Understanding NUMA Effects on Memory Allocation and Reclamation

by

Daewoo Kim

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Memory management in multicore systems is a well studied area. Many approaches to memory management have been developed and tuned with specific hardware architectures in mind, capitalizing on hardware characteristics to improve performance. In this thesis, the focus is on memory allocation and reclamation in multicore systems.

I first identify and diagnose a performance anomaly in epoch based memory reclamation (EBR), one of the most popular approaches to reclaiming memory in multicore systems. EBR experiences significant performance degradation when running on multiple processor sockets. This degradation is related to the fact that EBR is vulnerable to thread delays. Even minor delays can trigger a chain reaction that induces longer delays and more substantial performance problems. Moreover, I discover a negative interaction between EBR and popular memory allocators, caused by the fact that EBR frees batches of objects, and these allocators attempt to cache batches of objects for reallocation. The batches freed by EBR frequently overflow the allocator buffers, defeating their purpose and causing substantial performance overhead.

To solve these issues, an improvement to EBR, called amortized batch free is introduced to limit the amplification of delays and performance degradation when freeing. Amortized batch free gradually reclaims objects, and can drastically reduce the average time spent freeing an object. This technique is applied to a state of the art EBR algorithms, and significant performance improvements are shown experimentally.

This amortized batch freeing technique appears broadly applicable to other memory reclamation algorithms. As a first step in demonstrating this, I also apply it to a simple token based variant of EBR. Token EBR is conceptually simpler and easier to implement than the state of the art EBR algorithm, but has been shown in other work to perform poorly. When the amortized batch free technique is used, Token EBR performs similarly to (and sometimes slightly better than) the state of the art EBR algorithm.

Finally, I present a new design for an architecture aware memory allocator for multi-socket systems, using a state of the art allocator called Supermalloc as a starting point for my design. Several key bottlenecks in the original Supermalloc design are improved or eliminated in the new design. In particular, the new design dramatically improves performance when the address space is actively growing, reduces contention on shared resources, and optimizes memory accesses to reduce communication across processor sockets. Taking into account the lessons learned in the study of EBR, the new design also attempts to minimize the overhead of freeing objects. Experiments on a prototype of this new allocator show some performance improvement compared to the original Supermalloc allocator.

Acknowledgements

I would like to thank my supervisor Prof. Trevor Brown for his guidance and support throughout my Master's program. I would also like to thank Prof. Peter Buhr and Prof. Samer Al-Kiswany for serving as readers on my thesis committee. Finally, I would like to thank my family for their endless love and support.

Table of Contents

List of Figures

Chapter 1

Introduction

The performance of software is influenced by many factors, including synergies (or anti-synergies) with the hardware, operating system, and standard libraries. In this thesis, I focus on the performance of memory allocation and reclamation algorithms in multicore systems, and multi-socket systems with non-uniform memory architectures (NUMAs). Allocators and memory reclamation algorithms are critical for concurrent data structures, which serve as vital building blocks for concurrent software. A given concurrent data structure can be paired with different combinations of allocation and reclamation algorithms, resulting in vastly different levels of performance. The most popular allocators and memory reclamation algorithms typically provide high performance when running on a single socket, and sometimes even when running on two sockets. However, in this thesis, I present results showing that some high performance concurrent data structures perform surprisingly poorly when paired with leading allocators and memory reclamation algorithms, especially on systems with four sockets.

In Chapter 3, I perform a sequence of experiments studying the impact of memory reclamation on the performance of a leading concurrent tree data structure. The first experiment compares short executions that, respectively, leak memory and reclaim memory using a state of the art epoch based memory reclamation (EBR) algorithm called DEBRA [5], using various thread counts. This experiment reveals a severe performance problem when running on four sockets that appears to be caused by memory reclamation. Subsequent experiments delve into the root cause of this performance problem, which turns out to be a subtle interaction between the EBR algorithm and some popular allocators. I propose a simple algorithmic fix for the EBR algorithm and show that it significantly improves performance in a synthetic data structure benchmark. Furthermore, I show that this same fix can be applied to other epoch based memory reclamation algorithms. To

this end, I develop a variant of a simpler EBR algorithm, Token EBR, that is easier to implement and performs as well (and sometimes better) in practice than the state of the art.

In Chapter 4, I study several state of the art memory allocators: Hoard [1], TCmalloc [11], JEmalloc [10], Supermalloc [19] and Mimalloc [20]. Performance results are presented that suggest these allocators have primarily been optimized for single socket systems (and sometimes two socket systems). There have been some limited attempts to design NUMA aware allocators in the past. AMD developed a NUMA aware version of TCmalloc [15], but their implementation does not appear to be maintained and does not compile on modern systems. At present, the other allocators listed above do not appear to have NUMA aware counterparts. Using one of these allocators, Supermalloc, as a starting point, I propose a design for a NUMA aware allocator—i.e., an allocator that explicitly takes the topology of the system into account.

In this thesis, I make the following contributions:

- I demonstrate the root cause of a high impact performance problem in a state of the art EBR algorithm using a new visualization technique called a *timeline* graph.
- I propose an improved version of DEBRA [5] that addresses the aforementioned problem.
- I apply the same technique used to improve DEBRA to a simpler EBR algorithm called *Token EBR*, improving its performance substantially.
- Finally, I identify several performance problems in a state of art memory allocator, and propose and evaluate solutions to these problems. The solutions result in significant performance improvements in some workloads.

Chapter 2

Background

In this chapter, I briefly present background information on several memory reclamation algorithm and memory allocators.

2.1 Memory Reclamation

In a single threaded system, reclaiming memory is relatively easy, even in languages like C++ that require programmers to explicitly free objects. However, reclaiming memory in concurrent data structures is more difficult. If a thread calls `free()` to reclaim an object while another thread can still access that object, this can result in the program crashing.

2.1.1 Epoch Based Reclamation (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 [5] introduced DEBRA, which has been shown to be one of the fastest EBR algorithms. The improved algorithms presented in this thesis are compared with DEBRA experimentally.

2.2 Memory Allocation

Several allocators are studied in this thesis. I define an allocator as the library that implements `malloc()` and `free()`. Allocators use different amounts of memory and offer different levels of performance. Such details are discussed alongside experimental evidence in Chapter 4 where I justify the choice of which allocator I build upon in my work. Here I give a brief description of how a typical allocator works, and also describe some high level architectural differences among popular allocators.

2.2.1 Hoard

Hoard [1] is the earliest allocator I am discussing in detail, and it is one of the first allocators specifically designed for multiprocessor systems[1]. It uses per-processor heaps to improve performance, and a global heap to allow excess objects freed at one processor to be a reused at another processor.

At a high level, Hoard divides memory into superblocks, which are 2MB in size. One design goal in hoard was to avoid allocator induced false sharing, which occurs when multiple threads allocate objects in the same cacheline. The authors attempted to do this by having each thread memory map its own superblocks to allocate objects from.

When superblocks are nearly empty, they are moved from a per-processor heap to the global heap. This is ostensibly done to avoid wasting memory in producer consumer scenarios, where one thread repeatedly allocates objects and sends them to another thread who frees them (although it is not clear that this actually prevents unboundedly more memory than is necessary from being used in such scenarios).

Since threads can move their superblocks to the global heap before they are completely empty, and those superblocks can then be reused by other threads, it is possible for superblocks to contain objects allocated by multiple threads. So some allocator induced false sharing may still occur.

2.2.2 TCmalloc

TCmalloc (which is short for *thread caching malloc*) was developed by Google researchers, and released in 2005 [11]. Huge objects are allocated an appropriate number of whole pages

[1] Here, multiprocessor refers to the same thing as multicore (as opposed to multi-socket).

from a central page heap. Small objects are allocated in 170 different size classes. For each size class, there is a *central cache* and a *thread local cache* for each thread. Each central cache is protected by a lock.

When a thread allocates memory, it first tries to serve the allocation from its thread cache (for the appropriate size class). If there is no available object in the thread cache, it repopulates the thread cache using a batch of objects from the central cache (for that size class). When a thread frees an object, if the thread local cache is full, a batch of objects is moved from the thread local cache to the central heap. Accesses to the global heap can result is substantial contention in systems with many cores.

2.2.3 JEmalloc

JEmalloc [10] is an extremely popular allocator. It is the default allocator for FreeBSD, Firefox, and many other large software packages, and it has seen active development by large companies such Facebook.

Whereas TCmalloc uses private per thread caches, JEmalloc uses *arenas*, which *can be* shared among threads, but the number of arenas is made fairly large in an effort to reduce contention among threads. Specifically, by default there are $4n$ arenas, where n is the number of processors. A given thread is essentially hashed to an arena, and each arena is protected by a lock.

Many size classes reside in an arena. An arena has one chunk that it allocates 4KB pages from. If the chunk becomes full, a new chunk is `mmap`ed. When an allocation happens in size class x, it first checks the free list for that size class. If the free list is empty, it checks if there is an active page for that size class (containing some free space). If there is a free object, JEmalloc bump allocates from the page. If the page is full (no free space), JEmalloc bump allocates one or more new pages from the arena's chunk (and makes those the active page(s) for this size class).

2.2.4 Supermalloc

More details are given for Supermalloc [19] than for the other allocators, since part of my thesis work involved modifying Supermalloc.

Huge object sizes ($\geq$ 1MB) are mapped and unmapped directly, and metadata for them is stored in a large array. Small objects are allocated from and freed to Supermalloc's *object*

caches, which behave like free lists (or, more specifically, collections of free lists). Levels of object caches in Supermalloc are as follows:

- For each size class, there are thread local, processor local, and global object caches.
- Each thread local cache consists of two lists, each containing 8KB of objects.
- Each per-CPU cache consists of two lists, each containing 1MB of objects.
- Each global cache contains n linked lists, where n is the number of processors, each containing 1MB of objects.

There is also a *chunk* for each size class, where new objects are bump allocated when all relevant object caches are empty. When one chunk is completely filled (with no room to allocate new objects), a new chunk is allocated.

When `malloc()` is invoked by thread p, p first inspects the thread local cache of the appropriate size class to see if there is a free object. If so, it allocates from the thread local cache without having to acquire any locks.

Otherwise, p inspects the appropriate per-CPU cache (for the processor[2] it is running on), which requires locking. If the per-CPU cache is non-empty, then p moves many free objects from the per-CPU cache to its thread local cache (enough to fill half of the thread local cache), then p allocates an object from its thread local cache. Note that moving 8KB of objects from the per-CPU cache to the thread local cache involves traversing many nodes in a linked list at the per-CPU cache.

If the per-CPU cache is also empty, p inspects the global cache (for this size class) which, again, requires locking. If this global cache is non-empty, then p moves one full 1MB list of free objects from this global cache to p's per-CPU cache (and then allocates as above). Moving objects from the global cache to the per-CPU cache requires locking both caches.

Finally, if the global cache is empty, thread p acquires a global lock that is specific to this size class, and bump allocates one new object. It checks if there is any non-empty chunk that was previously `mmap`ed. If there is a non-empty chunk, p allocates an object from the chunk. If there is no non-empty chunk, it first `mmap`s a new chunk and allocates an object from the new chunk.

`free()` is similar to `malloc()`. p first tries to free to its thread local cache (for the appropriate size class). If the thread local cache is full, and there is room to accommodate

[2]I.e., core, or logical processor on a system with hyperthreading

this list in the per-CPU cache, then it pushes half of the cache up to its per-CPU cache (merging one 8KB list into an existing 1MB list). However, if there is insufficient space in the per-CPU cache to accommodate the thread local cache's 8KB list, then the global cache is checked to determine whether a 1MB list can be moved from the per-CPU cache to the global cache. If this cannot be done, because there is insufficient space in the global cache, then no lists are moved, and the object is simply freed directly to the page it was allocated from. This entails locking that page and modifying the metadata stored there.

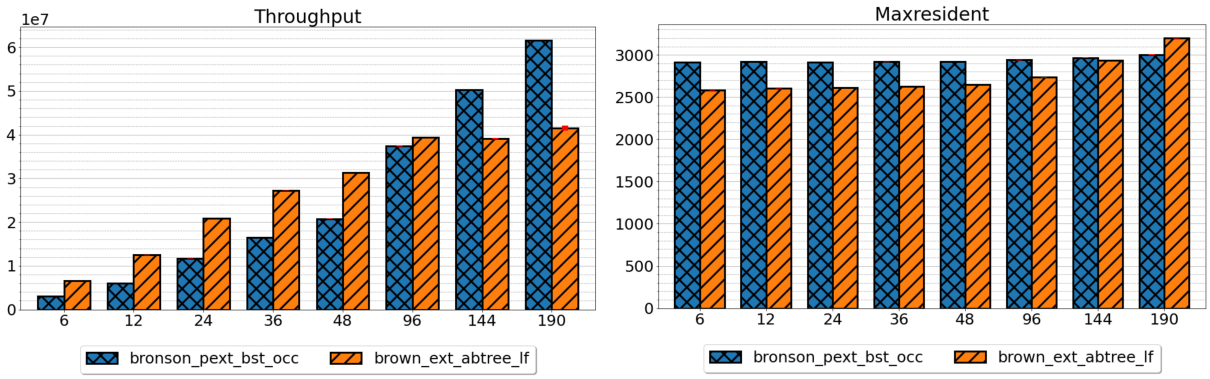
Chapter 3

Memory Reclamation

3.1 Motivation

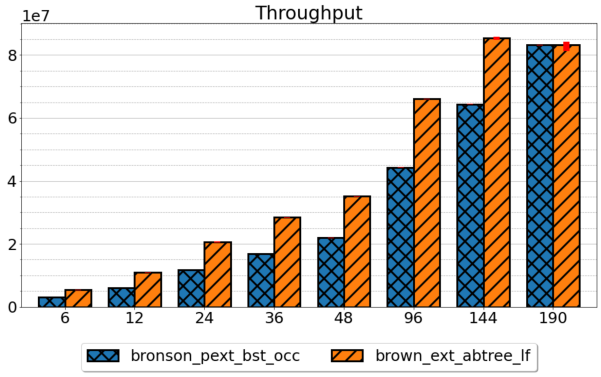
In running experiments on concurrent tree data structures, I observed that on large scale NUMA systems with four processor sockets, some data structures that scale similarly when running on a single socket scale drastically differently when running on four sockets. For example, consider an AVL tree by Bronson et al. [2] that uses optimistic concurrency control (hereafter, OCCtree), and a concurrency friendly variant of a B-tree by Brown [3] that uses lock-free techniques (hereafter, ABtree). I 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 $n \in \{6, 12, 24, 36, 48, 96, 192\}$, 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. This experimental methodology is similar to that in other papers that study concurrent memory reclamation (e.g., [5, 28, 26, 25]).

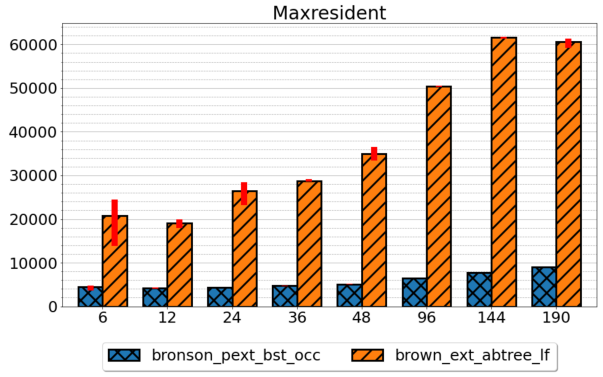


(a) Performance: DEBRA

(b) Peak memory usage (MiB): DEBRA



(c) Performance: leaky



(d) Peak memory usage (MiB): leaky

Figure 3.1: Performance and peak memory usage w/JEmalloc, for OCCtree and ABtree, using DEBRA vs. leaking memory.

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.)[1]

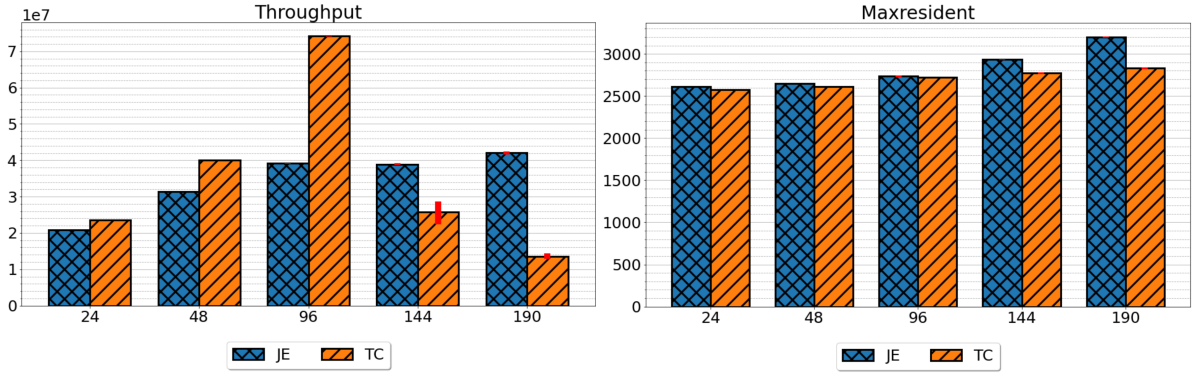
Results Figure 3.1a shows the results of this experiment. Each data point shows the average throughput over three trials, and the minimum and maximum throughput over these three trials is shown using error bars. Both algorithms scale well up to 48 threads, but the ABtree stops scaling above 96 threads, whereas the OCCtree continues to scale (albeit not as well as it did up to 96 threads). 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). Figure 3.1b shows the amount of memory used on average at each of these data points. Interestingly, peak memory usage for the ABtree is not much higher than for the OCCtree, although it is *somewhat* higher at high thread counts, where its scaling is diminished. One might expect the ABtree to have substantially higher peak memory usage, but this would not be correct. One possible explanation is that the memory reclamation algorithm simply spends much more time freeing garbage to maintain a low peak memory usage.

To confirm this hypothesis, I disable memory reclamation for both data structures, and

[1] I have included, in the system description, all of the details identified by Grammoli [13] as crucial for reproducing shared memory multicore experiments. That paper provides, among other things, a concise study of the impact of thread pinning and turbo boost on performance. In my experiments turbo boost is enabled, since it is typically enabled on real systems, and it has a substantial impact not only on low thread counts, but also on high thread counts.

On the experimental system, when the benchmark was run on one or two cores, the clock speeds at those cores were 3.5-3.6GHz. 4 cores ran at 3.4GHz, 8 at 3.3-3.4GHz, 12 at 3.2-3.3GHz, 24 at 2.8GHz, and the all-core turbo speed was 2.8GHz, despite the base frequency of the processor being 2.1GHz. This can make some of the scaling results in this thesis appear somewhat worse than they would be if turbo boost were disabled, however the absolute performance results at every thread count are better than they would be in that case.

simply leak memory, to see whether this closes the performance gap between them. The results in Figure 3.1c largely confirm this hypothesis. It is clear that the ABtree allocates much more memory (Figure 3.1d), 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, or possibly both. Note that the scaling of the OCCtree at higher socket counts also appears to have been primarily due to overheads associated with reclaiming memory, as its scaling appears to be nearly linear in the number of sockets in Figure 3.1c.



(a) Performance

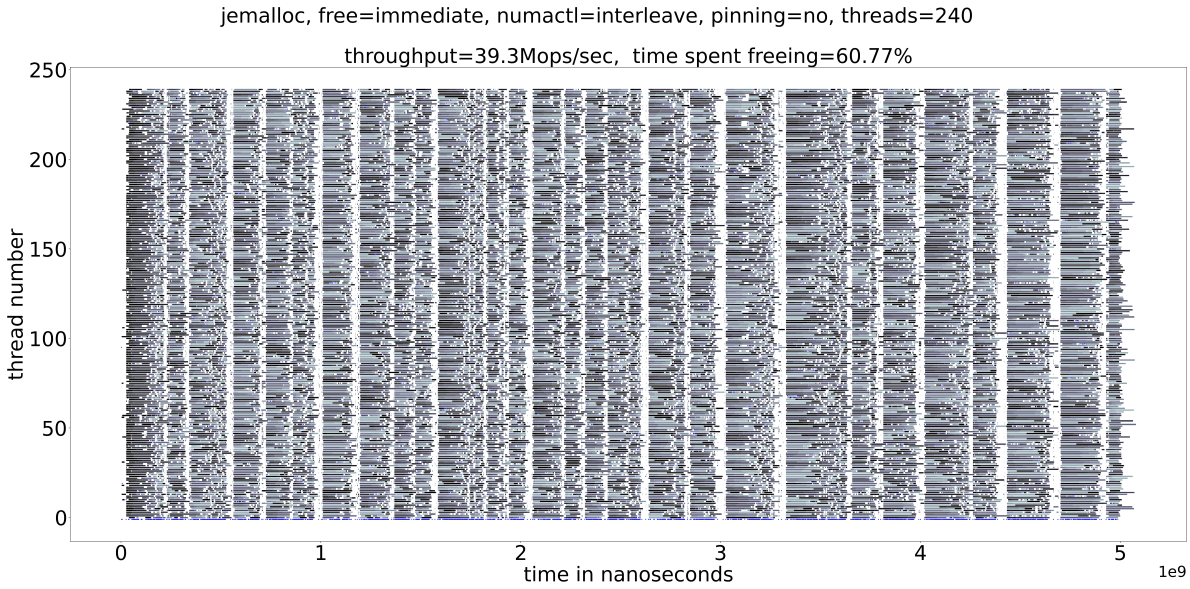
(b) Peak memory usage (MiB)

Figure 3.2: Performance and peak memory usage w/TCmalloc, for ABtree, using DEBRA

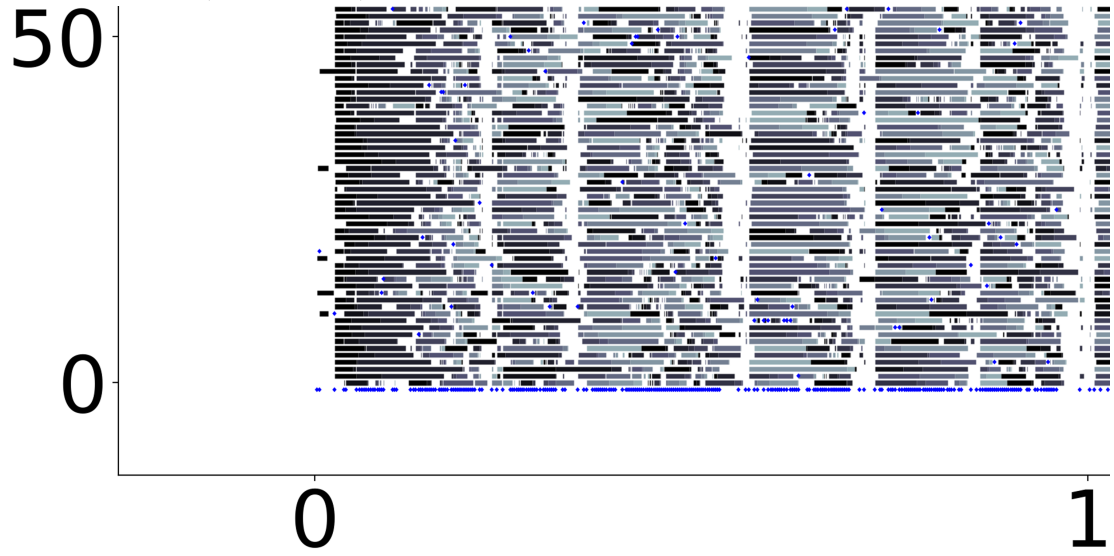
Replicating the Result with TCmalloc To understand whether this is just the result of an interaction between JEmalloc and DEBRA, I also repeated the same experiment with TCmalloc, another widely used allocator (Figure 3.2). As one can see in Figure 3.2, TCmalloc scales well up to two sockets but performs poorly when running on more than two sockets, suggesting that the problem may originate in DEBRA. So, I set out to modify DEBRA to try to fix this problem.

3.2 Investigating the Reclamation Overhead

Crucially, although DEBRA maintains a similar memory footprint for all of the thread counts in our experiment (Figure 3.1b), the performance of the ABtree substantially flattens at high thread counts in Figure 3.1a. This suggests that DEBRA is struggling to keep



(a) X-axis = time since first epoch, Y-axis = thread, colour = epoch, blue dots = epoch change, between boxes (white space) = time spent accessing ABtree



(b) Zoomed-in version of Figure 3.3a

Figure 3.3: Timeline graph showing how much time threads spend freeing nodes as epochs change in the ABtree with JEmalloc.

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. To determine if this is caused by thread delays, I visualized the behaviour of threads, specifically the time spent freeing objects, in a graph that I call a *timeline*. I have not seen timeline graphs used elsewhere in the literature, but my experience with them suggests they are a very useful tool for investigating performance problems caused by thread delays. My supervisor and I implemented a highly efficient mechanism to allow threads to record data in memory to be printed to files at the end of an experiment, with very little impact on performance. (I did not measure any significant impact on performance when recording up to 100,000 timeline events per thread.)

Figure 3.3 depicts a timeline graph showing how much time threads spend freeing nodes as epochs change in the ABtree with JEmalloc (and an enlarged region of the graph is shown in part (b) of that figure). Rows represent different threads, 240 running threads are shown in the figure. Boxes represent time spent freeing batches of old objects that were removed from the data structure in previous epochs. Boxes are coloured differently to make it clear which epoch a thread is in when it is freeing. Blue dots represent a thread successfully changing the global epoch number. The blue dots are projected at the bottom of the graph to show epoch changes made by all threads together. (This makes it easier to identify periods of time during which the epoch is not changed by any thread.) This figure depicts five seconds of execution time in the measured part of an experiment.

One can see that there are some *good* periods of time during which all threads generally spend relatively little time freeing small batches. There are also some vertical columns of white space followed by a flurry of activity wherein threads take a relatively long time to free their next (few) batch(es) of nodes.

When a *bad* period begins, more and more threads begin having trouble freeing all of their objects before the next epoch begins. To understand why, one needs to understand a little bit more about DEBRA. In DEBRA, while a thread is freeing a batch of objects, the epoch can change only once. (For it to change a second time, the thread would need to finish freeing its batch, and participate in a sort of barrier synchronization across all threads.) If a batch is big enough, this can have the effect of delaying an epoch change in the near future. Since objects can only be freed when the epoch changes, this effectively reduces how often batches are freed, which increases average batch sizes. The net effect is that more and more time is spent freeing. As more time is spent freeing, less time is spent actually updating the data structure and creating garbage to be freed, so threads eventually catch up, and enter a good period once more.

Figure 3.3a also shows the total time spent freeing. In this experiment, 60.77 percent

of the total execution time is spent freeing objects.

3.3 Amortized Batch Free: Improving DEBRA

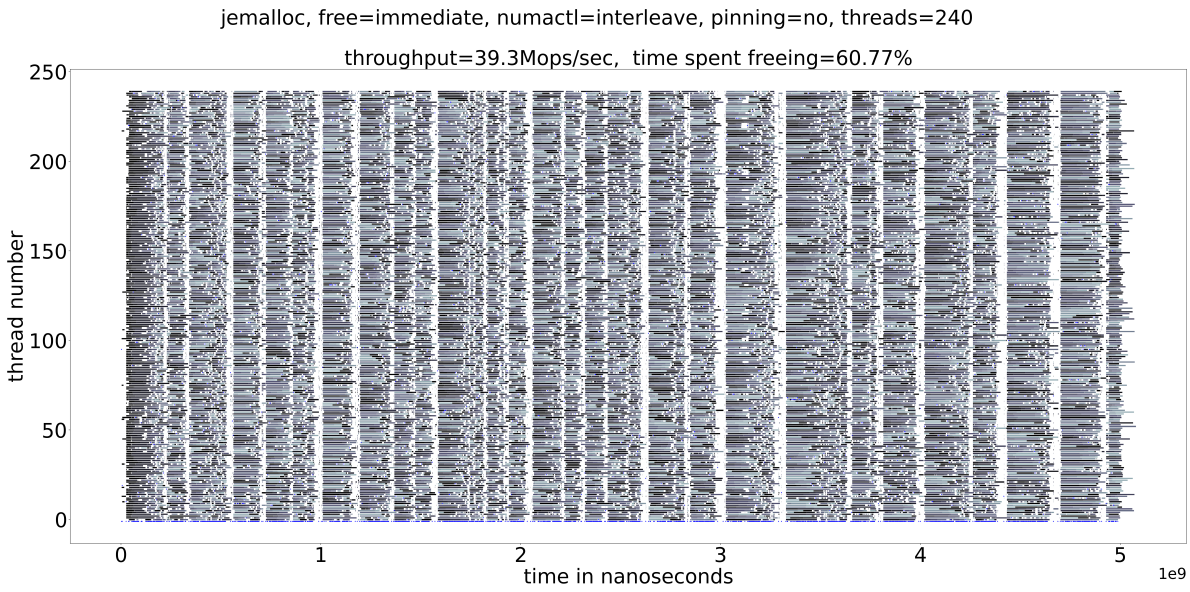
The performance problem in DEBRA seems to be caused by the batched free behaviour. Batching is inherent in EBR, but once a batch of nodes has been identified by EBR as "safe to free," one does not necessarily need to free them immediately as a batch. To improve DEBRA, I had the idea to amortize the cost of freeing over time, by gradually freeing one object at a time, each time an operation is performed on the data structure. I call this amortized batch freeing.

Figure 3.4a and Figure 3.5a show the timeline graphs of the original DEBRA and DEBRA with my amortized batch freeing modification. Note that Figure 3.4a is exactly the same as Figure 3.3a, but I have reproduced the graph here to make it easier to compare the two pairs of graphs in Figure 3.4 and Figure 3.5. Most of long batch frees shown in the original DEBRA are eliminated when amortized batch freeing is performed.

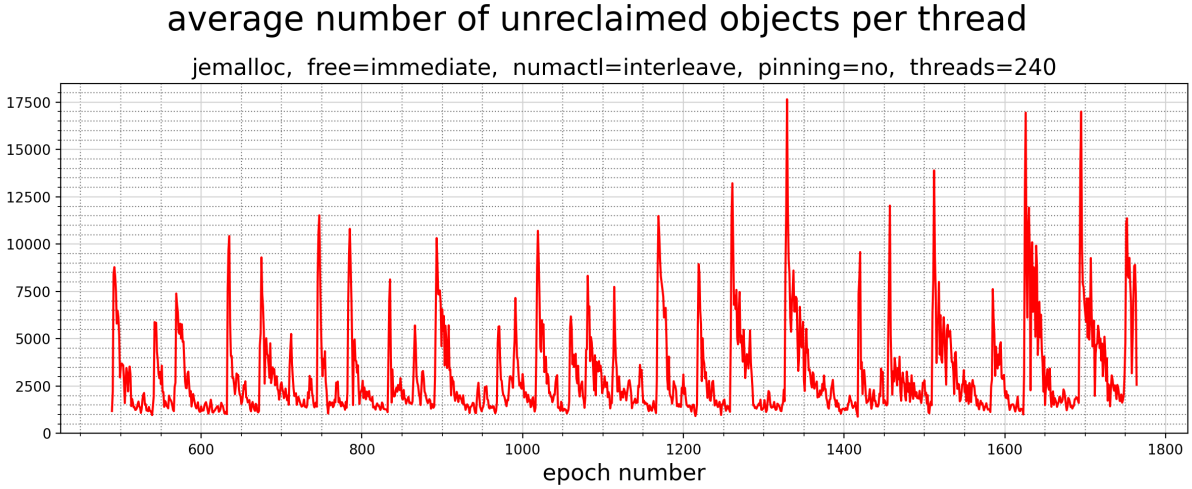
The impact of this change on the average number of unreclaimed objects per thread (i.e., the amount of garbage waiting to be freed) is shown in Figure 3.4b and Figure 3.5b. The amounts of unreclaimed objects in Figure 3.4b and Figure 3.5b are similar, but in the original DEBRA, the number of unreclaimed objects drops much more sharply as a batch is immediately freed.

Looking at the pair of graphs in Figure 3.4, one can see that the sharp drops in the number of unreclaimed objects precipitate large delays in the timeline graph. On the other hand, the pair of graphs in Figure 3.5 seem to suggest that the gradual change in the number of unreclaimed objects does not cause the same issues in the timeline graph.

Moreover, whereas 60.8 percent of the total execution time in the original DEBRA is spent freeing objects, resulting in a total throughput of 39.3 million operations per second (Figure 3.4a), the version with amortized batch freeing spends only 30.6 percent of the total execution time freeing objects, resulting in a total throughput of 91.0 million operations per second (Figure 3.5a). From these results, it seems that amortized batch freeing can yield substantial performance improvements by reducing the overhead of freeing nodes. Crucially, note that the algorithms that performs amortized batch freeing spends half as much time freeing nodes, even though it allocates and frees more than twice as many nodes (because of its increased throughput). This suggests the improvement in the overhead of managing memory is on the order of 400 to 500 percent.

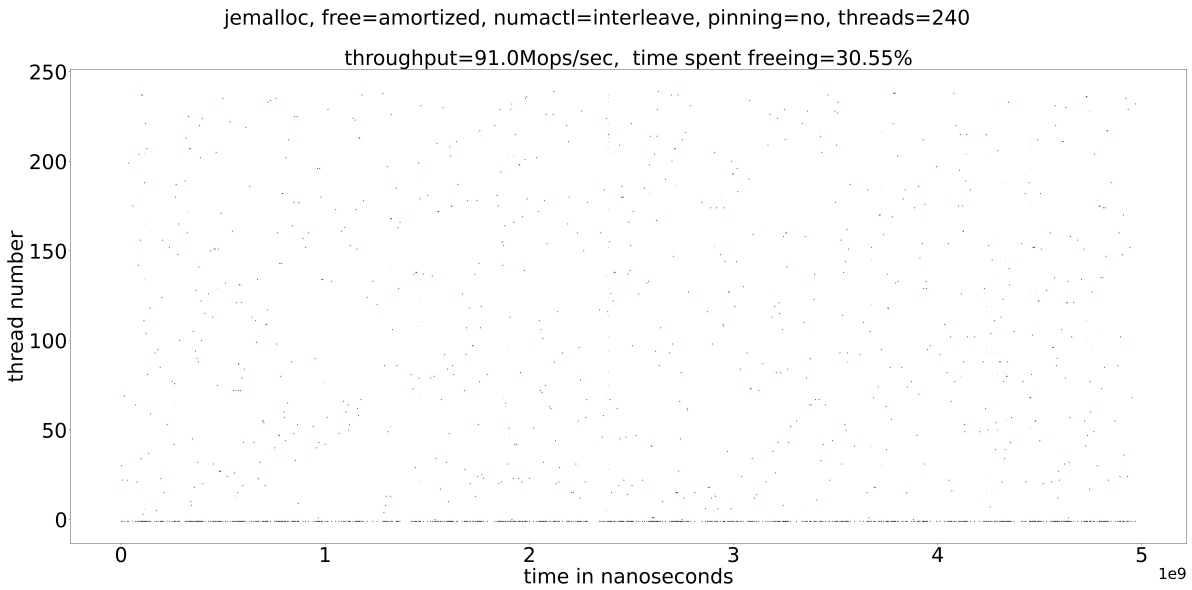


(a) X-axis = time since first epoch, Y-axis = thread, colour = epoch, blue dots = epoch change, between boxes (white space) = time spent accessing ABtree

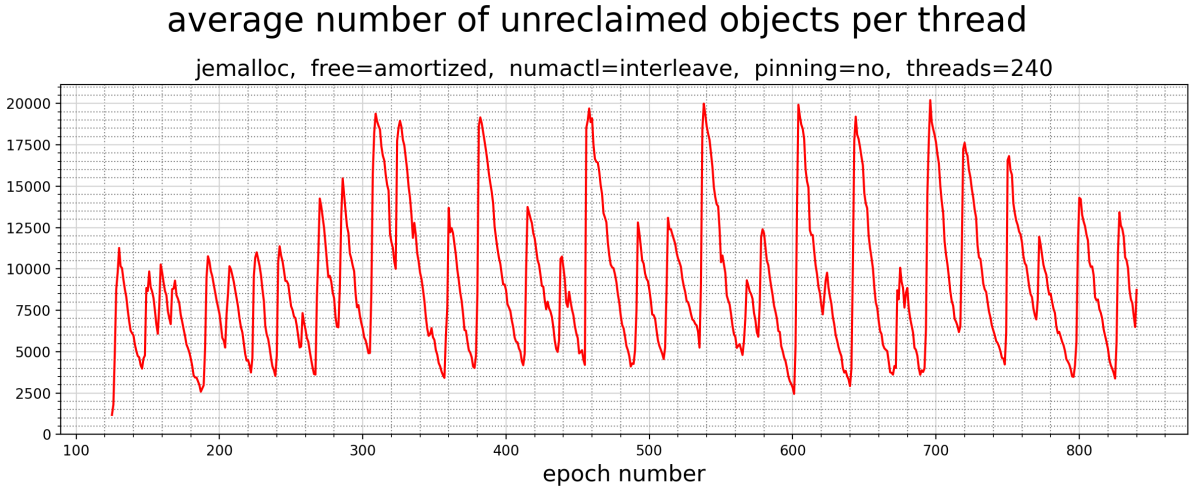


(b) X-axis = epoch number, Y-axis = average number of unreclaimed objects per thread

Figure 3.4: Timeline graph and average number of unreclaimed objects per thread of the original DEBRA in the ABtree with JEmalloc.

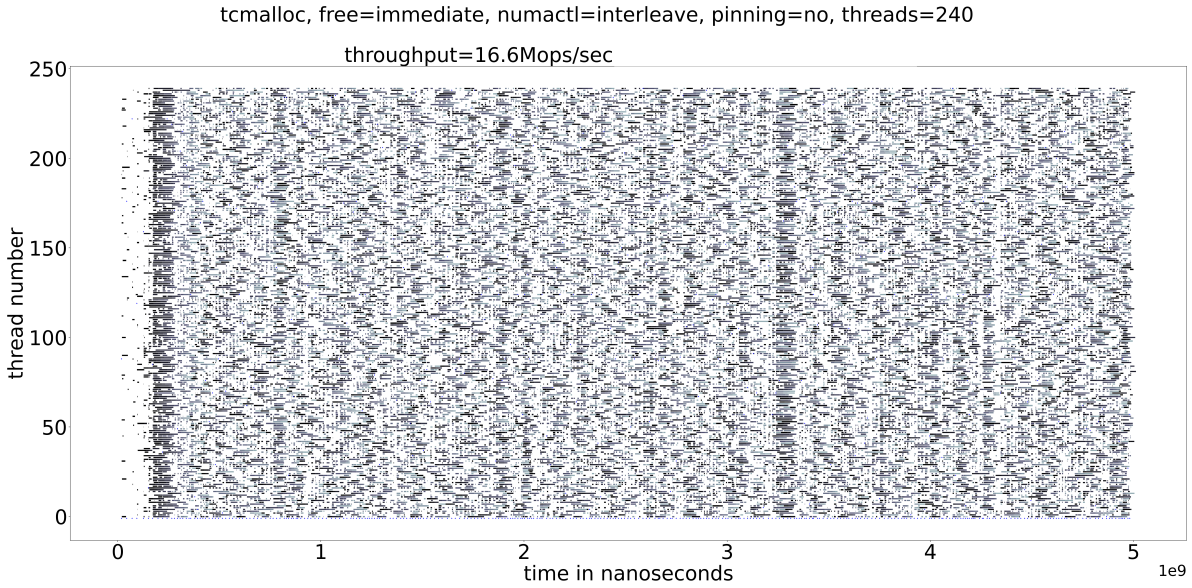


(a) X-axis = time since first epoch, Y-axis = thread, colour = epoch, blue dots = epoch change, between boxes (white space) = time spent accessing ABtree

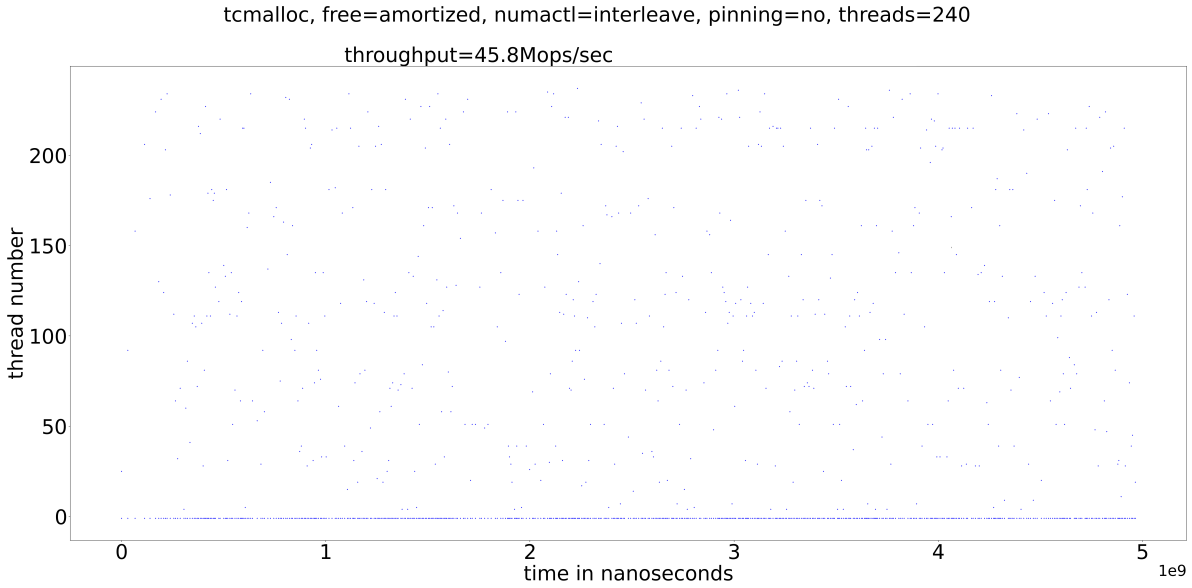


(b) X-axis = epoch number, Y-axis = average number of unreclaimed objects per thread

Figure 3.5: Timeline graph and average number of unreclaimed objects per thread of amortized batch freeing DEBRA in the ABtree with JEmalloc.



(a) X-axis = time since first epoch, Y-axis = thread, colour = epoch, blue dots = epoch change, between boxes (white space) = time spent accessing ABtree



(b) X-axis = time since first epoch, Y-axis = thread, colour = epoch, blue dots = epoch change, between boxes (white space) = time spent accessing ABtree

Figure 3.6: Timeline graph comparing the original DEBRA and amortized batch freeing DEBRA in the ABtree with TCmalloc.

Once again, to demonstrate that this problem is not specific to JEmalloc, I repeated the same experiment with TCmalloc. As one can see in Figure 3.6, amortized batch freeing also improves the performance of DEBRA drastically (by almost 300 percent) when TCmalloc is used.

3.4 A Simpler EBR Algorithm

This raises the question of whether amortized batch freeing would also improve performance in a more straightforward EBR algorithm. There are two reasons to consider this question. First, I want to understand if this is an optimization that is more widely applicable, both to other EBR algorithms, and perhaps to other algorithms. Second, if I can make a simpler algorithm perform as well or better than DEBRA, then that would be a worthwhile contribution.

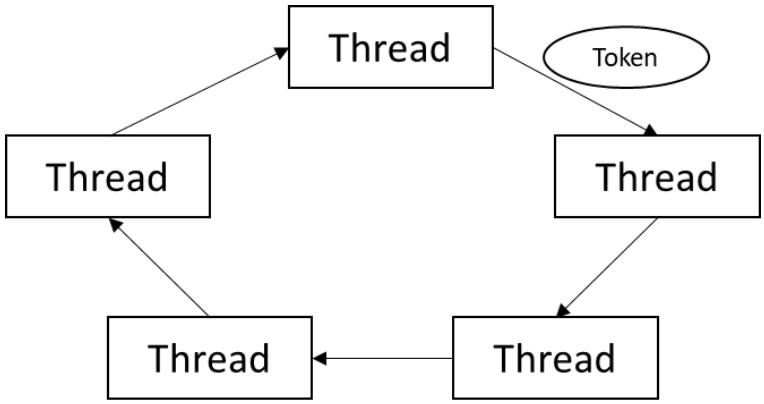
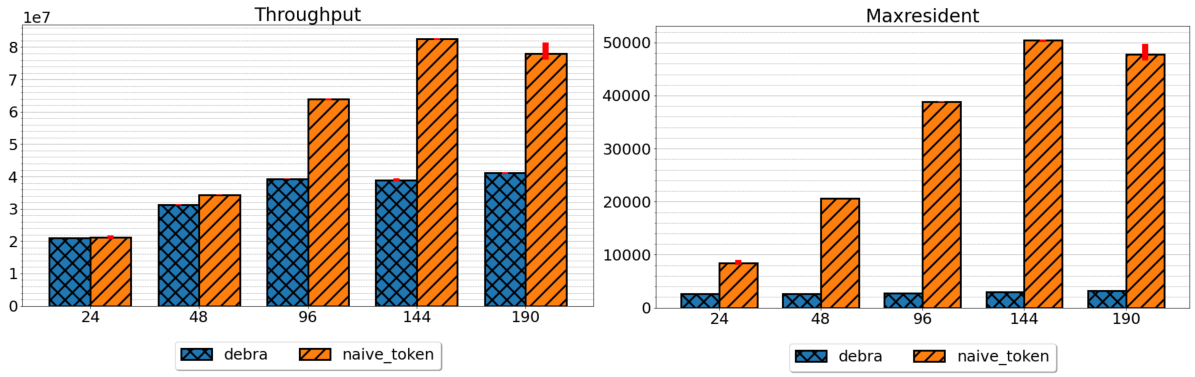


Figure 3.7: Diagram of Token EBR

One conceptually simple way of implementing epoch based reclamation is to arrange threads on a ring, and have them repeatedly pass a token around the ring (Figure 3.7). Whenever a thread begins a new data structure operation, it checks whether it currently owns the token, and if so, it passes it along to the next thread. This idea, which I call Token EBR, is described in a thesis by Tam [27], however only a high level sketch of the algorithm is provided.

To understand how this algorithm works, 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 removed from the data structure before the interval I begins. Each thread that begins a data structure operation before O is removed 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 removed from the data structure. So, no thread can access O, and O is safe to free. More broadly, every object that was removed from the data structure before I is safe to free after I.



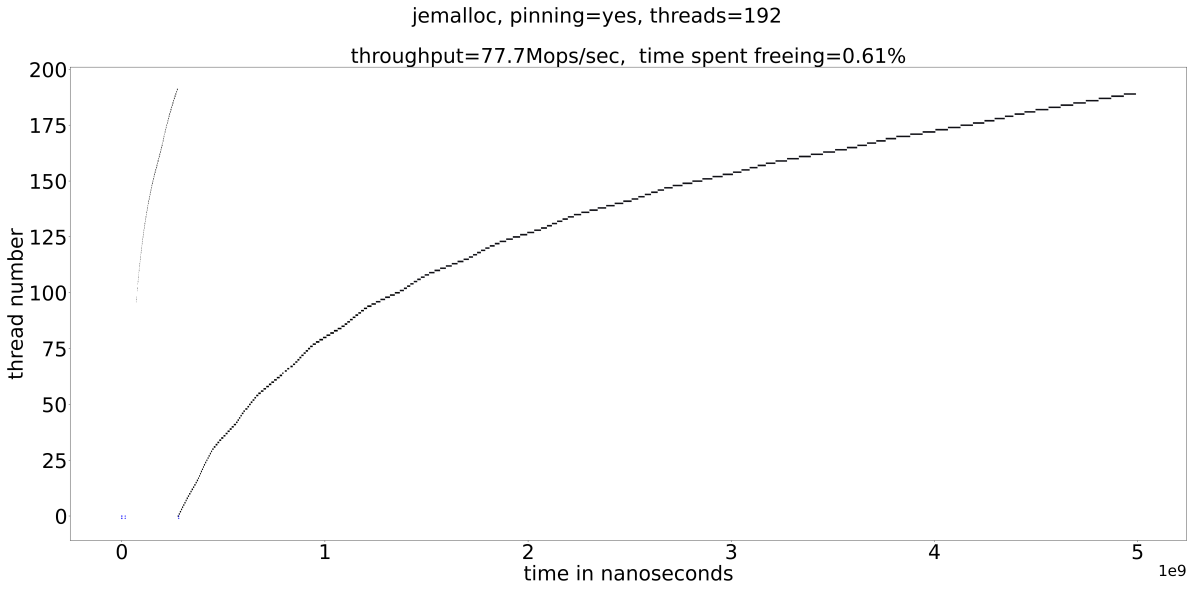
(a) Performance

(b) Peak memory usage (MiB)

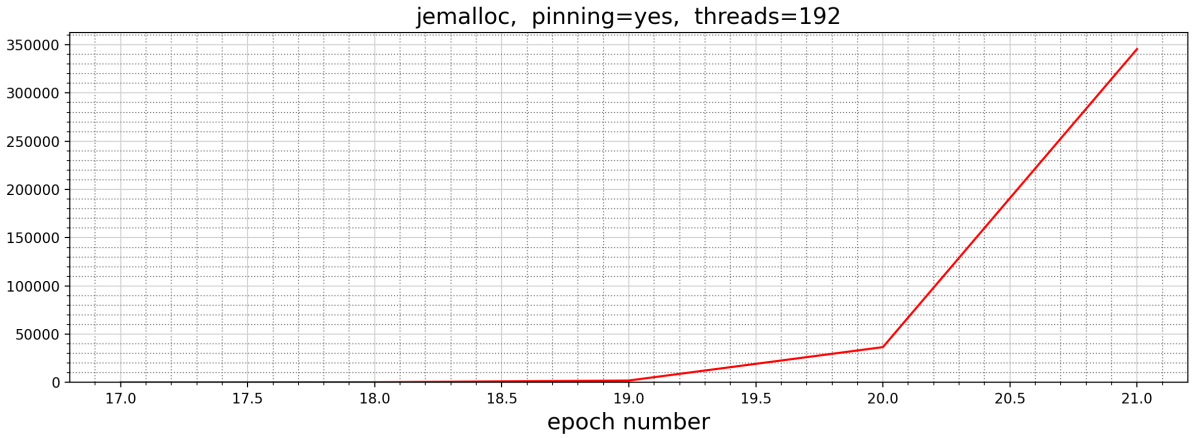
Figure 3.8: Performance and peak memory usage w/JEmalloc, for ABtree, using Naive Token EBR

Naive Token EBR I produced a *Naive* implementation of Token EBR as follows. Each thread maintains two bags of objects that are waiting to be freed, a *new* bag and an *old* bag. Whenever a thread removes an object from the data structure, it places it in its *new* bag. Whenever a thread begins a new data structure operation, it checks whether it has received the token. If so, it first frees all objects in its *old* bag, and swaps its *old* and *new* bags. After the swap, the *new* bag is empty, and the *old* bag contains objects waiting to be freed the next time the token is received. Finally, the thread passes the token to the next thread, and then proceeds to perform the data structure operation.

Like DEBRA, this Naive Token EBR algorithm does not perform well with JEmalloc (Figure 3.8a). This initial variant of the algorithm has some problems that limit its performance. Since a thread holds the token until it finishes freeing all of the objects in its *old* bag, other threads are not able to begin freeing their objects until this thread is finished. Those other threads are not prevented from accessing the data structure during this time,



(a) Timeline graph



(b) Average number of unreclaimed objects per thread

Figure 3.9: Naive Token EBR

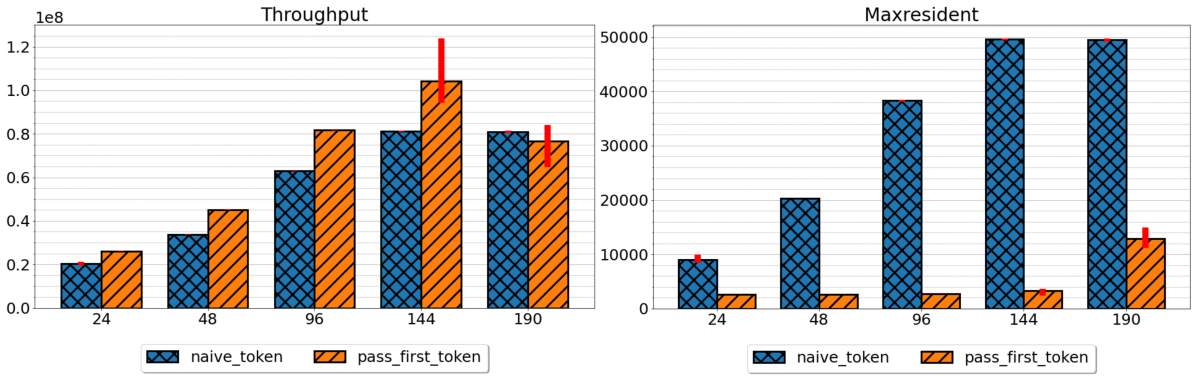
and they can modify the data structure, creating new garbage. (But, they cannot free that newly created garbage.)

If it takes a long time for a thread to free its objects, other threads may accumulate a large amount of garbage while they wait. In my experiments, I found that the amount of garbage quickly spirals out of control (Figure 3.8b). To see why, consider the following example. Suppose all threads start with the same amount of garbage that they are waiting to free, and the first thread begins freeing its garbage. While this thread is freeing, the $n-1$ other threads all continue to accumulate garbage. When the second thread receives the token, it has more garbage to free than the first thread did. Moreover, while the second thread is freeing, the third thread accumulates more garbage than either of the first two threads had, and so on.

This problem is self perpetuating, since in the time it takes for the token to travel around the ring, the first thread can accumulate far more garbage than it had when it previously freed. (And after the token passes around the ring again, the first thread will have even more garbage still.) I call this the *garbage pile up* problem. In Figure 3.9a, it is easy to see that only a single thread frees objects at a time (and the timeline graph looks like it consists of a pair of curves). Interestingly, Naive Token EBR is actually substantially faster than DEBRA, which came as a shock to me, since freeing nodes is such a highly serialized process. The reason for this is made clear by the graph showing its enormous peak memory usage (Figure 3.8b).

In essence, it takes so long to free batches of objects in Naive Token EBR that more than 90 percent of the execution time is spent in a single epoch (as shown in Figure 3.9a), and thus more than 90 percent of the garbage created in the execution is never freed[2]. This garbage can be seen piling up dramatically near the end of the execution in Figure 3.9b. The end result is that Naive Token EBR only spends 0.6 percent of its time freeing, but this is not because it is efficient; rather, it is investing far too little time to maintain a reasonable memory footprint. (And, the fact that the length of each successive epoch increases drastically is a major problem for this algorithm.)

[2]Recall that garbage created in one epoch can only be freed in a *later* epoch.



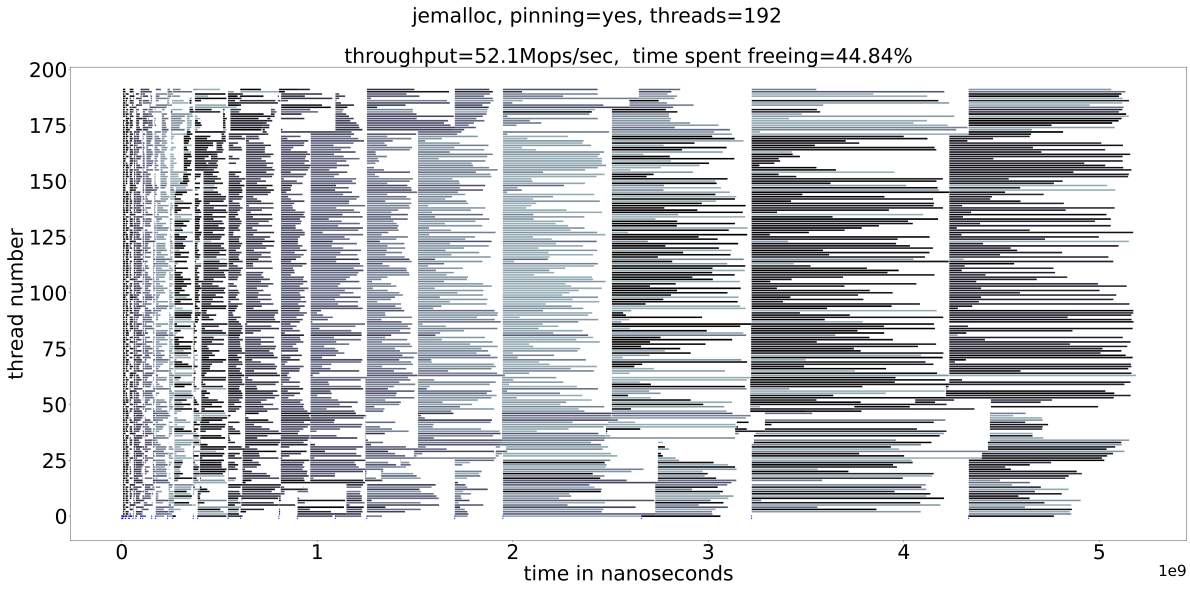
(a) Performance

(b) Peak memory usage (MiB)

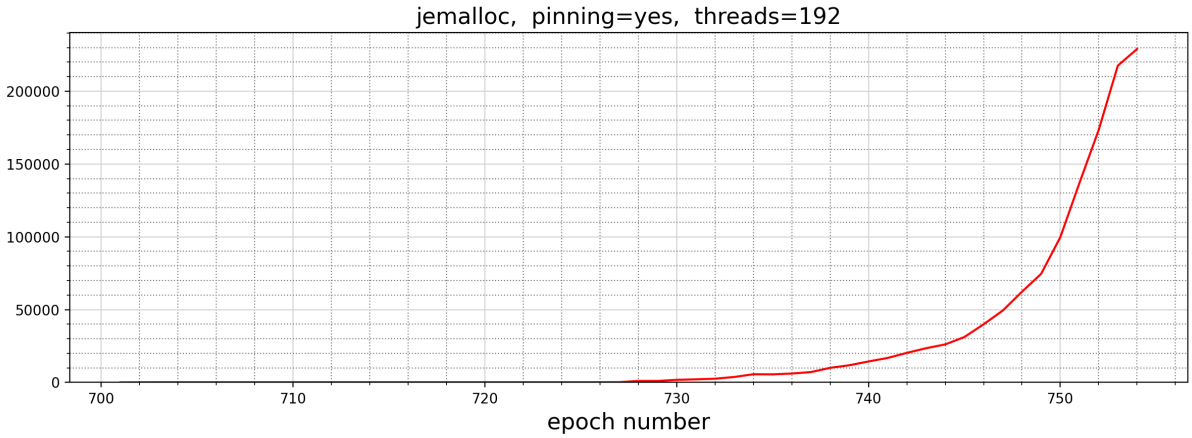
Figure 3.10: Performance and peak memory usage w/JEmalloc, for ABtree, using Pass-first Token EBR

Pass-first Token EBR One way to improve this is to realize that the token can be passed as soon as a thread receives it. The *receipt* of the token tells the thread that it is safe to free the objects in its *old* bag. It does not need to continue to hold the token while it frees. I call this algorithm *Pass-first* Token EBR, because a thread first passes the token, then begins freeing its objects. Pass-first Token EBR allows threads to free objects concurrently, so it is plausible that it could avoid the garbage pile up problem.

Figure 3.10 shows that Pass-first Token EBR both improves the performance of Naive Token EBR, and is less susceptible to the garbage pile up problem, but garbage pile up still occurs at high thread counts. It is perhaps surprising at first that garbage pile up is not prevented altogether, but consider the following. Suppose a thread receives the token, passes it to the next thread, and then begins freeing a large bag of objects. If, while the thread is still freeing its objects, it receives the token again, it will hold onto the token (potentially for a long time) until it has finished freeing all of its objects, and only then pass the token to the next thread. So, if a thread ever has a particularly large bag of objects to free, it can still needlessly prevent other threads from freeing their objects. In Figure 3.11b, the amount of unreclaimed garbage can be seen to increase rapidly towards the end of the execution. Due to this increase in unreclaimed garbage, the bars in the timeline graph (Figure 3.11a) grow longer over time.

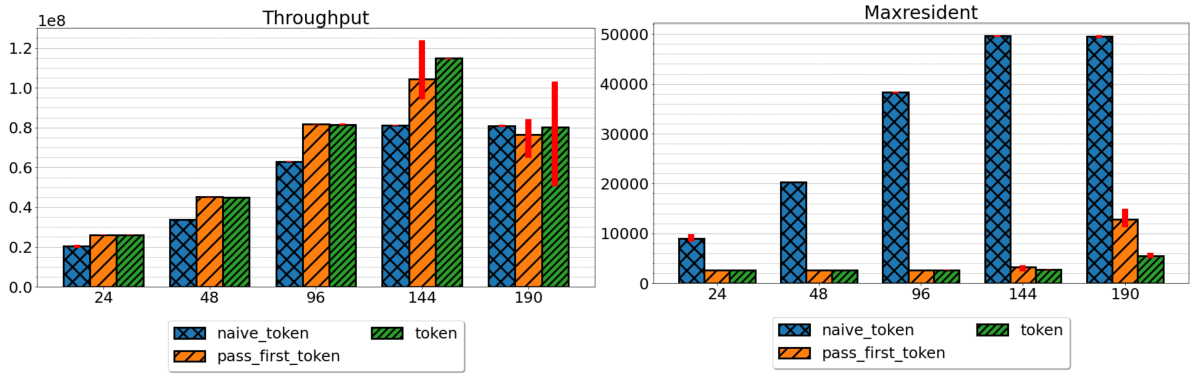


(a) Timeline graph



(b) Average number of unreclaimed objects per thread

Figure 3.11: Pass-first Token EBR



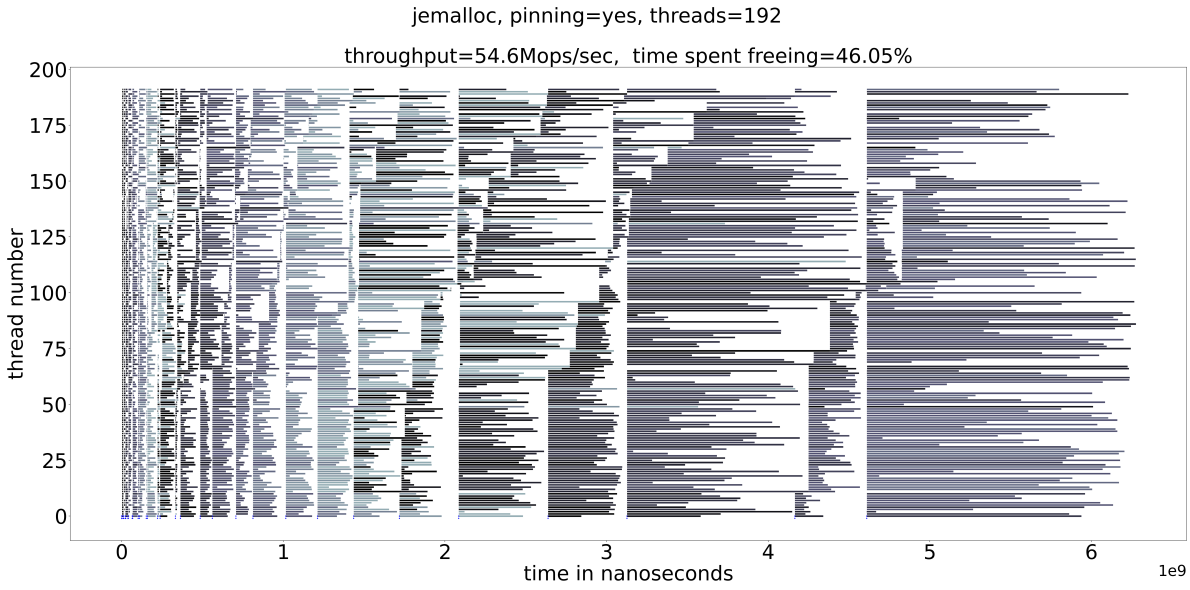
(a) Performance

(b) Peak memory usage (MiB)

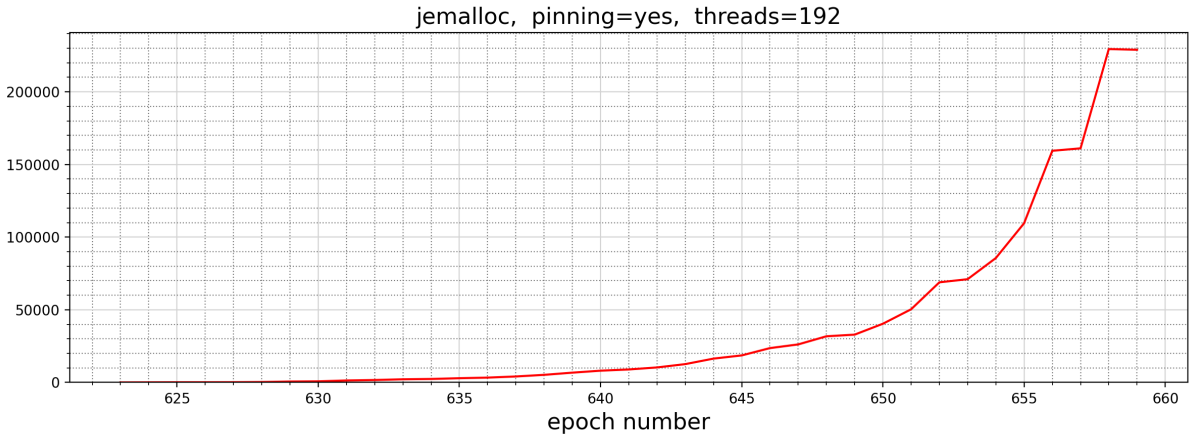
Figure 3.12: Performance and peak memory usage w/JEmalloc, for ABtree, using Token EBR

Token EBR This realization leads naturally to a superior algorithm, which I simply call Token EBR. While a thread is freeing a bag of objects, it periodically checks (in our experiments, after every 100 `free()` calls) to see if it has received the token. If so, it immediately passes the token to the next thread, and then resumes freeing its objects. This algorithm performs better than Pass-first Token EBR (Figure 3.12a), and is better able to avoid garbage pile up (Figure 3.12b).

However, the performance of Token EBR still drops significantly when running on four sockets (190 threads), and the peak memory usage is correspondingly higher than at lower socket counts. In the timeline graph in Figure 3.13a, the amount of time spent freeing individual batches still grows dramatically over time. (And, the amount of unreclaimed garbage grows correspondingly, as shown in Figure 3.13b.)

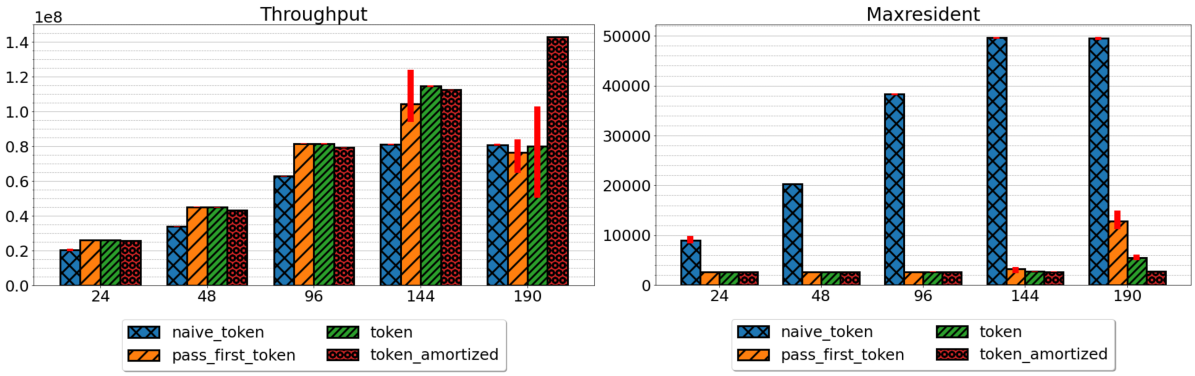


(a) Timeline graph



(b) Average number of unreclaimed objects per thread

Figure 3.13: Token EBR

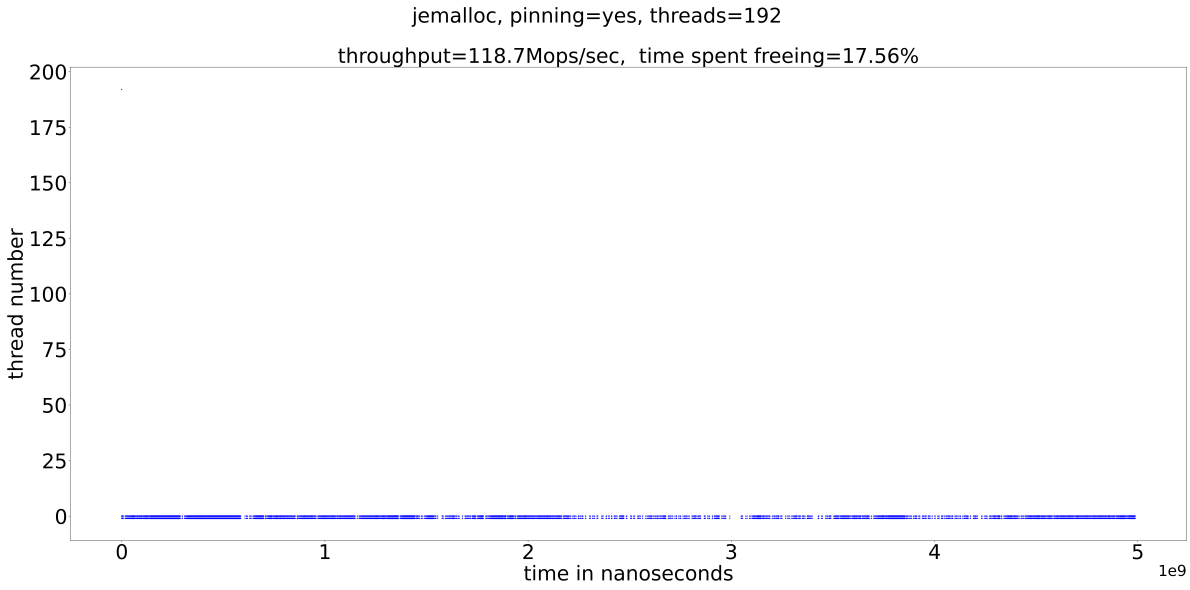


(a) Performance

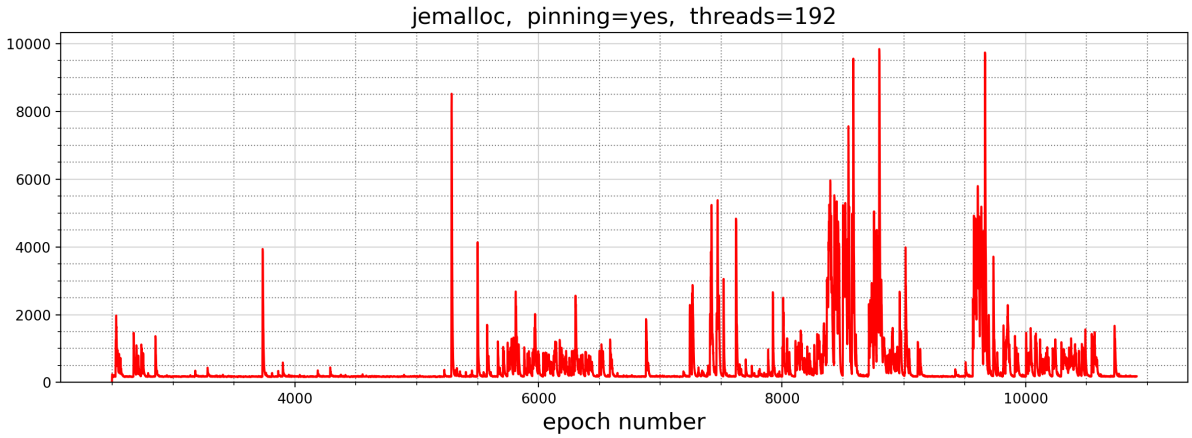
(b) Peak memory usage (MiB)

Figure 3.14: Performance and peak memory usage w/JEmalloc, for ABtree, using amortized batch freeing Token EBR

Token EBR with Amortized Batch Freeing I also tested Token EBR with amortized batch freeing. Figure 3.14 shows that amortized batch freeing offers large improvements in both performance and peak memory usage for Token EBR. The fact that such improvements are seen in two very different EBR algorithms suggests that amortized batch freeing is a general technique that can be used to improve other EBR algorithms. Amortized batch freeing stabilizes the amount of unreclaimed objects over time (Figure 3.15b), and improves the total time spent freeing garbage compared to Pass-first Token EBR. And, as expected, it transforms the timeline graph in Figure 3.15a similarly to how the timeline graphs for DEBRA were transformed.



(a) Timeline graph



(b) Average number of unreclaimed objects per thread

Figure 3.15: Amortized batch freeing Token EBR

3.5 Root Cause of the EBR Delays

Amortized freeing seems to be a feasible workaround for the problems associated with EBR batch freeing. But the question remains: what is the reason that some batches can be freed quickly (before the epoch changes), and other batches cannot?

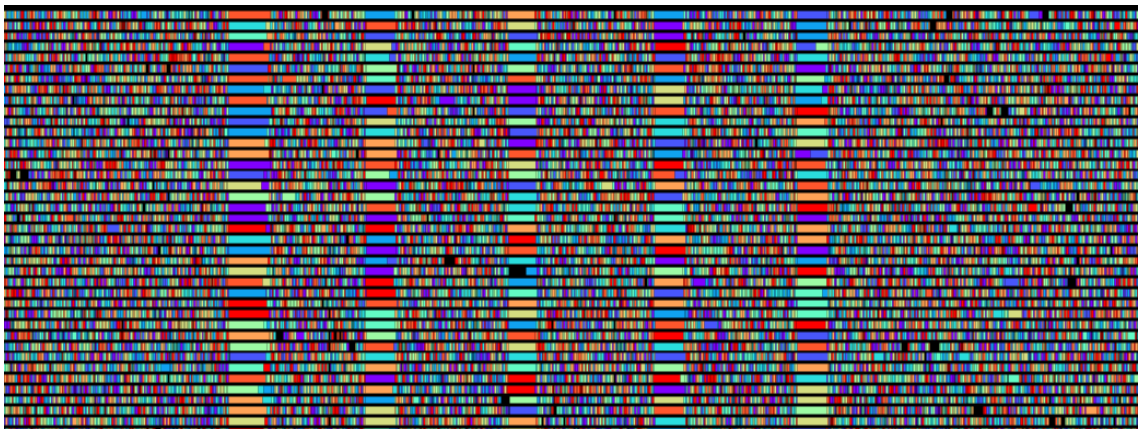


Figure 3.16: Page Migration

A Correlated, but Not Causal, Anomaly EBR algorithms are sensitive to delays, and the timeline graphs above suggest that such delays can lead to significant performance degradation. However, there is still a question of where these delays originate. In some of the timeline plots above, one can observe occasional gaps spanning all threads (i.e., columns of white space) during which the epoch is not advanced by any thread, and consequently threads cannot free memory. After such gaps, it often takes much longer to free one or more subsequent batches than it took to free previous batches. After a thorough investigation running many experiments and using Linux `perftools` extensively, I was able to determine that this peculiar visual anomaly is caused by page migration across NUMA domains (i.e., sockets).

Figure 3.16 displays the result of one of the final experiments my supervisor and I conducted to understand this issue. It depicts a cropped timeline graph where the intervals represent the time spent executing large *range query* operations, which search for many consecutive keys in the data structure at once, rather than batch free operations. These range queries have predictable lengths (accessing 200,000 objects each), and should each take approximately the same amount of time to run. Clearly, there are intervals of time during which all concurrent range queries run for an unusually long period of time.

Using `perf record` and `perf report` to sample call stacks during those time intervals, specifically, revealed that a large fraction of the cycles in such intervals are spent on kernel functions related to page migration and spinning on kernel locks.

Interestingly, these columns can often be eliminated by running with `numactl --interleave=all` to uniformly distribute all memory pages across all NUMA nodes. However, eliminating these columns does not seem to have a substantial impact on performance. So, while these columns are correlated with large subsequent batch frees, there does not seem to be a causal connection.

The Real Problem Further investigation using `perf` 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 two specific JEmalloc functions: `je_tcache_bin_flush_small` and `je_malloc_mutex_lock_slow`. According to the source code for this version of JEmalloc, when a thread invokes `free()`, it places the freed object in a buffer, and then checks whether the buffer is filled beyond a given threshold. If so, it consumes 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[3]. If the object 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 large batch, as one typically does in EBR, 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 heavyweight bookkeeping required to move objects to and from bins. In some timeline graphs with 192 threads, 30 percent or more of the total cycles were spent in `je_tcache_bin_flush_small`, and 15 percent or more of the total time in `je_malloc_mutex_lock` calls that originate from `je_tcache_bin_flush_small`.

Through further experimentation, I realized that similar effects also occur in TCmalloc. The fact that such anti-synergies between the allocator and reclamation algorithm are possible (and possibly are even common) leads naturally to a deeper study of the structure of modern allocators.

[3]In this thesis, I am 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).

Chapter 4

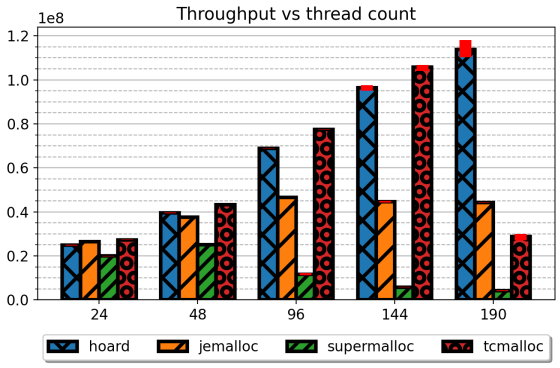
Memory Allocation

To start this chapter, I will address the question of why I chose Supermalloc to build upon in my research. The decision was based on formative experiments that I ran at the beginning of this work. In those experiments, I used a workload based on concurrent data structures to compare the performance of two of the most popular allocators: JEmalloc and TCmalloc, the venerable Hoard allocator, and the recent (at the time) Supermalloc.

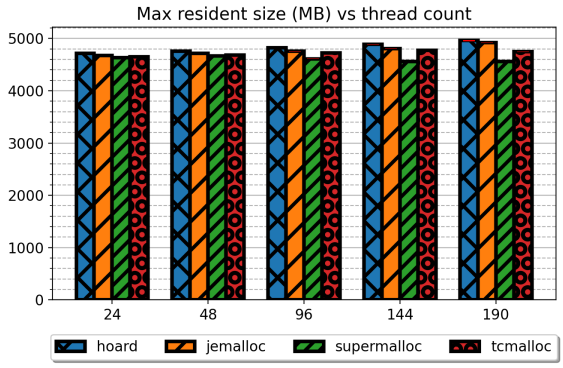
The experiments consisted of timed trials, in which n threads access a shared data structure for five seconds, and the total number of data structure operations per second is measured. Results were averaged over multiple trials. This methodology is common in the concurrent data structure literature. Five seconds may seem like a short time, but for data structures that often perform 100 million operations per second, a lot can be accomplished in five seconds. The results of these experiments can be found in Figure 4.1a and Figure 4.1b.

JEmalloc As expected for a popular allocator, JEmalloc performs quite well when using only a single socket. Its scaling is almost linear as the thread count increases. However, it scales relatively poorly as additional sockets are used, especially beyond two sockets. As was explained above in Section 3.5, batch freeing in EBR causes JEmalloc to acquires locks on remote resources for long periods of time while freeing large batches of objects. This process consumes a large fraction of the total cycles in an execution, especially on multiple sockets.

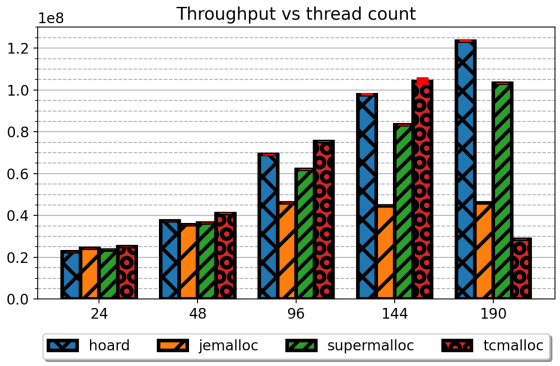
TCmalloc Like JEmalloc, TCmalloc performs poorly when running on multiple sockets. TCmalloc also has a similar performance problem to that described for JEmalloc. In fact,



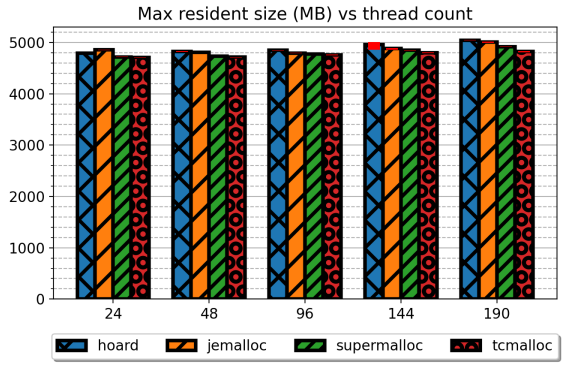
(a) Performance in a 5 second timed benchmark. X-axis: number of concurrent threads. Y-axis: data structure operations/second.



(b) Maximum resident memory size in 5 second benchmark. X-axis: number of concurrent threads. Y-axis: MiB.

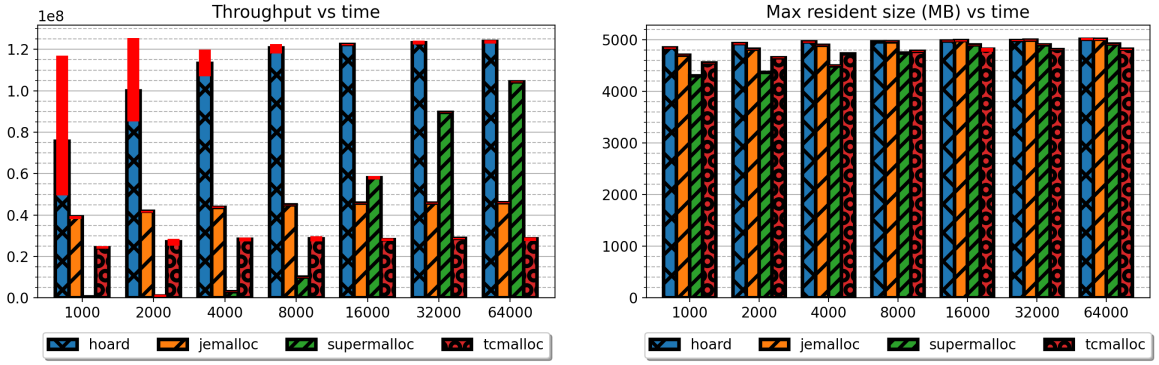


(c) Performance in a 60 second timed benchmark. X-axis: number of concurrent threads. Y-axis: data structure operations/second.



(d) Maximum resident memory size in 5 second benchmark. X-axis: number of concurrent threads. Y-axis: MiB.

Figure 4.1: 20M keys, Ins 50 Del 50, w/DEBRA



(a) Performance

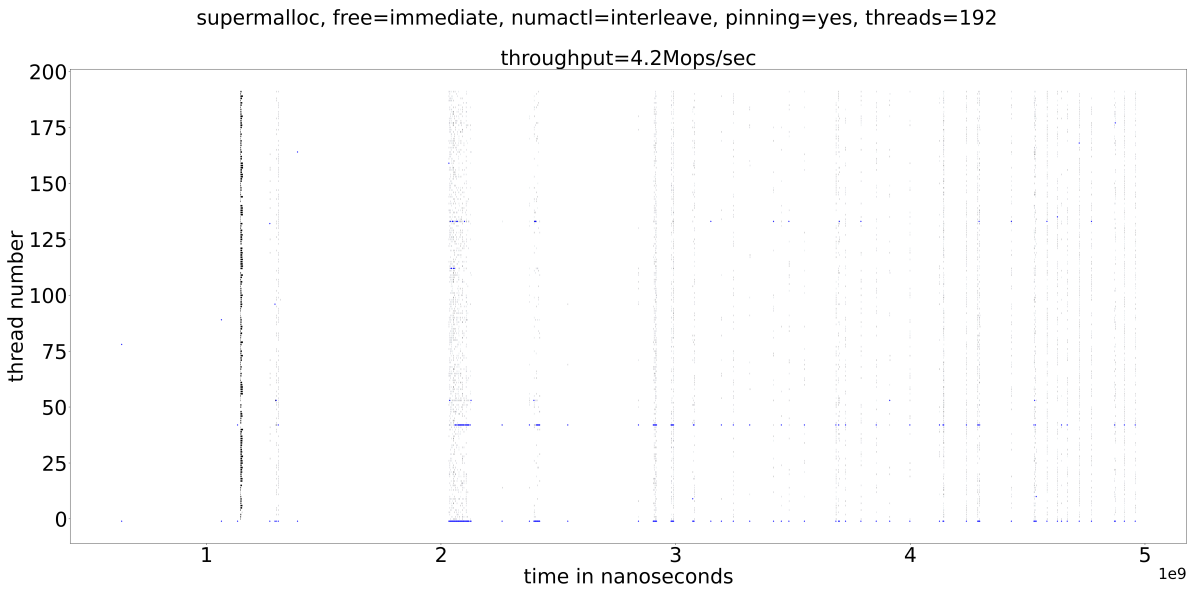
(b) Peak memory usage (MiB)

Figure 4.2: Performance and peak memory usage. 190 threads. X-axis: time (ms)

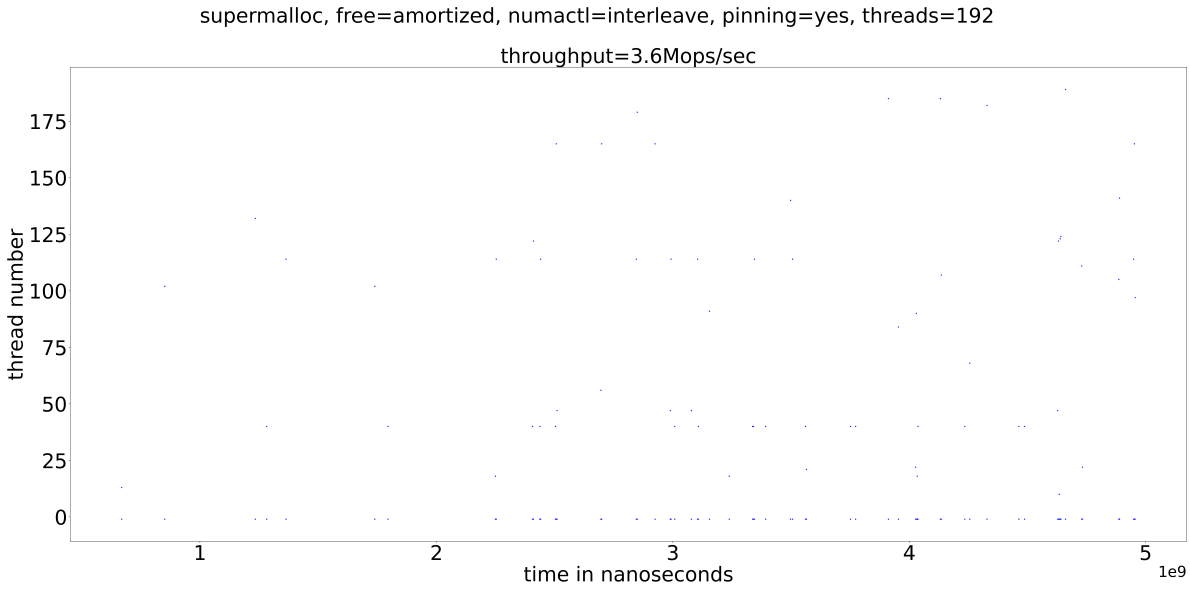
its performance running with four sockets is sometimes worse than its performance using only a single socket.

Hoard Perhaps surprisingly, Hoard, which is one of the oldest allocators designed for multicore systems, performs quite well, both when running on a single socket, and when running on multiple sockets. It achieves the best performance out of the four allocators that I am focusing on. Disappointingly, Hoard performs poorly when the system is oversubscribed (e.g., when running with 240 threads on 192 logical processors), as is demonstrated below in Figure 4.16. Additionally, whereas many modern allocators invest significant effort to bound the amount of memory usage beyond what is necessary to store allocated objects, it appears possible for Hoard to use unbounded memory usage in some workloads (although this did not arise in my experiments).

Supermalloc In the paper that introduced Supermalloc, it was shown to be substantially faster than the other allocators I have described here. However, in my initial experiments, I found it to be the slowest allocator, by a wide margin. As I investigated this result, I tried running experiments for longer periods of time. These results, which appear in Figure 4.1c, suggest that Supermalloc is actually quite fast when running for 60 seconds. In Figure 4.2, Supermalloc performs better over time. Additionally, as one can see in Figure 4.3a, unlike the other allocators, Supermalloc does not seem to induce the kinds of performance problems described above when it is paired with an EBR algorithm for reclaiming memory. And, unsurprisingly, amortized batch freeing does not improve its



(a) Supermalloc w/original DEBRA



(b) Supermalloc w/amortized batch freeing DEBRA

Figure 4.3: Timeline graph of Supermalloc (5 second benchmark)

performance (Figure 4.3b). This suggests that there is a different kind of performance problem to identify in Supermalloc, and that fixing such a problem might lead to a fast allocator that does not suffer from the same kinds of performance problems that necessitated amortized batch freeing in the first place.

4.1 Causes of Poor Supermalloc Performance

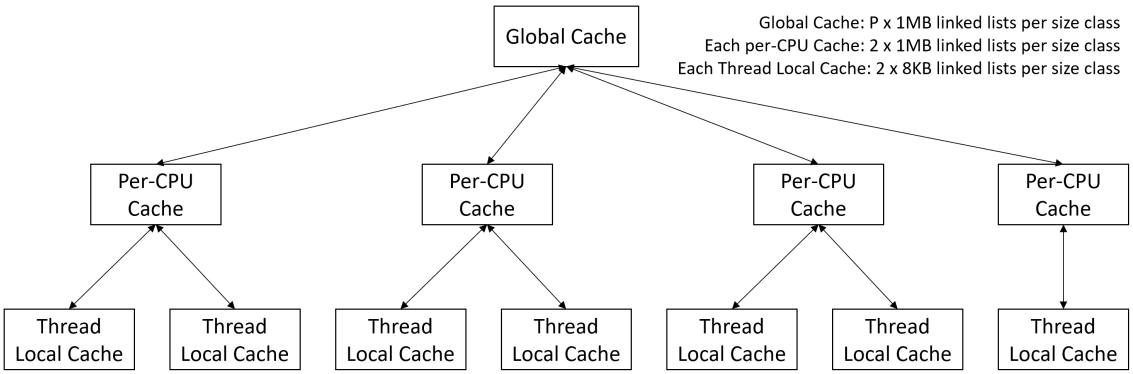


Figure 4.4: Cache Structure of a single size class in Supermalloc. The number and size of linked lists in each cache is specified at each level in the diagram. P is the number of cores. In this diagram there are two threads running on each of the first three cores, and only one thread running on the last core.

Supermalloc must access globally shared resources for every `malloc()` request in the early stages of an execution. At the beginning of an execution in Supermalloc, every global cache is empty, and so is every per-CPU cache, and every thread local cache (diagram in Figure 4.4). One might think Supermalloc would initially `mmap` some pages then use them to populate these caches with free objects. However, it does not. When these caches are empty, and the very first allocation is performed, Supermalloc `mmap`s a new page to be used for bump allocating new objects, and bump allocates a *single* object from that page. Until some objects are freed, all caches will remain empty, and each subsequent allocation will bump allocate a single object. Bump allocation is performed while holding a global lock, and in the initial stages of an experiment when the address space is growing, this global lock is a severe concurrency bottleneck. (Indeed, `htop` shows only a single processor is utilized at a time while the address space is growing.) Figure 4.5 to Figure 4.8 illustrate the allocation algorithm for the original Supermalloc allocator (before any of my modifications).

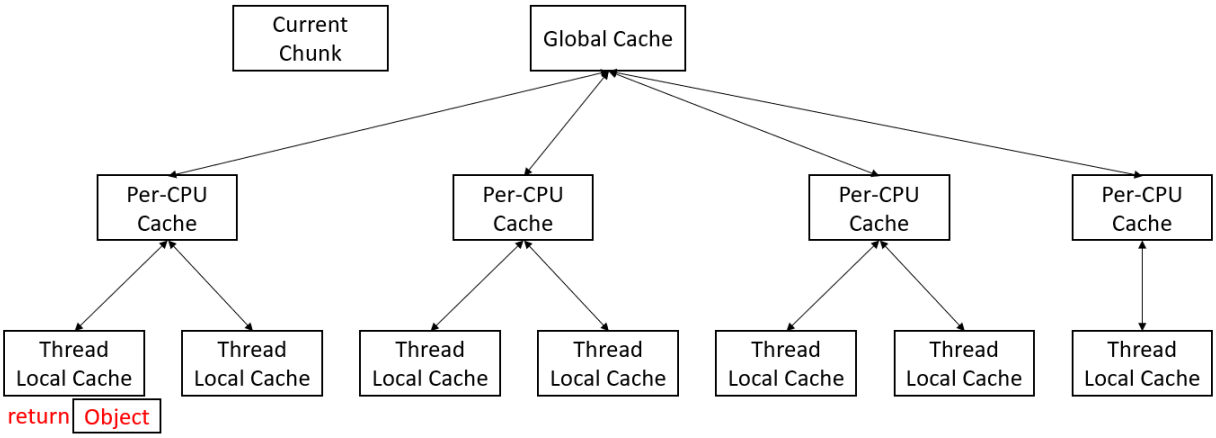


Figure 4.5: Supermalloc allocation when thread local cache is non-empty

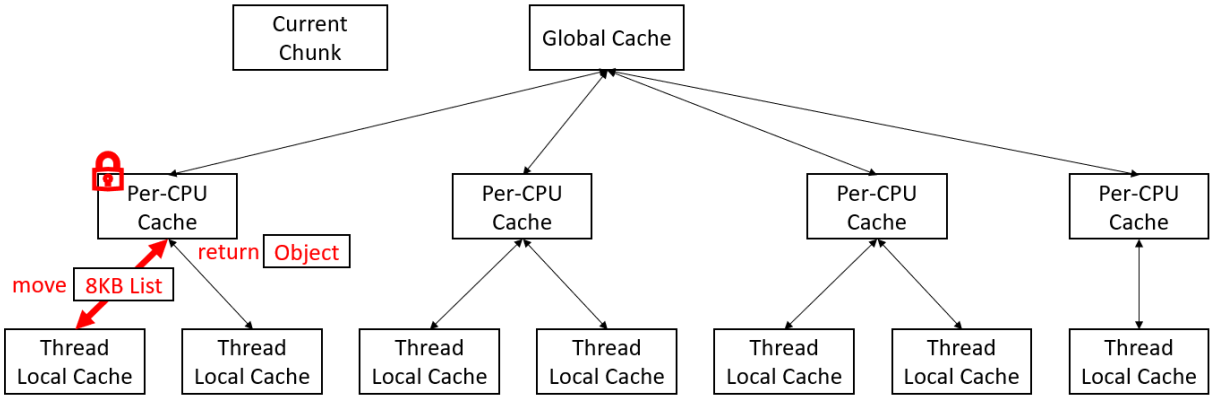


Figure 4.6: Supermalloc allocation when per-CPU cache is non-empty, and cache(s) below it are empty

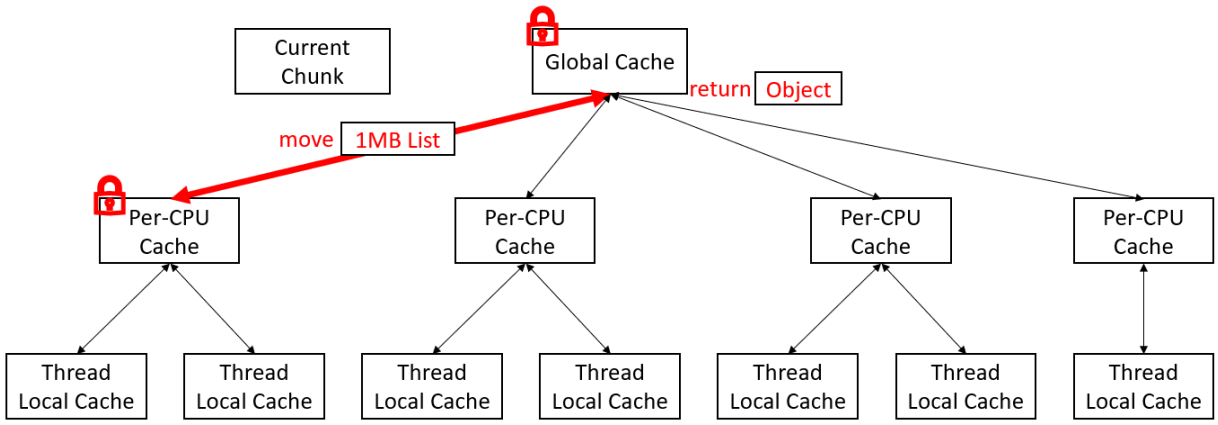


Figure 4.7: Supermalloc allocation when global cache is non-empty, and cache(s) below it are empty

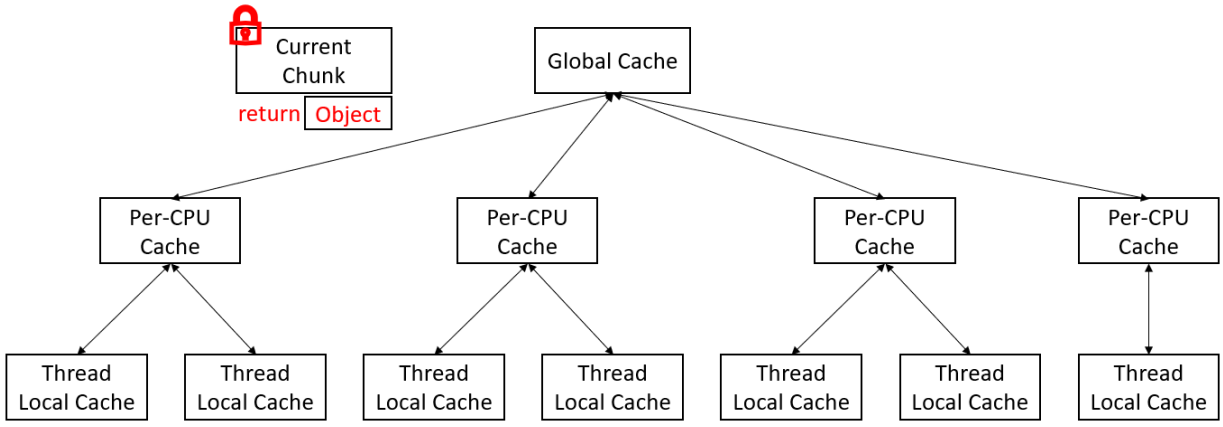


Figure 4.8: Supermalloc bump allocation when caches are empty

The sizes of the caches at each level in Supermalloc are tuned to suit the hardware. For example, thread local caches are similar in size to L1 processor caches. This is intended to improve performance, but because of the different cache sizes, the caches at different levels are stored in linked lists of different sizes. Global caches and per-CPU caches have the same 1MB size linked lists. When objects are moved between these caches, an entire linked list is moved. However, thread local caches use 8KB sized linked lists. So, when a thread moves objects from a per-CPU cache to its thread local cache, the thread cannot simply move an entire 1MB linked list. Instead, it traverses the linked list in the per-CPU cache and separates it into an 8KB list and a list containing the remaining elements, and moves the 8KB list into the thread local cache.

Another problem is high contention on global caches when running on multiple sockets. In Supermalloc, there is a huge difference between the amount of contention experienced at the per-CPU cache level and the global cache level. Whereas only two threads contend on a per-CPU cache (with two way hyperthreading), requests from *all* processors are directed to a single global cache (for a given size class), and accessing the global cache entails acquiring a global lock. This is the same global lock that is held when new objects are bump allocated to grow the address space. However, this is not the same performance problem as what I described above. As I explained above, acquiring a global lock to allocate only one object causes extreme degradation at the *beginning* of an execution, while the address space is growing. In contrast, the contention on global caches is a problem throughout the entire execution.

The lack of a caching level between the per-CPU caches and global caches makes the performance problems described above especially inefficient. If there were such a level, containing, for example, per-socket caches, then one could move entire 1MB lists to per-socket caches, and per-CPU caches could communicate with their own per-socket caches instead of accessing a global cache directly.

4.2 Modifications to Supermalloc

Diagrams depicting the allocation algorithm for the improved variants of Supermalloc appear in Figure 4.9 to Figure 4.13.

4.2.1 Improving NUMA Awareness

To improve the performance of Supermalloc on systems with multiple processor sockets, I added an additional level to the object cache hierarchy. To see why, consider the following. In the original Supermalloc design, all processors access the global cache. So, as the number of processor increases, the number of global cache accesses increases. Moreover, when the accesses come from processors on multiple sockets, the lock and metadata for the global cache must be transferred across sockets, at a much higher cost than when all accesses come from a single socket.

By adding per-socket caches, between the per-CPU and global caches, the total number of accesses to the global cache can be reduced. More specifically, the contention experienced by a per-socket cache is proportional to the number of processors on that socket, and the contention on the global cache is proportional to the number of sockets. Each per-socket cache contains up to P_S lists, each containing 1MB of objects, where P_S is the number of processors per socket.

4.2.2 Reducing Global Cache Contention

Supermalloc can avoid performing individual bump allocations while holding a global lock (which is especially impactful in the early stages of an execution) if it immediately populates a global cache with many free objects whenever it is found to be empty. More specifically, the very first `malloc()` call for a given size class bump allocates sufficiently many objects to fill the corresponding global cache to half of its maximum capacity. After this, many subsequent `malloc()` calls can avoid the overhead of bump allocating, and can simply move lists of objects from the global cache to per-socket caches, per-CPU caches and thread local caches. Using this approach, bump allocation only needs to occur when a global cache becomes empty, which should be relatively infrequent.

Conversely, when a global cache is full, and `free()` is performed, it repeatedly frees objects from the global cache, until it is only half full. This effectively reduces the cost of some future `free()` calls.

4.2.3 Optimizing Linked List Transfer

In the original Supermalloc design, 1MB lists are used for the global cache and per-CPU cache, and 8KB lists are used for thread local caches. In principle, using smaller 8KB lists for thread local caches limits the amount of memory wasted if there are many threads, that

each allocate a couple of objects from many different size classes, since only one 8KB list is "wasted" per size class. However, because the list size differs for per-CPU and thread local caches, there is significant overhead when transferring lists between these levels. For example, whenever a thread wants to transfer a list from a per-CPU cache to a thread local cache, the per-CPU cache list is traversed, link by link, until 8KB of objects have been accumulated, and the per-CPU cache list is split at that point. This traversal involves pointer chasing, and can be costly.

On modern machines with large amounts of memory, I would argue it is reasonable to use 1MB lists for thread local caches as well. And, by doing so, I can eliminate the need to traverse linked lists altogether, and allow lists to be transferred efficiently between all cache levels.

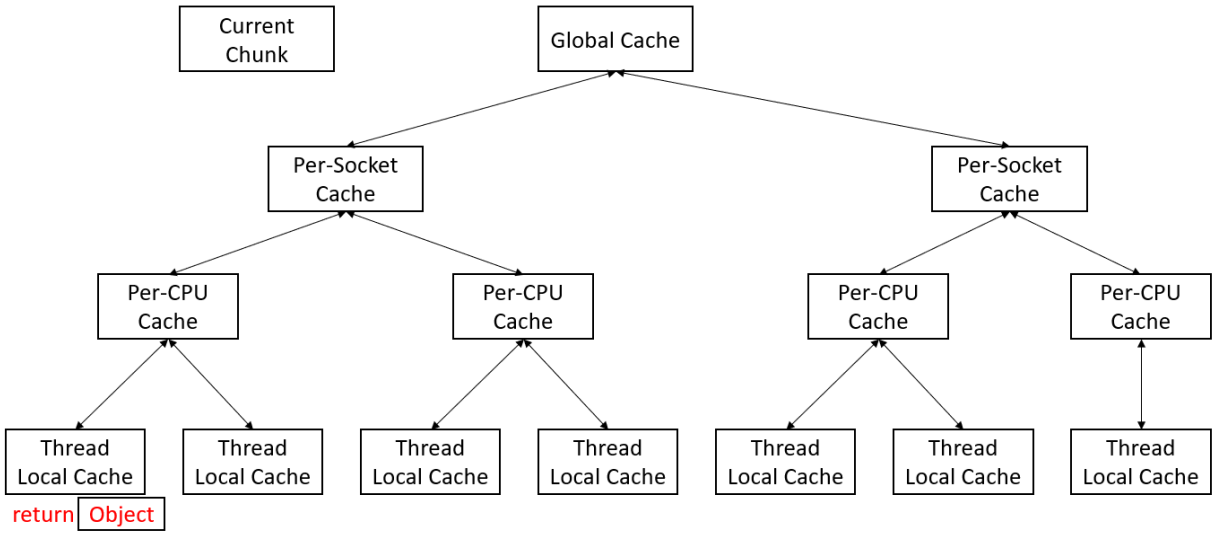


Figure 4.9: Improved Supermalloc allocation when thread local cache is non-empty

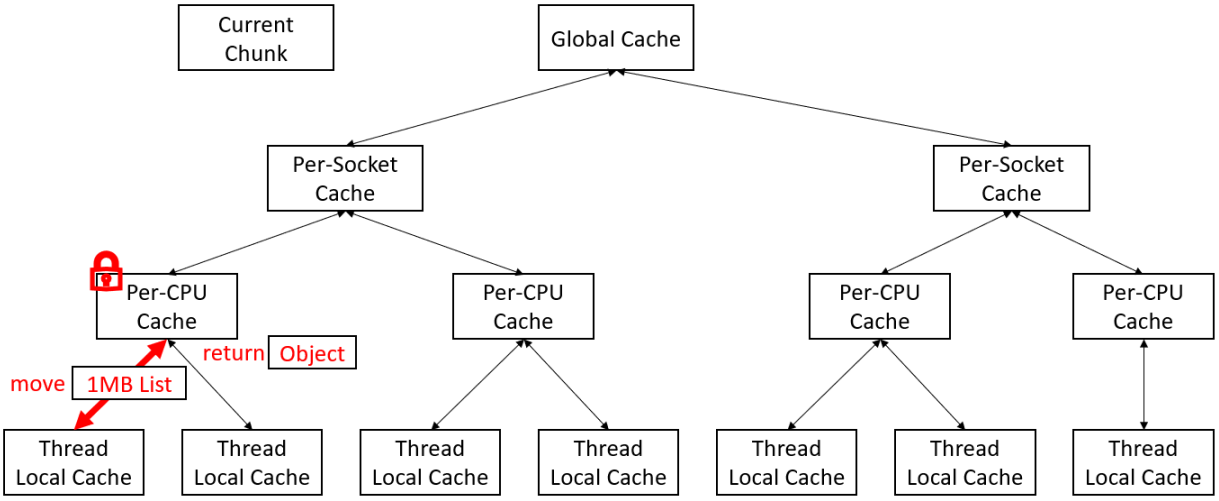


Figure 4.10: Improved Supermalloc allocation when per-CPU cache is non-empty, and cache(s) below it are empty

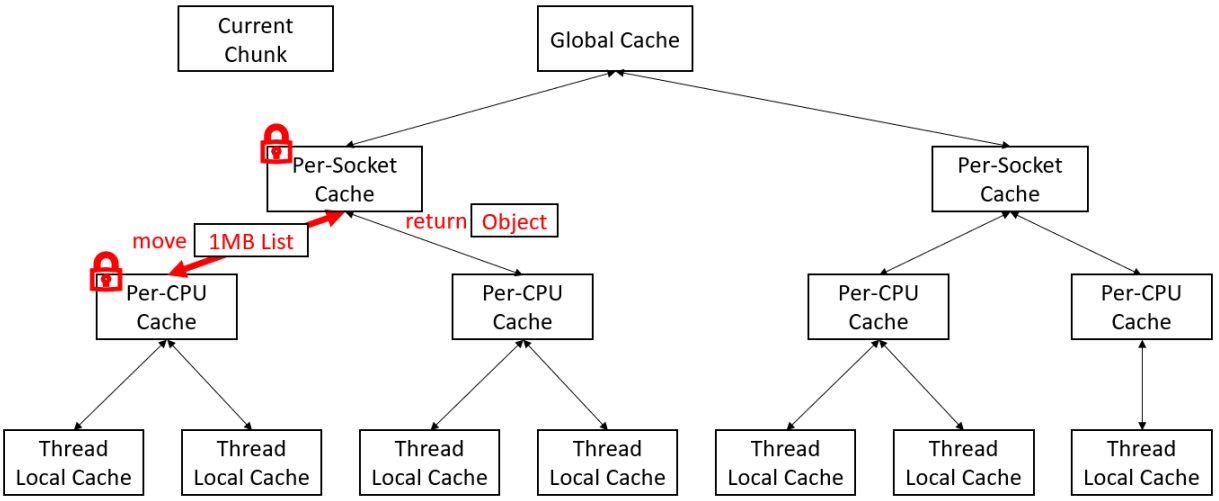


Figure 4.11: Improved Supermalloc allocation when the (newly added) per-socket cache is non-empty, and cache(s) below it are empty

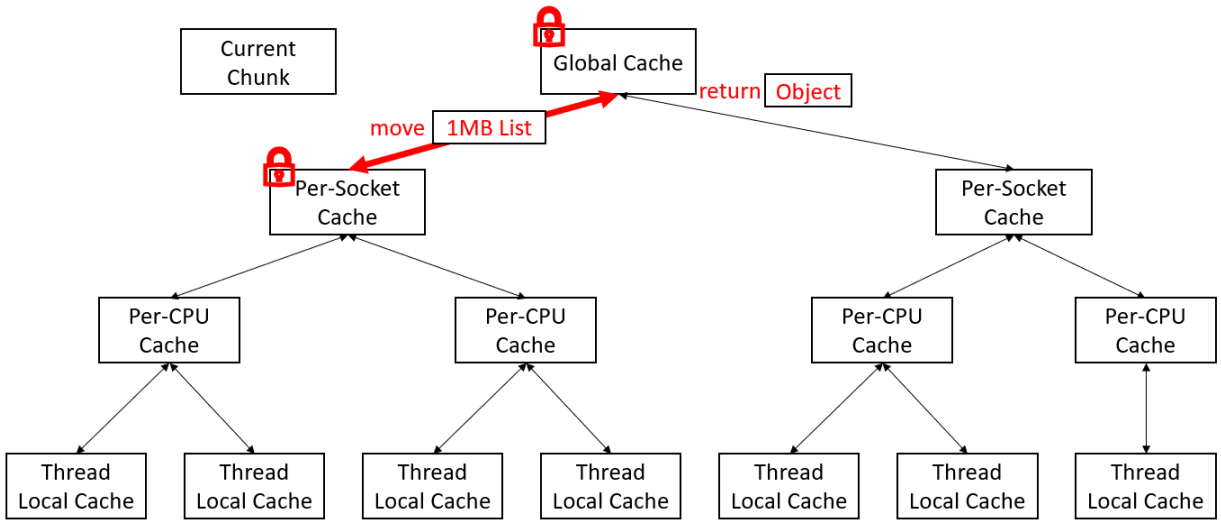


Figure 4.12: Improved Supermalloc allocation when global cache is non-empty, and cache(s) below it are empty

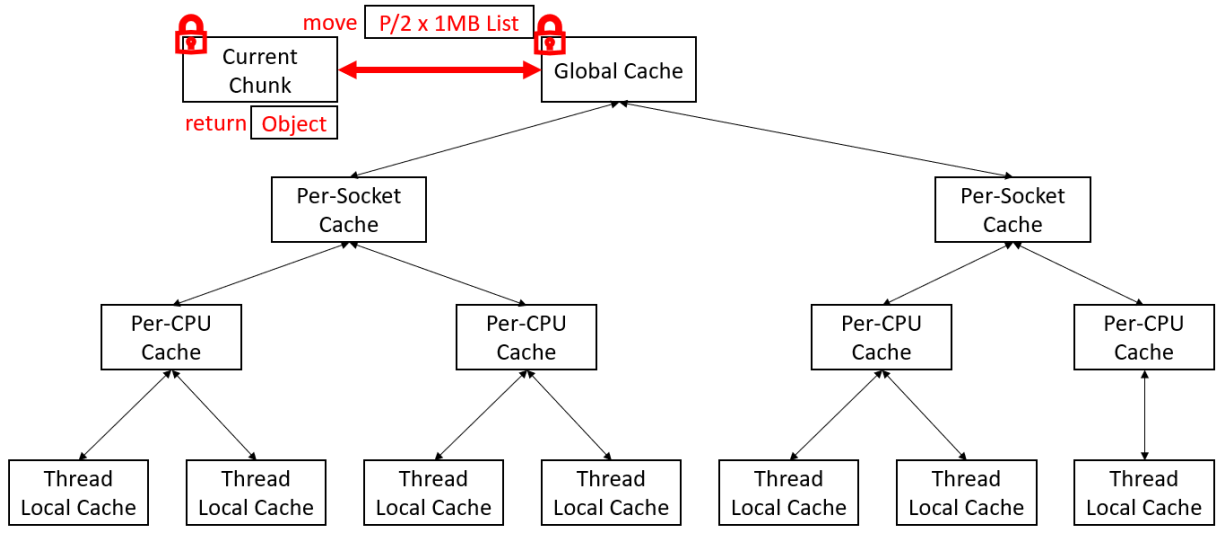


Figure 4.13: Improved Supermalloc bump allocation when caches are empty

4.3 Evaluation

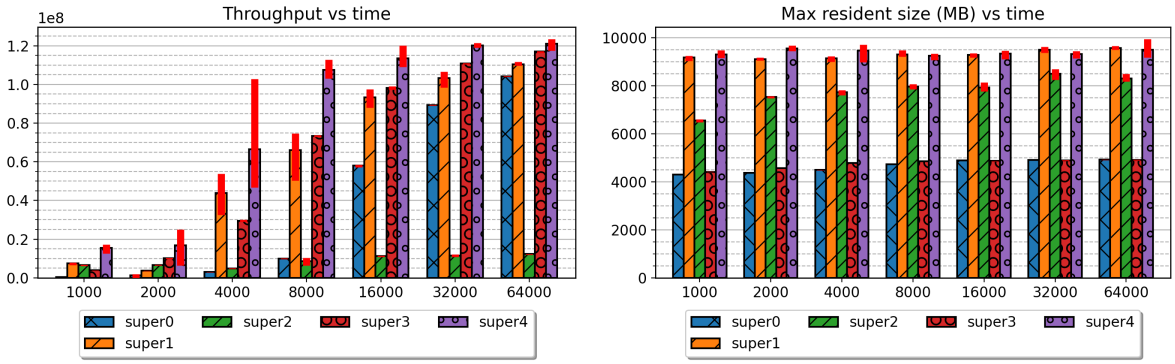
I implemented several of the modifications proposed in Section 4.2 and I evaluate the resulting variants of Supermalloc in this section.

The variants of Supermalloc are as follows:

- super0: original Supermalloc
- super1: multiple bump allocation to a global cache when caches are empty (proposed in Section 4.2.2)
- super2: super1 + multiple free from a global cache when caches are full (proposed in Section 4.2.2)
- super3: optimized linked list size (proposed in Section 4.2.3)
- super4: super1 + super3

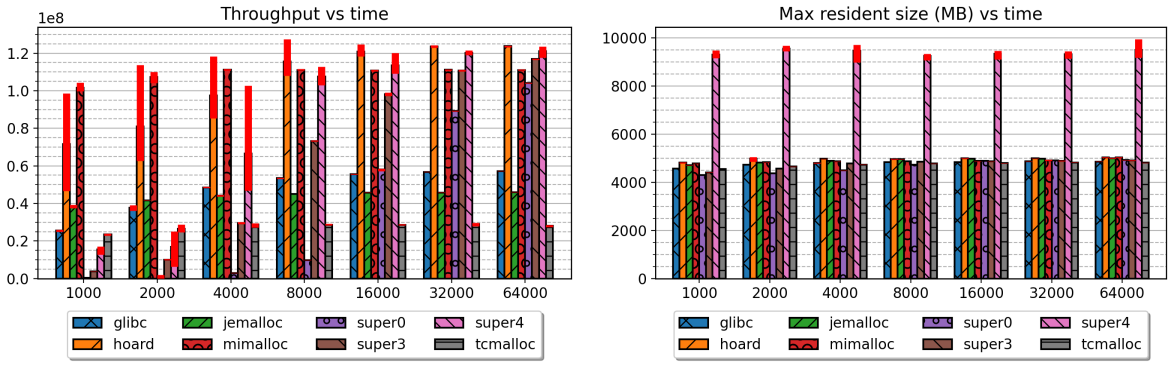
The performance results in this section include a recent allocator called Mimalloc that was developed by Microsoft Research concurrently with the work presented in this thesis. If I was to ignore Mimalloc, the performance results obtained from the improvements proposed in this thesis would be more impressive. However, since Mimalloc was released publicly *while* I was in the process of implementing some of the proposed improvements, it was important to compare these improvements with Mimalloc. As one of the following graphs demonstrates, Mimalloc represents a large leap forward in performance in certain workloads (although it is not always the fastest allocator).

As shown in Figure 4.14a, super1, super3 and super4 improve performance especially in the early stages of experiments. The super1, super2 and super4 variants have higher peak memory usage compared to super0 due to their aggressive global cache fill up (Figure 4.14b). More specifically, consider that in this somewhat simplistic improvement to Supermalloc, a single allocation of an object in any given size class will cause a corresponding global cache to be filled half way. So, if there are several size classes in which only a few objects are allocated, super1, super2 and super4 will each allocate many unnecessary objects. This could be improved with a more intelligent global cache filling algorithm. For example, rather than immediately prefilling to half of the cache's capacity, one might allocate only one object in the first allocation, two objects in the second allocation, four in the third, eight in the fourth, and so on. The study of such optimizations is left for



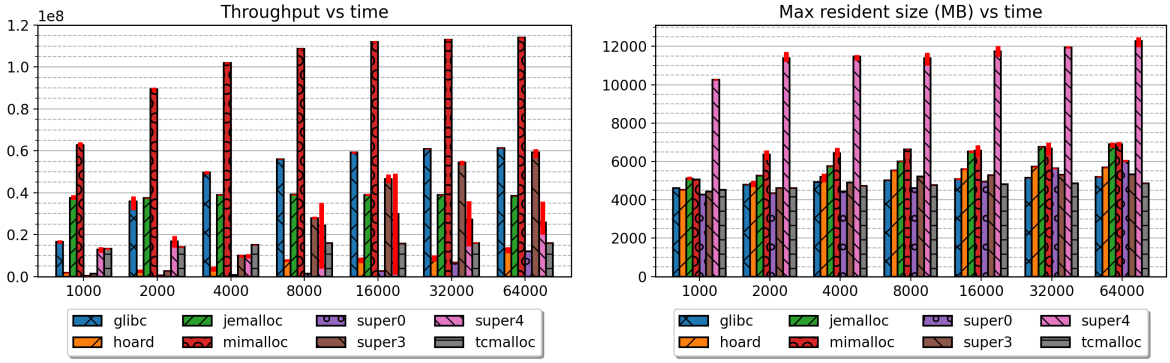
(a) Performance (b) Peak memory usage (MiB)

Figure 4.14: Performance and peak memory usage. 190 threads. X-axis: time (ms)



(a) Performance (b) Peak memory usage (MiB)

Figure 4.15: Performance and peak memory usage. 190 threads. X-axis: time (ms)



(a) Performance

(b) Peak memory usage (MiB)

Figure 4.16: Performance and peak memory usage. 240 threads. X-axis: time (ms)

future work. Note that although the absolute memory usage of super1, super2 and super4 is higher than in super0 or super3, it does not appear to grow over time.

Figure 4.15 compares super3 and super4 to popular allocators. They have exhibit competitive performance, and super3 is competitive in terms of peak memory usage. In Figure 4.16, 240 threads are run on 192 logical processors (and threads are not pinned). Mimalloc performs the best in this graph, followed by super3 and the default system allocator (glibc), followed by the other popular allocators: JEmalloc, hoard and TCmalloc. Among the different versions of Supermalloc, super3 has the best performance, and it also maintains a relatively small memory footprint.

Chapter 5

Related Work

Several popular memory allocators, and some memory reclamation algorithms, have already been described and cited above. So, this chapter focuses on describing the most closely related work that has not yet been addressed. Broadly, this related work falls into two categories: token based memory reclamation algorithms, and work that studies the impact of peak memory usage on performance.

5.1 Token Based Memory Reclamation

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 [29], and similar algorithms have been used to reclaim memory in operating system kernels [17]. Token based algorithms have also been used as part of complex garbage collection algorithms in the distributed setting [14], and token based EBR has been described in a thesis due to Tam in the shared memory multicore setting [27]. The multicore token based EBR algorithm in [27] essentially matches the description above. However, the experiments in that thesis show that the token based algorithm is inefficient, and the authors dismiss the design, partly on the ground that it is inefficient. This thesis shows it does not have to be inefficient. The new token based algorithm presented herein competes with the current state of the art, which is more impressive considering there has been more than 15 years of progress in EBR algorithms since the publication of [27].

Several algorithms have been introduced that are similar to, but not exactly the same as, EBR. Notable examples include IBR (interval based reclamation) [28], NBR [26], and

various algorithms such as beware & cleanup that blend EBR with other memory reclamation techniques [12, 22, 24, 23]. Additionally, many memory reclamation algorithms that are not as closely related to EBR nevertheless free objects in batches to amortize various overheads. Notable examples include hazard pointers, hazard eras, wait-free eras, crystalline, thread scan, fork scan and stack track. There are many other such algorithms. In fact, even traditional garbage collectors typically sweep to accumulate a large batch of objects to free, and then free them all at once. It seems highly likely that such algorithms will benefit (possibly dramatically) from the amortized free technique described in this thesis.

5.2 Performance Impact of Peak Memory Usage

In [21], Mitake et al. studied the question of how peak memory usage and the latency of database transactions are related, in in-memory database systems that use EBR to reclaim memory. They explain that some improvements in performance and latency can be gained by using asynchronous reclamation, meaning that a background thread continuously frees batches of objects that are safe to free. (I.e., when a batch of objects is eligible to be reclaimed, it is shipped to a background thread, which then attempts to free it all at once.) The high level idea is: Since a background thread is responsible for actually freeing objects, the threads participating in epochs can asynchronously proceed without waiting for that freeing to occur. A similar approach is taken in Hekaton [8] and Ermia [16]. Their work is clearly related in that they focus on EBR, and study peak memory usage and its impact on performance, but their conclusions are quite different. Rather than understanding why freeing batches using a background thread is not enough to improve performance to their satisfaction, and developing a new EBR variant that avoids those problems, they claim that it is simply crucial to reduce memory usage overall, and design a new database index structure that uses less memory. My conclusions are different: moving batch freeing to a background thread is not enough to avoid the problems I identify. Batch freeing itself is the problem. My amortized approach wherein objects are gradually freed over the course of many data structure operations, results in qualitatively different behaviour, and side steps a powerful negative interaction that can occur with several popular memory allocators.

Chapter 6

Conclusion

6.1 Contribution

In this thesis, I focus on memory allocation and reclamation in multicore systems. I first find the root causes of poor performance in EBR algorithms and improve a state of the art EBR algorithm with an algorithmic fix in Chapter 3. I also show the algorithmic fix can be applied to other EBR algorithms, to great effect. Finally, I propose a NUMA aware redesign of a conventional allocator, Supermalloc, in Chapter 4.

I did this by first showing that traditional EBR algorithms spend a huge amount of time freeing batches of retired objects in Section 3.2. By introducing a simple amortized freeing technique, I avoid freeing huge batches after an epoch change. This algorithmic change improves the total time spent freeing objects, and seems to have a smoothing effect on the distribution of execution times for individual `free()` calls in some allocators, such as JEmalloc (preventing large batch freeing events in the internals of the allocator).

The timeline graphs also appear to be novel in this context, and I think they can be a powerful tool for diagnosing behaviour in algorithms that require all threads to synchronize using barrier-like computations. It seems likely they would be useful regardless of whether such barriers are hard, as in a traditional barrier synchronization, or fuzzy, as in EBR where threads can *pass* a barrier to continue to execute operations, but cannot necessarily free objects.

To demonstrate that the amortized freeing technique is not specific to one particular EBR algorithm, I study a second class of EBR algorithm, wherein a token is passed between threads to determine when memory can be freed. Token based EBR is conceptually

simpler, and also easier to implement. Without the new amortized freeing technique, the performance of Token EBR is extremely poor in some workloads, for example, when it is paired with JEmalloc. When the amortized freeing technique is applied to Token EBR, it is competitive with the state of the art in EBR algorithms.

Second, I propose a NUMA aware design of Supermalloc. I investigate design decisions in Supermalloc that lead to extremely high overhead in the early stages of experiments that use Supermalloc. I discover a series of performance problems, including a global concurrency bottleneck on bump allocation of new objects, high contention on a global object cache, and differing list sizes at various levels of object caching in Supermalloc. Three structural changes are suggested to improve the performance and NUMA awareness of Supermalloc in Chapter 4.2.

6.2 Limitations and Future Work

It would be interesting to study how amortized freeing would affect performance for other memory reclamation algorithms. Since amortized batch freeing improves EBR algorithms, and the performance improvement seems to be due to the interaction between batch freeing and internal allocator buffers, it seems likely that this approach can improve performance generally for any reclamation algorithms that free objects in batches. Such algorithms include IBR (interval based reclamation) [28], NBR [26], and various algorithms such as beware & cleanup that blend EBR with other memory reclamation techniques [12, 22, 24, 23].

Additionally, many memory reclamation algorithms that are not as closely related to EBR nevertheless free objects in batches to amortize various overheads. Notable examples include hazard pointers, hazard eras, wait-free eras, crystalline, thread scan, fork scan and stack track. There are many other such algorithms. In fact, even traditional garbage collectors typically sweep to accumulate a large batch of objects to free, and then free them all at once. It seems likely that such algorithms will benefit (possibly significantly) from the amortized free technique described in this thesis.

There are some limitations to this work. Amortized batch freeing can cause problems if data structures have enormous nodes that really should be freed as soon as possible. For example, Brown et al. [4] introduced a data structure called a concurrent interpolation search tree, in which some nodes can be extremely large—many megabytes in size—and it may be advantageous to free such large nodes as soon as possible, rather than amortizing their reclamation over many data structure operations.

The simplest proposal for amortized batch freeing suggests freeing one object per data structure operation, which may be insufficient for data structures that free more than one object per data structure operation, on average. This is the case in some workloads in, for example, the lock-free binary search tree of Ellen et al. [9]. In such a data structure one might need to free more than one object per data structure operation, or perhaps occasionally perform a batch free to "catch up."

Rigorous implementation and optimization for Supermalloc is required to show that the proposed NUMA aware design yields tangible performance benefits. Some aspects of the design (such as the additional level of object caches) were not fully evaluated in my experiments due to the fact that I have only produced a naive implementation that is poorly optimized. It is also worth mentioning that I only studied some of most well known allocators. It is possible that some other allocators (such as TBB [18]) could achieve better performance in some workloads.

Finally, there are some memory allocators for non-volatile RAM (see, e.g., Poseidon [7] and Ralloc [6]), but it seems the problem of memory allocation for non-volatile RAM is relatively underexplored, and it would be interesting to apply the lessons learned regarding NUMA aware allocator design and amortized freeing to that setting.

References

[1] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 35(11):117–128, 2000.

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

[3] Trevor Brown. *Techniques for Constructing Efficient Lock-free Data Structures*. PhD thesis, University of Toronto, 2017.

[4] Trevor Brown, Aleksandar Prokopec, and Dan Alistarh. Non-blocking interpolation search trees with doubly-logarithmic running time. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 276–291, 2020.

[5] Trevor Alexander Brown. 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*, pages 261–270, 2015.

[6] Wentao Cai, Haosen Wen, H Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L Scott. Understanding and optimizing persistent memory allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pages 60–73, 2020.

[7] Anthony Demeri, Wook-Hee Kim, R Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st International Middleware Conference*, pages 207–220, 2020.

[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[9] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140, 2010.

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

[11] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc. *Retrieved from http://goog-perftools.sourceforge.net/doc/tcmalloc.html on January 27, 2023*, 2005.

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

[13] Vincent Gramoli. The information needed for reproducing shared memory experiments. In *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers 22*, pages 596–608. Springer, 2017.

[14] Richard L Hudson, Ron Morrison, J Eliot B Moss, and David S Munro. Training distributed garbage: The dmos collector. *Object-Oriented Programming Systems, Language and Applications*, 1997.

[15] Patryk Kaminski. Numa aware heap memory manager. *Retrieved from https://web.archive.org/web/20220306112649/https://developer.amd.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf on January 27, 2023*, 2009.

[16] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687, 2016.

[17] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark

Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[18] Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[19] Bradley C Kuszmaul. Supermalloc: A super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 International Symposium on Memory Management*, pages 41–55, 2015.

[20] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 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*, pages 244–265. Springer, 2019.

[21] Hitoshi Mitake, Hiroshi Yamada, and Tatsuo Nakajima. 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*, pages 3–18. Springer, 2019.

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

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

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

[25] Gali Sheffi, Maurice Herlihy, and Erez Petrank. Vbr: Version based reclamation. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 443–445, 2021.

[26] Ajay Singh, Trevor Brown, and Ali Mashtizadeh. Nbr: neutralization based reclamation. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 175–190, 2021.

[27] Adrian Tam. *QDo: A Quiescent State Callback Facility*. PhD thesis, University of Toronto, 2006.

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

[29] ycombinator. 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.