Efficiently Reclaiming Memory in Concurrent Search Data Structures While Bounding Wasted Memory

Daniel Solomon
Tel Aviv University
Israel

Adam Morrison
Tel Aviv University
Israel

Abstract

Nonblocking data structures face a *safe memory reclamation* (SMR) problem. In these algorithms, a node removed from the data structure cannot be reclaimed (freed) immediately, as other threads may be about to access it. The goal of an SMR scheme is to minimize the number of removed nodes that cannot be reclaimed—called *wasted memory*—while imposing low run-time overhead. It is also desirable for an SMR scheme to be *self-contained* and not require specific OS features.

No existing self-contained SMR scheme can guarantee a predetermined bound on wasted memory without imposing significant run-time overhead. In this paper, we introduce *margin pointers* (MP), the first nonblocking, self-contained SMR scheme featuring both predetermined bounded wasted memory and low run-time overhead. MP targets search data structures, such as binary trees and skip lists, which are important SMR clients and also victims of its high overhead. MP's novelty lies in its protecting *logical* subsets of the data structure from being reclaimed, as opposed to previous work, which protects *physical* locations (explicit nodes).

CCS Concepts: • **Theory of computation** → **Shared memory algorithms**.

Keywords: safe memory reclamation, hazard pointers

1 Introduction

Nonblocking concurrent data structures face a *safe memory reclamation* (SMR) [22] problem. It is not safe to immediately reclaim (free) a node's memory when the node is removed from the data structure, as other threads may be holding local references to the node and cannot be blocked from accessing it (e.g., via locking). In manual memory management settings, such as C/C++ programs, the SMR problem is handled by an *SMR scheme*. Instead of freeing a removed node, it is *retired*, indicating that no other node points to it. The SMR scheme reclaims a retired node once no thread refers to it locally.

It is important for an SMR scheme to guarantee a *predetermined* bound on the number of retired but unreclaimed nodes, or *wasted memory*. By "predetermined bound" we mean a bound such as $O(\text{number of threads})$, which holds independently of thread scheduling. We refer to providing a predetermined bound as *bounding wasted memory*.

Hazard pointers (HP) [22] bound wasted memory, but do so by tracking every thread-local reference (i.e., each pointer dereference), which imposes high run-time overhead on the client application. Nevertheless, bounding wasted memory is important enough that HP is being adopted in the C++ standard library as well as by applications with high-availability and soft real-time requirements [23].

Various HP-like SMR schemes reduce run-time overhead by leveraging specific operating system (OS) primitives [1–3, 6, 12, 25]. These schemes do bound wasted memory, but relying on OS-specific mechanisms makes them non-*self-contained*—e.g, unsuitable for adoption in a standard library.

Self-contained SMR schemes trade off bounding wasted memory to reduce overhead. Epoch-based reclamation (EBR) [15, 20] only tracks which threads are performing a data structure operation when a node is retired and might therefore locally refer to the retired node. Once all such threads complete their operation, the node can be reclaimed. If a thread stops taking steps mid-operation, however, no retired node can be reclaimed. EBR thus does not even satisfy *robustness* [3, 12, 29], which is defined as not allowing unbounded wasted memory.

Interval-based reclamation (IBR) [29] and hazard eras (HE) [27] are robust. They use node lifetimes to infer whether a retired node might be referenced locally by a stalled thread. Like EBR, they divide time into epochs. Each thread announces the epoch that it last observed, making that epoch *active*. A node can be safely reclaimed if it was retired before every active epoch or created after every active epoch. Therefore, a stalled thread cannot prevent the reclamation of any node created (and retired) after its active epoch, so the amount of wasted memory cannot grow unboundedly.

We observe, however, that robustness still allows for *arbitrarily large* amounts of wasted memory. For instance, the data structure can grow arbitrarily large before a thread stalls mid-operation; if other threads subsequently empty the data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441582

structure, none of the removed nodes can be reclaimed by IBR or HE. We thus argue that *robustness alone is not a helpful SMR property*. If an SMR scheme is only robust, a real system (with finite memory) can still end up with almost all of its memory taken up by unreclaimable nodes as a result of a thread stalling mid-operation. An SMR scheme should thus guarantee *a predetermined bound* on wasted memory, thereby making the amount of wasted memory independent of thread behavior.

This work. In this paper, we design a *self-contained, low-overhead SMR scheme that guarantees bounded wasted memory*. We introduce *margin pointers* (MP), an SMR scheme targeted at *search data structures* [10], such as binary trees and skip lists, which are important SMR clients and suffer greatly from its overhead.

Margin pointers encode "announcements" that protect *logical* subsets of the data structure from being reclaimed, as opposed to previous work, which protects *physical* locations (explicit nodes) [4, 18, 22]. In MP, similarly to HP, threads maintain multiple "pointer" records. In HP, each record protects some node from reclamation. In MP, in contrast, each record protects the interval of keys within some *margin* (distance) from a key. Threads can therefore ensure that the key of any node accessed is contained in some MP-protected interval. Because an interval can protect multiple nodes, a thread can perform multiple accesses without updating any MP record—significantly reducing run-time overhead. At the same time, controlling the margin allows MP to bound the protected interval size and bound wasted memory.

MP supports HP's SMR interface and extends it with optional method calls, without which it falls back to HP. Consequently, MP can be seamlessly plugged into any client program that uses the HP interface. Client search data structures can then use MP as an optimization, by modifying their code to invoke MP's optional methods.

We apply and evaluate MP on several nonblocking data structures: a binary search tree [24], a linked list [21], and a skip list [15]. MP shows a symbiotic relationship with the client data structure: the faster the data structure, the less overhead MP imposes. Overall, MP's performance is comparable to IBR and HE on fast data structures, while also guaranteeing bounded wasted memory.

MP show that bounded wasted memory does not necessary imply high SMR run-time overhead in practice. Our results motivate further exploration of the connection between bounded wasted memory and performance, e.g., by theoretical characterization or obtaining low-overhead, bounded wasted memory SMR for more data structures.

2 Model & Problem Statement

We consider a multi-threaded data structure, in which T threads communicate through a shared memory using standard read, write and atomic read-modify-write primitives such as `compare-and-swap` (CAS). The data structure consists of memory blocks referred to as *nodes*. We assume an environment without automatic garbage collection, as is the case for C/C++ programs. In this setting, the programmer must explicitly allocate and reclaim nodes. The *safe memory reclamation* (SMR) problem is to guarantee that a node is reclaimed only after no thread can access it. To this end, an *SMR scheme* supports the programming model detailed below, by providing the interface shown in Listing 1.

Listing 1. SMR API.

```
Function alloc(uint size): Node*
Function retire(Node* ptr): void
Function start_op(): void
Function end_op(): void
Function read(Node** ptr_addr,
             uint refno): Node*
Function unprotect(Node*,
             uint refno): void
```

Every node is first allocated using *alloc*. (The SMR scheme may use more memory than requested for a node, to piggyback its own per-node bookkeeping data.) The node is then *linked* into the data structure by updating some pointer to point to it. A node is *removed* from the data structure when no other node points to it. Once a node is removed, only threads that hold a local reference to it (having read a pointer to it) can access the node. A removed node is *retired* and passed to the SMR scheme by invoking *retire*. We assume that only removed nodes are passed to *retire*, and that a node is not retired more than once. The SMR scheme buffers every retired node in an internal *retired list*, and reclaims the node's memory once no thread refers to it locally.

To track possible references, the SMR scheme provides the following interface: Upon starting and completing a data structure operation, a thread must call *start_op* and *end_op*, respectively. Before accessing a node, a thread must pass the node's address to *read*. MP and similar designs require explicitly identifying the local references with a *refno* parameter (which is ignored in other designs) and performing an explicit *unprotect* call when a local reference is dropped (see § 3). As is standard, we assume that threads do not hold local references to nodes across data structure operations.

The SMR problem is a major concern for nonblocking algorithms, and so we require an SMR scheme to be nonblocking as well: After sufficiently many execution steps, *some* invocation of an SMR interface call must complete [17]. In particular, no SMR call may block, waiting for a thread to release local references. Instead, if nodes cannot be reclaimed, the retired list will grow in size, possibly unboundedly.

3 Background & Related Work

Here, we survey and compare existing SMR techniques. The main axes of comparison are the run-time overhead an SMR scheme imposes on the client data structure, the amount of wasted memory it allows to accumulate, and programmer effort required to utilize the scheme in a data structure. A secondary consideration is the node size overhead (if any) imposed by the scheme. Table 1 summarizes the discussion in this section and the properties of MP (§ 4).

Scheme	Run-Time Overhead	Wasted Memory Bound?	Data Struct. Integration Effort	Per-Node Overhead (# Words)
HP [22]	High	Bounded	Per-reference	-
DTA [4]	Low	Robust†	Harder than HP‡	2
OA [9]	Low	Recycle only	≈ HP	-
AOA [8]	Low	Recycle only	Rewrite DS to normal. form	-
FA [7]	Low	Recycle only	Automatic	-
EBR [15]	Low	Unbounded	Per-operation	-
HE [27]	Low	Robust	≈ HP	2
IBR [29]	Low	Robust	Per-operation	3
MP	**Low–Med. (Search DS)** / **= HP (Other DS)**	**Bounded**	**HP + extra method calls for full benefit (see § 4.1)**	**3**

† Amount of memory consumed by *frozen* nodes can be arbitrarily large.
‡ A data structure-specific *freezing* process must be designed.

Table 1. Comparison of memory reclamation schemes.

3.1 Pointer-Based Reclamation (PBR)

A PBR scheme protects specific nodes from being reclaimed. Each thread maintains a set of *pointer protection variables* (PPVs), in which it must "announce" any node to which it holds a local reference. (In our model, the SMR scheme's *read* procedure (Listing 1) performs this announcement.) A thread may only access a node if one of its PPVs has protected that node continuously from a time at which the node was linked into the data structure. To establish this property, a thread wishing to dereference a pointer p to node n first writes n's address to one of its PPVs, and then verifies that p still points to n. Success of this validation establishes that n was linked to the data structure throughout these steps, and so n is successfully protected. If the validation fails, the protocol is repeated for the node n' to which p now points. For correctness, the write to the PPV must be visible to all threads before p is verified, which requires executing a costly memory fence operation after writing the PPV. As a result, PBR schemes have high run-time overhead.

PBR schemes also require explicit management of the PPV's lifetimes by the programmer. The programmer must invoke the scheme's *unprotect* call whenever a PPV no longer needs to protect its node. There are often subtle rules about overwriting one PPV with another PPV's value [22]. PBR can be complex to use with some algorithmic patterns (e.g., binary tree rotations) [29].

In the canonical PBR schemes, PPVs are called *hazard pointers* (HP) [22] or *guards* [18]. We describe HP (guards are similar). HP stores retired nodes in thread-local retired lists. Whenever the number of retired nodes in a thread's list exceeds some bound, the thread attempts to reclaim nodes. It scans every thread's hazard pointers, and reclaims any retired node not protected by some hazard pointer. The threshold for attempting reclamation can be tuned to balance wasted memory overhead vs. the cost of these scans. Critically, however, wasted memory has a predetermined bound: at most $O(HT)$ retired nodes cannot be reclaimed, where H is the number of HPs per thread.

Drop the Anchor (DTA) DTA [4] attempts to reduce HP's overhead. DTA's PPV is called an *anchor*. When the anchor points to a node, it protects any node reachable from that node in k node traversals, where k is a DTA parameter. The anchor thus needs to be updated once every k node traversals, significantly reducing overhead. Retired nodes are managed similarly to HP, but the reclamation process is different. DTA first tries a variant of EBR. If a thread is stalled mid-operation, blocking reclamation, DTA makes use of that thread's anchor. It *freezes* the k nodes protected by the anchor, making them immutable, and replaces them with new copies. The idea is that the stalled thread can now only access a frozen node, so any non-frozen node can be reclaimed.

DTA has two main downsides. First, freezing is data structure-specific, and must be designed from scratch for each new client data structure. Currently, only a list freezing technique is known [4]. Second, freezing may lead to an arbitrarily large number of nodes becoming unreclaimable if a thread stalls. The reason is that, to guarantee that any node the stalled thread might access gets frozen, DTA freezes every node it encounters while traversing from the stalled thread's anchor, until it has frozen k nodes that were linked to the data structure before the stalled thread's operation started.[1] There can be an arbitrarily large number of nodes frozen this way, e.g., any number of nodes can be inserted between the stalled thread's anchor and its current position.

Optimistic Access (OA) OA [9] reduces HP overhead by replacing HP updates of read nodes with post-read validation checks. Write accesses are HP-protected as usual. OA thus allows a thread to read from a retired node. To prevent accessing reclaimed memory, OA *never reclaims nodes*. Instead, retired nodes are recycled and returned by future allocation requests. As a result, OA cannot provide a bound on used memory. OA requires the data structure to be implemented in a specific *normalized form*. Subsequent work [7, 8] has made OA-style SMR automatically applicable to arbitrary nonblocking data structures.

3.2 Epoch-Based Reclamation (EBR)

EBR schemes [15, 20] divide time into *epochs*, implemented by incrementing a global counter, with protection performed at epoch granularity. Each thread announces the latest epoch it has observed. A retired node can be reclaimed once all threads have announced later epochs than the epoch the node was retired at. EBR only requires to be informed when a thread starts and stops an operation. EBR schemes differ in how epochs are maintained. For instance, Fraser's scheme [15] increments the global counter once every thread

[1] To this end, DTA adds an insertion time field to the nodes.

has either observed the current epoch or is not performing an operation.

EBR schemes are not robust: a thread stalled mid-operation blocks memory reclamation, as the epoch counter stops advancing.

3.3 Robust EBR Extensions

Two recent schemes extend EBR ideas and obtain robustness by using node lifetimes to infer if a retired node might be accessible to a stalled thread. Hazard eras (HE) [27] extend HP with a variant of EBR. HE maintains node fields describing the node's *birth-death interval*, which indicates the epochs in which the node was allocated and retired. In HE, an HP declares the value of the global epoch when the thread wants to access the node. A retired node can be reclaimed if no thread declares an epoch in the node's birth-death interval. The global epoch advances every constant number of deletion operations. HE significantly reduces HP run-time overhead, as multiple nodes can be accessed without needing to update the HP as long as the global epoch does not change. Being based on HP, HE requires the same amount of effort to deploy.

Interval-based reclamation (IBR) [29] uses similar ideas, but without maintaining PPVs. Each thread only maintains an epoch interval, such that the birth epoch of any node it accesses is within that interval. A retired node can be safely reclaimed if for every interval, its retirement time is before the start of the interval, or its creation time is after the interval's endpoint. In IBR, the global epoch advances every constant number of node allocations. IBR has less run-time overhead than HE, because in HE, a global epoch change can cause a thread to update all its PPVs, whereas in IBR, it can only cause a thread to update its epoch interval.

HE and IBR are robust: A stalled thread does not indefinitely block memory reclamation, since nodes allocated (and retired) after the thread stalls can be reclaimed. Nevertheless, the number of retired nodes that cannot be reclaimed may be *arbitrarily large*, since it can reach the size of the data structure when the thread stalled.

3.4 Other SMR Approaches

Reference Counting In a reference counting scheme [11], each node contains a reference count which is atomically incremented or decremented when a thread obtains or drops a reference to the object, respectively. Thus, an object with a reference count of zero can be safely and immediately reclaimed. Reference counting overhead is usually not acceptable because of the contention caused by the atomic updates on read-only, frequently-accessed nodes such as list and tree heads [16]. We do not consider such schemes further.

Non-self-contained SMR Schemes Multiple SMR schemes leverage OS features to reduce HP overhead or handle stalled threads in EBR. ThreadScan [1] and ForkScan [2] use *signals* to force all threads to report their local references. DEBRA+ [6] uses signals to restart a stalled thread, releasing all its local references. Several schemes [3, 12, 25] use OS system calls to force threads to perform a memory fence on demand, and thereby relieve the threads from having to perform a fence on each HP write. Our focus is on self-contained schemes.

4 Margin Pointers

This section presents the margin pointers (MP) SMR scheme. We describe the main ideas and terms (§ 4.1) and define the search data structures that MP targets (§ 4.2). We then walk through the code (§ 4.3), prove that MP bounds wasted memory (§ 4.4), and prove MP's correctness (§ 4.5). In § 5, we apply MP to nonblocking linked list, skip list, and binary search tree algorithms.

4.1 Main Idea

MP is a pointer-based design. MP's core idea is for the PBR protection variables (PPVs) to protect *logical subsets* of the nodes rather than specific physical nodes. Protecting a subset of the nodes removes the requirement to update a PPV on each pointer dereference (unlike HP), which reduces the memory fence overhead. Protecting a logical subset allows MP to unambiguously identify protected retired nodes, and thereby bound wasted memory.

MP Concepts To identify logical subsets of nodes, MP assumes the client data structure is a search data structure (such as a binary tree or skip list) that satisfies certain properties (detailed in § 4.2). One such property is that nodes have immutable keys, which allows MP to identify nodes by their keys. MP also requires the keys to be totally ordered, which allows MP to efficiently encode protected subsets as intervals of keys. Each MP protects an interval of keys within some *margin* (distance) from a key. The margin is a predetermined MP parameter. This scheme allows encoding an interval with a key: if the margin is M, then an MP whose value is k protects the key interval $[k - M/2, k + M/2]$.

Performing protection logically creates a "chicken and egg" problem. To determine if node n is protected by an MP and can thus be accessed, a thread must know n's key—but it must access n to read the key. To address this problem, we use the assumption that keys are immutable and represent a pointer p as a tuple $p = (n, k)$ where p points to n and $n.key = k$. This representation allows a thread to determine a node's key given only a pointer that points to the node.

Making MP Practical Realizing the above concepts for general keys raises several challenges. First, in some key domains (e.g., reals) an interval $[k - M/2, k + M/2]$ contains infinitely many keys. This means that a single MP could block an unbounded number of nodes from reclamation. Second, it may not be simple and/or efficient to compute a

distance between keys. Finally, it is not practical to encode a pointer together with a large key, as, e.g., that prevents atomically updating a pointer (and its associated key) with a single memory write.

MP solves the above problems by *intelligently mapping keys to fixed-width integers* (e.g., 32-bit). Specifically, MP associates an *index* with each node, such that $n_1.key \leq n_2.key \iff n_1.index \leq n_2.index$ for any nodes n_1, n_2. (Below, we address the issue of index collisions between different nodes.) MP protection is then performed on indices, i.e., an MP with an index i protects the nodes whose indices are in the interval $[i - M/2, i + M/2]$. Index-based protection makes protection intervals bounded in size and distances trivial to compute. It also allows MP to efficiently encode pointers as pairs of node and index, as discussed in § 4.3.1.

MP Index Creation Crucially, MP does not use a static key-to-index mapping. MP creates the mapping at run-time, based on the *data structure dynamics*, in a way that strives to make the indices of nodes that are physically close (i.e., reachable by a few pointer dereferences) also logically close (i.e., with indices that are close), as described next.

In search data structures, a traversal searching for a key implicitly maintains an open interval of keys, each of which must either be reachable to the traversal (by following pointers) or is not present in the data structure. This interval also always contains the traversal's target key. The interval shrinks each time the traversal performs a key comparison and navigates through a node, until finally the target key is found or its absence is established. For instance, when navigating left/right in a binary search tree, the right/left end of the interval is updated. Data structure insertions perform a traversal and then, if the target key was not found, link a new node to the node where the traversal ended.

MP leverages this behavior to determine the new node's index, so that it is close to the index of its parent. MP exports *update_lower_bound* and *update_upper_bound* methods to allow a traversal to update MP when the lower and upper endpoints of the search interval are adjusted. If an insertion ends with its search range being (k_1, k_2) with indices (i_1, i_2), respectively, then MP assigns the newly linked node an index of $\frac{i_1 + i_2}{2}$. (Other policies are possible; we leave exploring them to future work.) The end result is that *indices approximate physical proximity in the data structure*. Thus, protecting a node's index with an MP is likely to protect the next few nodes on the traversal, and thereby reduce the number of MP updates (and corresponding memory fence overhead).

Index Collisions The main challenge created by mapping a (possibly unbounded) key domain to fixed-width integers is the possibility of multiple (possibly an unbounded number of) linked nodes obtaining the same index. In such cases, a single MP may again block reclamation of an unbounded number of nodes. MP handles such occurrences by falling back to a variant of HP and HE (§ 4.3.2). Crucially, due to the specific way MP uses EBR, a stalled thread can only block a *constant* number of retired nodes from reclamation, so MP's bound on wasted memory is not compromised (§ 4.3).

Bound on Wasted Memory Overall, intuitively, MP should bound the number of protected retired nodes by $T \times M \times \#MP$, where M is the margin and $\#MP$ is the number of MPs per thread. Our solution to index collisions introduces additional factors, and the full bound is detailed in § 4.4.

4.2 Search Data Structures

While MP can be used with any client code that is compatible with HP (because MP uses the same SMR interface), MP's low overhead protection of logical data structure subsets targets search data structures, such as trees, lists, and skip lists [5, 13, 15, 19, 21, 24, 26].

Here, we characterize the properties that define a search data structures which can use MP's extended interface and thereby benefit from reduced runtime overhead.

Definition 4.1. A data structure D is a *search data structure* if it has the following properties:

1. D deterministically implements a set or key/value data type over a totally ordered domain of keys.
2. D's nodes have immutable keys.
3. D's *insert* operation allocates a new node n only after locating n's position in the data structure.
4. D's structure orders nodes according to their keys. For every node n in D and every pointer p of n, either (1) $n'.key > n.key$ for all nodes n' reachable from n by following p, or (2) $n'.key < n.key$ for all nodes n' reachable from n by following p.
5. D's *insert* operation can be modified to explicitly maintain a search interval I (as described in § 4.3), such that when a node for key k is created, $I = [pred, succ]$, where *pred* and *succ* are the predecessor and successor of k in the data structure.
6. When D's *insert* operation links a new node n with key k to some existing node n', then $n'.key$ is k's successor or k's predecessor.

4.3 Algorithm Walk-Through

We first describe the basic MP algorithm, and then add handling for index collisions (§ 4.3.2). Listing 2 shows the shared and private structures used by the algorithm. The main structure, *mp_slots*, is an array of *margin* pointers, which are used to announce indices for protection. Each thread has a constant number of margin pointers. Similarly to HP, each thread has a local retired list. MP has two tunable

Listing 2. MP structures.

```
const uint thread_cnt
const uint MPs_per_thread

unsigned_int mp_slots[thread_cnt]
                    [MPs_per_thread]
thread_local list retired

const uint max_index
const uint margin // protected range
```

parameters: *max_index*, which is the maximal value of any assigned index, and *margin*, which is the size of the safety margin. In addition (not shown), MP requires nodes to be augmented with an index field. If the client does not rely on node sizes (e.g., to copy nodes), MP can add this field opaquely by reserving extra space during node allocation. Otherwise, the client must be aware of the index field.

Node Protection The *read* method (Listing 3) attempts to dereference a pointer. It extracts both the target node's address and its index from the pointer. (We discuss the encoding in § 4.3.1.) It then checks if the node's index is already protected by the reference's MP, i.e., if the index lies in that MP's protection interval. If so, the node is returned and the caller may access it. If no protection exists, the node's index is written to the specified slot in the thread's *margin* array. This write is then made globally visible by issuing a memory fence. If subsequently the node remains pointed to by the pointer, it can be safely accessed. The reason is that (as with HP) the protection was announced while the node was linked to the data structure.

Listing 3. Basic read.

```
Function read(Node** addr, refno):
 while true do
  <node, idx> := *addr
  mp := mp_slots[tid][refno]
  // idx protected by this mp?
  if idx ∈ [mp − margin/2, mp + margin/2]:
   return node
  // no protection
  mp_slots[tid][refno] := idx
  memory_fence

  // ensure node remains linked
  if <node, idx> = *addr:
   return node
```

Node Unprotection An *unprotect()* call has no effect (it is a no-op), so that the protection interval of the MPs can remain valid and protect future-accessed nodes. We only clear MPs when an operation ends, as described below.

Retirement & Reclamation Listing 4 shows the code for node retirement and reclamation. Retirement simply places the node in the thread-local retirement list. Every *empty_freq* retirements, the thread calls *empty* to reclaim retired nodes. This procedure walks through the retirement list and reclaims the memory of any retired node that is not protected by some MP in the system. To check if a node is protected, it first reads *idx*, the node's index, from the node. It then checks, for every MP that protects an index interval I, whether $idx \in I$. If the node's index is not in any protected interval, it is reclaimed. (This process can be optimized, e.g., by building and querying an interval tree with the intervals of all MPs.)

Listing 4. Retirement & Reclamation.

```
// frequency of reclamation:
const int empty_freq
// public (invoked by client)
thread_local int counter
Function retire(Node* ptr): void
 retired.append(ptr)
 counter++
 if counter % empty_freq = 0:
  empty()

const uint NO_MARGIN = 0xffffffff
// private: invoked internally by MP
Function empty(): void
 for node in retired:
  conflict := false
  idx := node->index
  for tid in threads:
   for mp in mp_slots[tid]:
    if mp ≠ NO_MARGIN and
       idx ∈ [mp − margin/2, mp + margin/2]:
     conflict := true
  if not conflict:
   free(node)
```

Index Creation & Node Allocation MP requires the client data structure to allocate a new node (for an immutable key) only after locating the node's parent in the data structure. This requirement allows MP to maintain its key-to-index mapping by observing how the inserting operation's search for the key navigates the data structure. Specifically, when a node n for key k is allocated, MP needs to know k's predecessor and successor in the data structure, respectively, *pred* and *succ*. MP then assigns n with index $(pred.index + succ.index)/2$, and thereby maintains the property that $\forall k', k \le k' \iff n_k.index \le n_{k'}.index$.

To know a new node's predecessor and successor, MP leverages the property in most comparison-based data structures (e.g., skip lists, binary trees) that the data structure is navigated by maintaining a search interval which shrinks throughout the navigation, until it finally reduces to an interval whose endpoints are the target key's predecessor and successor. The navigation then stops and determines whether k is absent from the data structure and can thus be inserted.

MP requires modifying the client's *insert* operation to explicitly maintain the search interval's endpoints using *update_lower_bound* and *update_upper_bound* calls (Listing 5). (In § 5, we demonstrate such modifications for various nonblocking data structures.) Ultimately, when the inserted key's desired position is located, and the operation allocates a node for the key, MP knows the indices of the key's predecessor and successor, and can assign it an index. Initialization allocations of sentinel nodes (for keys such as $\pm\infty$) require the client to specify the index.

Listing 5. Node creation.

```
thread_local uint lower_bound, upper_bound
Function update_lower_bound(Node* n):
 lower_bound := n->index
Function update_upper_bound(Node* n):
 upper_bound := n->index

Function alloc(Key key): Node*
 node := malloc(sizeof(Node))
 node->key := key
 node->index := ⌊(lower_bound + upper_bound)/2⌋
 return node
```

End of Operation When a client data structure operation completes, the thread's MP array is cleared, to signal that the thread is no longer protecting any nodes. All the writes are made visible with one memory fence.

4.3.1 Index Extraction & Pointer Encoding

MP requires a pointer p to node n to be represented as $(n, n.index)$, so that a thread can know a node's index without accessing the node. MP thus defines a pointer object for clients to use. MP's pointers support the usual read, write, and CAS operations, so converting client code to use MP's pointers is straightforward. Listing 6 shows MP's pointer representation. MP packs both pointer and index into a single memory word.

It relies on the fact that in many architectures, not all the bits of a virtual address are used. For example, x86 and ARM reserve the top 16 bits. MP therefore packs the index into those reserved bits. (A similar approach is used in DTA [4].) The subtlety here is that indices might not fit in 16 bits. In such a case, MP stores the 16 most significant bits of an index in the pointer, losing some precision: Observing a pointer (n, i) now means that $n.index \in range(i) = [i \times 2^{16}, i \times 2^{17} - 1]$. Therefore, MP considers i to be protected by an MP only if the MP's interval contains $range(i)$. Moreover, the margin must be larger than 2^{16}, otherwise this condition can never be satisfied.

Listing 6. Pointer representations.

```
struct MP_CAS_Ptr {
 uint ptr = {
  next_index : 16 bits
  next : 48 bits
 }
 Function read_tuple():
// called only by MP code
  return <this.next, this.next_index>
 // called by client:
 Function deref(): Node*
  return (Node*) this.ptr.next
 Function write(MP_WAS_PTR new):
  this := new
 Function CAS(MP_CAS_PTR old,
              MP_CAS_PTR new):
  return CAS(&this, old, new)
}
```

4.3.2 Handling Index Collisions

Because MP maps a large key domain to fixed-width indices, there may be arbitrarily many keys mapping to the same index, which can cause a single MP to protect an unbounded number of retired nodes. We leverage the way indices are created to address this problem by carefully falling back to a variant of HP and HE. Appendix A provides the full code of the algorithm, which adds index collision handling to the code shown in the previous sections.

We first observe that the only way an index collision can initially occur is when there is no room for a new unique index between the indices of a new node's predecessor and successor. MP identifies this case and, instead of assigning a colliding index to the new node, uses a special *USE_HP* index value. The idea is that nodes stamped *USE_HP* must be protected with hazard pointers and not margin pointers, making the indices of all MP-protected nodes unique.

We add an array of hazard pointers to each thread, and when trying to dereference a pointer (n, USE_HP), the MP *read* function sets an HP to the target's address instead of an MP. The reclamation code similarly checks for HP or MP protection, depending on whether the retired node's index is *USE_HP* or not.

Falling back to HP solves the problem of index collisions for nodes in the data structure, but it is also possible for the indices of retired nodes to collide. Imagine the same key being repeatedly inserted and then deleted from the same position in the data structure. Each such deleted node will have the same index. To address this problem, we combine MP with ideas from HE. We add a global epoch counter that each thread increments every constant number F of node unlinks, and stamp each node with birth (creation) and retirement epochs. As in HE, each thread announces the global epoch it observed when starting an operation. The memory reclamation code checks if an MP protects a retired node *only if* the epoch of the MP's thread lies inside the retired node's birth-death interval. If a thread observes the epoch changing during its operation, it switches to using HPs during that operation, i.e., any subsequent *read* of a node that is not already protected by one of the thread's MPs will protect the access with an HP. The code in Appendix A omits this latter change for simplicity.

4.4 Predetermined Bound on Wasted Memory

We prove the following theorem:

Theorem 4.2. *MP has a predetermined wasted memory bound.*

Proof. Consider the number of retired nodes protected by some thread t. Each of t's HPs protects at most one node. If the thread is using only HP, we are done. Otherwise, each of t's MPs protects at most M (i.e., *margin*) indices. We need to show why MP's HE-like technique does not allow an arbitrary number of retired nodes to be protected by one of these indices. To see why, let e be t's announced epoch. We use the fact that each thread increments the global epoch once every F nodes it unlinks (see § 4.3.2). Thus, after FT node unlinks, the global epoch advances at least once.

Now, consider each index i protected by one of t's MPs. Retired nodes whose retirement epoch is $< e$ are not protected by that MP, as it could not have observed them. There may well be an unbounded number of retired nodes $n_1, n_2, \ldots$ whose retirement epoch is $\geq e$ and all have index i, but no two such nodes could exist together in the data structure, because MP-protected nodes in the data structure have unique indices. It follows that for every n_i, n_j, either n_i is unlinked before n_j is inserted or vice versa. Therefore, nodes $n_{FT+1}, \ldots$ all have a birth epoch $> e$. But such nodes are not considered protected by t's MP, since t's epoch is e.

Overall, the number of retired nodes protected by a thread t is at most $\#HP + \#MP \times M + \#MP \times M \times FT$. □

Theorem 4.2 shows the *existence* of a wasted memory bound. This guaranteed bound may be very large in practice, depending on the concrete values of T, F, and M used. We believe that future work can improve the bound. For example, if we advance the global epochs on every node unlink (as in HE), the per-thread bound improves to $\#HP + O(\#MP \times M)$. We leave exploring the related trade-offs to future work.

4.5 Safety & Progress Proofs

An SMR scheme is considered *reclamation safe* if it is guaranteed that a node cannot be accessed after it has been reclaimed.

Theorem 4.3. *The MP scheme is reclamation safe.*

Proof. We prove by contradiction. Assume that MP is not reclamation safe. Hence, there exists a node n with index i,

which was reclaimed (freed) at time t_r by thread j and thread k accessed node n at time $t_a > t_r$. n's lifetime is $[t_b, t_{r'}]$ where t_b and $t_{r'}$ are the birth and retirement times of n, respectively, which satisfy: $t_b < t_{r'} < t_r < t_a$.

Since thread k accessed n at time t_a, there must exist a time $t_p \in (t_b, t_{r'})$ in which thread k holds a pointer to n, otherwise it cannot access it (as before t_b the node n does not exist and at $t_{r'}$ the node was disconnected from the data structure). Thread k can hold a reference to node n only by using the SMR *read* method, which will place a protection on node n for thread k whether implicitly by protecting its index (if not already covered) using a margin pointer or explicitly by protecting its address (if not already protected) using a hazard pointer. This means that for any time t between $[t_p, t_a]$, node n was protected by thread k. At time t_r, thread j reclaims node n and it can only do so by running the internal *empty* method that reclaims nodes retired by j if no other thread protects them. Notice that $t_r \in [t_p, t_a]$, and so at time t_r, either n is protected by one of thread k's HPs or the protection interval of one of thread k's MPs contains i. Thus, the internal *empty* method does not reclaim n—a contradiction. □

Theorem 4.4. *The MP scheme is nonblocking.*

Proof. All methods but *read* are wait-free. As can be seen in Listing 10, *read* may loop forever only if the protected node pointer keeps changing. This implies that some other thread is making progress. □

5 Data Structure Integration

Here, we present the process of the using MP in a search data structure. We first detail the high-level steps that need to be taken (§ 5.1). We then demonstrate the methodology on nonblocking lists and skip lists (§ 5.2) and on the Natarajan-Mittal [24] nonblocking binary search tree (§ 5.3). In the full version of this paper, we also describe how MP naturally applies to tree rotations [28, § 4.4.5] and analyze the number of index collisions in these data structures both theoretically and empirically [28, § 4.6].

5.1 Methodology

In general, applying MP to a nonblocking search data structure (that satisfies Definition 4.1) involves the following:

1. Adapt the client algorithm to HP [22] by using the SMR interface.
2. Replace pointers with MP's pointer objects, and pointer accesses with the corresponding MP interface calls.

At this point, the algorithm can be used safely (as MP will fall back to HP), but to benefit from MP's reduced overhead, the following steps need to be taken as well.

3. Choose appropriate indices for sentinel nodes (if any), according to their location in the totally ordered key space. For instance, a maximum (minimum) sentinel should be assigned the maximal (minimal) index.
4. Modify insert operations to invoke *update_lower_bound* and *update_upper_bound* to update MP about the search interval while searching for the inserted key's location.
5. Node allocations should be performed using the SMR *alloc* call, after the inserting operation has determined the key's location.

5.2 Lists & Skip Lists

We first apply MP to a variant of Michael's nonblocking linked list [21] and then generalize to skip lists [15].

Linked Lists The list maintains keys in sorted order, with head and tail sentinel nodes, holding keys $-\infty$ and ∞, respectively. (The tail sentinel is not present in Michael's original version, but is trivial to add.) The list implements the standard set interface. The algorithm has an internal *seek* procedure, which is used by all operations to search for a key. We assume 32-bit indices, and so assign the head sentinel index 0 and the tail sentinel index $max_index = 2^{32} - 2$. The index $2^{32} - 1$ stands for *USE_HP*.

Applying HPs to the list is standard [21]. The additional changes required to benefit from MP are mostly trivial (e.g., changing pointer fields and accesses to use the MP pointer objects). The only non-trivial change is the maintenance of the search interval during insertion, which we add to the *seek* procedure (Listing 7, which omits HP handling). *Seek* traverses the list from its head, searching for some key k. It maintains *prev* and *curr* pointers. The search terminates upon finding a node with key $k' \geq k$, in which case *seek* returns the tuple $(prev, curr)$ to the calling operation. If k is not in the list, this pair forms k's predecessor and successor, allowing an *insert* operation to link a new node for k between *prev* and *curr*. Accordingly, *seek*'s search interval starts at $(-\infty, \infty)$, shrinks to (x, ∞) after traversing through a node with key x, and finally becomes $(prev.key, curr.key)$ when *seek* returns. Updating MP about the search interval is thus straightforward, requiring only two lines of code (bolded in Listing 7).

Listing 7. Linked list seek (MPs/HPs not shown).

```
Function seek(Key k): Node*, Node*
 try_again:
 prev := list->head
 curr := prev->next
 while curr ≠ NULL do
  next := curr->next
  if next & 1:
   if !CAS(prev, curr, next - 1):
    goto try_again
   retire(curr)
   curr := next - 1
  else:
   if curr.key < k:
    update_lower_bound(curr)
    prev := curr
    curr := next
   else:
    update_upper_bound(curr)
    return (prev, curr)
 return (prev, NULL)
```

MP requires no algorithmic changes to the list, e.g., to the list's two-step node deletion process. To delete key k in node n, the list algorithm first sets (with CAS) a reserved *deleted* bit in *n.next* (the least significant bit). Subsequently, n is physically removed from the list by splicing it from the list. A *seek* that observes a pointer with the *deleted* bit set splices

the marked node out of the list. This logic is not affected by MP, which is oblivious to how the client interprets node addresses (i.e., the meaning of the *deleted* bit).

Skip Lists In a skip list, each node is linked into multiple sorted lists (via an array of *next* pointers). The lists are ordered by containment. Every node is linked to the bottom-most list, and each higher level list contains a constant fraction of fewer nodes. A search for key k navigates the highest level (thereby "skipping" over multiple nodes) until finding the largest key $k' < k$ in that level. It then continues from that node in the next lower level, and so on, until either finding k or terminating in the bottom-most list.

We apply MP to Fraser's nonblocking skip list [15]. This algorithm, as other nonblocking skip lists, essentially maintains multiple Michael-style linked lists. Similarly to the list, a search operation requires two MPs—for *prev* and *curr* nodes—as it navigates the data structure. Insertions/removals need two MPs per level, since they can potentially modify every level. Maintaining the search interval of an operation works exactly as for the single list, i.e., we update MP about the search interval shrinking as each list level is searched.

5.3 Binary Search Tree (BST)

We consider the nonblocking BST of Natarajan and Mittal [24], which implements the standard set data type. The data structure is an unbalanced external (leaf-oriented) tree, in which the leaf nodes store the actual keys and the keys stored in internal nodes are used only to guide searches. This design choice simplifies deletions, which need handle only leaves. Deletions are implemented by marking edges. An edge has tail (points from) and head (points to) nodes. An edge can be marked as *tagged*, meaning the tail node is deleted, or *flagged*, meaning both head and tail nodes are deleted. These marks are implemented by stealing the two least significant bits from pointers. Once an edge is marked, it cannot be modified.

All tree operations are based on an internal *seek* procedure, which traverses the tree while maintaining a seek record (Listing 8). When *seek* returns, the seek record contains the leaf where the search stopped, its parent, an untagged successor of the leaf, and an untagged ancestor of the leaf. If the leaf is not undergoing deletion, the latter two are its parent and grandparent (i.e., the seek record effectively contains only 3 nodes). The idea is that the seek record can be used to link a new node to the tree. Refer to [24] for the full details. For our purpose, only the following details are important.

Listing 8. NM Tree API.

```
struct seek_record {
 Node* leaf,
 Node* parent,
 Node* successor,
 Node* ancestor,
}
Function seek(Key k): seek_record
```

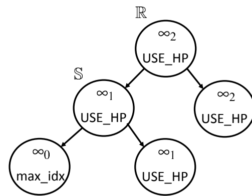


Figure 1. NM Tree initial state.

First, the initial state (Figure 1) contains 3 sentinel nodes, to ensure that a seek record's contents is always well defined. These sentinels have keys $\infty_0 < \infty_1 < \infty_2$, and any other key is considered as $< \infty_0$. Additionally, the initial state contains two routing nodes, $\mathbb{R}$ and $\mathbb{S}$, which can never be removed. For MP, we assign ∞_0 with index *max_index*. The remaining initial nodes are assigned index *USE_HP* for completeness. These nodes are never removed and the client need not use the SMR interface to access them.

The only other change required is to update MP about the search interval in the *seek* procedure. As in the list algorithm, this is straightforward: when the search interval is (x, y) and *seek* decides to navigate left (respectively, right) at a node with key k, the interval shrinks to (x, k) (respectively, (k, y)). Listing 9 shows the *seek* procedure with the modifications (two bolded lines).

Listing 9. NM Tree seek.

```
Function seek(Key k): seek_record
 seek_record sr = {
  .ancestor := R,
  .successor := S,
  .parent := S,
  .leaf := (S->left).ptr,
 }
 parent_node := (sr.parent)->left
 current_node := (sr.leaf)->left
 current := current_node.ptr
 while current ≠ NULL do:
  if !parent_node.tag:
   sr.ancestor := sr.parent
   sr.successor := sr.leaf
  sr.parent := sr.leaf
  sr.leaf := current
  parent_node := current_node
  if k < current->key:
   update_upper_bound(current)
   current_node := current->left
  else
   update_lower_bound(current)
   current_node := current->right
  current := current_node.ptr
 return sr
```

6 Performance Evaluation

We now evaluate the amount of run-time overhead and wasted memory of MP compared to prior SMR schemes.

Experimental Setup: We use a machine with two Intel Xeon E5-2699 v4 processors. Each processor has 22 cores, each with 2 hardware threads (HTs), for a total of 88 HTs in the machine. We run experiments with software threads pinned to HTs in a *spread* manner: threads are first pinned to the first HT of every core in a round robin manner across processors (e.g., a 4-thread configuration uses two cores per processor). Once all cores are used, we continue pinning to sibling HTs in the same round robin manner. All algorithms are implemented in C++ and compiled with g++ 5.3.0. We use the scalable `jemalloc` memory allocator [14].

SMR Schemes: We compare (1) **MP**: Listing 10, with indices in the range (0, 2^{32}); (2) **IBR**: the default tagged pointer interval-based reclamation [29]; (3) **HE**: hazard eras [27]; (4) **HP**: hazard pointers [22]; and (5) **DTA**: drop the anchor [4].

Parameters: For MP, we use a margin of size 2^{20} (we select this value based on a sensitivity study of margin sizes, presented below). We use an anchor of size 100 for DTA, following [4]. For common SMR parameters, we use the same values as [29]: each thread tries to reclaim nodes once every

30 *retire* calls and, for schemes with global epoch counters, each thread increments the counter once every $150T$ allocations, where T is the number of threads in the benchmark.

Client Algorithms: We evaluate the SMR schemes on three nonblocking data structures: Fraser's skip list [15], the Natarajan-Mittel BST [24], and Michael's linked list [21] (§ 5). We evaluate DTA only on the list, since it is not known how to apply DTA to the other client data structures.

Workloads: We use integer keys and values. Each test comprises a 5-second benchmark in which threads repeatedly invoke a random operation on a uniformly random key. We run with thread counts in the range [1, 100]. We purposefully exceed the number of available HTs to evaluate the effect of thread stalls caused by the resulting context switches.

We evaluate three workloads: (1) **read-dominated**, where *contains* is invoked with probability 90% and *insert* or *remove* each with an equal 5% probability; (2) **write-dominated**, where *insert* or *remove* are invoked with an equal 50% probability; and (3) **read-only**, where only *contains* is invoked. Using equal probabilities for *insert*/*remove* keeps the data structure size roughly constant throughout the experiment.

Unless stated otherwise, the data structure is initialized by inserting S uniformly random keys from a range of size $2S$. During the experiment, the operations choose keys from this range. We evaluate the skip list and BST with $S = 500\,K$ and $S = 50\,K$ (due to space constraints, the latter results appear in the full version [28]). For the list, we use $S = 5\,K$, as its linear complexity makes larger sizes unlikely in practice.

We report throughput (aggregated over all threads) and wasted memory (average number of retired but unreleased nodes). Each data point in the graphs is the average of 10 runs, whose variance is low.

Optimizations to IBR Framework: Our code is based on the framework of IBR [29]. However, our results for HP and HE are better than reported by IBR [29], as we performance-optimized the implementation of HP-based schemes in the framework. For instance, we tune HP, HE, and MP so that when an operation ends and clears all its hazard pointers, it executes a memory fence once (instead of after clearing each pointer). Similarly, we optimize the *empty* reclaiming procedure to first snapshot all hazard pointers in the system and then work with that snapshot, instead of accessing the original hazard pointers for each retired node. This optimization dramatically improves performance of HP-based schemes.

6.1 Results

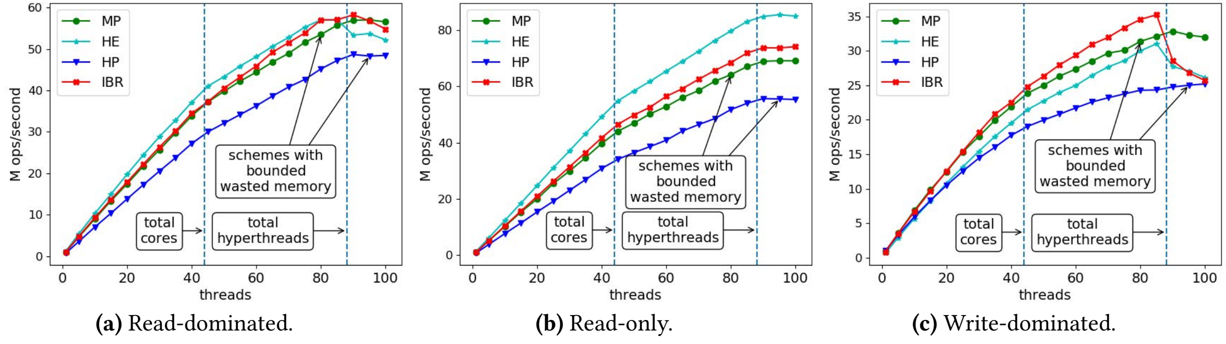
Throughput Figures 2 and 3 show the throughput of the 500 K node BST and skip list experiments, respectively. In non-read-only workloads, which are representative of practical usage, MP performs comparably to IBR and HE on the BST, and comparably to HE on the skip list. HP, in contrast, is $\approx 1.3\times$–$2\times$ slower than all of them. Here, MP does not "pay" for its guaranteed pre-determined wasted memory bound, unlike HP. Moreover, the throughput of both IBR and HE decreases when the workload exceeds the maximum number of HTs, which correlates with their wasted memory increasing (see below). At this point, MP often outperforms them (e.g., by $\approx 1.3\times$ in the write-dominated BST experiment).

In the read-only workloads, however, MP is slower than the best EBR-based scheme by $\approx 20\%$ and $\approx 30\%$ on the BST and skip list, respectively. The reason is that a read-only workload emphasizes SMR-related overheads, which do not dominate in a non-read-only workload in which operations experience cache misses due to update operations, perform atomic operations, etc. MP has a higher overhead than IBR because MP has per-dereference overhead whereas IBR's overhead is per-operation. While HE also has per-dereference overhead, in a read-only workload it is lower than MP's, because it consists of only reading the global epoch, whereas MP occasionally updates a margin pointer, which involves issuing a memory fence.

MP exhibits a symbiotic relationship with the data structure. On the BST and skip list, where operations take logarithmic time, MP's overhead is significantly lower than on the list, whose operations are linear. Figure 4 shows the throughput obtained on the 5 K node linked list. Here, IBR outperforms MP at high thread counts by $2\times$ to $3\times$ in non-read-only and read-only workloads, respectively. This symbiotic property means that MP should be useful in practice, where programmers prefer efficient search data structures. On the list benchmark, DTA outperforms both MP and HP but, unfortunately, it is a scheme that cannot currently be applied to other data structures.

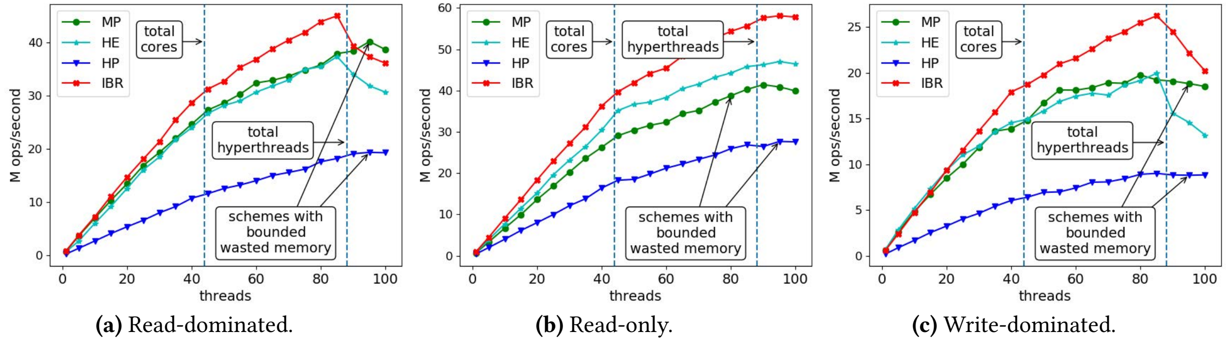
Memory Fences in MP vs. HP MP outperforms HP in all experiments. The reason is that MP executes fewer memory fences than HP, because setting a single MP protects the multiple nodes whose index falls within the margin, whereas HP must set a HP for every pointer dereference. Figure 5 quantifies this effect by comparing the average number of fences issued by MP and HP per traversed node in the read-only workload, on all the evaluated data structures. MP issues $\approx 2\times$ fewer memory fences than HP on all data structures.

Wasted Memory Figure 6 shows the average number of objects in a thread's retired list (i.e., retired but unreclaimed) at the start of each operation in the read-dominated workload, on all evaluated data structures. (Results for other workloads are similar [28, § 5].) Only MP and HP, which bound wasted memory, have close to no wasted memory on all data structures. HE and IBR have non-negligible memory overhead (up to orders of magnitudes worse than MP and HP), which gets worse as the thread count grows and with it, the number and duration of thread stalls due to context switches. DTA (which is evaluated only on the list) also has little wasted memory, because the behavior that can cause its



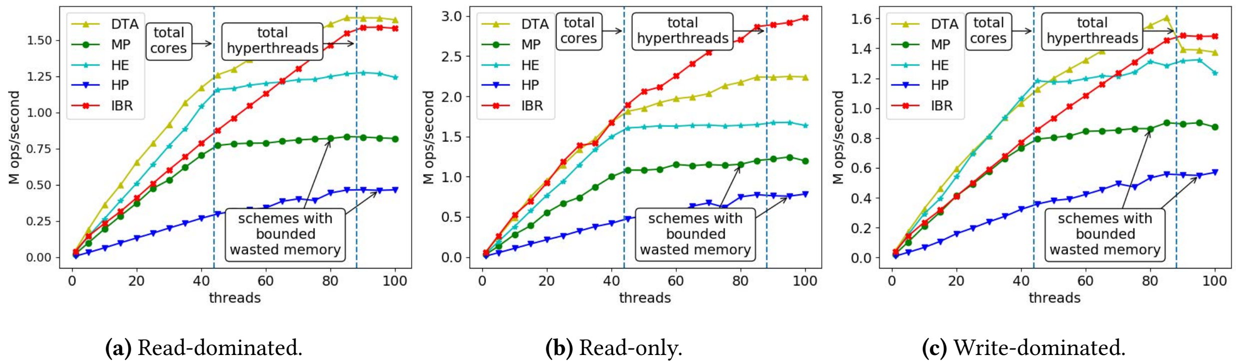
(a) Read-dominated. **(b)** Read-only. **(c)** Write-dominated.

Figure 2. Natarajan-Mittel BST (500 K nodes) throughput.



(a) Read-dominated. **(b)** Read-only. **(c)** Write-dominated.

Figure 3. Skip list (500 K nodes) throughput.



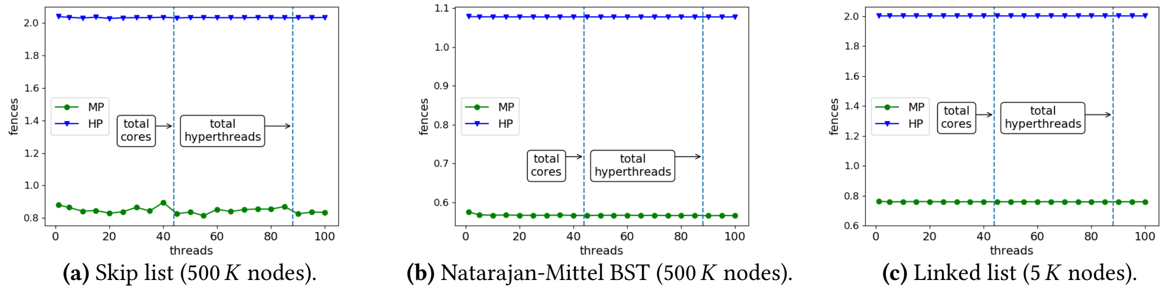
(a) Read-dominated. **(b)** Read-only. **(c)** Write-dominated.

Figure 4. Linked list (5 K nodes) throughput.

memory overhead to grow arbitrarily (§ 3.1) does not occur in our experiments.

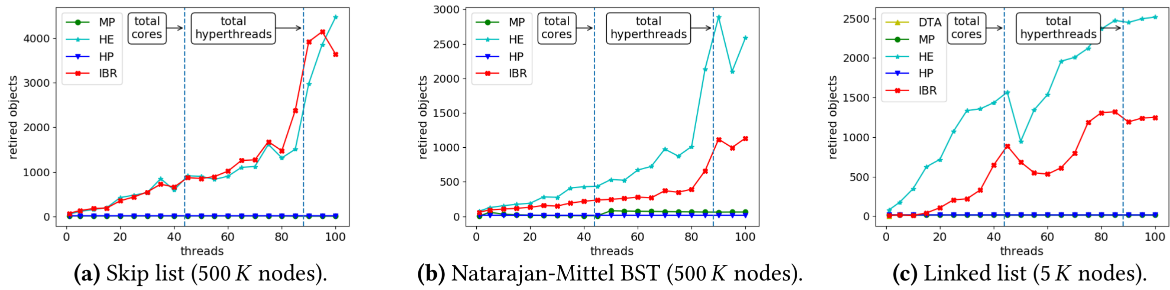
Key Distribution & MP Index Collisions Because a node's immutable index is determined on insertion, the efficacy of MP's margin-based protection mainly depends on how keys are distributed *inside* the data structure, not on the distribution of queried keys (e.g., whether it is uniform vs. skewed). Node layout in memory determines the probability for index collisions, which in turn, cause MP to fall back onto HPs with increased overhead. To study MP's sensitivity to node layout, we evaluate MP on a worst-case scenario [28, § 4.6.1]: a linked list constructed by inserting keys in ascending order. In this scenario, each insertion halves the remaining index range. Since we use 32-bit indices, the indices of all nodes but the first 32 will collide. Figure 7a shows the search throughput (read-only workload) of MP vs. HP on such a 5 K-node list. MP gracefully degrades to HP's performance, without imposing additional overhead due to index collisions. We conclude that for clients prioritizing wasted memory bounds, applying MP is a net win; there is no risk of experiencing larger overhead than HP.

Margin Size Sensitivity Analysis MP has an overhead-/wasted memory trade-off that is influenced by the margin size. Increasing/decreasing the margin size increases/decreases the number of nodes that can be protected by a single MP, resulting in lower/higher run-time overhead but a higher/lower wasted memory bound (Theorem 4.2). To



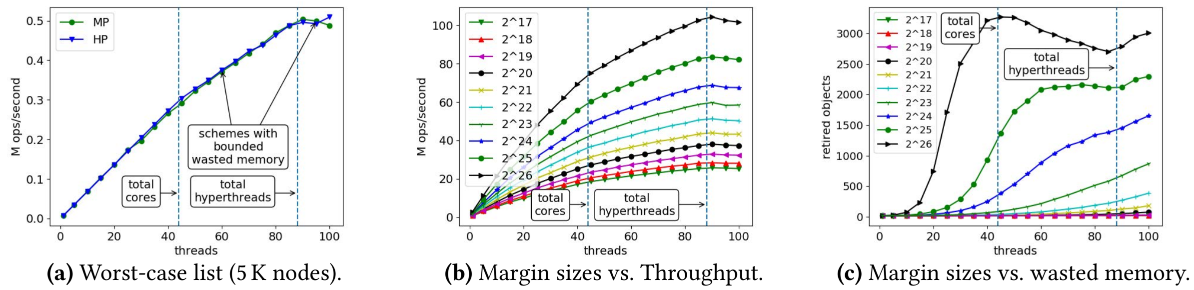
(a) Skip list (500 K nodes). **(b)** Natarajan-Mittel BST (500 K nodes). **(c)** Linked list (5 K nodes).

Figure 5. SMR-related fences per searched node (read-only workload).



(a) Skip list (500 K nodes). **(b)** Natarajan-Mittel BST (500 K nodes). **(c)** Linked list (5 K nodes).

Figure 6. Retired objects per operation (read-dominated workload).



(a) Worst-case list (5 K nodes). **(b)** Margin sizes vs. Throughput. **(c)** Margin sizes vs. wasted memory.

Figure 7. Left: Throughput of list read-only workload when list is initialized by inserting keys in ascending order. **Middle–Right:** Margin size sensitivity analysis (write-dominated workload).

quantify this effect, Figures 7b and 7c show an analysis of how margin size affects throughput and wasted memory (retired but unreclaimed nodes) on the write-dominated workload with the 500 K-node BST. We examine the margin sizes $2^{17}, 2^{18}, \ldots, 2^{26}$. As expected, both throughput and wasted memory size increase monotonically with the margin size. Based on this study, we pick a margin size of 2^{20} for all other experiments. This is the largest margin that maintains a roughly constant amount of wasted memory with only a small increase as the thread count grows, i.e., it has the lowest run-time overhead out of all margin sizes that yield a constant experimental bound on wasted memory.

Evaluation Takeaways ① MP is the best performer in its category of SMR schemes with bounded wasted memory. ② In the real-world cases of efficient data structures with non-read-only workloads, MP performs comparably to EBR-based schemes, despite its stronger wasted memory bound guarantee, and can outperform them in the presence of thread stalls. ③ MP wastes less memory than EBR-based schemes, not only in theory but in practice.

7 Conclusion

We introduce *margin pointers* (MP), the first nonblocking, self-contained SMR scheme featuring both bounded wasted memory and low run-time overhead. MP protects logical subsets of the data structure from being reclaimed, in contrast to prior approaches that protect explicit nodes. MP is most effective on efficient search data structures, such as binary trees and skip lists. MP significantly outperforms HP and can be competitive with EBR-based schemes, all while guaranteeing bounded wasted memory.

Acknowledgments This research was funded in part by the Israel Science Foundation (grant 2005/17) and the Blavatnik Family Foundation.

A Full MP Algorithm

The full MP algorithm is presented in Listing 10. We explain the differences from the previous code listings:

1. In addition to the *margin* pointer array, each thread has a *hazard* pointer array. Protection is checked against both arrays. When ending an operation, the thread clears both its margin and hazard pointer arrays.
2. Each thread also has a local epoch. When starting a new operation, the thread announces the current epoch in which it operates.
3. After protecting a node with a margin pointer, the thread checks that the global epoch did not advance. If it has, MP will fall back into using only HP.
4. Every node is associated with two additional fields, *birth*, which is filled when the node is allocated, and *retired*, which is filled when the node is retired.
5. When emptying the retired list, a thread's margin/hazard pointers are checked only if the thread's epoch intersects the retired node's lifetime. Moreover, nodes whose index is the reserved value *USE_HP* are not checked for margin protection.
6. When trying to protect a node, the extracted index may not be precise enough, so lower and upper bounds are calculated.
7. When checking if a retired node's index is covered by a *margin* pointer, due to lack of precision, the margin must cover both lower and upper bounds of the index.

Listing 10. Full MP pseudo code.

```
uint global_epoch          // global maintained epoch
uint local_epochs[thread_cnt] // local per-thread start epochs

const uint NO_MARGIN = 0xffffffff
const void*       NO_HAZARD = NULL
const uint USE_HP   = 0xffffffff

uint mp_slots[thread_cnt][MPs_per_thread]
void* hp_slots[thread_cnt][MPs_per_thread]

thread_local uint retired_counter
thread_local list retired

thread_local uint deletions_counter // total deletions by thread

const int empty_freq // frequency of trying to reclamation
const int epoch_freq // frequency of deletions to increment epoch

const uint max_index // MP indices will be between 0 and max_index
const uint margin  // protected margin range

struct Node {
 // .. client fields ...
 uint   next_index,
 uint   birth,
 uint   retired,
}

// public (invoked by client)
Function retire(Node* ptr): void
 retired.append(ptr)
 node->retired := global_epoch
 counter++
 if counter % empty_freq = 0:
  empty()

// private - invoked by the MR algorithm
Function empty(): void
 for node in retired:
  conflict := false
  idx := node->index
  for tid in threads:
   if local_epochs[tid] ∉ (node->birth, node->retired):
    continue
   if idx ≠ USE_HP:
    for mp in mp_slots[tid]:
     if mp ≠ NO_MARGIN and idx ∈ [mp − margin/2, mp + margin/2]:
      conflict := true
   else:
    for hp in hp_slots[tid]:
     if hp ≠ NO_HAZARD and hp = node:
      conflict := true
  if not conflict:
   free(node)

Function start_op(): void
 local_epochs[tid] := global_epoch
 upper_bound := lower_bound := 0
 memory_fence

Function end_op(): void
 for i in range(MPs_per_thread):
  mp_slots[tid][i] := NO_MARGIN
  hp_slots[tid][i] := NO_HAZARD
 memory_fence

Function alloc(Key key): Node*
 node = malloc(sizeof(Node))
 node->key := key
 if | upper_bound - lower_bound | ≤ 1:
  node->index = USE_HP
 else:
  node->index := ⌊(lower_bound + upper_bound) / 2⌋
 node->birth := global_epoch
 return node

Function read(Node** addr, uint refno): Node*
 while true do
  <node, idx> := *addr

  idx_lower_bound := idx << PRECISION
  idx_upper_bound := idx_lower_bound + ((1 << PRECISION) - 1)

  if idx_upper_bound = USE_HP:
   // collision - HP fall back
   hp_slots[tid][refno] := node
   memory_fence
   if <node, idx> := *addr:
    return node
   continue

  mp := mp_slots[tid][refno]
  // check if index is protected by a mp
  if mp − margin/2 ≤ idx_lower_bound and idx_upper_bound ≤ mp + margin/2:
   return node

  hp := hp_slots[tid][refno]
  // check if node is protected by a hp
  if hp := node:
   return node

  // no protection
  mp_slots[tid][refno] := idx
  memory_fence

  // ensure node remains linked to the data structure
  if <node, idx> = *addr:
   // ensure epoch did not advance
   if global_epoch ≠ local_epochs[tid]:
    use HPs from now (but old MPs remain)
    continue
   return node
```

References

[1] Dan Alistarh, William Leiserson, Alex Matveev, and Nir Shavit. 2015. ThreadScan: Automatic and Scalable Memory Reclamation. In *SPAA '15*.

[2] Dan Alistarh, William Leiserson, Alex Matveev, and Nir Shavit. 2017. Forkscan: Conservative Memory Reclamation for Modern Operating Systems. In *EuroSys '17*.

[3] Oana Balmau, Rachid Guerraoui, Maurice Herlihy, and Igor Zablotchi. 2016. Fast and Robust Memory Reclamation for Concurrent Data Structures. In *SPAA '16*.

[4] Anastasia Braginsky, Alex Kogan, and Erez Petrank. 2013. Drop the Anchor: Lightweight Memory Management for Non-blocking Data Structures. In *SPAA '13*.

[5] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A General Technique for Non-Blocking Trees. In *PPoPP '14*.

[6] Trevor Alexander Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There Has to Be a Better Way. In *PODC '15*.

[7] Nachshon Cohen. 2018. Every Data Structure Deserves Lock-free Memory Reclamation. *Proceedings of the ACM on Programming Languages* 2, OOPSLA, Article 143 (2018).

[8] Nachshon Cohen and Erez Petrank. 2015. Automatic Memory Reclamation for Lock-free Data Structures *(OOPSLA 2015)*.

[9] Nachshon Cohen and Erez Petrank. 2015. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *SPAA '15*.

[10] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *ASPLOS '15*.

[11] David L. Detlefs, Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. 2002. Lock-free Reference Counting. *Distributed Computing* 15, 4 (2002).

[12] Dave Dice, Maurice Herlihy, and Alex Kogan. 2016. Fast Non-Intrusive Memory Reclamation for Highly-Concurrent Data Structures. In *ISMM '16*.

[13] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. 2010. Non-Blocking Binary Search Trees. In *PODC '10*.

[14] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD.

[15] Keir Fraser. 2004. *Practical lock-freedom*. Ph.D. Dissertation. University of Cambridge.

[16] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *DISC '01*.

[17] Maurice Herlihy. 1991. Wait-free synchronization. *TOPLAS* 13 (January 1991), 26 pages. Issue 1.

[18] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. 2002. Dynamic-Sized Lock-Free Data Structures. In *PODC '02*.

[19] Shane V. Howley and Jeremy Jones. 2012. A Non-Blocking Internal Binary Search Tree. In *SPAA '12*.

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

[21] Maged M. Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *SPAA '02*.

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

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

[24] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *PPoPP '14*.

[25] Matthew Parkinson, Dimitrios Vytiniotis, Kapil Vaswani, Manuel Costa, Pantazis Deligiannis, Dylan McDermott, Aaron Blankstein, and Jonathan Balkind. 2017. Project Snowflake: Non-blocking Safe Manual Memory Management in .NET. *Proceedings of the ACM on Programming Languages* (2017).

[26] Arunmoezhi Ramachandran and Neeraj Mittal. 2015. A Fast Lock-Free Internal Binary Search Tree. In *ICDCN '15*.

[27] Pedro Ramalhete and Andreia Correia. 2017. Brief Announcement: Hazard Eras - Non-Blocking Memory Reclamation. In *SPAA '17*.

[28] Daniel Solomon. 2020. *Efficiently Reclaiming Memory in Concurrent Search Data Structures While Bounding Wasted Memory*. Master's thesis. Tel Aviv University.

[29] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-based Memory Reclamation. In *PPoPP '18*.