RESEARCH-ARTICLE

Publish on Ping: A Better Way to Publish Reservations in Memory Reclamation for Concurrent Data Structures

AJAY SINGH, University of Waterloo, Waterloo, ON, Canada

TREVOR ALEXANDER BROWN, University of Waterloo, Waterloo, ON, Canada

Published: 28 February 2025

Citation in BibTeX format

PPoPP '25: The 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming
March 1 - 5, 2025
NV, Las Vegas, USA

Conference Sponsors:
SIGPLAN
SIGHPC

PPoPP '25: Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (February 2025)
https://doi.org/10.1145/3710848.3710890
ISBN: 9798400714436

Publish on Ping: A Better Way to Publish Reservations in Memory Reclamation for Concurrent Data Structures

Ajay Singh
University of Waterloo
Canada
ajay.singh1@uwaterloo.ca

Trevor Brown
University of Waterloo
Canada
trevor.brown@uwaterloo.ca

ABSTRACT

Safe memory reclamation techniques that utilize per read reservations, such as hazard pointers and hazard eras, often cause significant overhead in traversals of linked concurrent data structures. This is primarily due to the need to announce a reservation, and fence to make it globally visible (and enforce appropriate ordering), before each read. In real world read-intensive workloads, this overhead is amplified because, even if relatively little memory reclamation actually occurs, the full overhead of reserving records before use is still incurred while traversing data structures.

In this paper, we propose a novel memory reclamation technique by combining POSIX signals and delayed reclamation, introducing a publish-on-ping approach. This method eliminates the need to make reservations globally visible before use. Instead, threads privately track which records they are accessing, and share this information on demand with threads that intend to reclaim memory. The approach can serve as a drop-in replacement for hazard pointers and hazard eras. Furthermore, the capability to retain reservations during traversals in data structure operations and publish them on demand facilitates the construction of a variant of hazard pointers (EpochPOP). This variant uses epochs to approach the performance of epoch-based reclamation in the common case where threads are not frequently delayed (while retaining the robustness of hazard pointers).

Our publish-on-ping implementations based on hazard pointers and hazard eras, when applied to various data structures, exhibit significant performance improvements. The improvements across various workloads and data structures range from 1.2X to 4X over the original HP, up to 20% compared to a heavily optimized HP implementation similar to the one in the Folly open-source library, and up to 3X faster than hazard eras. EpochPOP delivers performance similar to epoch-based reclamation while providing stronger guarantees.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent programming languages**.

KEYWORDS

safe memory reclamation, concurrent data structures, memory management, Fast Hazard Pointers

1 INTRODUCTION

There is a rich literature of concurrent data structures that can be used as building blocks for concurrent software. However, many concurrent data structure designs assume automatic garbage collection, and cannot be used (without modification) in environments with no garbage collection, such as C/C++. In such environments, concurrent data structures typically must be paired with safe memory reclamation (SMR) algorithms [2–4, 8–10, 13, 15–17, 22–24, 27, 29, 31, 33, 34, 41, 46, 46–48, 50, 54, 55, 57, 60] to prevent use-after-free errors. As an example of a use-after-free error, consider a traversal of a linked-list based set that does not use locks or other synchronization in its traversals: While one thread deletes and subsequently reclaims a node another thread could concurrently access it, potentially leading to a segmentation fault.

Many of the most common SMR algorithms are *pointer based*. The most notable example, the celebrated hazard pointer (HP) algorithm [41], has received a Dijkstra award, and is set to be included in the C++26 standard [42]. Unfortunately, in HPs, whenever a thread encounters a new shared object, such as a node in a list, the thread must *reserve* a pointer to the node by (1) storing it in a single-writer multi-reader (SWMR) slot in a shared array, (2) executing memory fence instruction(s) to *publish* this reservation, making it visible to other threads (and preventing instruction reordering), and (3) re-reading a pointer from which this node was encountered to verify that the reserved node is still reachable at some time after the reservation was published.

The need to fence every time a new node is encountered can cause high overhead in linked data structures. This overhead is even more pronounced in common read-intensive workloads wherein, even though memory reclamation may be infrequent, and relatively lightweight, the overhead that fencing imposes on read-only operations is not. A more recent technique, called hazard eras (HEs) [50], uses monotonically increasing global timestamps and reservation of timestamps, instead of pointers. Roughly speaking, HEs is more coarse grained, as a timestamp represents many nodes. The use of timestamps reduces how often memory fences are needed. For example, if a thread is about to reserve a timestamp that it has already reserved (because a new node being accessed corresponds to the same timestamp as a previously accessed node), the previously published timestamp can be reused (without additional fencing). However, the overhead of HEs can still be substantial.

Another alternative is to use fast epoch based reclamation (EBR) algorithm [26, 29, 39]. However, these techniques are not robust:

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710890

A delayed thread can prevent all threads from reclaiming memory (forever) resulting in unbounded memory consumption. Subsequent reclamation techniques like NBR [54, 57], VBR [51] and IBR [60] provide a variety of robustness guarantees, but they have other trade-offs in, for example, ease of integration, applicability to classes of data structures, assumptions regarding whether memory pages are returned to the operating system, and so on. NBR requires data structure operations to have a particular structure, consisting of read- and write-phases, with specific requirements in each phase. Optimistic techniques like VBR require a type preserving allocator, and cannot allow memory pages to be returned to the operating system (without tricks involving trapping and ignoring segmentation faults). Moreno and Rocha designed a custom allocator that leverages the virtual memory subsystem to solve this problem by customizing page fault handling [44]. However, these techniques cannot be used as drop-in replacements for hazard pointers without making sacrifices in ease of use, portability, or applicability to certain data structures.

In this paper, we propose a technique called *publish-on-ping (POP)* which can be used to design fast and robust drop-in replacements for pointer based techniques, such as hazard pointers and hazard eras. The POP technique can be applied to hazard pointers to significantly improve performance, resulting in an algorithm we call *HazardPtrPOP*. Perhaps surprisingly, EBR can also be incorporated into HazardPtrPOP, allowing different threads to *simultaneously* operate in epoch-based and hazard pointer modes. This results in an algorithm called *EpochPOP* that achieves performance similar to EBR, while retaining the robustness guarantees of HazardPtrPOP.

The key to safety from use-after-free errors in most pointer based techniques is that threads traversing a data structure eagerly reserve and publish (making the reservations visible to all threads) before accessing nodes during traversals, and threads reclaiming objects must first scan all the reservation to avoid freeing reserved objects. It is highly pessimistic to publish reservations (and fence) just because there *might be* some concurrent reclaimer thread about to free some nodes. Even if there is very little (or no) reclamation in a given workload, traversals must pay this cost for every node visited. In this paper, we ask the question: What if each traversal could publish its set of reservations *precisely* when a reclaimer is about to scan threads' reservations?

As in the recent NBR algorithm, POSIX signals are the key to our POP technique. However, signals elicit different behaviour from the recipient in POP than in NBR. In NBR, a thread that wants to reclaim memory sends signals to all other threads, causing those threads to discard all pointers they hold and *restart* their current traversals (unless they have already performed writes to the data structure). In POP, a thread that wants to reclaim memory sends signals to all other threads (as in NBR), but a thread receiving a signal does not need to restart its operation or otherwise change its control flow. Rather, it simply publishes its reservations (which it had been tracking *locally* until this point), increments a SWMR counter that tracks how many times it has published reservations, and issues a single memory fence. Once all threads have published their reservations, the reclaimer can scan them and free nodes that are not reserved. In a nutshell, this is how HazardPtrPOP works.

EpochPOP builds on HazardPtrPOP as follows. In the common case, threads execute data structure operation largely as they would in epoch based reclamation [26, 29, 39]. A data structure operation begins by reading a global epoch number (a timestamp), and announcing the value it read in a per-thread SWMR slot in a shared array, then proceeds as usual. Nodes unlinked from the data structure are stored in a per-thread list. Periodically, threads increment the global epoch, and scan other threads' announcements to identify the oldest announced epoch. Threads can free objects unlinked from the data structure in epochs older than the oldest announced epoch. However, unlike traditional EBR, all threads also keep executing the steps of the HazardPtrPOP algorithm, locally reserving nodes before accessing them. If threads announce the epochs sufficiently often, they are able to reclaim memory regularly to keep the size of the lists with unlinked objects below a user-specified threshold, then the POP mechanism is not needed at all! But, if thread delays hinder frequent announcements of epochs, preventing reclamation of the lists, then POP is activated to enable reclamation.

The dual mode of operation in EpochPOP differs significantly from the fast-path/slow-path approach used in Qsense [8]. In Qsense, all threads globally switch (together) between a hazard pointer based mode, and an epoch based mode. This involves additional synchronization and background threads, and necessitates careful decision making about *when* all threads should switch between modes. In the epoch based mode, if a single thread is delayed, all threads must switch to the hazard pointer mode.

In contrast, EpochPOP does not require any global mode switching (or requisite tuning). Threads can effectively run in both modes simultaneously. (One reclaimer may observe a thread delay, while another does not, and the latter can simply continue reclaiming with epochs while the former pings all threads.) QSense and POP both avoid fencing after individual reservations, but the techniques are quite different, and QSense requires additional background threads (one per core), whereas POP only needs signals.

Other signal-based techniques, such as NBR[54], DEBRA+ [13], HP-BRCU [36], VBR [51] and PEBR [34] require programmers to either insert checkpoints and carefully identify safe regions for restarting or to add custom recovery code in data structure operations. While this may be straightforward in some cases, it can be challenging or even infeasible in others [14, 51].

Overall, POP is no more difficult to use than hazard pointers and is easier to implement than existing signal-based techniques that force thread restarts, like NBR. Similar to other signal-based approaches, POP algorithms avoid signal overhead when threads are not actively reclaiming memory. However, unlike these techniques, POP algorithms do not require thread restarts, making them well-suited for long-running read operations.

1.0.1 Outline: The remaining paper is organized as follows. In Section 2, we recall Hazard Pointer and EBR to understand the algorithms presented in this paper. Then, in Section 4 we describe our Publish on Ping based algorithms– HazardPtrPOP, HazardEraPOP and EpochPOP– in detail followed by a discussion on correctness. In Section 5, we evaluate and compare the performance of HazardPtrPOP, HazardEraPOP and EpochPop with some previous techniques. Finally, in Section 6 we discuss some technique similar in goals as ours and conclude in Section 7.

2 BACKGROUND

A data structure object from its allocation to its freeing back to the underlying allocator can be in the following different states. *reachable*: when the object after being allocated memory is inserted in to the data structure and threads could access it. *deleted*: when the object is logically marked for removal from the data structure. *retired*: when the object has been removed from the data structure, but yet not freed. Threads which gained access to the object before it was removed could still have access to it. *free*: when the object has been returned to the allocator and could be recycled for subsequent allocation requests.

Terminology. A data structure object is said to be *safe* for reclamation if it is in the retired state and no thread could access it. Otherwise, it is *unsafe* for reclamation. Typically, in *deferred memory reclamation*, every thread has a local list, called *retire list*, to which they add the *retired* objects until they are safe to be freed. This ensures retired objects are safe to be freed and amortizes reclamation overhead. When a retire list reaches a threshold, threads free the objects using the reclamation technique's synchronization. In the context of reclamation algorithms, a thread with a reference to a shared object in the data structure is termed a *reader*, while a thread deleting an object from the data structure is called a *reclaimer*. Readers can save a pointer or a timestamp, representing a collection of objects, in their local or at shared memory locations. These are referred to as *reservations*. These *reservations* are made globally visible for reclaimers, referred to as *publishing*. Threads are considered to be in a *quiescent state* between consecutive data structure operations, meaning they do not access any shared objects.

The synchronization mechanisms, vary between techniques, and ensure that the reclaimers only free *safe* objects and *readers* do not access *unsafe* objects. For example, in hazard pointers, synchronization happens by requiring *readers* to reserve and publish objects before accessing them, and *reclaimers* to first scan reservations to identify all the *unsafe* objects, and subsequently freeing only the *safe* objects from their retire list. Similarly, in epoch based techniques, *reclaimers* must wait before freeing an object in their retire list until all threads have gone quiescent at least once since the object was retired.

This section revisits hazard pointers (HP) [41] and read-copy-update (RCU) style [29] implementations of epoch based reclamation (EBR). HP is used in MongoDB and in Folly- Facebook's open source C++ library, and RCU is part of the Linux OS. Both techniques have been proposed for the C++ standard library [42, 43]. Therefore, we have chosen these two techniques for discussion in the background section and will later build upon them to explain our publish on ping implementations of these algorithms.

2.1 Hazard Pointers

2.1.1 *The Overview.*

The key principle underlying the operations of Hazard Pointers (HP) involves a contract between *readers* and *reclaimers*. *Readers reserve* and *publish* pointers to objects currently being accessed at single-writer multi-reader (SWMR) locations for *reclaimers*. *Reclaimers*, in turn, ensure they scan all the SWMR locations to collect all reserved objects and only free those whose reservations have not been *published*. This ensures safety from use-after-free errors.

Crucial to publishing of reservations in HP is that every read of a shared object pointer should execute a memory fence. Specifically, *readers* should follow these steps to timely publish their reservation to a pointer: 1) save the pointer to the object being read, 2) execute a memory fence, 3) validate that the pointer saved in the step 1 is still *reachable*, if not, retry or abort the operation. This ensures that the pointer was *reachable* at the time it was reserved (in step 1) –a crucial condition for correct application of HPs [41]. Without the memory fence, the reservation of the pointer in step 1 could be reordered after the validation of reachability step (step 3). This could lead to reservation of an object which has already been deleted, resulting in unsafe accesses in the future.

2.1.2 *The Problem.*

The per read memory fences incur high overhead, leading to poor scaling of data structures. Our Perf tool analysis, on a Hazard-Michael list of size 100 nodes where 128 threads execute 50% inserts and 50% delete operations showed that searches approximately spend ≈ 50% of CPU cycles on reading HPs, whereas searches in a leaky implementation of the same list only spends *approx* 15% of CPU cycles in reading the HPs.

Several subsequent techniques have aimed to reduce or eliminate the need for memory fence during traversals in HP. For example, Cadence [8] utilizes system level memory fences triggered by context switches. This allows *readers* to defer publishing reservations (using explicit memory fence) until a context switch occurs. *Reclaimers*, to free a retired object, wait for a context switch to occur since the object was retired. This waiting period ensures that the reservations to the object, if any, become visible and the *reclaimer* can free it if it is not found to be reserved.

Cadence employs auxiliary threads pinned to each core to force the context switches at regular intervals. These auxiliary threads compete with primary worker threads that execute data structure operations, impacting performance. Moreover, the ***assumption*** that rescheduling always triggers a memory fence is architecture dependent [8, 38]. This mechanism requires the instrumentation of data structure objects with a timestamp to determine whether a global memory fence has occurred since the object was retired, affecting the memory layout of the underlying data structure. Additionally, the overhead due to auxiliary threads enforcing a global memory fence, though amortized over multiple operations, is incurred even if threads do not reclaim. So, the issue of uneven overhead between readers and reclaimers is still present.

Dice et al. in [23] discuss a technique to eliminate memory fences from traversals by leveraging write-protect feature in modern operating systems, which enforces a global memory barrier. *Reclaimers* briefly write-protect all memory pages associated with hazard pointers and then remove the protection, ensuring a global memory fence is executed. This makes all the hazard pointers visible before attempting to free retired objects. However, this technique requires all threads to block in an interrupt handler until the write protection is revoked. In the same paper, the authors also discuss a hardware extension suggesting the addition of a dedicated store buffer unit along with two new instructions to maintain hazard pointers.

Another technique, Hazard Eras (HE) [50], which emerged after the former two, attempts to amortize the cost of the per-read memory fences by using timestamps. *Readers* reserve current eras, instead of pointers, represented by monotonically increasing global timestamps, whenever they read a pointer. The memory fences to timely publish the reserved eras are only incurred when the epoch changes concurrently with a read of an object pointer. In more detail, Hazard Eras leverages infrequent changes in epochs to avoid incurring memory fences at every read. *Reclaimer* free only those nodes whose lifetime does not intersect with epochs reserved by all threads. Although the amount of garbage is bounded but it still can be very high proportional to the all the memory whose lifetime intersects with a reserved era, and is still slower in many workloads as observed in our experiments.

The process-wide memory barrier `sys_membarrier`[21] on Linux reduces the asymmetric overhead between readers and reclaimers and is used in both the RCU in the kernel and Folly's HP implementation. However, the availability and implementation of `sys_membarrier` (which can be blocking) vary across kernel versions and architectures. When unavailable, the system may fall back to `mprotect`, which can degrade performance[20]. Additionally, `sys_membarrier` has been linked to security vulnerabilities [1]. While HP with `sys_membarrier` offers significant improvement over the original algorithm, it can still be 12%-40% slow, as our experiments will show.

2.2 Epoch Based Reclamation

2.2.1 The Overview. Epoch based reclamation (EBR) has multiple variants, such as those proposed by Harris [28], Fraser [26], RCU [29], and Brown's DEBRA [13]. The key concept across these algorithms is threads execute data structure operations in sequence of monotonically increasing timestamps called epochs. *Readers* publish the current epoch they are executing in and *reclaimers* can free objects retired in or before a given epoch e once every thread has completed execution in epoch e or earlier and has transitioned to a more recent epoch (greater than e). Implying that for objects retired on or before epoch e, all threads have gone quiescent at least once, making the objects *safe* for reclamation. In RCU [43] terminology, while executing data structure operations, threads are assumed to be executing a read-side critical section. A *reclaimer* can only free objects that are *retired* before the beginning of the oldest read-side critical section. Pseudocode appears in the extended version of the paper [53].

2.2.2 The Problem. The main concern with EBR is that a thread could get stuck in a data structure operation, leading to a delayed exit from an old read-side critical section due to arbitrary system level reasons, such as page fault servicing, thread scheduling etc. These delays might be significant enough to cause the minimum declared epoch to lag far behind the current global epoch. This situation prevents all threads from freeing their retire lists, resulting in a drastic increase in system memory consumption and possibly leading to out-of-memory errors. This issue is commonly referred to as the lack of *robustness*.

Various techniques, such as NBR [54, 57], VBR [51], and HP-BRCU [36], achieve robustness and efficiency but compromise the programmability of original HPs to varying degrees, in line with the ERA theorem [52]. These methods typically require programmers to install checkpoints (one per read phase in NBR, and multiple per read phase in VBR and HP-BRCU) and assume that a thread can abort and restart from a previous checkpoint while it is executing in an arbitrary code region.

Additionally, in NBR and HP-BRCU, programmers must identify regions in code that are *unsafe* for signal-induced restarts, such as those where threads might have taken locks or performed partial updates [19, 28, 30]. They must also identify and reserve all nodes that could be accessed within these regions beforehand. This process can be challenging for arbitrary data structures [14, 36, 51]. These restarts can introduce varying degrees of overhead and non-trivial reasoning about the correctness of the memory reclamation scheme for practitioners.

2.3 Summary: The Problem and The Solution

Programmers face unattractive choices: choose robust but slow hazard pointers[41], opt for robust and fast ad hoc era or pointer reservation-based techniques [8, 23, 50], but tolerate intrusiveness or less portable hardware dependent solutions, or opt for a fast but not robust EBR[13, 26, 28, 29] or opt for hybrid techniques that are fast and robust but tradeoff programmability in varying degrees [36, 51, 54, 57].

Consequently, it will be desirable to have a simpler solution that is fast and robust and yet maintains a similar programmability property like the original hazard pointers. Hazard eras and `sys_membarrier` was on right path for achieving this but it seems to be slow, as we will show in our experiments.

This paper introduces a fast and non-intrusive publish-on-ping paradigm that uses signals without inducing complexity of signal-based restarts like the earlier techniques to accelerate hazard pointers. We apply publish-on-ping to hazard pointers and hazard eras that result in the new algorithms HazardPtrPOP and HazardEraPop, respectively. Building upon HazardPtrPOP, we implement a variant of hazard pointers using EBR to accelerate hazard pointers in the common case where threads are not frequently delayed. In rare cases where thread delays are suspected, *reclaimers* can continue freeing their retire list using publish-on-ping. The technique is unique in the sense that it does not require switching between the two reclamation schemes and two threads could be reclaiming at the same time in either common EBR or less common HazardPtrPOP way. These techniques are backward compatible with hazard pointers and hazard eras, retaining their ease of programmability and apply to all data structures hazard pointer applies to.

3 PUBLISH ON PING

3.1 The Overview

A key aspect of our technique is that threads can read new pointers and reserve them locally without immediately publishing these reservations to *reclaimers*. This approach eliminates the need for costly memory fences on every read. *Reservations* are published only when a thread attempts to reclaim its retire list. Specifically, when a thread wants to reclaim its retire list, it signals all participating threads to publish their local *reservations* to shared locations. The *reclaimer* then scans these shared locations to collect all the *reservations* and subsequently frees all retired objects in its retire list that are not reserved by other threads.

The publish-on-ping (POP) behavior is implemented using a POSIX signal and a corresponding signal handler. *Reclaimers* use the `pthread_kill` call to signal all other threads in the system (**ping**). The other threads execute a signal handler to assign their local reservations to the corresponding shared locations (**publish**) for reclaimers.

In summary, POP removes the memory fence overhead from the main traversal path by having threads publish reservations only when reclamation events occur, with a simple signaling mechanism. This avoids forcing thread restarts (unlike previous signalling based techniques, such as NBR(+), DEBRA, and PEBR) and reduces overhead on the read path, which is particularly beneficial for data structures with read-dominated workloads. Moreover, POP is equally effective for other related reservation based techniques, incurring similar per-read memory fences during traversals, such as, Hazard Eras (HE). We apply POP to HE and observe performance improvements, as demonstrated in Section 5.

4 ALGORITHMS

4.1 HazardPtrPOP

Having discussed publish-on-ping, we now apply it to HP to eliminate the need for publishing reservations during traversals or reads of new objects in data structures. The resulting algorithm is termed Hazard Pointer Publish-on-Ping (HazardPtrPOP). First, we provide an overview of a typical HP implementation.

4.1.1 Programmer's view of HP. For programmers, a standard Hazard Pointers reclamation interface includes the following three main functions:

(1) READ( ): Used for every read of a new data structure object. *Readers* use READ to ensure that threads *reserve* and *publish* the object to a shared location. (2) CLEAR( ): Employed to remove reservations when threads finish accessing an object or exit the operation. (3) RETIRE( ): Used during update operations, *reclaimers* employ RETIRE to append *deleted* objects to their retire lists. During the RETIRE, *reclaimers* collect every other threads's published reservations. Subsequently, in an iterative fashion, they free those objects from their retire list that were not present in their collected reservations.

Algorithm 1 and Algorithm 2 show the implementation of the proposed HazardPtrPOP algorithm. From a programmer's perspective, it maintains the same interface as in HP, making it backward compatible with HP. Similarly to HP, threads maintain a list per thread, depicted as `retireList` (line 3) to which they append the retired objects with call to RETIRE( ) within their update operations.

Contrary to the publish eagerly paradigm in HP, threads in HazardPtrPOP locally save the objects at their own list of slots in `localReservations` (line 4) array during READ. The READ (line 8) procedure repeatedly reads the pointer to the object, saves it in a corresponding slot in `localReservations`, and then rereads the pointer. Assuming a system with maximum of `NTHREADS` threads, every thread has the same fixed number of slots represented by `MAX_HP`. The loop exits only when the pointer remains unchanged on the second read, upon which the pointer is returned. When a thread is about to go *quiescent*, i.e., while exiting the data structure operation, it resets the local reservations by setting the corresponding slots to NULL (line 25) by using CLEAR. This allows the reserved nodes to get freed when they are not in use. These local reservations are published to a corresponding shared array of single-writer multi-reader slots, called `sharedReservations` (line 5).

Algorithm 1 HazardPtrPOP: Hazard Pointer Publish-on-Ping.

```
 1: const int reclaimFreq                         ▷ frequency of reclaiming retire list
 2: thread_local int tid                                        ▷ current thread id
 3: list<T*> retireList [NTHREAD]
 4: T* localReservations [NTHREAD][MAX_HP]
 5: atomic<T*> sharedReservations [NTHREAD][MAX_HP]
 6: atomic<int> publishCounter [NTHREAD]
 7: thread_local int collectedPublishCounters [NTHREAD]
 8: procedure T* READ(atomic<T*> &ptrAddr, int slot)
 9:     repeat
10:         T* readPtr ← *ptrAddr
11:         localReservations[tid][slot] ← readPtr
12:         ▷ no store load fence needed.                                        ◁
13:     until readPtr = *ptrAddr
14:     return readPtr

15: procedure RETIRE(T* ptr)
16:     myRetireList ← retireList[tid]
17:     myRetireList.append(ptr)
18:     if myRetireList.size() ≥ reclaimFreq then
19:         COLLECTPUBLISHEDCOUNTERS( )
20:         PINGALLTOPUBLISH( )
21:         WAITFORALLPUBLISHED( )
22:         RECLAIMHPFREEABLE(myRetireList)

23: procedure CLEAR( )
24:     for $slot = 0, \dots, MAX\_HP$ do
25:         localReservations[tid][slot] ← NULL
```

Algorithm 2 HazardPtrPOP: Continued.

```
26: procedure RECLAIMHPFREEABLE(myRetireList)
27:     ▷ collect all published reservations                                     ◁
28:     set<T*> collectedReservations ← {}
29:     for all <tid, slot> ∈ sharedReservations[tid] do
30:         objPtr ← sharedReservations[tid][slot]
31:         collectedReservations.insert(objPtr)
32:     ▷ free all objects not reserved                                          ◁
33:     for all objPtr ∈ myRetireList do
34:         if objPtr ∉ collectedReservations then
35:             free(objPtr)

36: procedure PINGALLTOPUBLISH( )
37:     for all othertid ≠ tid do
38:         pthread_kill(othertid, . . . )
39: ▷ signal handler                                                             ◁
40: procedure PUBLISHRESERVATIONS( )
41:     for $ihp = 0, \dots, MAX\_HP$ do
42:         sharedReservations[tid][ihp] ← localReservations[tid][ihp]
43:     publishCounter[tid] ← publishCounter[tid]+1

44: procedure COLLECTPUBLISHEDCOUNTERS( )
45:     for all tid do
46:         collectedPublishCounters[tid] ← publishCounter[tid]

47: procedure WAITFORALLPUBLISHED( )
48:     ▷ establish all threads have executed signal handler                     ◁
49:     for all tid do
50:         repeat
51:             collectedPublishingCounter[tid] ← publishCounter[tid]
52:         until collectedPublishingCounter[tid]+1 ≥
            publishCounter[tid]
```

In the RETIRE procedure (line 15), the *reclaimer* triggers the publishing of local reservations by calling PINGALLTOPUBLISH when the size of its `retireList` exceeds the threshold set with `reclaimFreq` (line 1). Within PINGALLTOPUBLISH (line 36), the *reclaimer* employs `pthread_kill()` to send pings to all threads. The threads that receive these pings execute a signal handler, named PUBLISHRESERVATIONS, (line 40), which writes all their local reservations to the shared array. Once all threads have completed PUBLISHRESERVATIONS, the reservations become visible to the *reclaimer*. After all threads' reservations are published, the *reclaimer* can then call RECLAIMHPFREEABLE (line 22), which similar to HP gathers up all the reservations and frees those that are not part of the collected reservations (line 28-line 35).

A key factor in ensuring safe reclamation (avoiding use-after-free) is determining the point at which we can confirm that all threads have completed their PUBLISHRESERVATIONS( ) calls, allowing the *reclaimer* to safely free its `retireList` once the PINGALLTOPUBLISH( ) function returns. To establish such a time, HazardPtrPOP uses `publishCounter` array (line 6) where each slot is a monotonically increasing counter assigned to a specific thread. Each thread in the system increments its slot in the `publishCounter` (line 43) after finishing publishing.

A *reclaimer* monitor each thread's `publishCounter` value before and after pinging all threads to assert that every thread has completed publishing, based on comparing the previously read `publishCounter` values with the reread values. Specifically, the *reclaimer* read each thread's `publishCounter` value into their thread-local `collectedPublishCounters` array using COLLECTPUBLISHEDCOUNTERS( ) at line 19. Then, it calls PINGALLTOPUBLISH( ) (line 20), followed by WAITFORALLTOPUBLISHED( ) (line 21), during which the reclaimer repeatedly reread each thread's `publishCounter` value and compare it with the previously recorded values, only exiting the loop when all threads have incremented their `publishCounter` value at least once since the *reclaimer* collected the `publishCounter` values (line 49 - line 52). (When multiple reclaimers send signals simultaneously, the signals are effectively coalesced, and a reader publishing reservations once is sufficient to satisfy all concurrent reclaimers interested in knowing about those reservations.)

4.1.2 Limitations of POP and signals. The POP mechanism assumes that, upon being signaled, all threads will complete executing their signal handlers within a bounded time. This assumption is crucial for ensuring that all threads publish their reservations within a finite number of steps, allowing the *reclaimer* to eventually exit its WAITFORALLTOPUBLISHED loop. In practice, signal delivery occurs in bounded time, as demonstrated by prior signal-based techniques [3, 4, 13, 34, 54] and verified by studies on the timeliness of signal delivery [57] on modern architectures. Note that signal delivery timings are expected to get an order of magnitude faster with user space IPIs [40].

The key advantage of using POSIX signals is that we can relax the traditional thread failure model in the asynchronous shared-memory setting. In theory, a thread can run arbitrarily slowly or halt altogether, and one cannot distinguish between a slow thread and a halted one. In practice, threads that appear to be delayed are either busy with other work, trapped in infinite loops, or descheduled. The operating system knows which threads are running, descheduled, or have terminated or become zombies (awaiting events to terminate).

The *pthread_kill* function used to signal (ping) threads returns an error code if a thread is a zombie, or terminated, allowing a reclaimer to ignore such threads. This leaves running threads and descheduled threads. Running threads will be interrupted by a signal, and quickly publish their reservations. As for descheduled threads, modern schedulers are (somewhat) fair, and ensure each thread is scheduled to run periodically, at which point a thread will publish its reservations.

The worst case scenario for POP is an *oversubscribed* system with many more threads than CPUs, as a reclaimer will need to wait for all threads to be scheduled to publish reservations. However, concurrent data structures tend to scale negatively as the number of threads grows beyond the number of CPUs, so this scenario is not our focus. That said, our experiments include oversubscription, and POP performs surprisingly well despite moderate oversubscription.

4.1.3 Correctness and progress.

ASSUMPTION 1. *Threads publish their reservations in a bounded time after being pinged.*

PROPERTY 2. *(Safety) HazardPtrPOP avoids use-after-free errors.*

In order to prove HazardPtrPOP is safe, we need to establish that any *reclaimer* will not free an object which other threads could subsequently access.

$Wlog.$, by the way of contradiction, let us assume, a thread $T1$ frees an object o at a time $t1$ which is subsequently accessed by another thread $T2$ at a later time $t2$ ($t1 < t2$). In order to access o, $T2$ must successfully protect it at an earlier time $t2'$, such that $t2' < t2$. Similarly, to free o, $T1$ retires, then pings all threads to publish their reservations, and then for a bounded time waits to ensue that all threads complete publishing their reservations at a time $t1'$, such that $t1' < t1$. Now, Two cases arise. First, $t1' < t2'$, i.e., $T2$ published all its reservations for $T1$ before it reserved o at $t2'$. In this case, since n was already retired by $T1$, $T2$ will fail validation while reserving o. Thus, $T2$ can not access o without successfully reserving (HP requirement that threads reserve before access). In the second case, $t2' < t1'$, i.e o was successfully reserved before $T1$ requested $T2$ to publish the reservations. Note, in this case, $T2$ would have published the reservation which $T1$ will scan and skip freeing n as it is guaranteed to find it in the reservation list of $T2$ (Assumption 1). Hence, HazardPtrPOP is immune to use-after-free errors.

PROPERTY 3. *(Liveness) HazardPtrPOP is robust.*

A thread accumulates r nodes in its retire list of which $N \times H$ nodes may be reserved, where N is the number of threads and H is the maximum number of reservations that N threads could hold at a given instance in time. This implies, at a given time, a thread could free at least $r - N \times H$ nodes and at most $N \times H$ nodes may not be freed. Since $N \times H$ is a constant, the amount of unreclaimed garbage per thread is always constant. Therefore, popHP is robust.

4.1.4 HazardEraPOP. POP can be applied similarly to Hazard Eras. Hazard Eras [50] has similar interface to Hazard Pointers, but unlike hazard pointers maintain a global monotonically increasing epoch variable. Each thread reserves and publishes the current value of this epoch when accessing a node, rather than reserving the node itself. Additionally, each node maintains its birth epoch and retire epoch, representing its lifespan during which it is reachable in a data structure. The key idea is that, before freeing a node, a thread compares the node's lifespan to the reserved epochs. If no thread has reserved an epoch that intersects with the node's birth and retire epochs, then no thread could hold a hazardous reference to the node, making it safe to free. To obtain HazardEraPOP, we change the READ to only reserve epochs locally, and then publish for reclaimers by writing the reservations to shared memory when a reclaimer pings. The full pseudocode of HazardEraPOP along with its description and proof of safety and robustness appear in [53].

4.1.5 Note on a useful property of HazardPtrPOP. HazardPtrPOP enables threads to privately track reservations using a lightweight READ and publish them on demand with a single fence when required by a *reclaimer*. In essence, even if threads are stuck in a long-running execution, one can ping the stalled thread to learn which object it might currently be accessing. This feature allows us to develop an efficient variant of hazard pointers that approaches the performance of EBR. Specifically, in common cases, threads follow a fast path similar to EBR algorithms and when threads suspect delays, the publish-on-ping mechanism is used to assist a thread who is unable to reclaim. In the next section, we introduce this variant of hazard pointers that incorporates EBR.

4.2 EpochPOP

4.2.1 The overview. In EpochPOP, threads operate in epochs, announcing their entry into and exit from the *quiescent state* using a global epoch variable, similar to the EBR algorithm. However, unlike EBR, threads also privately track local reservations during traversals, utilizing the lightweight reads of HazardPtrPOP (no fence overhead), as if they are simulatenously executing HazardPtrPOP. This dual-mode operation occurs without the need for special synchronization (or alternation) between the two modes.

Reclaimers, much like in EBR, periodically scan all announced epochs to free objects retired before the minimum announced epoch. In rare cases where thread delays prevent a *reclaimer* from freeing its retired objects—such as when the retire list size remains above a certain threshold even after reclaiming in EBR mode—the *reclaimer* employs publish-on-ping to force all threads to publish their current reservations. This action empties the retire list while skipping the reserved objects. This approach contrasts with techniques like neutralizing in NBR [54, 57] or ejecting in PEBR [34] and HP-BRCU [36], in which signals are used to force threads to restart from programmer-defined checkpoints.

We describe EpochPOP as simultaneously operating in dual modes, deliberately avoiding the term *fast-path slow-path* approach to avoid confusion with other techniques. Unlike fast-path slow-path techniques such as Qsense [8], EpochPOP does not require switching between modes (and sychronizing between them). Instead, threads can seamlessly operate in both modes simultaneously, with different threads concurrently reclaiming objects in each mode.

Algorithm 3 EpochPOP

```
 1: const int reclaimFreq, epochFreq, C
 2: atomic<int> epoch
 3: thread_local int tid, counter
 4: atomic<int> reservedEpoch[NTHREAD]
 5: list<T*> retireList [NTHREAD]
 6: T* localReservations [NTHREAD][MAX_HP]
 7: atomic<T*> sharedReservations [NTHREAD][MAX_HP]
 8: atomic<int> publishCounter [NTHREAD]
 9: int collectedPublishCounters [NTHREAD]

10: procedure STARTOP( )
11:     if 0 == ++counter % epochFreq then
12:         epoch.fetch_add(1)
13:     reservedEpoch[tid] ← epoch

14: procedure T* READ(atomic<T*> &ptrAddr, int slot)
15:     repeat
16:         T* readPtr ← *ptrAddr
17:         localReservations[tid][slot] ← readPtr
18:     until readPtr = *ptrAddr
19:     return readPtr

20: procedure RETIRE(T* ptr)
21:     myRetireList ← retireList[tid]
22:     myRetireList.append(ptr)
23:     ptr.retireEpoch ← epoch
24:     if 0 == myRetireList.size() % reclaimFreq then
25:         RECLAIMEPOCHFREEABLE(myRetireList)
26:         if myRetireList.size() ≥ C*reclaimFreq then
27:             COLLECTPUBLISHEDCOUNTERS( )
28:             PINGALLTOPUBLISH( )
29:             WAITFORALLPUBLISHED( )
30:             RECLAIMHPFREEABLE(myRetireList)

31: procedure RECLAIMEPOCHFREEABLE(myRetireList)
32:     minReservedEpoch ← reservedEpoch.min()
33:     for all objPtr ∈ myRetireList do
34:         if objPtr.retireEpoch < minReservedEpoch then
35:             free(objPtr)

36: procedure RECLAIMHPFREEABLE(myRetireList)
37:     ▷ Same as in Algorithm 2 ◁

38: procedure ENDOP( )
39:     reservedEpoch[tid] ← MAX
40:     CLEAR( )
```

4.2.2 Description of algorithm. Algorithm 3 describes EpochPOP building upon an EBR implementation. All line reference in this section refer to Algorithm 3. Each thread maintains a `retireList` to collect objects retired by it (line 5), a SWMR slot in `reservedEpoch` array to announce the current epoch it is executing in (line 4), and a monotonically increasing `epoch` variable (line 2). The `epoch` is incremented periodically using the value of `epochFreq` (line 1).

Similar to original EBR, a thread announces it is exiting the quiescent state by reserving the current epoch at an appropriate slot in `reservedEpoch` with a call to STARTOP( ). Additionally, to access new data structure objects within an operation, the thread uses READ( ), privately reserving them (without fencing), similar to HazardPtrPOP. Threads enter the *quiescent state* by announcing the latest global epoch by invoking ENDOP( ) and clearing the local reservations (line 40).

Reclaimers free nodes in their retire lists as they normally do in EBR (without considering threads' private reservations) unless a thread delay is suspected. During RETIRE calls, threads append retired objects to their retire lists by associating the current epoch as their `retireEpoch`. When the list reaches a threshold size, they invoke RECLAIMEPOCHFREEABLE (line 25). This procedure identifies the minimum epoch reserved by a thread from the `reservedEpoch` array and then frees the objects that were retired before that epoch.

If a *reclaimer* suspects a thread is delayed, during the RETIRE call the reclaimer invokes the robust reclamation process of HazardPtrPOP. That is, if after attempting EBR style reclamation (line 25), the reclaimer finds that its `retireList` is still too large (say, more than half of the `retireList` remains unreclaimed), it assumes that some thread reserving an older epoch has been delayed. The reclaimer therefore calls RECLAIMHPFREEABLE (line 30) to ping all threads, wait for reservations to be published, and free from its `retireList`.

Note that the private tracking of reservations is key in allowing reclaimers to continue to free objects in the presence of thread delays. A thread that is delayed by other work will be interrupted by the reclaimer's signal, and will publish its reservations, allowing the reclaimer to precisely determine which objects the delayed thread will potentially access. This allows the *reclaimer* to safely free its retire list, skipping only a bounded set of reserved objects.

4.2.3 *Correctness and progress.*

PROPERTY 4. *(Safety) EpochPOP avoids use-after-free errors.*

EpochPOP mostly runs classic EBR synchronization between *readers* and *reclaimers*. In scenarios where no delayed threads are detected, *reclaimers* only free objects whose retire timestamp indicates that they were retired before the oldest announced timestamp across all threads. The success of the aforementioned condition indicates that all threads have gone quiescent at least once since the object (which a *reclaimer* wishes to free) was retired. Consequently, no thread could hold a reference to this object. In the event that a delayed thread is detected, a *reclaimer* (1) signals all threads to timely publish the reservations they were maintaining all along (Assumption 1), (2) in a bounded waiting loop waits to establish that all reservation were published before it proceeds to free its retire list, (3) scan the reservations and frees only the objects that are not reserved (and thus cannot be accessed by any thread).

PROPERTY 5. *(Liveness) EpochPOP is robust.*

This is ensured by the ability of the algorithm to maintain reservations while traversing the data structures and detect delayed threads. This allows threads to ensure continuous reclamation of objects in its retire list, only skipping a bounded set of reserved objects across all threads.

4.2.4 Ease of Use and Applicability. The interface of the POP algorithms, including HazardPtrPOP, HazardEraPOP, and EpochPOP, is the same as that of hazard pointers. Additionally, these POP algorithms do not impose any extra usability limitations; thus, they are compatible with the same data structures as hazard pointers.

5 EXPERIMENTAL EVALUATION

We use the publicly available C++ benchmark from NBR(+) [54, 57], which includes several safe memory reclamation algorithms: Hazard Pointers (**HP**) [41], Hazard Eras (**HE**) [50], **NBR+** [54, 57], **IBR** and **EBR** (RCU-style as in IBR) [60]. We extended the benchmark by implementing an optimized Linux `sys_membarrier`-based version of HP (**HPAsym**, similar to Folly's HP implementation) and adding our three algorithms: **HazardPtrPOP**, **HazardEraPOP**, and **EpochPOP**. All algorithms were integrated with five data structures present in the benchmark: Brown's (a,b)-tree (**ABT**) [12], the external binary search tree by David, Guerraoui, and Trigonakis (**DGT**) [19], the Harris-Michael list (**HML**) [41], a lazy list (**LL**) [30], and a hashtable (**HMHT**) based on HML. These data structures account for several contention levels and memory access patterns. We also ran data structures without reclamation, which is shown as **NR** in the plots for a rough baseline.

The experiments for results presented in this paper were conducted on an Intel CascadeLake server - 144 threads, 4 NUMA nodes, 18 cores per node (2-way hyperthreaded), 99MiB L3 cache, 2.2 GHz frequency, 188GB RAM. Additional experiments are provided in the appendix of the extended version of the paper [53], where we also compare with the recent Crystalline [49] algorithm, showing that POP algorithms outperform it as well. We also tested our benchmark on a 192 thread Intel Skylake SP, and obtained similar results.

5.0.1 Experimental Setting. Our benchmark, compiled with C++14 and `-O3` optimization, ran on Ubuntu 20.04 (kernel 5.8.0-55) using `numactl -interleave=all` and the mimalloc allocator. We used mimalloc (not jemalloc[25]), as suggested by Brown et al. [35], to avoid negative interaction between deferred memory reclamation and jemalloc, affecting scalability. Mimalloc[37], with multilevel sharding for free lists, resolves this issue. Thus, we used mimalloc in our benchmark. Unless otherwise mentioned, we show results for DGT with a maximum size of 200K nodes, ABT with 20M size, lists (LL and HML) with 2K size and HMHT with 6M size and load factor of 6. We used 24K as max size of the retire list after which a reclamation event is triggered for all reclamation algorithms in all our experiments unless mentioned otherwise. [a]

5.0.2 Experimental Methodology. In each trial of our experiment, threads prefill the data structure up to half of the maximum fixed key range (size). Subsequently, they enter an execution phase where, they perform data structure operations for 5 seconds. During execution, repeatedly a randomly chosen insert, delete, or contains operation is invoked with a key randomly selected from a given key range of the data structure. We report throughput (millions of operations per second) and memory consumption (maximum garbage collected per thread) for read-heavy workloads with 90% contains, 5% inserts and 5% deletes, as well as update-heavy workloads with 50% inserts and 50% deletes for thread in the range of 1 to 288. The system is oversubscribed after 144 threads. The variance in the trials was below 5%.

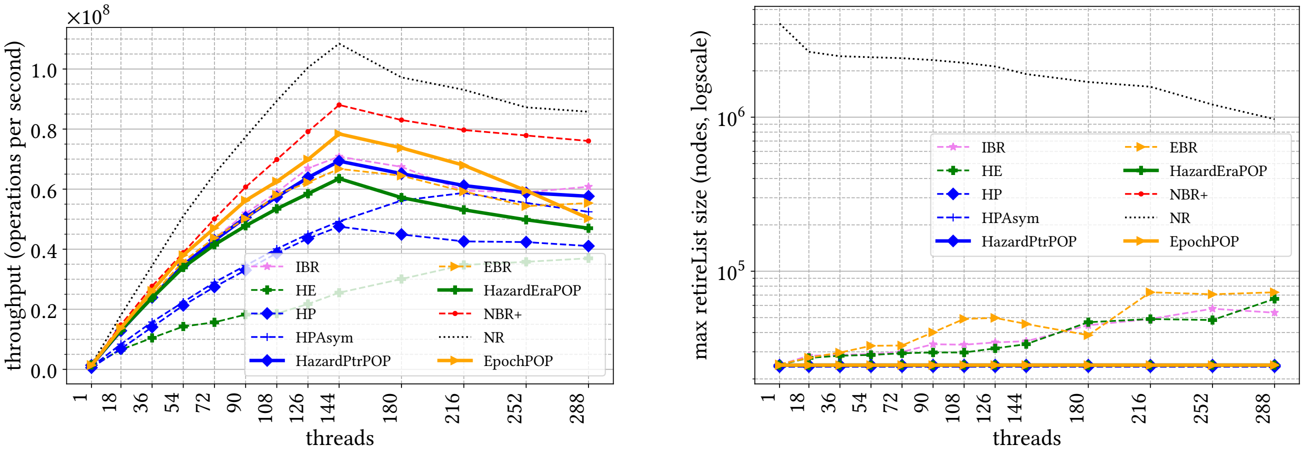
5.1 Results

In all the data structures across read-heavy and update-heavy workloads, publish-on-ping algorithms exhibit low memory footprint, and the impact of reduction in overhead due to the elimination of eager publishing of reservations with every pointer access translates to improvement in overall throughput and scalability.

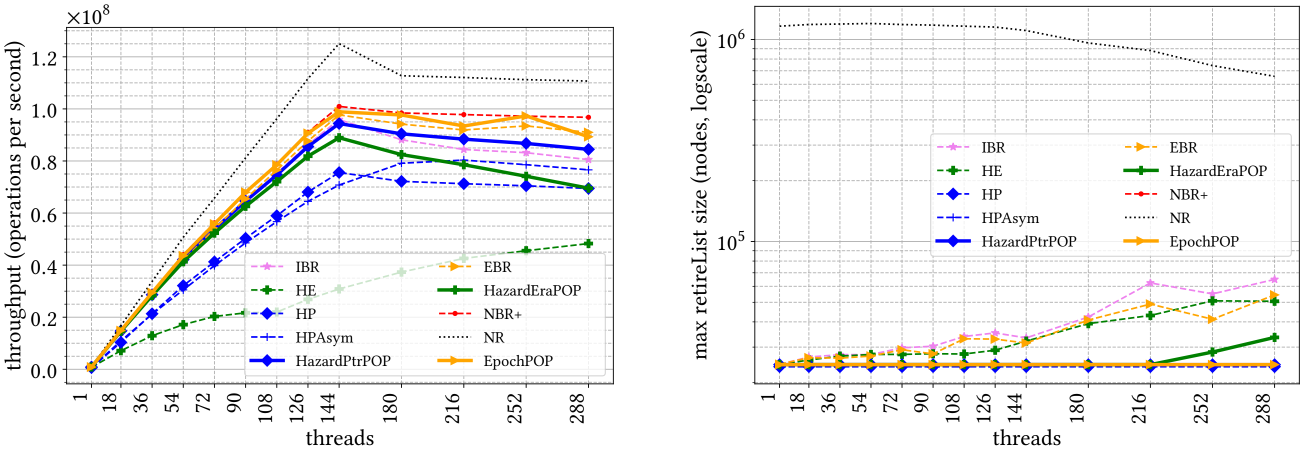
5.1.1 Update-heavy and Read-heavy Workload: For update-heavy workloads shown in Figure 1 and Figure 2. publish-on-ping (POP) algorithms consistently perform better or are similar and exhibit a lower memory footprint compared to the original algorithms on which they are based. Specifically, HazardPtPOP is on average up

---

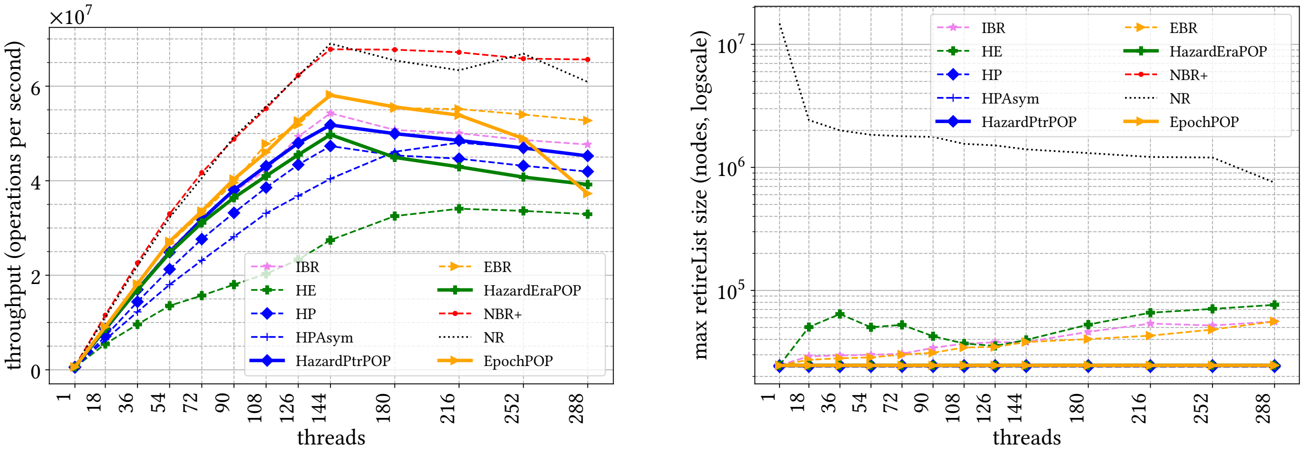
[a] The NBR [54] authors used retire list of 32K to ensure fair comparison and avoid excessive signal overhead.



(a) Ext. Binary Search Tree (DGT). Size 200K

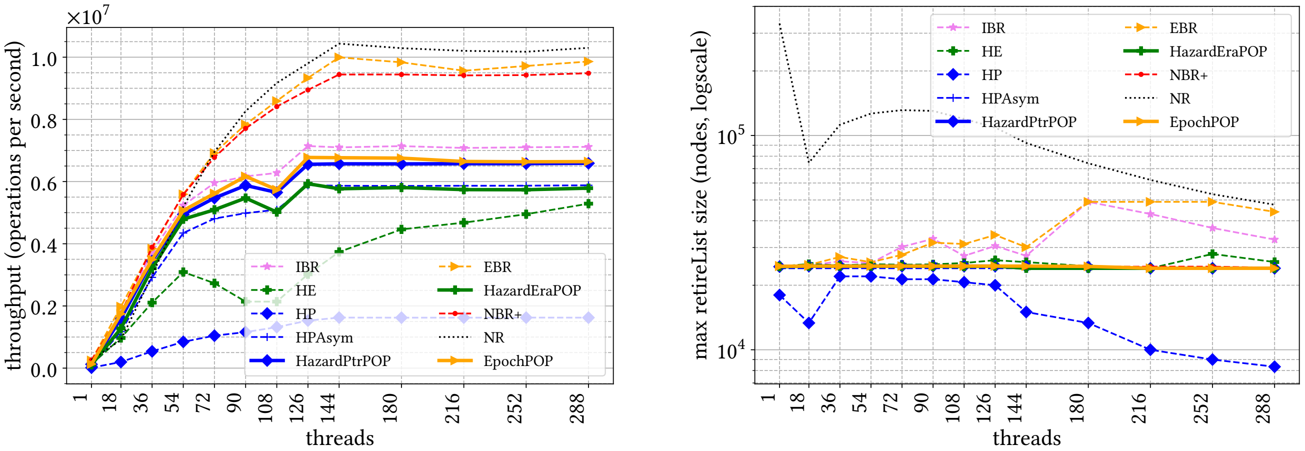


(b) Hash Table with Harris Michael List (HMHT). Size 6M.

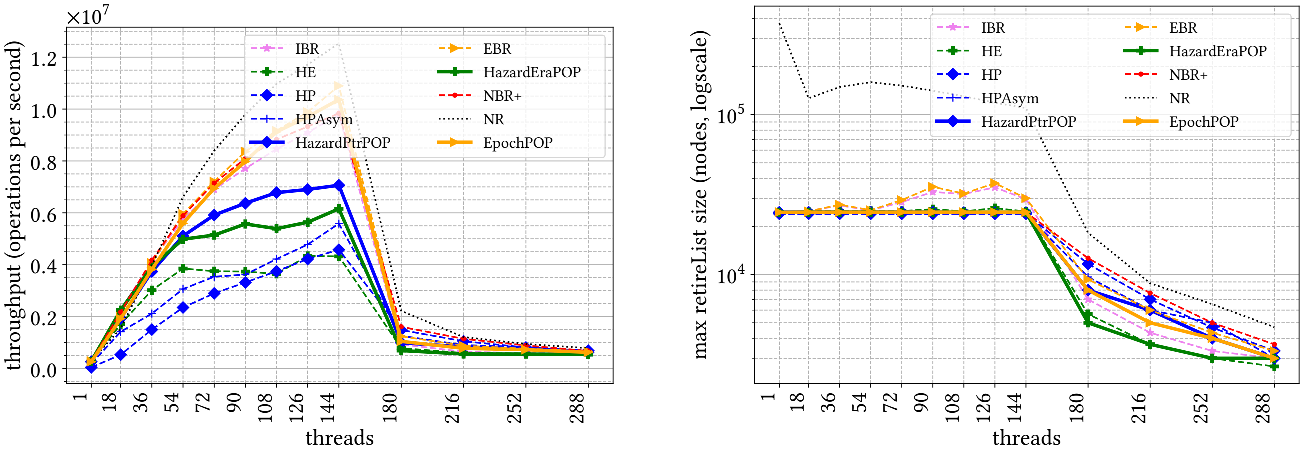


(c) (a,b) Tree (ABT). Size 20M.

Figure 1: Workload: Update-heavy. Throughput and memory consumption across varying threads for different data structures.



(a) Harris-Michael List (HML). Size 2K



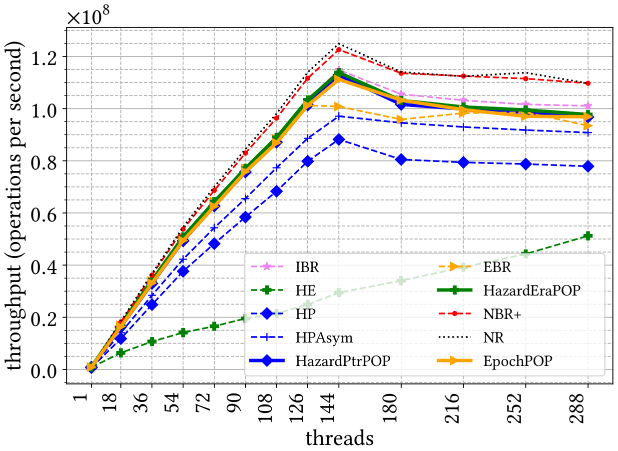
(b) Lazy List(LL). Size 2K.

Figure 2: Workload: Update-heavy. Throughput and memory consumption across varying threads for different data structures.

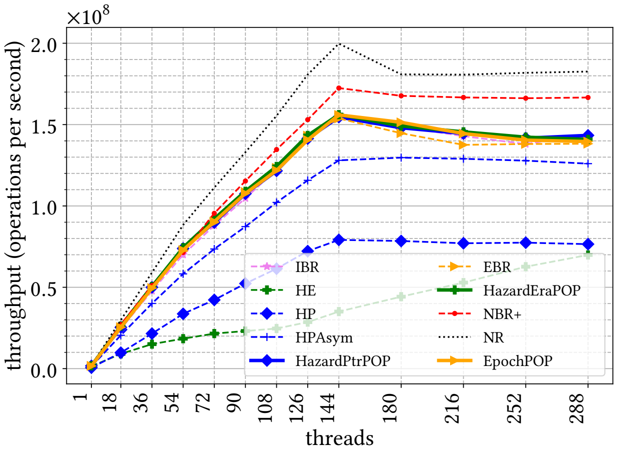
to 70% faster than HP and up to 20% faster than HAasym. HazardEraPOP is on average up to 2x faster than HE. And EpochPOP is similar in performance to EBR and IBR while maintaining strictly lower memory consumption. Note that at high thread counts, especially at 288 threads (oversubscribed), EpochPOP is slower in trees (DGT and ABT) but is comparable for in the hash table (HMHT). This predominantly is due to excessive signalling required to reclaim garbage, which is a reasonable trade-off to maintain a strictly lower memory footprint.

In read-heavy workloads all POP algorithms are similar or, in some cases especially, at oversubscription marginally better than EBR and IBR, as shown in Figure 3. Individually, here again, HazardEraPOP is on average up to 3× faster than HE, HazardEpochPOP is marginally (up to 15%) better than HPAsym. *Plots for additional data structures under read-heavy workloads can be found in the extended version [53].*

NBR+ demonstrates exceptional speed across various workloads and data structures due to the absence of read overhead and its minimal memory footprint, which improves cache performance. Although the POP algorithms eliminate the significant overhead of memory fences during reads, they still require the local reservation of pointers or eras, which are then published upon signaling for each read. This can be one reason for the slower performance of the POP algorithms compared to NBR+. However, as we will show in Section 5.1.2, NBR+ incurs a high overhead, when retire list size is set very low (2K in our experiments), for long-running read operations (first shown by Kim, Jung and Kang in HP-BRCU [36]), leading to frequent restarts from entry points in data structures. This causes a substantial drop in read throughput. In contrast, our POP variants perform significantly faster in these scenarios, as they do not require threads to restart. Another notable advantage of

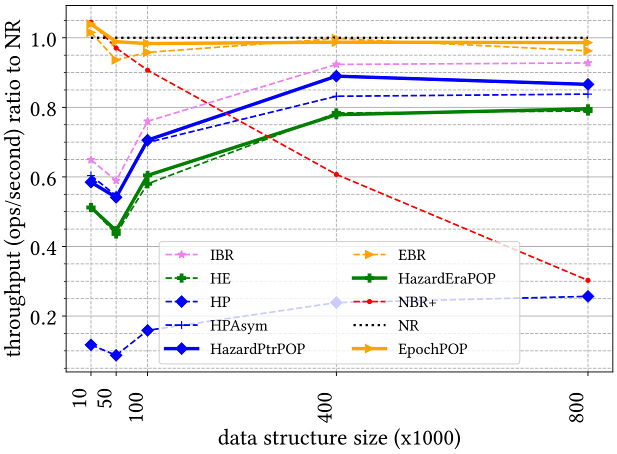


(a) ABT. Size 20M

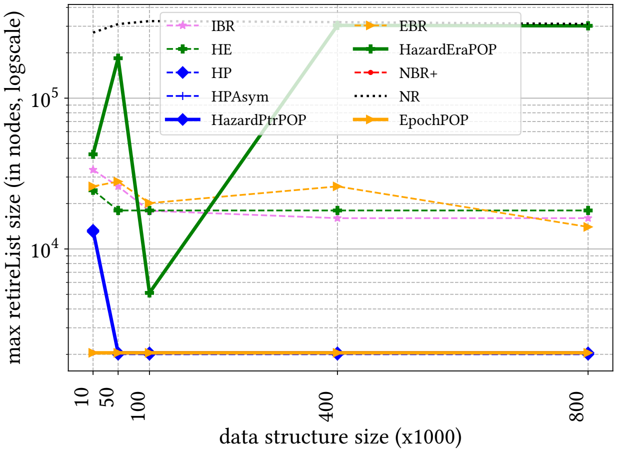


(b) DGT. Size 200K

Figure 3: Workload: Read-heavy. Throughput for ABT and DGT.



(a) Throughput.



(b) Memory Consumption.

Figure 4: Workload: Long Running Reads. HML list.

HazardPtrPOP over NBR+ is its compatibility with some data structures [11, 41] that cannot be paired with NBR+ but are compatible with HP [57], and hence with HazardPtrPOP. For these data structures, HazardPtrPOP can provide better performance than existing pointer reservation based algorithms.

5.1.2 *Long Running Reads Workload:*

To demonstrate the advantage of POP algorithms over NBR+ for long-running reads, we conducted an experiment similar to the one in [36] on a 192-thread Intel machine. We used the HML list with sizes ranging from 10K to 800K nodes. In this setup, 96 threads exclusively performed search operations, while another 96 threads executed update operations near the head of the list. We set the retire list size of all reclamation algorithms to 2K to frequently trigger reclamation events. This setup ensures that NBR+ frequently sends signals to restart the long-running read threads, resulting in a very low probability of completing reads.

Figure 4 presents the read throughput ratio (read throughput of other reclamation algorithms compared to NR) and memory consumption behavior for all algorithms in this experiment. The POP algorithms maintain high read throughput since they do not require read threads to restart when a thread reclaims memory, while also maintaining low memory consumption. In contrast, NBR+ slows down due to excessive restarts induced by reclaimers.

Subsequent techniques like HP-BRCU [36] and VBR [51] reduce the restart overhead for readers by introducing intermediate checkpoints, allowing threads to resume from the last checkpoint instead of the data structure's entry point during long-running searches. However, HP-BRCU is implemented in Rust, and VBR requires a type-preserving memory allocator, making direct comparison difficult. Qualitatively, POP variants are superior as they avoid restarts entirely, while HP-BRCU and VBR still incur overhead, depending on checkpoint frequency.

6 RELATED WORK

Research in solutions to the problem of safe memory reclamation for non-blocking data structures gained popularity in early 2000s with the appearance of epoch based [26, 28, 29], pointer reservation based techniques [31, 32, 41] and reference counting [18, 22, 27, 58, 59]. Since then, multiple manual and automatic techniques [5, 7, 15–17, 51] have appeared with varying properties, such as performance, robustness, ease of use, and extent of applicability. Among these, some are designed with careful use of features in modern operating systems and architecture [2–4, 10, 24, 33, 34, 54–57] while others leverage earlier techniques [6, 46–48, 50, 60]. There are others which contribute to the improvement of earlier techniques [8, 13, 23, 44]. In this section, we exclusively focus on the deferred reservation-based techniques which aim to eliminate or reduce the overhead on the traversal path.

One approach in [45] assumes an alternative memory model called temporally bounded total store order (TBTSO) and applies it to HPs to guarantee that reserved pointers will be published to all threads within a bounded time, facilitating safe memory reclamation. In [8], Balmau et. al. employ context switches triggered by periodic scheduling of auxiliary processes per core to timely publish hazard pointers. This technique incurs overhead to periodically publish reservations, even when threads might not be reclaiming. Dice et. al. [23] advocate using the write-protection feature of memory pages that triggers a global barrier to facilitate timely publication of hazard pointers before threads reclaim. However, the issue with it is that it could block threads trying to reserve an object if a reclaimer stalls after write protecting the page the object resides on.

Hazard Eras [50] reserves epochs which reduces the frequency of memory fences to publish reservations. Although the frequency of memory fences is reduced, the overhead remains substantial as seen in our experiments. Another technique, Conditional Access [56] utilizes hardware-software co-design to eliminate explicit synchronization required between reclaimers and readers at the programming level by leveraging existing synchronization at the cache coherence level. However, this might not be available in hardware any time soon.

The question of determining which objects are safe to free compelled earlier techniques using signals, such as DEBRA+ [13] and NBR [54] to forcibly change control flow, compelling threads to restart, inducing overheads for long-running read operations such as in OLTP workloads [36]. However, EpochPOP *does not* need to alter the control flow of threads, due to its ability to track reservations privately and publish them on demand.

Kim, Jung and Kang proposed HP-RCU and HP-BRCU [36] that execute in a sequence of HP and RCU phases and amortize the per read traversal cost by HP-based checkpoints from which traversals have to restart in case signalled for reclamation. Compared to NBR [54], HP-BRCU turns the coarse-grained restart overhead into fine-grained with the use of intermediate checkpoints. But, like other checkpoint-based techniques (VBR[51]), it could be complex to use and requires programmers to identify regions in the code for safe application. Furthermore, installing checkpoints determining safe points in data structure code to enable restart may be difficult for arbitrary data structures [14]. POP algorithms on the other hand are simpler, are able to avoid per-node overhead without inducing signal-based rollbacks and retain the original programmability of hazard pointers.

7 CONCLUSION

We have proposed the publish-on-ping approach for reclamation, utilizing POSIX signals to expedite pointer based techniques like hazard pointers and hazard eras. It can be used easily as a replacement for hazard pointers–a technique set to be included in the C++26 standard library. Furthermore, we integrate epochs alongside the publish-on-ping variant of hazard pointers. The resulting algorithm, EpochPOP, performs similar to epoch-based reclamation, while providing bounded garbage. Overall, publish-on-ping algorithms retain ease of programming in comparison to prior signal-based approaches, like DEBRA+ and NBR, as the former do not necessitate a reclamation-triggered change in control flow in data structure operations. In the future, it would be worth exploring whether other synchronization problems exhibiting a safe memory reclamation-like pattern can leverage our approach.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) Collaborative Research and Development grant: 539431-19, the Canada Foundation for Innovation John R. Evans Leaders Fund (38512) with equal support from the Ontario Research Fund CFI Leaders Opportunity Fund, NSERC Discovery Program Grant: 2019-04227, NSERC Discovery Launch Grant: 2019-00048, and the University of Waterloo. The findings and opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. We also thank the anonymous reviewers for their thoughtful feedback.

REFERENCES

[1] 2024. CVE-2024-26602. Available from MITRE, CVE-ID CVE-2024-26602.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-26602

[2] Dan Alistarh, Patrick Eugster, Maurice Herlihy, Alexander Matveev, and Nir Shavit. 2014. Stacktrack: An automated transactional approach to concurrent memory reclamation. In *Proceedings of the Ninth European Conference on Computer Systems.* 1–14.

[3] 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.* 483–498.

[4] Dan Alistarh, William Leiserson, Alexander Matveev, and Nir Shavit. 2018. Threadscan: Automatic and scalable memory reclamation. *ACM Transactions on Parallel Computing (TOPC)* 4, 4 (2018), 1–18.

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

[6] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2021. Concurrent deferred reference counting with constant-time overhead. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 526–541.

[7] Daniel Anderson, Guy E Blelloch, and Yuanhao Wei. 2022. Turning manual concurrent memory reclamation into automatic reference counting. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 61–75.

[8] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and robust memory reclamation for concurrent data structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 349–359.

[9] Guy E Blelloch and Yuanhao Wei. 2020. Concurrent Reference Counting and Resource Management in Wait-free Constant Time. *arXiv preprint arXiv:2002.07053* (2020).

[10] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the anchor: lightweight memory management for non-blocking data structures. In *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. 33–42.

[11] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. *ACM Sigplan Notices* 45, 5 (2010), 257–268.

[12] Trevor Brown. 2017. Techniques for Constructing Efficient Lock-free Data Structures. *arXiv preprint arXiv:1712.05406* (2017).

[13] 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*. 261–270.

[14] Nachshon Cohen. 2016. *Memory Management: From Theory to Practice.* Ph. D. Dissertation. Technion - Israel Institute of Technology, Israel. https://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2016/PHD/PHD-2016-11

[15] Nachshon Cohen. 2018. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–24.

[16] Nachshon Cohen and Erez Petrank. 2015. Automatic memory reclamation for lock-free data structures. *ACM SIGPLAN Notices* 50, 10 (2015), 260–279.

[17] Nachshon Cohen and Erez Petrank. 2015. Efficient memory management for lock-free data structures with optimistic access. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*. 254–263.

[18] Andreia Correia, Pedro Ramalhete, and Pascal Felber. 2021. Orcgc: automatic lock-free memory reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 205–218.

[19] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

[20] Mathieu Desnoyers. 12 March, 2015. Alternative to signals/sys_membarrier() in liburcu. Retrieved online. https://lists.lttng.org/pipermail/lttng-dev/2015-March/024270.html [Accessed 15-08-2024].

[21] Mathieu Desnoyers. April, 2015. sys_membarrier(): system-wide memory barrier (x86) [LWN.net] — lwn.net. https://lwn.net/Articles/640239/. [Accessed 15-08-2024].

[22] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele Jr. 2001. Lock-free reference counting. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 190–199.

[23] 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*. 36–45.

[24] Aleksandar Dragojević, Maurice Herlihy, Yossi Lev, and Mark Moir. 2011. On the power of hardware transactional memory to simplify memory management. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. 99–108.

[25] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada.*

[26] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report. University of Cambridge, Computer Laboratory.

[27] Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. 2008. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems* 20, 8 (2008), 1173–1187.

[28] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*. Springer, 300–314.

[29] ThomasE Hart, PaulE McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270–1285.

[30] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. 2005. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems*. Springer, 3–16.

[31] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2005. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems (TOCS)* 23, 2 (2005), 146–196.

[32] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2002. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *International Symposium on Distributed Computing*. Springer, 339–353.

[33] 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*. 213–226.

[34] 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*. 314–328.

[35] Daewoo Kim, Trevor Brown, and Ajay Singh. 2024. Are Your Epochs Too Epic? Batch Free Can Be Harmful. arXiv:2401.11347 [cs.DC]

[36] 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*. 1–13.

[37] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. Mimalloc: Free list sharding in action. In *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17.* Springer, 244–265.

[38] Paul E McKenney. 2010. Memory barriers: a hardware view for software hackers. *Linux Technology Center, IBM Beaverton* (2010).

[39] Paul E McKenney and John D Slingwine. 1998. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, Vol. 509518.

[40] Sohil Mehta. September, 2021. User Interrupts – A faster way to signal. Retrieved online. https://lpc.events/event/11/contributions/985/attachments/756/1417/User_Interrupts_LPC_2021.pdf [Accessed 15-08-2024].

[41] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15, 6 (2004), 491–504.

[42] Maged M Michael, Michael Wong, Paul McKenney, Arthur O'Dwyer, and David Hollman. 2017. *Hazard Pointers: Safe Resource Reclamation for Optimistic Concurrency.* Technical Report. Technical Report P0233R3. C++ SG14 Working Group.

[43] Maged M Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, David S Hollman, JF Bastien, Hans Boehm, David Goldblatt, et al. 2017. Proposed Wording for Concurrent Data Structures: Hazard Pointer and Read-Copy-Update (RCU).

[44] Pedro Moreno and Ricardo Rocha. 2023. Releasing Memory with Optimistic Access: A Hybrid Approach to Memory Reclamation and Allocation in Lock-Free Programs. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*. 177–186.

[45] Adam Morrison and Yehuda Afek. 2015. Temporally bounding tso for fence-free asymmetric synchronization. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 45–58.

[46] Ruslan Nikolaev and Binoy Ravindran. 2019. Hyaline: fast and transparent lock-free memory reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 419–421.

[47] Ruslan Nikolaev and Binoy Ravindran. 2020. Universal wait-free memory reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 130–143.

[48] Ruslan Nikolaev and Binoy Ravindran. 2021. Brief announcement: Crystalline: Fast and memory efficient wait-free reclamation. (2021).

[49] Ruslan Nikolaev and Binoy Ravindran. 2024. A Family of Fast and Memory Efficient Lock-and Wait-Free Reclamation. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 2174–2198.

[50] Pedro Ramalhete and Andreia Correia. 2017. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 367–369.

[51] 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

[52] Gali Sheffi and Erez Petrank. 2023. The ERA Theorem for Safe Memory Reclamation. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 435–437.

[53] Ajay Singh and Trevor Brown. 2025. Publish on Ping: A Better Way to Publish Reservations in Memory Reclamation for Concurrent Data Structures. arXiv:2501.04250 [cs.DC] https://arxiv.org/abs/2501.04250

[54] 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*. 175–190.

[55] Ajay Singh, Trevor Brown, and Michael Spear. 2023. Efficient Hardware Primitives for Immediate Memory Reclamation in Optimistic Data Structures. *arXiv preprint arXiv:2302.12958* (2023).

[56] A. Singh, T. Brown, and M. Spear. 2023. Efficient Hardware Primitives for Immediate Memory Reclamation in Optimistic Data Structures. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE Computer Society, Los Alamitos, CA, USA, 112–122. https://doi.org/10.1109/IPDPS54959.2023.00021

[57] 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

[58] Håkan Sundell. 2005. Wait-free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 10–pp.

[59] John D Valois. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed*

computing. 214–222.

[60] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H Alan Beadle, and Michael L Scott. 2018. Interval-based memory reclamation. *ACM SIGPLAN Notices* 53, 1 (2018), 1–13.

A ARTIFACT DESCRIPTION

This section provides a step by step guide to run our artifact in a docker container.

The artifact can be found at the following links:

- zenodo (with docker image): https://doi.org/10.5281/zenodo.14219155.
- gitlab (repo without docker image): https://gitlab.com/aajayssingh/pop_setbench

If you prefer to use the artifact directly without using the docker container please refer to the accompanying README file in the source code.

The following instructions will help you load and run the provided Docker image within the artifact downloaded from Zenodo link. Once the docker container starts you can use the accompanying README file to compile and run the experiments in the benchmark.

Steps to load and run the provided Docker image:

Note: Sudo permission may be required to execute the following instructions.

(1) Install the latest version of Docker on your system. We tested the artifact with the Docker version 24.0.7, build 24.0.7-0ubuntu2 20.04.1. Instructions to install Docker may be found at https://docs.docker.com/engine/install/ubuntu/. Or you may refer to the "Installing Docker" section at the end of this README.
To check the version of docker on your machine use:

```
$ docker -v
```

(2) Download the artifact from Zenodo at URL: https://doi.org/10.5281/zenodo.14219155.

(3) Extract the downloaded folder and move to *pop_setbench/* directory using *cd* command.

(4) Find docker image named *pop_docker.tar.gz* in pop_setbench/ directory. And load the downloaded docker image with the following command:

```
$ sudo docker load -i pop_docker.tar.gz
```

(5) Verify that image was loaded:

```
$ sudo docker images
```

(6) Start a docker container from the loaded image:

```
$ sudo docker run --name pop -i -t \
--privileged pop_setbench /bin/bash
```

(7) Invoke *ls* to see several files/folders of the artifact: Dockerfile, README.md, common, ds, install.sh, lib, microbench, pop_experiments, tools.

Now, to compile and run the experiments you could follow the instructions in the README file.