

# Are Your Epochs Too Epic? Batch Free Can Be Harmful

Daewoo Kim  
University of Waterloo  
Canada  
daewoo.kim@uwaterloo.ca

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

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

## Abstract

Epoch based memory reclamation (EBR) is one of the most popular techniques for reclaiming memory in lock-free and optimistic locking data structures, due to its ease of use and good performance in practice. However, EBR is known to be sensitive to thread delays, which can result in performance degradation. Moreover, the exact mechanism for this performance degradation is not well understood.

This paper illustrates this performance degradation in a popular data structure benchmark, and does a deep dive to uncover its root cause—a subtle interaction between EBR and state of the art memory allocators. In essence, modern allocators attempt to reduce the overhead of freeing by maintaining bounded thread caches of objects for local reuse, actually freeing them (a very high latency operation) only when thread caches become too large. EBR immediately bypasses these mechanisms whenever a particularly large batch of objects is freed, substantially increasing overheads and latencies. Beyond EBR, many memory reclamation algorithms, and data structures, that reclaim objects in large batches suffer similar deleterious interactions with popular allocators.

We propose a simple algorithmic fix for such algorithms to amortize the freeing of large object batches over time, and apply this technique to ten existing memory reclamation algorithms, observing performance improvements for nine out of ten, and over 50% improvement for six out of ten in experiments on a high performance lock-free ABtree. We also present an extremely simple token passing variant of EBR and show that, with our fix, it performs 1.5-2.6× faster than the fastest known memory reclamation algorithm, and 1.2-1.5× faster than not reclaiming at all, on a 192 thread four socket Intel system.

**CCS Concepts:** • Computing methodologies → Concurrent algorithms; Concurrent programming languages.

**Keywords:** safe memory reclamation, memory allocator, concurrent data structures, memory management

## 1 Introduction

Concurrent data structures serve as crucial building blocks for high performance multicore applications. Many concurrent data structures assume that memory is automatically garbage collected, which means that in unmanaged environments such as C/C++, they must be paired with a separate safe memory reclamation (SMR) algorithm. SMR algorithms ensure that a thread can free an object only if no other thread can possibly access the object (or else a segmentation fault could occur). Epoch based reclamation (EBR) is one of the most widely used reclamation algorithms for concurrent data structures, largely owing its popularity to its ease of use and extremely low overhead.

To obtain the highest performance from a modern concurrent data structure, one must typically also pair it with a fast memory allocator that is engineered for highly concurrent systems. JEmalloc [15] and TCMalloc [17] are two of the most popular choices. Both of these allocators have seen widespread use and undergone significant development by industry. For example, JEmalloc is the standard allocator for the FreeBSD operating system, as well as the allocator for the Firefox web browser.

SMR algorithms and memory allocators have historically been developed and optimized for relatively small scale systems with one or two processor sockets with fairly uniform memory architectures. However, in recent years, processor and system designs have become increasingly non-uniform. Recent AMD processors follow a hierarchical chiplet design, wherein a 64 core processor is actually a set of eight interconnected chiplets of eight cores each, with their own internal interconnects and local caches. In server environments, large multi socket servers are becoming common, with Amazon AWS M7i cloud servers running on four socket, 192 hardware thread Intel configurations similar to the experimental system we use in this paper. It is important to consider the impact such increasing non-uniformity has on the performance of popular algorithms and system software.

To this end, we conduct a rigorous study of the performance of a fast concurrent data structure paired with a state of the art implementation of EBR called DEBRA [9] and two popular memory allocators—JEmalloc and TCMalloc. Our

---

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](mailto:permissions@acm.org).  
*PPoPP '24, March 2–6, 2024, Edinburgh, United Kingdom*

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0435-2/24/03...\$15.00

<https://doi.org/10.1145/3627535.3638491>

experiments show that common workloads that show strong scaling up to moderate thread counts (up to two processor sockets) can experience severe performance degradation when running at very high thread counts (using three or four sockets). Subsequent experiments delve into the root cause of this performance problem, which turns out to be a subtle interaction between the EBR algorithm and the memory allocator.

With the help of a new visualization technique that we call timeline graphs, we discover that algorithms such as DEBRA that free objects in large batches circumvent a key optimization in JEmalloc. This optimization is intended to avoid the overhead of returning an object to a remote thread that allocated it (i.e., its *owner*), by instead placing the object in a local buffer that the local thread can subsequently allocate from. Every object allocated locally from this buffer is an object that does not need to be freed remotely, back to its owner. Freeing a large batch of objects can overflow this buffer, triggering an extremely high latency free call (on the order of tens of milliseconds or more) in which many objects are removed from this buffer and freed remotely to their respective owners, incurring extremely high lock contention in the process. We call this the *remote batch free* (RBF) problem.

Beyond EBR, the RBF problem appears to broadly affect any kind of data structure or algorithm that reclaims memory in batches. Many algorithms, including nearly all modern SMR algorithms, treat batching objects to free them together as a kind of optimization. In this work, we argue that this is actually an anti-pattern.

We propose a simple algorithmic fix called *amortized free* (AF) to mitigate the RBF problem. AF essentially reverses the typical batch freeing optimization. Whenever an algorithm would normally free a batch of nodes, we instead place them in an auxiliary list and free them gradually over time. Freeing objects gradually creates opportunities for a thread to reallocate these objects locally, rather than returning them to their owners.

To demonstrate the generality of this work, we reproduce the problem in both JEmalloc and TCmalloc, implement our solution in *ten* different SMR algorithms, and leverage new insights to design a new, exceedingly simple, high performance SMR algorithm. Although our primary goal is to shed light on the subtle interactions between modern algorithms and system software, our experiments show that our solution is strikingly effective, and it results in dramatic improvements over prior art, as we explain below.

**In summary, we make the following contributions.**

- We reveal a subtle and highly impactful negative performance interaction between modern memory allocators and algorithms that free objects in batches, as well as a simple algorithmic fix for this problem called amortized freeing.
- We introduce timeline graphs, a new visualization that makes it substantially easier to understand the behaviour of threads in workloads with high latency operations.
- We apply amortized freeing to *ten* of the most popular SMR algorithms, resulting in performance improvements in nine of the ten algorithms, *doubling* the performance of six of the algorithms on average with 192 threads.
- Leveraging the insights gained in this work, we develop an extremely simple variant of EBR, *Amortized-free Token-EBR*, that outperforms the fastest existing SMR algorithm by 2.6× with 192 threads.
- To our knowledge, this is the most rigorous and in depth analysis of the behaviour of threads in EBR algorithms, and of the interaction between SMR and memory allocators, to date.

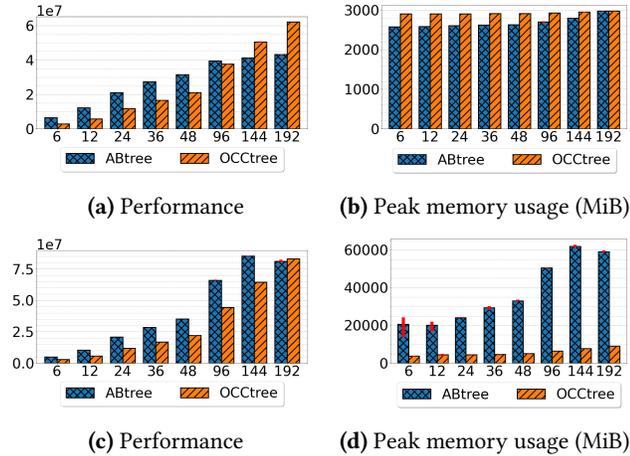
The paper is organized as follows. We discuss background in Section 2, diagnose the RBF problem, identify its root causes, and describe a simple solution in Section 3. Section 4 presents our new SMR algorithm as a sequence algorithmic improvements. Along the way, with the help of our timeline graphs, we describe in detail the lessons learned about the behaviour of threads in each variant that lead to our improvements. Our evaluation appears in Section 5. Finally, we survey related work in Section 6 and conclude in Section 7.

## 2 Background

In this paper, we study two of the most popular allocators, JEmalloc [15] and TCmalloc [17], as well as a new allocator from Microsoft Research called MImalloc [26]. Necessary details about the allocators' designs are included inline in the paper, but a more detailed description of the three allocators appears in the supplementary material.

We give a brief primer on EBR. Epoch based reclamation is one of the most widely used memory reclamation algorithms for concurrent data structures, particularly because it is easy to use and offers high performance. At a high level, in EBR, instead of freeing objects immediately, threads store these objects in a buffer called a *limbo bag*, to be freed later as a batch. The program execution is logically divided into *epochs*, and whenever the epoch changes, objects that were placed in a limbo bag in older epochs are freed.

Brown [9] introduced DEBRA, which has been shown to be one of the fastest EBR algorithms. A basic understanding of DEBRA will be important for our performance investigation below. In DEBRA, there is a global epoch number, and a single writer multi reader announcement array with one slot per thread, in which each thread stores the number of the epoch it is currently in. Threads update their announced epoch number at the start of each data structure operation. The global epoch can be advanced when all threads have announced it. To efficiently determine whether all threads



**Figure 1.** Performance (operations per second) and peak memory usage with JEmalloc, for OCCtree and ABtree, using DEBRA (upper) vs leaking memory (lower).

have announced the current epoch, each thread periodically (once every  $k$  operations) reads *one* other thread’s epoch, proceeding in round robin order. The first thread to notice that all threads have announced the current epoch will update the global epoch.

### 3 Diagnosing the RBF Problem

In running experiments on concurrent tree data structures, we observed that on large scale NUMA systems with four processor sockets, some data structures that scale similarly when running on a single socket scale drastically differently when running on four sockets. For example, consider an AVL tree by Bronson et al. [7] that uses optimistic concurrency control (hereafter, OCCtree), and a concurrency friendly variant of a B-tree by Brown [8] that uses lock-free techniques (hereafter, ABtree). We performed experiments to compare the performance of these data structures in a simple microbenchmark. Both data structures allocated memory using JEmalloc and reclaimed memory using DEBRA.

**Experimental Methodology.** For each thread count  $n \in \{6, 12, 24, 36, 48, 96, 144, 192\}$ , three trials were performed. In each trial,  $n$  threads access the same data structure, and for five seconds, repeatedly: flip a coin to decide whether to insert or delete a key, and perform the resulting operation on a uniform random key in a fixed key range  $[0, 2 \times 10^7)$ . Note that with a fixed key range, and 50 percent insert operations and 50 percent delete operations, in the steady state (after threads have run for a long time), the data structure should contain half of the key range. To avoid measuring the performance of a data structure as its size is changing at the beginning of the trial, the five second measured portion of the experiment begins once the size of the data structure stabilizes. In each trial, the total number of insert and

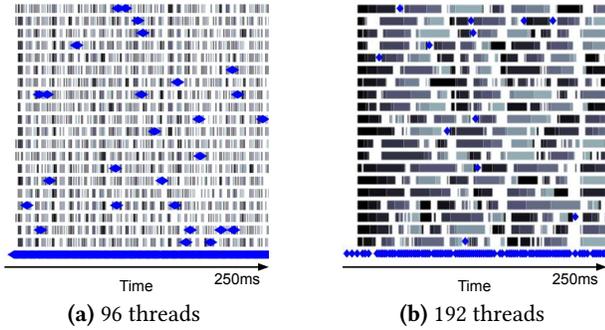
delete operations performed per second, across all threads (i.e., *throughput*), is reported. Each data point shows the average throughput over three trials, and the minimum and maximum throughput over these three trials is shown using error bars. This experimental methodology is similar to that in other papers that study concurrent memory reclamation (e.g., [9, 34, 35, 39]).

**System.** This experiment was run on a four socket Intel Xeon Platinum 8160 with 384 GB of DRAM. Each socket has 24 cores running at 2.1GHz nominal frequency with turbo boost up to 3.7GHz, and 48 logical processors with hyperthreading enabled, for a total of 96 cores and 192 hyperthreads across all sockets. The operating system was Ubuntu 20.04 LTS, with kernel version 5.8.0-55. Code was compiled with g++ 9.3.0-17 with `-O3` optimization and `std=c++14`. All experiments were run with `numactl -interleave=all` and threads were pinned to logical processors such that thread counts 1-24 run on a single socket, without hyperthreading, 25-48 run in a single socket with hyperthreading, and as additional threads are added, the same pinning pattern is applied to additional sockets as needed. (Sockets are populated with one thread per logical processor before additional sockets are used.)

**Symptom: poor scaling on NUMA.** Figure 1a clearly demonstrates the scaling issue described above. Both data structures scale well up to 48 threads, but the ABtree stops scaling above 96 threads, whereas the OCCtree continues to scale. One significant difference between the ABtree and the OCCtree is that the ABtree allocates one or two large nodes (240 bytes each) per insert or delete operation, whereas the OCCtree only allocates one small node (64 bytes) per insert operation (and does not allocate memory in a delete operation).

**Hypothesis: memory reclamation is a bottleneck.** Figure 1b shows the peak memory used (on average over three trials) for each of the data points in Figure 1a. Interestingly, peak memory usage for the ABtree is not much higher than for the OCCtree. This is perhaps surprising, since the ABtree allocates and reclaims substantially more memory per operation, on average, than the OCCtree. One might thus expect the ABtree to have substantially higher peak memory usage, especially at low thread counts, where it also performs *more operations* than the OCCtree. But, that is clearly not the case. One possible explanation is that, compared to the OCCtree, the ABtree spends a larger fraction of the execution time allocating and reclaiming memory than performing other useful work in the data structure (traversing and/or modifying it), limiting its performance.

To confirm this hypothesis, we disable memory reclamation for both data structures, and simply leak memory, to see whether this closes the performance gap between them. The results in Figure 1c largely confirm this hypothesis. It



**Figure 2.** Timeline graphs showing how much time threads spend freeing **batches** of nodes as epochs change with JEmalloc. (Y-axis = thread ID, blue dot = epoch change, space between boxes = time spent accessing the data structure.)

is clear that the ABtree allocates much more memory (Figure 1d), which means the allocator must do more work, and the memory reclamation algorithm must do much more work to maintain a small memory footprint. This suggests the performance degradation comes from memory management, either in the allocator or reclamation algorithm.

### 3.1 Investigating the Reclamation Bottleneck

Crucially, although DEBRA maintains a similar memory footprint for all of the thread counts in our experiment (Figure 1b), the performance of the ABtree substantially flattens at high thread counts in Figure 1a. This suggests that DEBRA is struggling to keep up with the amount of garbage being produced. DEBRA, like all EBR algorithms, is very sensitive to thread delays: *a single delayed thread can prevent all threads from reclaiming garbage* [35, 37]. To determine if this performance degradation is caused by thread delays, we visualized the behaviour of threads, specifically the time spent freeing objects, in a graph that we call a *timeline*.

**Timeline Graphs.** We have not seen timeline graphs used elsewhere in the literature, but our experience with them suggests they are a very useful tool for investigating performance problems caused by thread delays. We implemented a highly efficient mechanism to allow threads to record data (specifically two time stamps and a user specified value) in memory to be printed to files at the end of an experiment, with very little impact on performance. We did not measure any significant impact on performance when recording up to 100,000 timeline events per thread. Our code will be made available publicly when this paper is published.

Figure 2a and Figure 2b are timeline graphs showing how much time threads spend freeing nodes as epochs change in the ABtree with JEmalloc for 96 threads (left) and 192 threads (right). Rows represent different threads, and the x-axis represents time. For clarity, only 20 of the running threads and a representative 250ms of the 5 second trial are shown. Full graphs showing all threads and the entire 5 seconds of the

trial appear in the supplementary material. Each box represents a *reclamation event*, i.e., the time spent freeing a *batch* of objects removed from the data structure in a previous epoch. Boxes are coloured to make it easier to differentiate neighbouring events. Blue dots represent a thread successfully changing the global epoch number. All blue dots are also projected at the bottom of the graph to give a visual indication of how often the epoch changes overall. (This makes it easier to identify periods of time during which the epoch is not changed by any thread.)

**Comparing Timelines: 96 vs 192 Threads.** Comparing Figure 2a and Figure 2b, it is clear that more time is spent freeing objects, and individual reclamation events are much longer, with 192 threads than with 96 threads. (The time scales for these two enlarged graphs are the same.) Recall from Section 2 that DEBRA effectively amortizes the cost of scanning threads’ epoch announcements over many operations. This keeps per operation overhead low, but it means that doubling the number of threads should, on average, double the length of time needed to advance the epoch. This should in turn, double the amount of garbage to be reclaimed in each epoch, and double the length of time needed to free a batch, on average. So, we would *expect* the lengths of the reclamation events to be *twice* as long for 192 threads as for 96 threads.<sup>1</sup> However, we see that these events are many times longer than expected, which suggests there is an additional factor at work.

### 3.2 Root Cause of Long Reclamation Events

Further investigation using Linux Perftools led to the realization that poor performance in JEmalloc, such as when running on four sockets, is usually accompanied by a large fraction of the total cycle count being spent in function called `je_tcache_bin_flush_small`. Table 1 summarizes perf results to support the following discussion, and also quantifies how the total number of epochs changes as the thread count increases. These results confirm that the cost of freeing objects becomes prohibitive at high thread counts, preventing the data structure from scaling.

According to the source code for this version (5.0.1-25) of JEmalloc, when a thread invokes `free`, it places the freed object in a thread local buffer, and then checks whether the buffer is filled beyond a given threshold. If so, it takes a large number of objects from that buffer (approximately 3/4 of the buffer), and for each object, does the following. First, it identifies which *bin* the object belongs to<sup>2</sup>. If the object

<sup>1</sup>And, indeed, the lengths of reclamation events approximately double from 48 threads to 96 threads. See supplementary material for additional JEmalloc, TCMalloc and Mlmalloc results. Timeline graphs for TCMalloc showed similar behaviour.

<sup>2</sup>In this paper, we are not giving a detailed description of the “bin an object belongs to,” but intuitively, one can imagine it is the heap from which the object was originally allocated (and, hence, the heap to which it should be returned).

threads	ops/s	epochs	% free	% flush	% lock
48	35.9M	12631	11.5	9.9	4.9
96	45.3M	5176	39.3	38.3	24.6
192	43.4M	1980	59.5	58.8	39.8

**Table 1.** JEmalloc free overhead. % free = time spent in free. % flush = time spent in `je_tcache_bin_flush_small`. % lock = time spent in `je_malloc_mutex_lock_slow`.

was originally allocated by a different thread, this bin might reside on a remote core, or even a remote socket. The thread locks the bin, then iterates over all objects in its buffer (while holding the lock), and for each object that belongs to this bin, it performs the necessary bookkeeping to free the object to that bin.

Freeing a batch, especially the very large batches induced by high thread counts in DEBRA, triggers this mechanism often. This defeats the purpose of the buffer, which is intended to give a thread an opportunity to reallocate freed objects from the buffer, rather than always performing the bookkeeping required to move objects to remote bins.

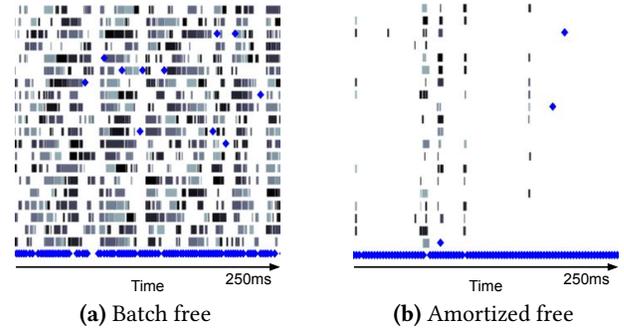
Most of the overhead of `je_tcache_bin_flush_small` comes from lock contention, as we discovered using `perf`. With 192 threads, 39.8% of the total time was spent in the function `je_malloc_mutex_lock_slow`, compared to 9.9% with 48 threads. After applying the solution we present below in Section 3.3, the time spent in this function is reduced to **5.5% with 192 threads** and less than 0.1% with 48 threads.

In summary, it is extremely expensive in JEmalloc for free to return objects to the remote threads that allocated them. To avoid this overhead, a thread frees to a local buffer, and subsequently allocates from that buffer. Every object allocated from that buffer is an object that *does not need to be freed to a remote thread in future*. Freeing a large batch overflows the buffer, forcing all objects to be freed remotely, causing extremely high lock contention.

### 3.3 A Simple Solution: Amortized Free

Given the above, it is clear that freeing large batches should be avoided wherever possible, at least when using JEmalloc. Although batching is inherent in EBR (and is also common in many other memory reclamation algorithms), once a batch of nodes has been *identified as safe to free*, one does not necessarily need to *free them immediately* as a batch. One could instead, for example, place the batch in a thread local *freeable* list, and gradually free objects one by one, each time a data structure operation is performed. We call this approach *amortized free* (AF).<sup>3</sup>

<sup>3</sup>As we will see, the amortized free approach is quite effective in improving multiple allocators and memory reclamation algorithms. However, another option is to modify the allocator itself to be sensitive to the possibility



**Figure 3.** Timeline graphs showing how long **individual** free calls take for batch free vs amortized free. 192 threads.

approach	ops/s	freed	% free	% flush	% lock
JE batch	43.4M	114M	59.5	58.8	39.8
JE amort.	111.3M	292M	19.2	17.6	5.5

**Table 2.** Amortized free vs. batch free. 192 threads. freed = number of objects freed. % free = time spent freeing. % flush = time spent in `je_tcache_bin_flush_small`. % lock = time spent in `je_malloc_mutex_lock_slow`.

In addition to gradually freeing objects from this list, one *could* optimize further by *allocating* objects from this list directly. This would essentially turn this approach into object pooling, avoiding most interaction with the allocator altogether. We want to show that we can make interaction with the allocator fast—not avoid it—so we do **not** perform this optimization.<sup>4</sup>

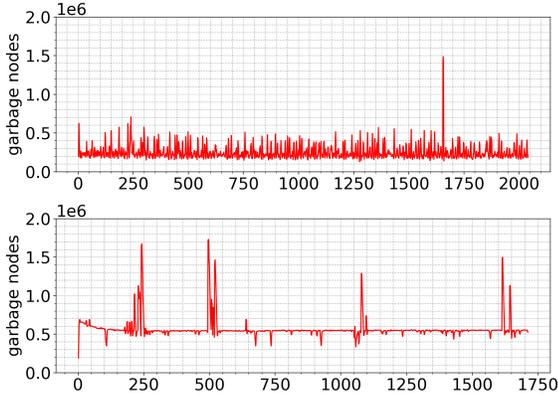
In the rest of the paper, unless otherwise stated, **every graph shows results for the ABtree and JEmalloc with 192 threads**, and follows the same methodology as described at the beginning of Section 3.

**Effect on the Overhead of Reclamation.** Figure 3a and Figure 3b illustrate the impact of amortized freeing on a timeline graph. Whereas Figure 2 visualized the time to free *batches* of objects, these new timelines show **individual free calls**. In both Figure 3a and Figure 3b the vast majority of free calls are too short to be visible on the graph. However, there is a clear difference: the batch free approach performs many more high latency free calls. The small number of high latency free calls that remain in the amortized free graph are further analyzed in the supplementary material.

The supporting `perf` results in Table 2 show that the amortized free algorithm spends  $\approx 3\times$  fewer cycles freeing

of batch frees coming from the reclamation algorithm. This would be an interesting direction for future work.

<sup>4</sup>The results in this paper can also help to explain why memory reclamation algorithms that free large batches, but use object pooling, such as Version Based Reclamation (VBR) [34], have been shown to outperform some prior EBR algorithms that interact directly with the allocator [19, 39].



**Figure 4.** Comparing the number of garbage nodes in each epoch for batch free (upper) and amortized free (lower) reveals the latter has a smoothing effect on memory usage.

memory and  $\approx 7\times$  fewer spinning on locks. However, the amortized free algorithm is  $2.6\times$  faster. These results suggest that amortized freeing can yield substantial performance improvements by reducing the overhead of freeing nodes.

Crucially, note that the algorithms that perform amortized freeing spend a third as much time freeing nodes (or less), even though they allocate and free more than twice as many nodes (because of the increased throughput). Effectively, JE batch spends  $5 \times 0.595 \approx 2.98$  seconds freeing 114M objects, averaging 38.3M objects freed/second, whereas JE amortized averages 304M objects freed/second. This suggests the improvement in the overhead of managing memory is on the order of  $8\times$ .

**Effect on Unreclaimed Garbage.** Figure 4 shows how the amount of unreclaimed garbage changes as the epoch advances for the original batch free approach (upper) and the amortized free approach (lower). To be precise, these graphs show: for each epoch, the sum, over all threads  $t$ , of the number of unreclaimed nodes  $t$  had in its limbo bag when  $t$  began that epoch. Amortized freeing substantially reduces the number of peaks in the graph while maintaining only a slightly larger amount of garbage on average.

**These Results Generalize to TCmalloc.** To determine whether these results are specific to JEmalloc, or are more general, we repeated all of the above experimentation and analysis with TCmalloc, another popular allocator. We found that the same problem arose and the same solution yielded large improvements. A small preview of the results appears in Table 3, where amortized freeing can be seen to improve performance by  $3.25\times$ . Additional results appear in the supplementary material.

**MImalloc Sidesteps the Problem Altogether.** MImalloc, on the other hand, is essentially immune to the problem we describe above by design. In MImalloc, a remote free synchronizes on a particular **page’s** free list. In contrast, in

approach	ops/s	freed	% free
TC batch	25.7M	69M	52.6
TC amort.	83.5M	219M	11.8
MI batch	104M	273M	15.6
MI amort.	95.0M	249M	17.2

**Table 3.** Analysis for additional allocators. 192 threads.

JEmalloc, a remote batch free synchronizes on one of  $4T$  arenas, where  $T$  is the number of hardware threads, meaning if  $T$  threads all perform remote free operations, we would expect approximately  $1/4$  of them to contend with one another. In TCmalloc, a remote batch free synchronizes on a global cache, which is even worse. This makes it relatively inexpensive to immediately free an individual object to a remote thread in MImalloc, as doing so will cause contention only if another thread is simultaneously freeing another object that was allocated *from the same page*. On the other hand, if a non-trivial number of threads are performing remote batch frees in JEmalloc or TCmalloc, they are much more likely to experience contention. As expected, Table 3 shows that amortized free does not improve performance in MImalloc, and in fact worsens it slightly.

MImalloc is quite unique in its approach. To our knowledge, no other allocator maintains per-page free lists. As a result, we expect that many other allocators would suffer similar RBF problems to JEmalloc and TCmalloc. It is worth noting that the JEmalloc results above with amortized freeing are faster than MImalloc, and that programmers are not always free to change the allocator in their environment, so finding workarounds for deficiencies in the most popular allocators is worthwhile.

## 4 Token-EBR: A Simpler EBR Algorithm

In this section, we investigate the question of whether an extremely simple EBR algorithm, paired with amortized freeing, can match or exceed the performance of the state of the art. To this end, we revisit an old idea: token rings. To our knowledge, the idea of passing a token around a ring to establish an epoch (which we call *Token-EBR*) has not been implemented or evaluated in the peer reviewed literature on safe memory reclamation, appearing only in a thesis by Tam [38] where the idea is simply sketched at a high level, then dismissed as inefficient.

We consider a sequence of possible implementations of the abstract algorithm, study their characteristics, show that all of the implementations that do not use amortized freeing are unusable in practice, and derive a final implementation that outperforms the state of the art in concurrent memory reclamation. It was surprising to us that such a simple EBR implementation outperformed the state of the art when paired with *Amortized Freeing*.

We start by explaining the abstract algorithm and why it is correct.

**Abstract Algorithm.** Conceptually, the algorithm is straightforward. All threads,  $T_1, T_2, \dots, T_n$ , are arranged in a ring. The token is passed around the ring to define epochs. More specifically, all threads begin in epoch zero, and enter a new epoch whenever they receive the token. Each thread has two limbo bags: one for the current epoch (the *current bag*), and one for the previous epoch (the *previous bag*), both initially empty. Objects that a thread unlinks from the data structure are always placed in the thread’s current bag.

Whenever  $T_i$  begins a new data structure operation, it checks whether it has received the token. If so, it conceptually enters a new epoch, and can (1) pass the token to the next thread, and (2) free all objects in its previous bag. (We will discuss the safety of freeing objects below.) Since its previous bag is now empty, it can simply swap its current and previous bags. The result is an empty current bag, and a previous bag that contains objects unlinked in the previous epoch (as it should).

**Correctness Sketch.** To understand why this algorithm ensures safe memory reclamation (i.e., ensures that a thread frees an object only if no other thread has a pointer to it), consider a time interval  $I$  during which the token makes its way around the entire ring. During this interval, each thread has passed the token, so each thread has begun a new data structure operation. Suppose an object  $O$  is unlinked from the data structure (but not yet freed) before the interval  $I$  begins. We argue that  $O$  is safe to free at the end of  $I$ .

Each thread that begins a data structure operation before  $O$  is unlinked could potentially still access  $O$  until it finishes its operation. However, at the *end* of interval  $I$ , each thread has finished its current operation and started a new operation. And, each thread’s new operation began after  $O$  was already unlinked from the data structure, and so cannot have obtained a pointer to  $O$  by accessing the data structure. So, no thread can access  $O$ , and thus  $O$  is safe to free. More broadly, every object that was removed from the data structure before  $I$  is safe to free after  $I$ .

### 4.1 Naive Token-EBR

In our first implementation of *Token-EBR*, which we call *Naive Token-EBR*, at the start of a data structure operation, a thread first (1) checks whether it has received the token, (2) if so frees the contents of its previous bag, swaps its bags and passes the token, in that order.

At first glance, Figure 5a suggests that *Naive Token-EBR* is a better algorithm than DEBRA. However, Figure 5b reveals that it does a terrible job of actually reclaiming memory. And, all of that time spent not reclaiming memory can be directed towards performing data structure operations, artificially inflating its throughput—at least until the system runs out of memory.

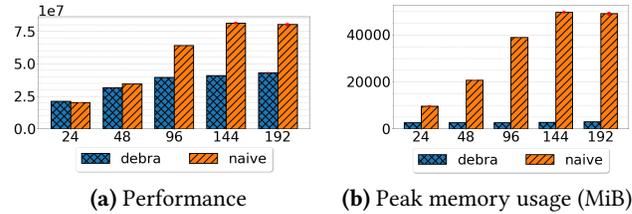


Figure 5. Performance and peak memory usage with JEmalloc, for ABtree, using *Naive Token-EBR*.

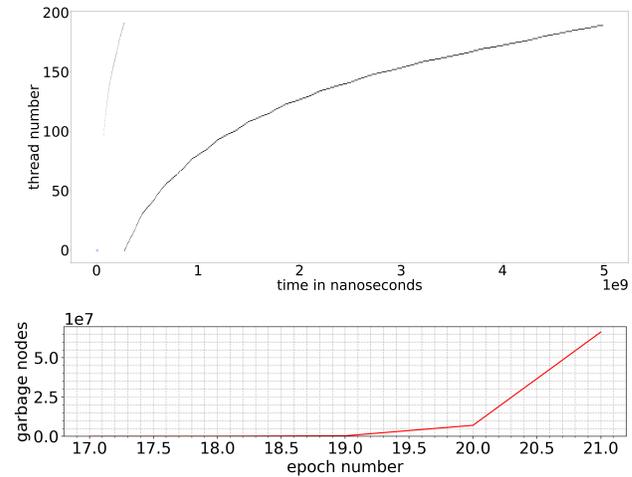
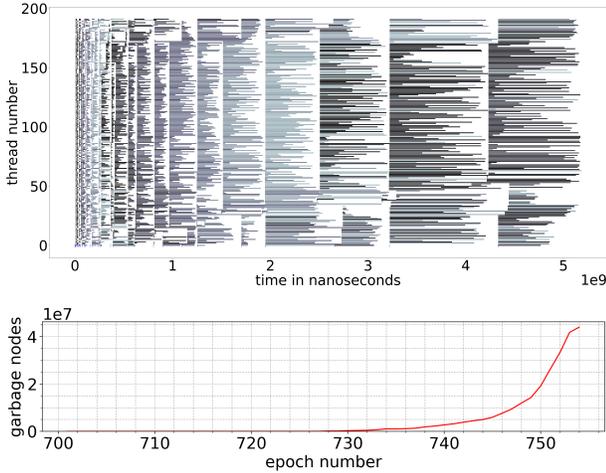


Figure 6. Timeline graph showing **batch** frees (upper) and number of garbage nodes (lower) for *Naive Token-EBR*.

The problem is illustrated in Figure 6, which looks quite different from the timeline graphs shown in Section 3. Visually, the graph looks like a continuous curve, but the “curve” is in fact a sequence of batch frees performed by individual threads, one after another, with no two threads freeing objects concurrently.

This serialization is a direct consequence of the decision to free the contents of the previous bag *before* passing the token. The next thread cannot free until it receives the token—after the previous thread finishes freeing. Worse still, while one thread is freeing,  $n - 1$  other threads are continually performing data structure operations without freeing, accumulating more and more garbage. Consequently, each thread finds itself with more garbage to free than the previous thread, and the problem compounds in each epoch. We call this the *garbage pile up* problem.

This problem is so pronounced, that the last epoch dominates the graph, lasting from  $\approx 0.25$  seconds until 5 seconds. The previous epoch lasts approximately a fifth of a second, and is visible as a very faint curve near the beginning of the time axis. There are 21 epochs in total, and the first 19 are too short to see.



**Figure 7.** Timeline graph showing **batch** frees (upper) and number of garbage nodes (lower) for *Pass-first Token-EBR*.

The bottom plot in Figure 6 shows the drastic increase in the amount of garbage accumulated in each epoch. To understand why the garbage pile up is so large in the final epoch, note that more than 90% of the execution is spent in that epoch, and after the first thread reclaims memory at the *start* of that epoch, it never reclaims again (and similarly for the second thread, and so on).

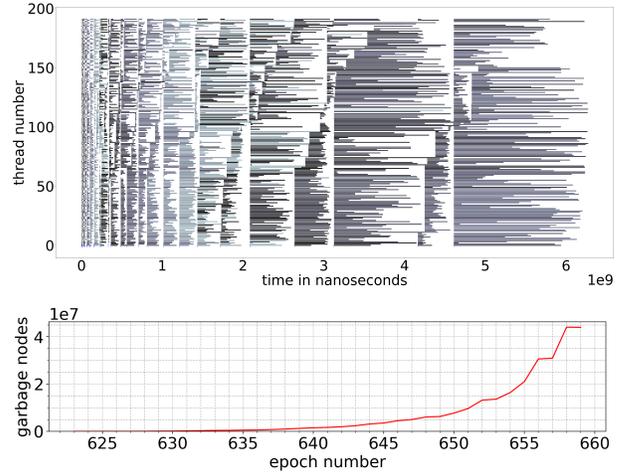
**Pass-first Token-EBR.** In our second algorithm, *Pass-first Token-EBR*, the token is passed *before* the contents of the previous bag are freed. Note that, in terms of correctness, it does not matter whether the thread frees and then passes the token, or passes then frees—it is the *receipt* of the token that tells the thread it can safely free its previous bag.

Figure 10 shows that *Pass-first Token-EBR* improves performance and reduces peak memory usage compared to *Naive Token-EBR*. Now that threads can free their bags concurrently, Figure 7 looks a little bit more like the timeline graphs in Section 3. However, the algorithm is clearly still susceptible to garbage pile up, as the lengths of batch free operations are increasing over time. This is further confirmed in Figure 7.

One reason for this is that a thread that receives the token, passes it to the next thread, and begins freeing a large bag of objects, may actually receive the token again before it is finished freeing. It will then hold onto the token until it has finished freeing, which can potentially be a long time.

**Periodic Token-EBR.** In our third algorithm, *Periodic Token-EBR*, a thread passes the token, then begins freeing the contents of its previous bag. However, as it is freeing objects, it periodically checks, every  $k$  free calls (100 in our experiments), whether it has received the token, and if so passes it along.

As Figure 10 shows, this approach performs somewhat similarly to *Pass-first Token-EBR*, but has significantly lower

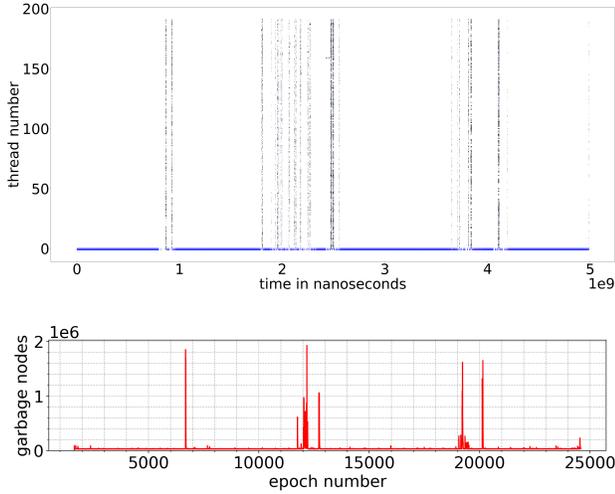


**Figure 8.** Timeline graph showing **batch** frees (upper) and number of garbage nodes (lower) for *Periodic Token-EBR*.

peak memory usage with 192 threads. Unfortunately, as Figure 8 shown, there is *still* a garbage pile up problem. This result may seem surprising at first. If we free no more than 100 objects while holding the token before passing it, how can threads have an opportunity to accumulate more than 40 million garbage nodes without passing the token all the way around the ring and advancing the epoch?

It turns out we have already learned the answer to this question above. In JEmalloc, whenever a thread overflows its internal buffer of freed nodes, and triggers a high latency remote free operation, a **single free call** can run for a very long time—on the order of milliseconds. As a result, it does not matter whether a thread checks if it holds the token after every 100 free calls, or after *every* free call. Unless it can somehow perform this check *during* a single long free call (which presumably would require modifying the allocator itself), the epoch cannot advance until that long free call has finished. And, as we saw in Figure 3a, large batch free operations can and do frequently cause such long free calls.

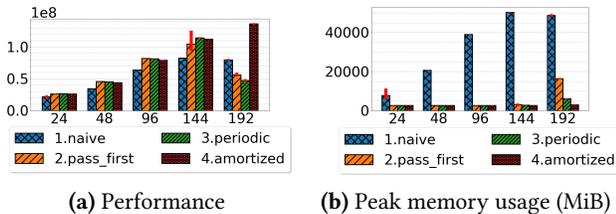
One might also wonder why peak memory usage is lower for this approach, even through the epoch counts in Figure 7 and Figure 8 are similar, and the problem of high latency free operations still exists. Since epochs are increasing in length as time passes, peak memory usage is essentially dictated solely by the final epoch. Moreover, a thread that begins freeing in the final epoch has typically accumulated so much garbage that it will not finish freeing until the experiment is over (as evidenced by the fact that batch frees do not finish until after the 5 second time limit). In the final epoch in *Pass-first Token-EBR*, such a thread will not pass the token, so all threads after it in the token ring will accumulate garbage in the final epoch, increasing peak memory usage. On the other hand, in *Periodic Token-EBR*, although such a thread



**Figure 9.** Timeline graph (upper) and number of garbage nodes (lower) for *Amortized-free Token-EBR*. This timeline graph shows **individual** free calls longer than 0.1ms.

will be unable to pass the token *during* a high latency free operation, if the thread performs at least *two* high latency free operations in the final epoch, then it can still pass the token in between high latency free operations. This enables additional threads to free their limbo bags concurrently in the final epoch, reducing peak memory usage.

**Amortized-Free Token-EBR.** Our final algorithm applies the amortized freeing approach to *Periodic Token-EBR*. Figure 10 shows this offers drastic improvements in both performance and peak memory usage with 192 threads. As Figure 9 shows, amortized free mitigates the garbage pile up problem (as well as greatly increasing the epoch count). Table 4 summarizes the impact of each algorithmic change on performance, time spent freeing, and the total number of objects freed.



**Figure 10.** Performance and peak memory usage with JE-malloc, for ABtree, using *Amortized-free Token-EBR*.

## 5 Evaluation

We evaluate the impact of the amortized freeing technique using a benchmark that implements the ABtree with 10

algorithm	ops/s	% free	freed
Naive	73.7M	3.3	7M
Pass-first	52.4M	45.4	98M
Periodic	54.4M	47.1	118M
Amortized	123.7M	14.7	323M

**Table 4.** Analysis of Token EBR variants. 192 threads.

memory reclamation algorithms, encompassing *Amortized-free Token-EBR* (token\_af), debra [9] and its amortized free version (debra\_af), hazard eras (he) [33], hazard pointers (hp) [28], interval based reclamation (ibr) [39], two neutralization based reclamation techniques [35], nbr and nbr+, quiescent state based reclamation (qsbr) [20], read-copy-update (rcu) [20] and wait free eras (wfe) [32] in two different experiments. The methodology and system are the same as in Section 3, except for the selected thread counts.

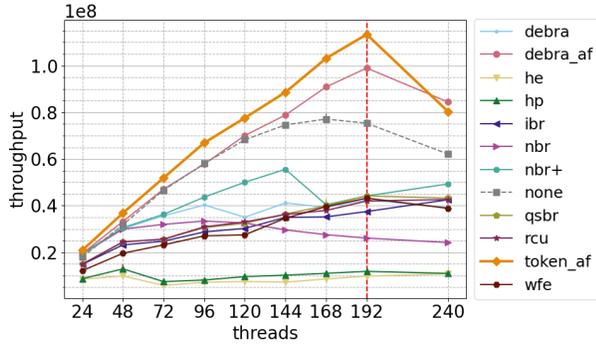
Note that our code will be submitted for artifact evaluation, and will be made public when the paper is published.

**Experiment 1:** We compare the performance of token\_af and debra\_af against a broad cross section of the state of the art in reclamation techniques (in Figure 11a). The vertical red line denotes the number of hardware threads in the system. A leaky implementation (none) is also included.

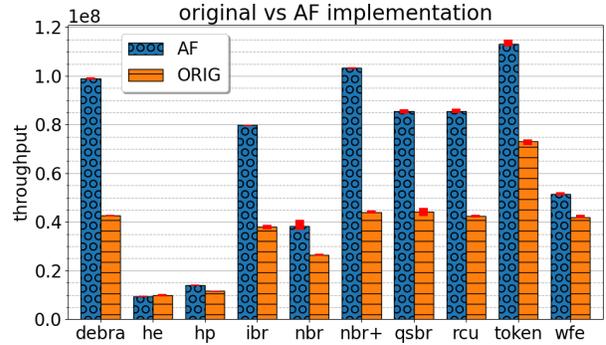
Our token\_af algorithm outperforms all other techniques. Averaging the results across all thread counts, it is  $\approx 1.7\times$  fast than the next fastest algorithm, nbr+, and  $\approx 7-9\times$  better than the slowest algorithms, hp and he. Surprisingly, both of our amortized free algorithms token\_af and debra\_af significantly outperform none, which is often (incorrectly) described as an upper bound on the performance of a reclamation algorithm in the literature. Of course, prior reclamation algorithms have been shown to outperform leaky implementations previously [9], citing improved locality.

**Experiment 2:** We assess the potential for the amortized free technique to improve other reclamation algorithms by implementing amortized free (AF) versions of *all 10* of the reclamation algorithms studied in Experiment 1. We then compare their throughput with their original (ORIG) implementations with 192 threads (Figure 11b). All of these algorithms accumulate a batch of garbage before freeing, and we use a uniform batch size of 32K nodes for all algorithms. This particular size was chosen because (1) each algorithm performed best with this size, and (2) the nbr algorithm already uses this bag size by default [35].

Figure 11b shows that the AF algorithms outperform their ORIG counterparts for 9 algorithms: debra, ibr, nbr/nbr+, qsbr, rcu, token, hp and wfe, demonstrating up to  $2.3\times$  improvements in throughput. The he algorithm does not improve, whereas hp and wfe show modest improvements of  $\approx 1.2\times$ . The lack of improvement (or limited improvement) in



(a) Experiment 1: Comparison of proposed *Amortized-free Token-EBR* (token\_af) with other reclamation techniques across threads.



(b) Experiment 2: Comparison with amortized free versions of various reclaimers at 192 threads.

**Figure 11.** Data structure: Brown’s ABtree. Workload: 50% inserts and 50% deletes. Size: 20M. Allocator:JEmalloc.

these algorithms is likely due to the fact that they have much higher synchronization overhead than the other algorithms, which prevents any improvement from manifesting. Additional experiments showing AF vs ORIG improvements at various thread counts appear in the supplementary material.

## 6 Related Work

The idea of passing a token around to establish a time when it is safe to free is not new. Practitioners have discussed implementing EBR in this way on online discussion forums [40], and similar algorithms have been used to reclaim memory in operating system kernels [25]. Token based algorithms have also been used as part of complex garbage collection algorithms in the distributed setting [22], and token based EBR has been described in a thesis due to Tam in the shared memory multicore setting [38], where it was dismissed as inefficient. Our new token based algorithm surpasses the state of the art, which is especially impressive considering there has been more than 15 years of progress in EBR algorithms since the publication of [38].

The study of concurrent memory reclamation saw multiple fundamental results appear in the early 2000s, including EBR algorithms [16, 27], pointer based reclamation techniques [21, 28] and reference counting techniques [13]. Since then, many new algorithms have appeared [1–6, 9–12, 14, 18, 20, 23, 24, 30–32, 32, 33, 33–35, 39], nearly all of which free batches of objects.

Most recently, a hardware-software co-design called *Conditional Access* [36] has appeared that facilitates the immediate reclamation of individual object, rather than batches. Experiments using JEmalloc showed significant performance improvement vs traditional memory reclamation algorithms that free batches of objects, but the authors did not establish a concrete reason for the improvement. Our work sheds new light on why techniques like *Conditional Access* that immediately free individual objects perform well in practice.

Mitake et al [29] studied the impact of peak memory usage on in-memory database transaction latencies that used epoch based reclamation. They suggested freeing batches using a separate background thread to increase the frequency with which the main worker threads could participate in advancing the epoch. In light of our work, moving batch freeing to a background thread appears to be insufficient to avoid the RBF problem. Batch freeing is, itself, the problem.

## 7 Conclusion

In this paper, we identify a performance problem that limits the performance and scaling of concurrent memory reclamation algorithms on modern multi socket systems. We perform a rigorous study of the root causes of this problem, and identify subtle interactions between popular allocators and memory reclamation algorithms that free batches of objects.

We present a simple workaround, amortized freeing, and demonstrate its effectiveness by applying it to ten existing memory reclamation algorithms, and a new reclamation algorithm token\_af. As our experiments show, the potential impact of the presented ideas on the memory reclamation literature is substantial. With 192 concurrent threads, amortized freeing on average *doubles the performance of the six fastest existing memory reclamation algorithms*, and token\_af *outperforms the state of the art by 2.6x*.

Amortized freeing was a natural fit for the ABtree data structure that we used in our experiments, since each operation frees at most one object on average. In data structures that free more than one object per operation on average, amortized freeing should be tuned to free more than one object per operation. Amortized freeing will be most effective if the number of objects freed and allocated per operation is similar, so that objects gradually freed to internal thread-local buffers in the allocator can be reallocated locally.

Finally, we have timeline graphs to thank for many of our insights. We hope this new visualization tool will prove useful in designing and profiling new concurrent algorithms.

## 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 comments and insights.

## References

- [1] 2021. Crystalline: Fast and Memory Efficient Wait-Free Reclamation. Ruslan Nikolaev and Binoy Ravindran (Eds.). *CoRR* abs/2108.02763. arXiv:2108.02763 <https://arxiv.org/abs/2108.02763>
- [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. 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.
- [7] 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.
- [8] Trevor Brown. 2017. *Techniques for Constructing Efficient Lock-free Data Structures*. Ph. D. Dissertation. University of Toronto.
- [9] 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.
- [10] Nachshon Cohen and Erez Petrank. 2015. Automatic memory reclamation for lock-free data structures. *ACM SIGPLAN Notices* 50, 10 (2015), 260–279.
- [11] 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.
- [12] Andreia Correia, Pedro Ramalheira, 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.
- [13] 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.
- [14] 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.
- [15] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the BSDcan conference, Ottawa, Canada*.
- [16] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.
- [17] Sanjay Ghemawat and Paul Menage. 2005. TCMalloc: Thread-caching malloc. Retrieved from <http://goog-perftools.sourceforge.net/doc/tcmalloc.html> on January 27, 2023 (2005).
- [18] Anders Gidenstam, Marina Papatrifaftilou, Håkan Sundell, and Philippos 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.
- [19] Timothy L. Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*. Springer, 300–314.
- [20] Thomas E. Hart, Paul E. 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.
- [21] 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.
- [22] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. 1997. Training distributed garbage: The DMOS collector. *Object-Oriented Programming Systems, Language and Applications (1997)*.
- [23] Jaehwang Jung, Janggun Lee, Jeonghyeon Kim, and Jeehoon Kang. 2023. Applying Hazard Pointers to More Concurrent Data Structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2023, Orlando, FL, USA, June 17-19, 2023*, Kunal Agrawal and Julian Shun (Eds.). ACM, 213–226. <https://doi.org/10.1145/3558481.3591102>
- [24] 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.
- [25] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. 2006. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review* 40, 4 (2006), 133–145.
- [26] 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.
- [27] 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. Citeseer, 509–518.
- [28] 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.
- [29] Hitoshi Mitake, Hiroshi Yamada, and Tatsuo Nakajima. 2019. Looking into the Peak memory consumption of epoch-based reclamation in scalable in-memory database systems. In *Database and Expert Systems Applications: 30th International Conference, DEXA 2019, Linz, Austria, August 26–29, 2019, Proceedings, Part II 30*. Springer, 3–18.

- [30] 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.
- [31] 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.
- [32] 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.
- [33] Pedro Ramalhete and Andreia Correia. [n. d.]. Brief announcement: Hazard eras-non-blocking memory reclamation. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures*. 367–369.
- [34] 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*. 443–445.
- [35] 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.
- [36] Ajay Singh, Trevor Brown, and Michael Spear. 2023. Efficient Hardware Primitives for Immediate Memory Reclamation in Optimistic Data Structures. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 112–122. <https://doi.org/10.1109/IPDPS54959.2023.00021>
- [37] 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>
- [38] Adrian Tam. 2006. *QDo: A Quiescent State Callback Facility*. Ph. D. Dissertation. University of Toronto.
- [39] 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.
- [40] ycombinator. 2017. Why is memory reclamation so important for lock-free algorithms? Retrieved from <https://web.archive.org/web/20200223075152/https://news.ycombinator.com/item?id=15269628> on January 27, 2023.

## A Artifact Description

The artifact containing the source code, scripts to run all experiments, and a detailed readme file is present at the URL: <https://doi.org/10.5281/zenodo.10226261>. If you prefer to run the artifact locally on your machine (without using the docker container) please directly 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. Once the docker container starts you can use the accompanying readme file to compile and run the experiments on the benchmark.

### Steps to load and run the provided Docker image in the artifact:

Note: Sudo permission may be required to execute the following instructions. The following instructions will help you install and directly run the docker container for `amortizedfree-setbench`.

1. Install the latest version of Docker on your system. We tested the artifact with the Docker version 20.10.2, build 20.10.2-0ubuntu1 20.04.2. Instructions to install Docker may be found at <https://docs.docker.com/engine/install/ubuntu>.  
\$ docker -v
2. Download the artifact named `amortizedfree-setbench.zip` from the artifact submission link: <https://doi.org/10.5281/zenodo.10226261>.
3. Extract the downloaded folder and move to `amortizedfree-setbench/` directory using `cd` command.
4. Find docker image named `amortizedfree_docker.tar.gz` in `amortizedfree-setbench/` directory. And load the downloaded docker image with the following command:  
\$ sudo docker load -i amortizedfree\_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 amortizedfree -it \ --privileged amortizedfree-setbench /bin/bash
7. Invoke `ls` to see several files/folders of the artifact: Dockerfile README.md, common, ds, install.sh, lib, microbench, af\_experiments, tools.

Now, to compile and run the experiments you could follow the instructions in the readme file. You may need to change the thread counts in scripts to suit the configuration of your machine.