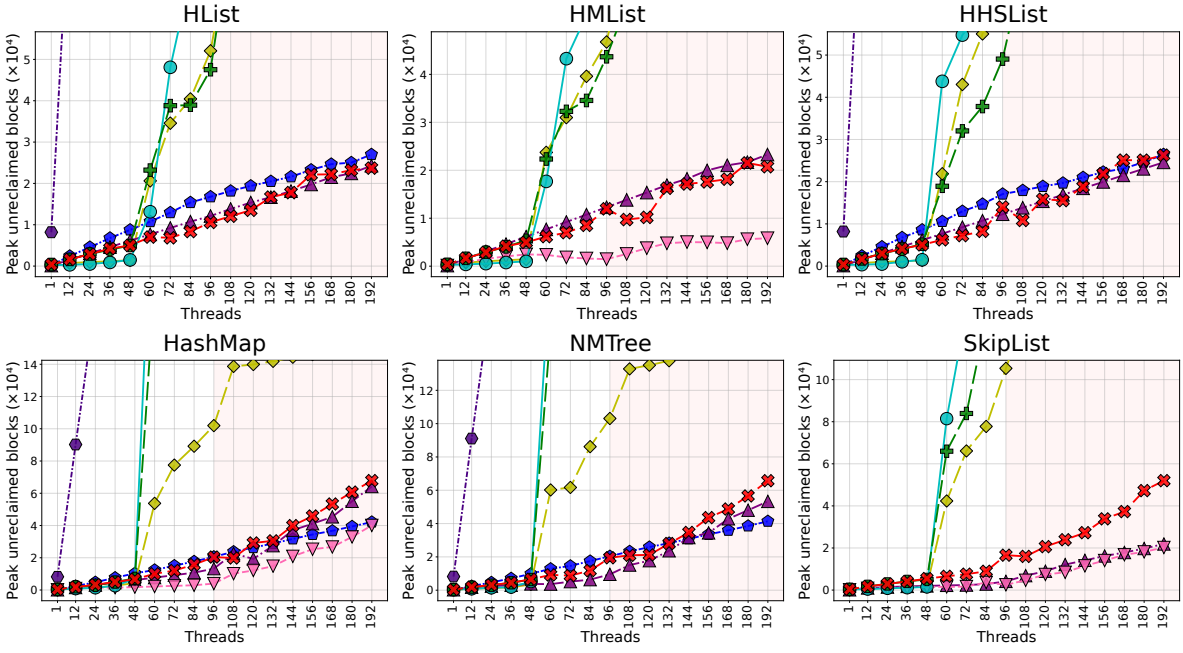


Figure 28: Peak number of unreclaimed blocks of read-intensive workloads for a varying number of threads.

C.1.4 Read-only Workloads.

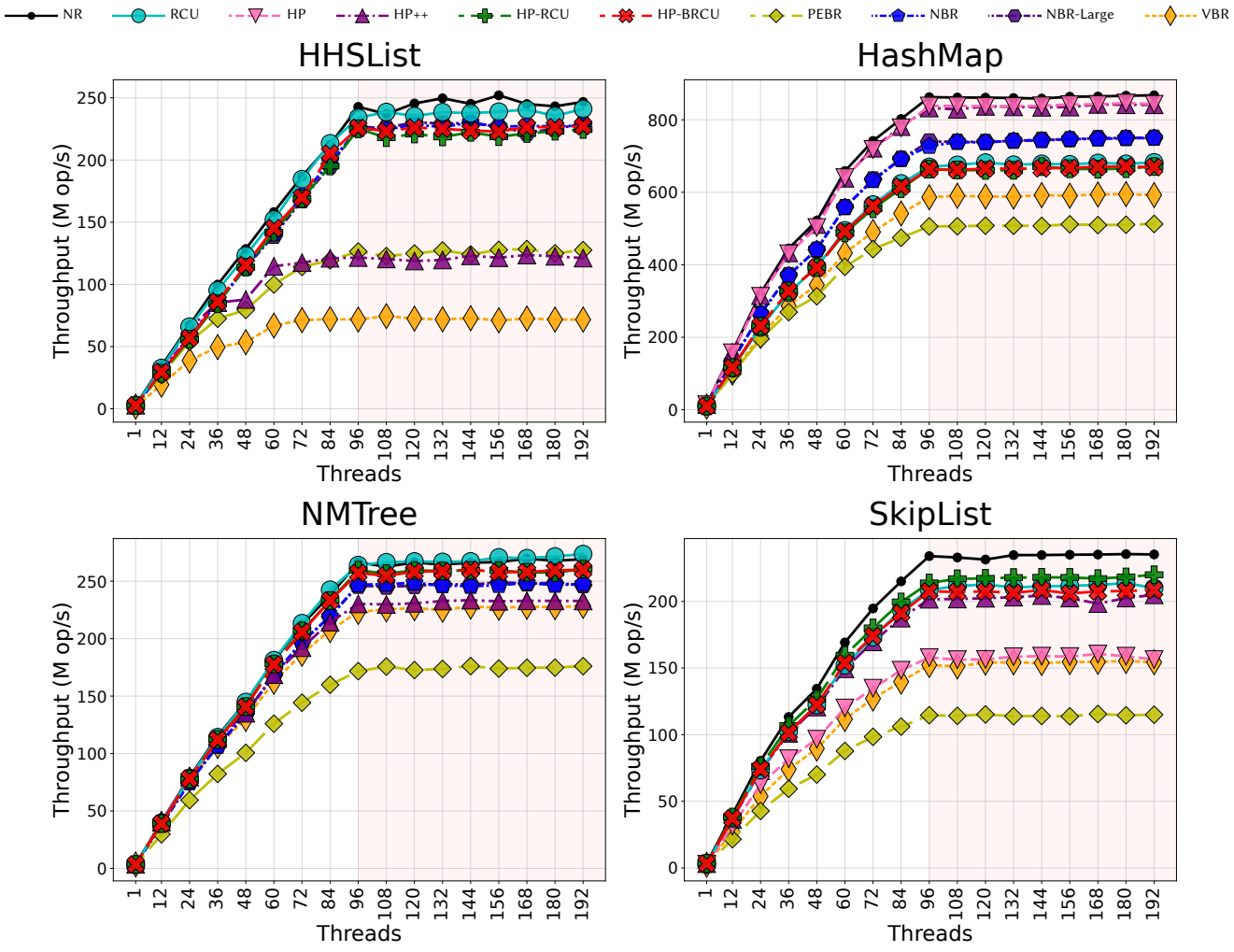


Figure 29: Throughput (million operations per second) of read-only workloads for a varying number of threads.

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

C.2.1 Write-only Workloads.

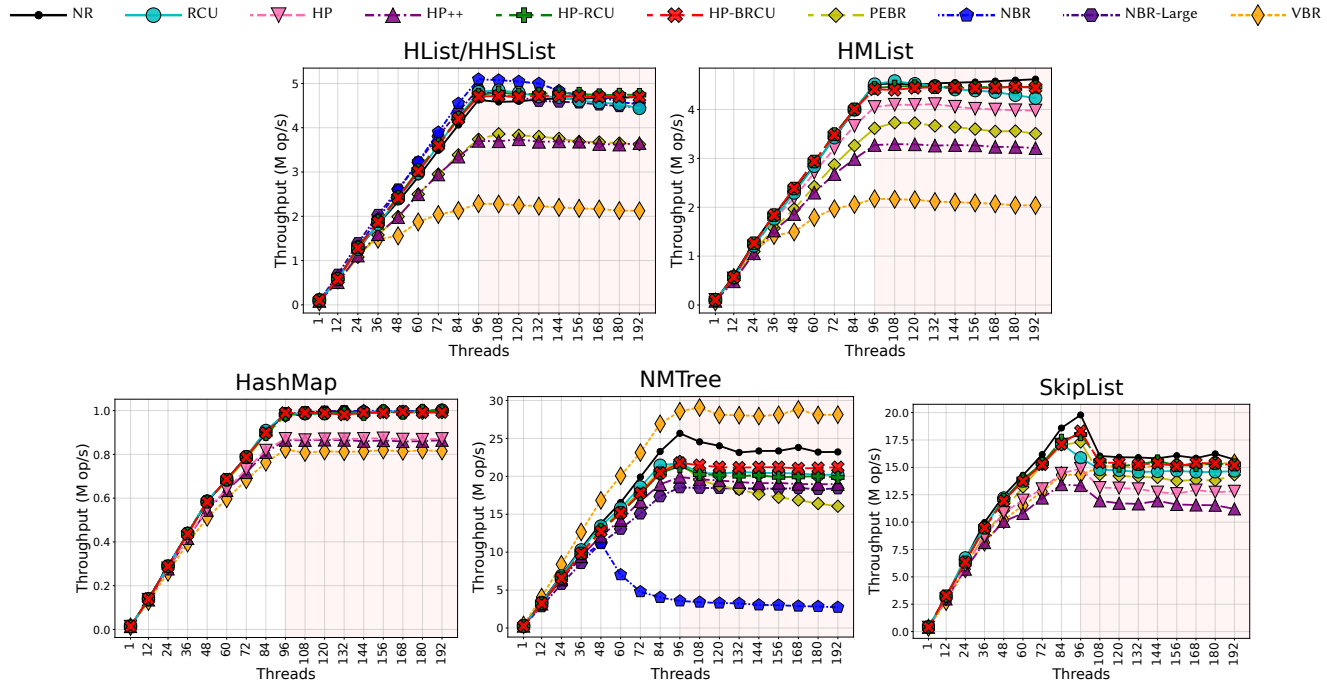


Figure 30: Throughput (million operations per second) of write-only workloads for a varying number of threads.

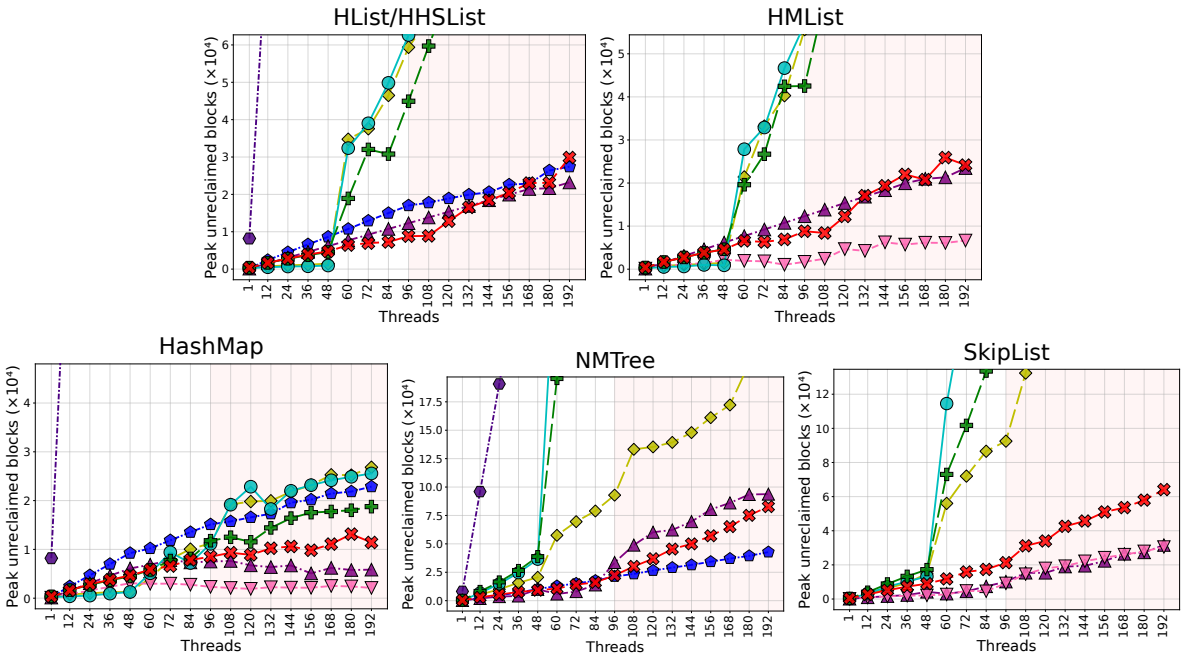


Figure 31: Peak number of unreclaimed blocks of write-only workloads for a varying number of threads.

C.2.2 *Read-write Workloads.*

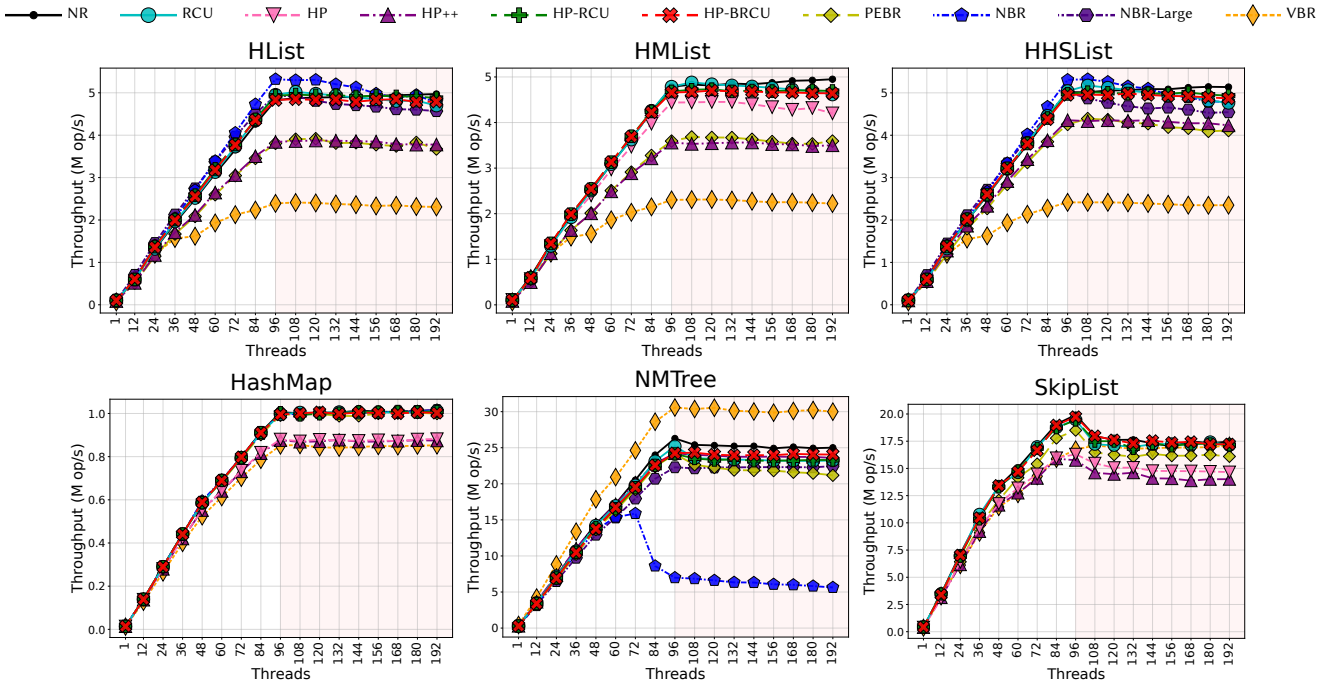


Figure 32: Throughput (million operations per second) of read-write workloads for a varying number of threads.

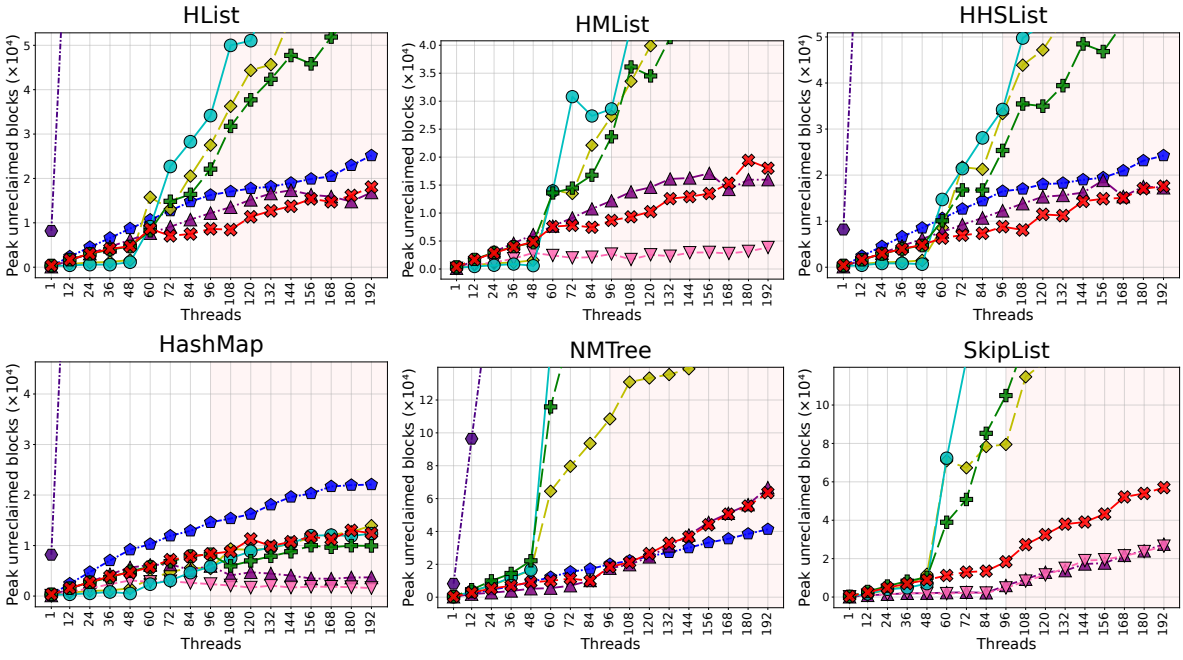


Figure 33: Peak number of unreclaimed blocks of read-write workloads for a varying number of threads.

C.2.3 *Read-intensive Workloads.*

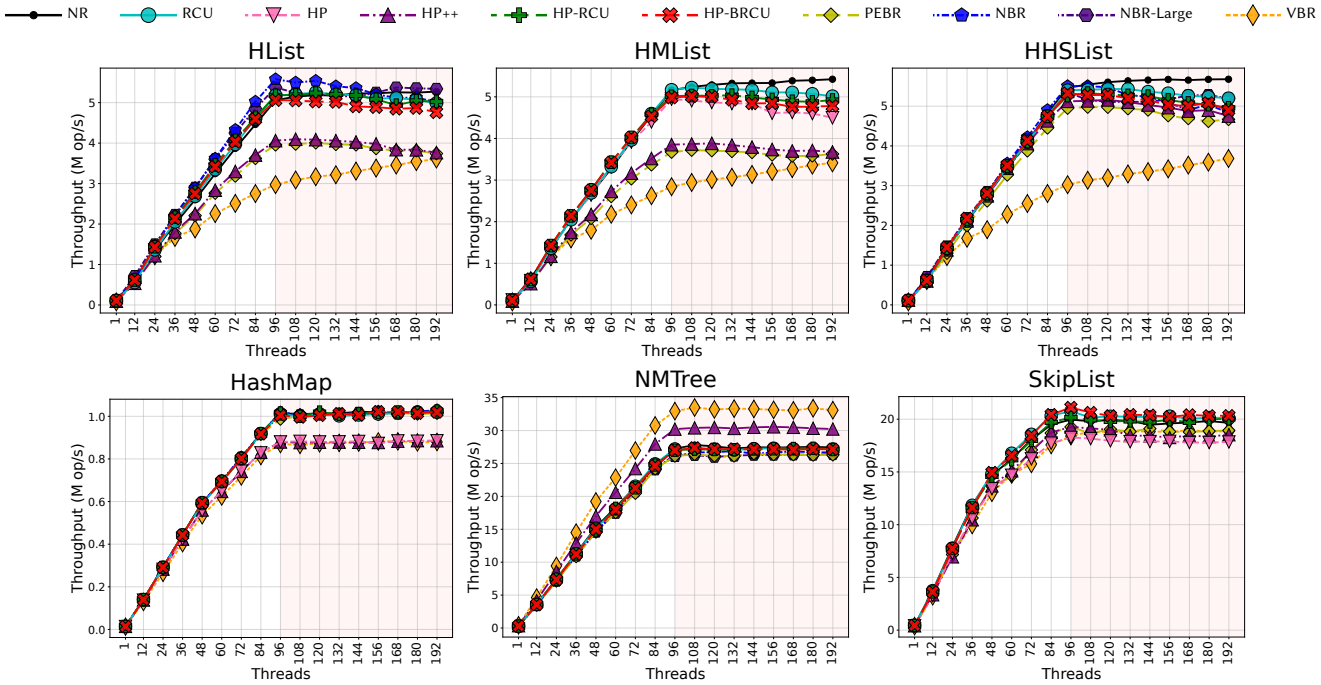


Figure 34: Throughput (million operations per second) of read-intensive workloads for a varying number of threads.

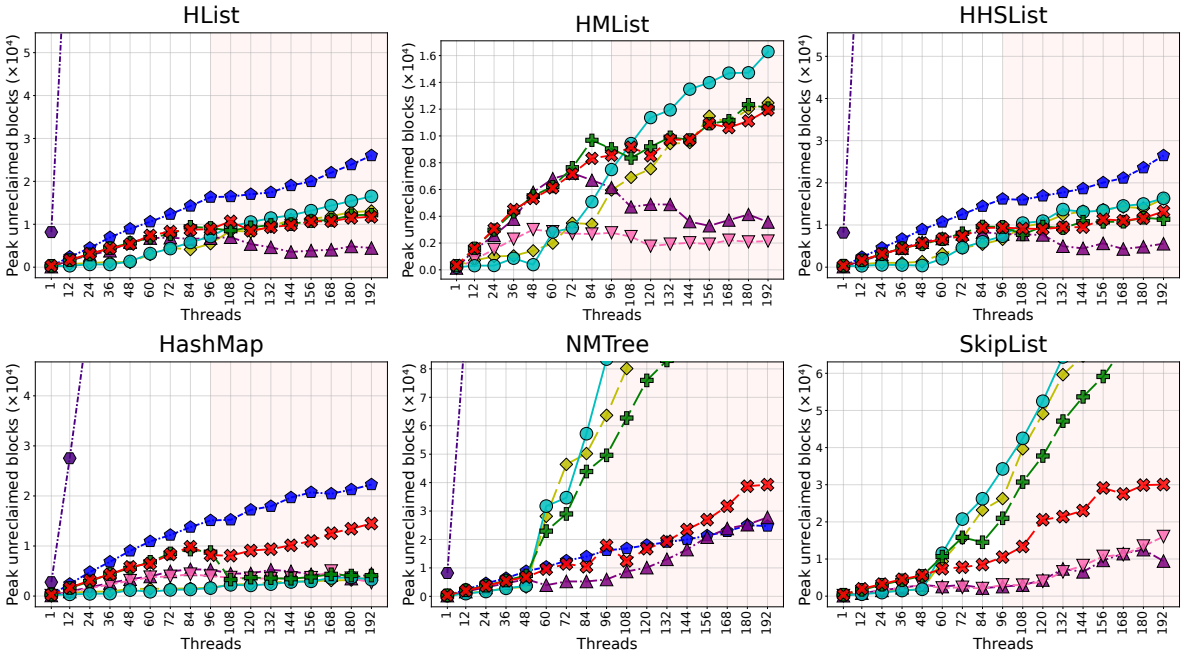


Figure 35: Peak number of unreclaimed blocks of read-intensive workloads for a varying number of threads.

C.2.4 Read-only Workloads.

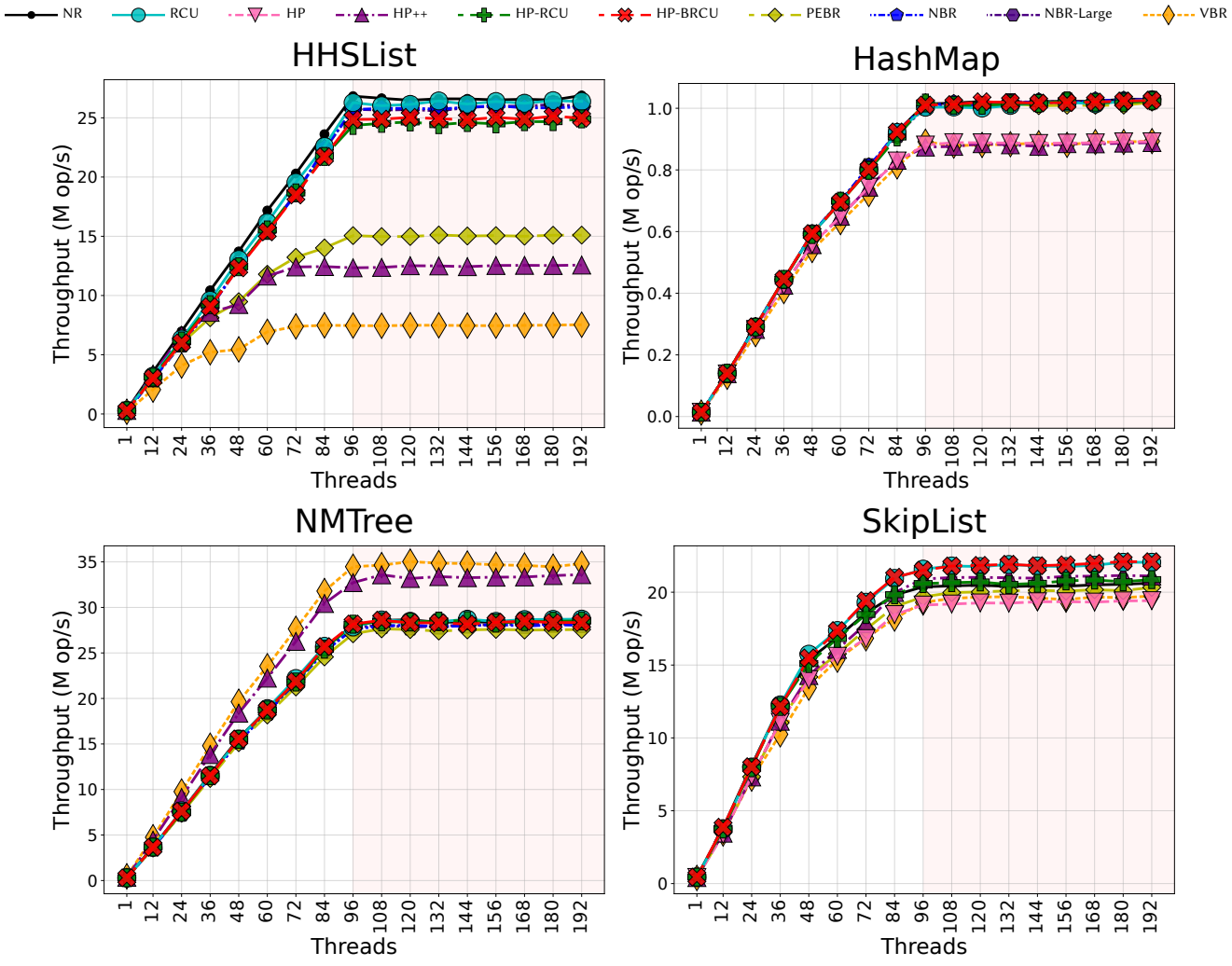


Figure 36: Throughput (million operations per second) of read-only workloads for a varying number of threads.

C.3 Long-Running Operations

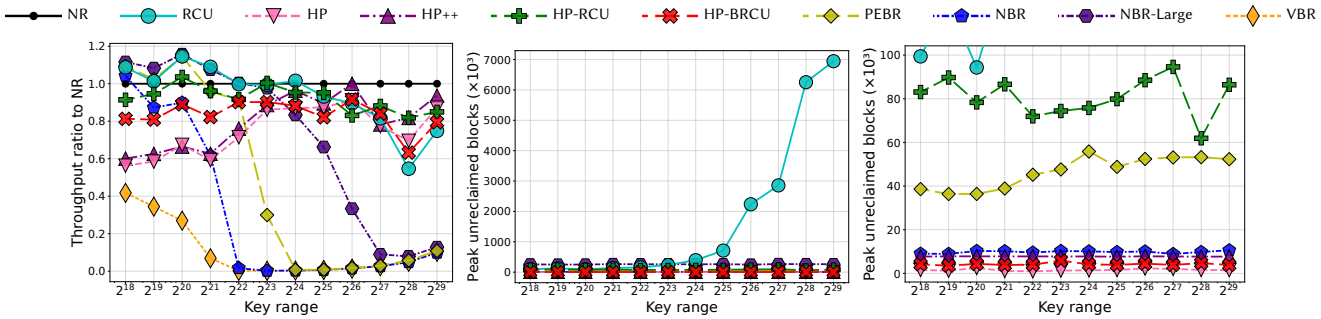


Figure 37: Throughput (million operations per second) and peak number of unreclaimed blocks of long-running read operations.