Adaptive Huge-Page Subrelease for Non-moving Memory Allocators in Warehouse-Scale Computers

Martin Maas
Google, USA

Chris Kennelly
Google, USA

Khanh Nguyen*
Texas A&M University, USA

Darryl Gove
Google, USA

Kathryn S. McKinley
Google, USA

Paul Turner
Google, USA

Abstract

Modern C++ server workloads rely on 2 MB huge pages to improve memory system performance via higher TLB hit rates. Huge pages have traditionally been supported at the kernel level, but recent work has shown that user-level, huge page-aware memory allocators can achieve higher huge page coverage and thus performance. These memory allocators deal with a trade-off: 1) allocate memory from the operating system (OS) at the granularity of a huge page, achieve high performance, but potentially waste memory due to fragmentation, or 2) limit fragmentation by breaking up huge pages into smaller 4 KB pages and returning them to the OS, but reduce performance due to lower huge page coverage. For example, the state-of-the-art TCMalloc allocator handles this trade-off by releasing memory to the OS at a configurable release rate, breaking up huge pages as necessary.

This approach balances performance and fragmentation well for machines running one workload. For multiple applications on the same machine however, the reduction in memory usage is only useful to overall performance if another workload uses this memory. In warehouse-scale computers, when an application releases and then reacquires the same amount or more memory quickly, but no other application uses the memory in the meantime, the release causes poorer huge page coverage without any system-wide benefit.

We introduce a metric, *realized fragmentation*, to capture this effect. We then present an adaptive release policy that dynamically determines when to break up huge pages and return them to the OS to optimize system-wide performance. We built this policy into TCMalloc and deployed it fleet-wide in our data centers, leading to an estimated 1% fleet-wide throughput improvement at negligible memory overhead.

*Work done while at Google.

ISMM '21, June 22, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8448-3/21/06.
https://doi.org/10.1145/3459898.3463905

CCS Concepts: • **Software and its engineering → Allocation / deallocation strategies**; **Main memory**.

Keywords: C++ Memory Allocation, Huge Pages, Fragmentation, Cluster Scheduling, TCMalloc

ACM Reference Format:
Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. 2021. Adaptive Huge-Page Subrelease for Non-moving Memory Allocators in Warehouse-Scale Computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management (ISMM '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3459898.3463905

1 Introduction

Modern C++ server workloads rely on 2 MB huge pages for performance. Workloads with large heap sizes put significant pressure on the translation lookaside buffer (TLB), causing the CPU to spend a substantial fraction of its cycles traversing the page table, especially when using traditional 4 KB pages. The speed-up from using 2 MB pages has been reported to be as high as 53% for realistic workloads [13].

Modern operating systems transparently support huge pages and recent research improves huge page mechanisms at the kernel-level [13, 15, 16]. A less explored problem is that for user-level applications to use these huge pages effectively, they need to manage memory at huge page-granularity as well. Past work shows that to make maximum use of huge pages, the user-level allocator needs to cooperate by allocating memory in chunks of 2 MB from the OS [10, 14].

This approach leads to a new fragmentation problem for non-moving memory managers [14], since the probability that a given 2 MB page frees up completely and can be returned to the OS is much smaller than the probability that a 4 KB page can be returned. Allocators therefore have to decide between keeping a 2 MB page fully allocated to the application, at the cost of incurring memory overheads in the form of fragmentation, or to break this page up and return a subset of its constituent 4 KB pages back to the OS, a process known as subrelease. The latter incurs performance overheads due to decreased huge page coverage, while the former incurs memory overheads from fragmentation. Previous work has shown that only using huge pages without subrelease can increase physical memory size by 23-69% [13].

Huge page-aware memory allocators such as TCMalloc handle this choice as part of their memory release policy: A configurable release rate tells the allocator how many MBs per unit time it should return to the OS, and the allocator will break up huge pages with free 4 KB sub-pages as necessary to meet this rate. While this policy is suitable for an individual application or benchmark in isolation, server workloads often share a machine with many other applications, which are assigned by a cluster scheduler that dynamically allocates resources such as cores or memory between them. We find that the above release policy is suboptimal in this setting.

Cluster schedulers take a certain amount of time to schedule a job. For the purpose of this paper, we call this time interval $T_{schedule}$. The specific value varies; for example, prior work by Verma et al. implies that it is on the order of minutes [21]. This delay means that if resources are returned to the OS for less than $T_{schedule}$ and then again requested, they cannot actually be used by any other workload.

This OS reuse policy leads to a subtle interaction with the application-level memory allocator. When the memory allocator returns memory immediately based on its release rate, it may return memory that the application needs again quickly. This introduces significant overheads from several sources: 1) additional syscalls and page faults to return and re-allocate memory to and from the OS, and 2) unnecessarily breaking up huge pages, thus reducing huge page coverage (the fraction of memory covered by huge pages), which reduces the overall application throughput.

Importantly, these are problems that are not visible when evaluating an allocator on benchmarks or on a single application – these problems only manifest through the interaction between memory manager, OS, and cluster scheduler in datacenter deployments, and can account for performance overheads of 3-5% (or more) for some workloads. We make several contributions to quantify and address this problem:

- We identify a source of inefficiency in C++ memory allocators that arises from the fact that current memory allocator policies do not take cluster scheduling delays into account.
- We show that this effect can lead to misleading fragmentation measurements and introduce a new metric, *realized fragmentation*, to capture long-lived free memory that can be returned to the OS. This metric therefore enables making better system-wide fragmentation and performance trade-offs.
- We introduce an algorithm and implement it in TCMalloc that exploits these insights to significantly improve huge page coverage and thus performance.
- We present a thorough evaluation of our algorithm in warehouse-scale datacenter deployments and demonstrate a 1% performance improvement across a global fleet of C++ servers, with speed-ups of 5% for some important workloads that rely on huge pages.

Our approach has been deployed fleet-wide in Google's datacenters. While a 1% saving may seem small, these savings represent realized end-to-end improvements across all servers in a heavily optimized global fleet and observed in a real production deployment, not an experimental setting. This includes servers that do not benefit from the optimization (e.g., because they don't release memory). We therefore demonstrate with high confidence that our technique is effective across a wide range of server workloads.

Section 2 introduces background about huge pages and memory management. In Section 3, we show that existing metrics used to present fragmentation can be misleading and propose a better metric to measure fragmentation in memory allocators. Section 4 then introduces our adaptive huge page release technique, while Section 4.1 presents implementation details. We then evaluate our technique using measurements from a hyperscale production deployment (Section 5), followed by related work (Section 6) and conclusions (Section 7). Our implementation is available open-source in TCMalloc.

2 Background

We now introduce background material on huge page management, TCMalloc, and cluster scheduling. We also describe interactions between the operating system, the user-level memory allocator, and the cluster scheduler.

2.1 Huge-Page Management

Huge pages are crucial for efficient execution of workloads with large memory footprints [13–16]. Traditionally, huge pages were supported at the kernel level through transparent huge-page support (THP), which describes techniques where user-level applications do not need to be aware of the underlying page size, can request memory from the OS at any granularity, and the OS then automatically determines how to rearrange/compact these pages to maximize huge page coverage (which is the fraction of used memory that is backed by huge pages).

While there has been a large amount of work on improving transparent huge page support [13, 15, 16], there are several fundamental challenges of this approach. First, achieving high huge page coverage without assistance from the application-level memory allocator is challenging: Since the kernel can only relocate memory at a base page granularity, efficiently packing memory at the kernel level does not automatically translate into larger huge page coverage, because contiguous base pages in physical memory also have to be contiguous in virtual memory in order to be backed by huge pages (otherwise, the pages still need to be backed using base pages). Further, not all pages allocated at the kernel level are movable, and unmovable pages can lead to fragmentation that reduces the availability of 2 MB pages [16]. Finally, compaction is not free and can incur substantial cost from memory copying overheads.

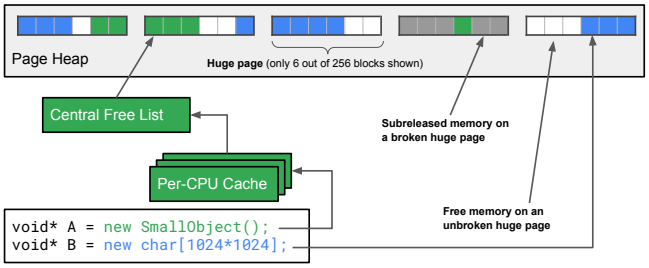


Figure 1. High-level overview of TCMalloc's allocation paths. The huge page-aware page heap manages memory at the granularity of 2 MB pages divided into 8 KB blocks. These pages can be broken up to subrelease portions of them back to the operating system.

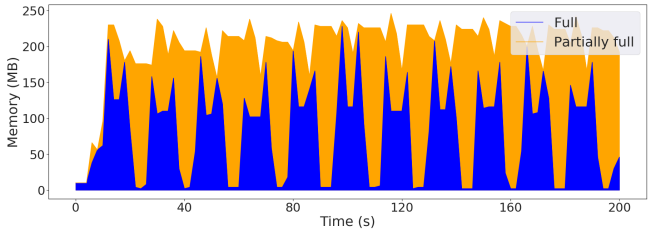


Figure 2. Memory consumption of a Redis server workload over time (running against synthetic requests). A substantial amount of live memory is in pages that are partially full and therefore either have to be broken up or retained.

For these reasons, transparent huge page mechanisms at the kernel level can further improve huge page coverage with assistance from the user-level application. If the user-level application allocates pages at multiples of 2 MB, these regions can be trivially backed by huge pages at the kernel level without creating fragmentation that reduces the availability of 2 MB pages. This requirement to support huge pages at the application-level is increasingly being recognized by modern memory allocators [10, 12, 14].

2.2 TCMalloc

Throughout this paper, we will use the huge page-aware allocator [10] in TCMalloc [8] to describe our techniques. We note that the techniques described here are not limited to TCMalloc specifically but apply to any memory allocator that manages memory at the granularity of huge pages and has the ability to break up huge pages as necessary.

Like most allocators, TCMalloc consists of two parts: A small-object allocator that is based on thread and CPU-local caches and serves allocations from a fixed set of size classes through free lists, and a *page heap* allocator that serves large allocations and backs the memory used by the small-object allocator. TCMalloc provides two implementations of the page heap: a legacy version that is not huge page-aware and a huge page-aware page heap that manages memory at the granularity of 2 MB ranges, thus enabling the OS to back these ranges with huge pages. The details of the huge page-aware allocator are described in [10].

At a high level, each 2 MB range managed by the huge page-aware allocator is logically subdivided into 8 KB blocks. The page heap keeps track of them in a bitmap and each block is either marked as "used" (e.g., to back free lists for small objects or large objects directly) or free. When a huge page is entirely free, it can be fully returned to the OS. However, if the huge page is partially free, the allocator can choose to leave the entire huge page allocated or "break it up" and return a subset of its constituent free pages back to the OS, a process called *subrelease* in TCMalloc (Figure 1).

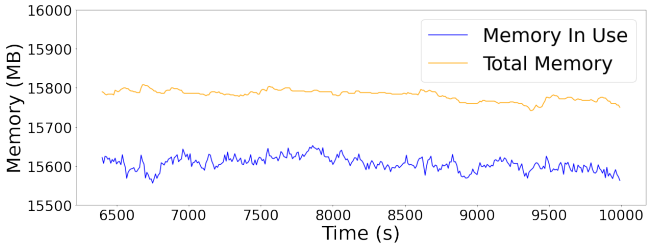
To serve large allocations, TCMalloc tries to binpack them into the existing huge pages, in order to minimize fragmentation. Huge pages that are entirely free are put into a cache and can be released when the allocator is asked to free memory (e.g., to satisfy a statically configured release rate or an explicit memory release request). If there are not enough fully free huge pages in the cache, then the allocator will break and subrelease partially full huge pages.

Many workloads – including server workloads in data centers – have highly varying memory footprints, which in turn leads to a large fraction of partially full huge pages [14]. Partially full pages are often the result of containing one or more objects that are longer-lived than the rest of the objects on that page. The probability that a given page contains a long-lived object increases with the page size. After running for a sufficiently long time, there is a high probability that most huge pages contain at least one long-lived object. When the memory footprint decreases, all of these pages will be partially full and cannot be released without breaking them up. Figure 2 shows this empirically for a Redis workload: a large fraction of allocated memory is on partially full huge pages. For this memory, the allocator has to make a choice whether to retain the memory and incur fragmentation, or whether to return it to the OS, breaking up huge pages and thus reducing huge page coverage. Figure 3 demonstrates this trade-off by looking at the memory from one of TCMalloc's microbenchmarks that mimics real allocation patterns.

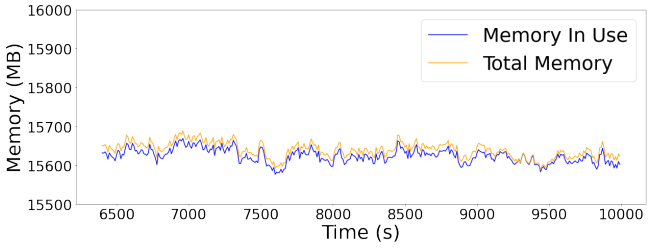
2.3 Cluster Scheduling

Machines in data centers are usually shared between several workloads (jobs), and a cluster scheduler is responsible for assigning jobs to machines. There are many published examples of cluster job schedulers deployed in large-scale production data centers, as well as a large body of academic research (e.g., [3, 9, 19, 21]).

One fundamental property these schedulers share is that they have to binpack jobs onto available machines. When a job arrives at the cluster scheduler, a coordination service determines machines that have enough spare capacity to schedule the job into (e.g., memory, CPU cores, or GPUs).



(a) Low release rate



(b) High release rate

Figure 3. Demonstrating TCMalloc's trade-off between fragmentation and huge page coverage (numbers exclude full huge page allocations). (a) has a low release rate and high huge page coverage, but wastes memory to fragmentation. (b) has a high release rate and wastes less memory, but does so at the cost of breaking up huge pages.

Fluctuations in resource usages induce a fundamental delay in how quickly spare resources can be reused. Resources need to be free for long enough that the cluster scheduler can develop high confidence that these resources will actually remain available to the scheduler and not be needed by another job on the machine. When the scheduler identifies that there is a margin of available resources available on this machine, and once a suitable job arrives to schedule into this gap, it will be assigned to the machine. In practice, these steps cause several minutes delay between the time resources are freed and the time they are reused for a new job. In our cluster environment using Google's Borg scheduler [21], we empirically assume this interval to be 5 minutes.

Due to this delay, resources that are returned to the OS by a job are not actually available to the cluster scheduler unless they remain free for at least a particular time interval. We call this interval $T_{schedule}$.

3 A Better Metric for Fragmentation

The goal of reporting fragmentation is to quantify the amount of memory assigned to the application but that is not actually used. This value is then used to trade off memory overheads and application performance. For example, increasing the release rate lowers fragmentation while incurring additional overheads from breaking up huge pages and interacting with the OS to acquire and release memory.

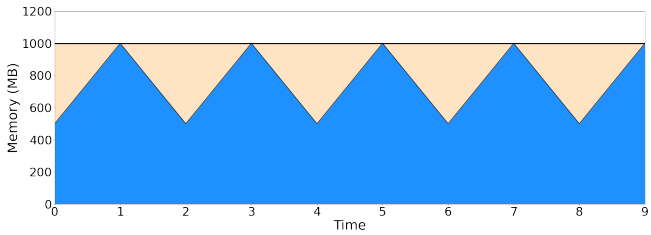


Figure 4. Reporting the current free memory leads to a reported average fragmentation of 250 MB. In practice, however, none of this fragmented memory could actually be used otherwise if the timescale is short.

There are several definitions of fragmentation. For example, Hoard [1] defines fragmentation as the ratio between the maximum amount of memory allocated from the OS and the maximum amount of memory required by the application. Such a metric is appropriate when running a single application in isolation, but does not apply in a setting of multiple long-running server workloads with time-varying demands. A more common metric is therefore *average fragmentation*, which takes time variations into account by periodically sampling the memory-in-use and mapped-memory statistics of the memory allocator, and reporting this ratio of used to mapped memory averaged over a time series.

3.1 Shortcomings of Average Fragmentation

Average fragmentation can be a misleading metric. Consider the example in Figure 4, which shows a workload with 1 GB of mapped memory and whose memory usage is fluctuating between 500 MB and 1 GB. If fragmentation is measured by periodically sampling this workload and reporting the average, the reported free memory from average fragmentation will be 250 MB. However, if the fluctuations between peak and trough are very short (e.g., if the unit of the x axis is in milliseconds), these 250 MB don't actually capture memory that is wasted in a sense that it could be more effectively used otherwise. Even if some of this memory could be returned to the OS, it would immediately be needed again.

At the same time, if the metric on the x axis was in days, 250MB would clearly be an adequate metric to measure the fragmentation, since the memory footprint is changing slowly enough that it is effective to return memory to the OS at any given time and bring it back when it is required again. It is therefore clear that average fragmentation is not the best way to measure fragmentation for the purposes of trading performance vs. memory usage. In our data center deployments, we previously used average fragmentation to determine how much memory we were wasting at any given time. However, we found that much of the fragmentation we were capturing this way was actually of the short-lived kind that could not have been used otherwise. We therefore need a better metric to measure fragmentation.

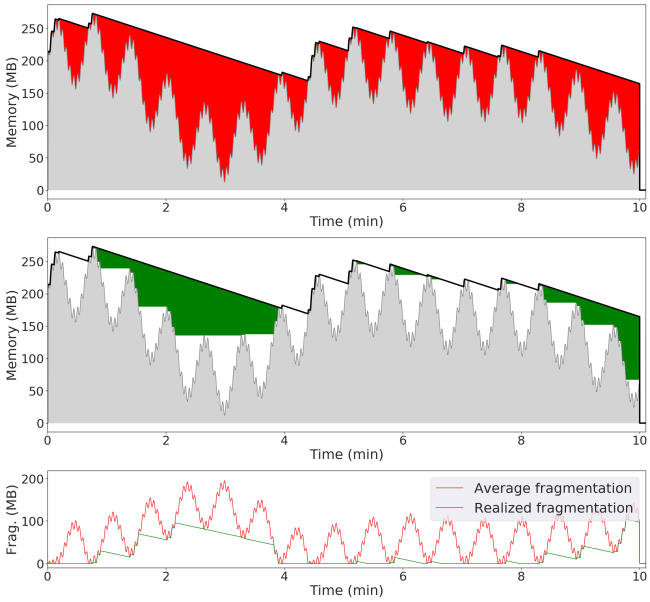
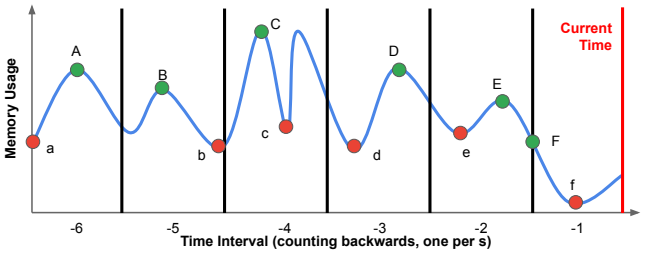


Figure 5. Average fragmentation vs. realized fragmentation, demonstrated on a synthetic memory allocation trace with $T_{schedule}$ = 1 minute. The top two graphs show in-use memory in gray. The top graph shows average fragmentation in red and the middle shows in green realized fragmentation, i.e., how much of the red above fragmentation persists for over 1 minute. The bottom graph presents the two metrics together. Realized fragmentation is much lower than average fragmentation, as expected.



	-1	-2	-3	-4	-5	-6	...
At minimum usage	Number of used pages (f), free pages, broken hugepages, unbroken hugepages	e	d	c	b	a	...
At maximum usage	Number of used pages (F), free pages, broken hugepages, unbroken hugepages	E	D	C	B	A	...
At minimum number of live hugepages	...	...	...	...	...	...	...
At maximum number of live hugepages	...	...	...	...	...	...	...

Figure 6. Data structure holding the time series data for tracking realized fragmentation. Time intervals are tracked in a circular buffer and for every interval, we store the statistics at all possible inflection points. This captures all local peaks and troughs.

3.2 Realized Fragmentation as a Better Metric

We define *realized fragmentation* to be the maximum amount of memory that is free and, if returned to the OS, would not be requested back within less than $T_{schedule}$. As such, this metric captures unused memory that, if it were released, could be productively reused by another workload. In contrast to average fragmentation, realized fragmentation cannot be computed from the current state of the allocator alone but requires a time series of previous activity.

Given a time series with a length of at least $T_{schedule}$, realized fragmentation can be calculated in closed form. For every point in the time series, we compute the amount of memory that is both mapped (i.e., allocated to the application) and free. Realized fragmentation is then the minimum of this value over the past $T_{schedule}$ interval. (This is equivalent to the previous definition: Since the memory is free for the entire $T_{schedule}$ interval, it would not be requested back within less than $T_{schedule}$ if returned to the OS. Meanwhile, we could not return more memory without requesting it back within less than $T_{schedule}$, since we are looking at the minimum across the time series. It is therefore the maximum amount of such memory.)

3.3 Tracking Realized Fragmentation

Figure 5 shows the difference between the two metrics visually. While average fragmentation is large, realized fragmentation is a much smaller amount, showing what may actually be returned to the OS and potentially used by other applications. Realized fragmentation thus more accurately represents wasted memory at the system-wide level.

Recording realized fragmentation requires new functionality in the memory allocator to capture a time series of memory use over the last $T_{schedule}$ interval. While it might seem intuitive that we would only need to capture the last peak, it turns out that this is not sufficient – the reason is that if we only keep track of one peak, as soon as that peak moves out of the $T_{schedule}$ time interval, we would not know where the next-largest peak is in the time series history. We therefore need a way to not only capture the time series of the last $T_{schedule}$ interval, but also do so in a way that accurately captures the inflection points and does not affect performance of the memory allocator.

We achieve this goal by dividing the time series into 1s epochs. We do not update the time series on every allocation but only on those that cause changes to the page heap portion of the allocator (see Section 2.2). Within the allocator, we first check whether we need to advance the time series (which can be done with a quick timestamp counter read) and then record the maximum and minimum used/free memory for the current epoch, which gives us the inflection points, as depicted in Figure 6. Note that all of these operations take constant time and overheads are therefore bounded.

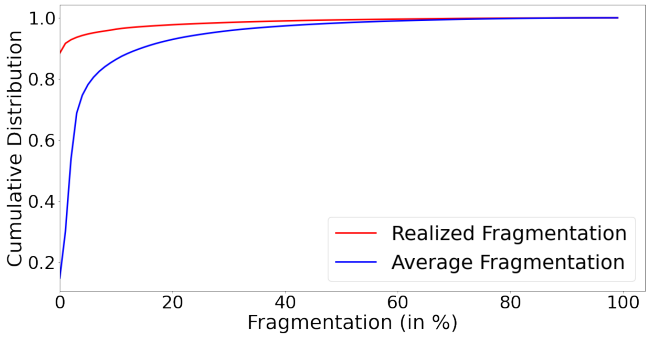


Figure 7. Fleet-wide distribution of fragmentation for a single day, measured by realized fragmentation ($T_{schedule}$ = 5 minutes) and average fragmentation. The y-axis normalizes to workload samples (not memory weighted). The average metric significantly over-reports fragmentation. While close to 85% of workloads have non-zero average fragmentation, almost 90% of workloads actually do not have any fragmentation that could be reused.

When TCMalloc metrics are queried, we go through the time series and look up the minimum free, mapped memory across all epochs – which we then report as the overall fragmentation for the workload. The metric is available upstream in open-source TCMalloc [8], listed in the "`time series over 5 min interval`" section of TCMalloc's statistics (in addition to other metrics derived from the time series).

3.4 Realized Fragmentation in Production

In our production deployment, we have a continuous monitoring system that regularly queries TCMalloc's various metrics across all workloads in the fleet [18], which allows us to collect realized fragmentation and other metrics fleet-wide. We configured $T_{schedule}$ = 5 minutes in our fleet wide metrics, which works well with other scheduling and reporting intervals in our fleet.

Figure 7 shows the cumulative distribution function (CDF) of both metrics for one day in the fleet. The realized fragmentation metric was zero for 90% of workloads in our deployment, whereas average fragmentation was zero for 15% of workloads using TCMalloc with huge pages and subrelease. These results revealed that TCMalloc was too aggressively releasing memory. We hypothesize that this is because engineering teams configure workloads based on the telemetry that they have and the existing telemetry suggested that there was a lot of fragmentation, implying that the release rate should be set higher to reduce this fragmentation. However, releasing this memory did not lead to any memory savings that could be reclaimed by the cluster scheduler. Furthermore, it reduced huge page coverage and induced system call overhead, causing applications to run slower than they would have with a lower release rate. With the high release-rate configuration, fleet-wide TCMalloc-level huge page coverage was at only 58%, indicating significant memory on broken huge pages (note that here and throughout, we exclude parts of the heap not managed by TCMalloc's *filler* [10], such as large allocations that are a multiple of 2 MB). We therefore set out to address this problem.

4 Adaptive Huge Page Release

While releasing memory back to the OS is an inherently dynamic problem, current memory allocators treat it as mostly static. Typically, there is a statically configured rate that determines how quickly free pages should be returned to the operating system (e.g., "release X free pages per second"). In addition, many allocators include a mechanism for an application to explicitly request freeing memory.

For example, jemalloc has a decay rate [5] that determines how rapidly pages are returned to the OS. In contrast, ptmalloc in glibc [7] allows setting a "trim" threshold to determine at how much free memory the allocator starts returning pages back to the OS. Finally, TCMalloc [8] has a static release rate that indicates the amount of free memory to return to the OS per unit time, if this memory is available. This rate is 1 MB/s per default. When used with huge pages, TCMalloc has an additional optimization that delays returning an entirely free huge page to the OS: It first places it into a cache, to allow the application time to use it and maintain this huge page. When returning memory to the OS, TCMalloc prefers to first release memory from this huge page cache before starting to break up partially free huge pages.

One of the shortcomings of these approaches is that they do not take into account when the memory that they are releasing will be needed again. These policies thus lead to the allocator releasing memory and then quickly requesting memory again from the OS. TCMalloc's huge page cache and jemalloc's decay rate try to partially address this problem for entirely free pages by allowing the allocator to buffer these pages for a period of time before returning them, but neither of these strategies can make good decisions when the trade-off is whether or not to break up a huge page. Without huge pages, returning and reacquiring memory has a fixed cost that only affects the operation itself. Breaking up a huge page, however, has performance implications for all future allocations and program TLB performance. Furthermore, neither policy is optimized for the scenario where the cluster scheduler cannot reuse memory unless it has been free for a particular amount of time.

Instead of using a static release rate, we therefore change the allocator to dynamically decide how much memory to release, based on predicted future demand. We make a simple assumption: We predict future from past memory demand by predicting that memory that has been used within the last time interval Δ will be used again within the next time interval Δ. This use of historical memory usage together with a statically or dynamically configured release rate leads

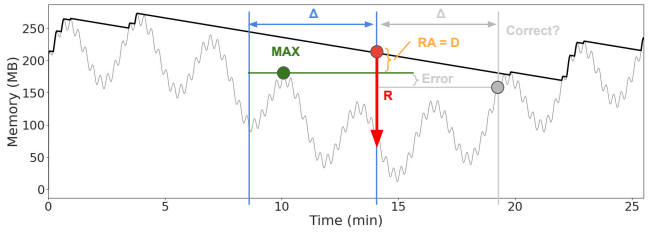


Figure 8. The adaptive subrelease policy and how our mechanism determines correctness after another interval Δ.

to the following adaptive release policy (which is triggered every time the memory allocator tries to release memory to meet its release rate):

1. Assuming there is sufficient free memory to satisfy a release request of X MB, we handle as much of this request as possible using unbroken free huge pages in the huge page cache. Let R equal any remainder, which requires breaking up huge pages.
2. We limit the actual release amount (RA) as follows. Let MAX equal the largest amount of used memory in the last interval Δ. We compute D as the difference between MAX and the currently mapped amount of memory. If $D \geq R$, $RA = R$. Otherwise ($D < R$), we release less than requested: $RA = \max(D, 0)$.
3. We then break up huge pages and subrelease any free constituent 4 KB pages until the amount totals to RA. The order in which to break up huge pages follows the allocator's existing heuristics.

Figure 8 visualizes this policy. While this policy still acts on a static parameter (i.e., the MB/s release rate parameter), it takes dynamic actions that depend on the state of the heap and the recent peak memory usage. The overall effect of this policy is that the amount of mapped (i.e., allocated to the application) memory never drops below the largest usage peak in the recent past. Over time, it will still return memory to the cluster scheduler, but at a rate such that the cluster scheduler is able to take advantage of the free memory and assign it to other applications, because this application is much less likely to need this memory immediately. We do not apply this policy to release requests due to reaching a memory limit, to avoid introducing out-of-memory errors.

4.1 Implementation

We implemented the release policy described in the previous section in TCMalloc by piggybacking on the fragmentation, in-use, and other memory metrics that it already gathers. Specifically, build on the time series data from Section 3. Adaptive huge-page subrelease is enabled and configured by setting a non-zero time interval Δ[1]. The allocator is thus still configured using a static release rate, but when the allocator releases memory due to a periodic release event (not through a user request), then the amount of memory to release is adjusted such that the allocated (mapped) memory never drops below the highest usage point within the last time interval Δ. The implementation examines the time series data (Section 3) and finds the largest usage at any given point in time. Since the largest usage amount is an inflection point, it is part of the time series.

Throughout the remainder of the paper, we examine heap usage history over the past 60 seconds (by setting Δ = 60 s). We use a default release rate of 1 MB/s in most of our experiments (10 MB/s in some of them).

4.2 Accuracy Metric

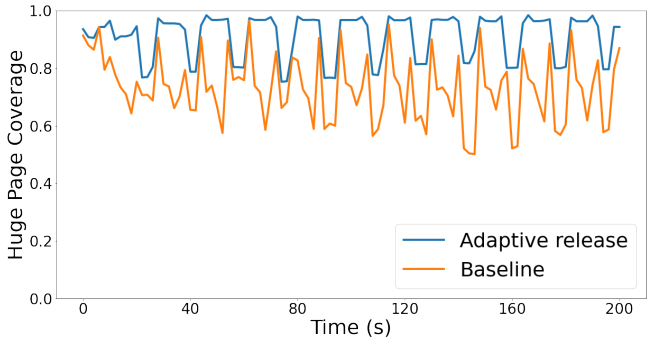
An important question is how often these subrelease decisions were correct. One specific concern when initially deploying adaptive huge-page subrelease was whether users would see their average fragmentation increase and believe that they are seeing a regression. To preempt this problem, we added a mechanism to compute how many of the skipped release decisions were correct, i.e., were indeed quickly needed again within the interval Δ.

This check can only be performed another time period Δ later, when we know whether or not a subsequent peak usage exceeded the target amount of used memory. Note that the mechanism can be partially right. For example, assume a workload with current usage of 500 MB, a recent peak usage of 1 GB, and mapped memory of 2 GB (i.e., 1.5 GB is currently free but mapped). Given a release request for 1.5 GB, our policy only releases 1 GB because of the recent 1 GB peak, even though in theory it could release up to 1.5 GB. If there is another 1 GB peak within the next interval Δ, this was the correct decision. However, if the next peak is only 750 MB, there are 250 MB that were incorrectly not released.

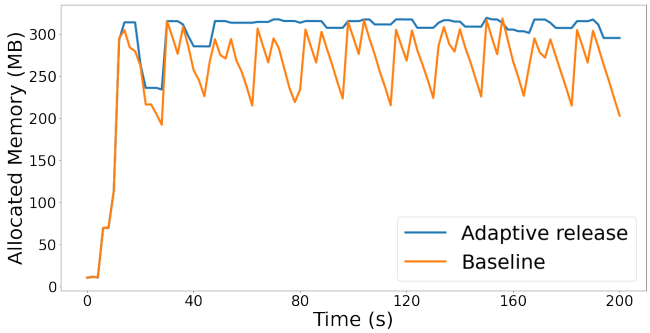
To track accuracy, we introduce a second time series tracker. When we forego releasing memory, we increment a per-epoch counter to keep track of how much memory we did not release due to a recent peak. Then, whenever we reach the end of an epoch in the future and there are pending subrelease decisions that were not confirmed yet, we go through the last interval Δ of the time tracker to check whether there were any skipped subreleases that are below the peak associated with the current epoch. If so, we can mark these as correct (note that we need to be careful not to double-count them in the future; in practice, this means that when going through the time series, we need to look at previous peaks and ignore anything below them). If a subrelease decision makes it to the end of the tracked range of the time series and has not been marked as correct, we can count it as incorrect. Figure 8 visualizes this approach (shown in gray).

We found the correctness tracking feature helpful for two reasons. 1) It helped us determine that the algorithm behaved as expected in the fleet, and 2) We used it to calibrate the

[1]This parameter is called `tcmalloc_skip_subrelease_interval` in upstream TCMalloc and can be configured using the `MallocExtension` interface (setting it to 0 disables the mechanism).



(a) Huge page coverage over time



(b) Average Fragmentation over time

Figure 9. Redis key-value store with and without adaptive subrelease. In figure (a), we see that huge page coverage improves substantially with the adaptive release policy. In (b), we see that the average fragmentation is slightly increasing. However, the residual fragmentation in both cases is 0 throughout the entire execution of the workload.

time interval Δ. The larger the Δ value, the more likely we are going to falsely hold on to too much memory. The shorter the Δ value, the more likely we are to falsely release it. By tracking the correctness, we can find a value for Δ that maximizes the amount of correct decisions. While this parameter could be tuned for every workload, we found Δ = 60 s to be a suitable parameter that worked reasonably well for most of the applications we looked at.

5 Evaluation

We now evaluate the effectiveness of our adaptive subrelease policy. We do so through a combination of experiments and real-world workloads deployed in the fleet. Our implementation is available in upstream TCMalloc [8].

5.1 Redis Workload

To evaluate how well our technique improves huge page coverage, we ran a Redis workload with and without the adaptive subrelease policy. We use the benchmark program that is part of Redis and configured it to run batches of 200K requests, with 1K parallel connections and 1K byte entries. We first pre-populate the Redis database with 200K such entries. We then run 10 iterations of adding 200K entries, waiting for 5 seconds, and then removing them again – emulating variation in demand. We ran with a 10 MB/s release rate and a 60s adaptive subrelease interval Δ. For reporting realized fragmentation, we use $T_{schedule}$ = 5min.

The adaptive release policy significantly improves huge page coverage (Figure 9a). With adaptive subrelease, the average huge page coverage is 91% vs. 73% without. Without adaptive subrelease, huge page coverage declines downwards over time. Note that while the average fragmentation increases slightly (Figure 9b), the realized fragmentation remains unchanged (0 in both cases).

5.2 Proprietary Workloads

We ran A/B experiments with a number of proprietary workloads in our fleet. By comparing the results of the system with and without adaptive huge page subrelease, we demonstrate the impact of the mechanism.

Web server. This workload serves queries for a large-scale web service. To compute the impact of adaptive huge page subrelease, we compared two different runs of an integration test of this service, one with adaptive huge page subrelease (Δ = 10min) and one without. Both used a 10 MB/s memory release rate and we ran both of them for ≈15 hours. Without adaptive huge page release, we observed a drop in huge page coverage to 87% (continually dropping) while huge page coverage remained at 98% with adaptive huge page subrelease. The effect was a 0.9% improvement in server throughput as measured in QPS. 60% of skip subrelease decisions were correct according to our measurements.

Storage service. We ran a test deployment of the adaptive huge page mechanism with a storage service backend. When deploying the mechanism, we observed a drop of about 50% in minor page faults – a meaningful reduction of cycles in its own right. We also observed that the number of huge pages that were broken up reduced by about 90% relative to the baseline without adaptive huge page subrelease.

Data analytics framework. We ran an A/B experiment with a large-scale data analytics framework. By deploying adaptive huge page subrelease to 50% of the services, those services saw a reduction in CPU cycles of 4-5%, as well as a 15%+ latency reduction at the 99%ile. Huge page coverage improved by 3.12×, which goes hand-in-hand with a reduction of broken huge pages by 4.5×.

These experiments demonstrated that the technique not only improved low-level metrics such as huge page coverage but also that it led to significant improvements in application throughput. Note that all of these workloads are highly tuned production services and that 1% improvement in either is a large improvement in these globally deployed services.

Table 1. Summary of our fleet-wide evaluation results (*latecy-sensitive* and *latency-tolerant* refers to workload type).

Metric	Control	Experiment
Application Throughput (latency-sensitive)	100%	100.77±0.03%
Application Throughput (latency-tolerant)	100%	101.44±0.03%
Cycles in DTLB load misses	11.71%	11.39%
Cycles in minor page faults	3.56%	3.39%
Huge page coverage	58%	72%
Number of hugepages broken (normalized)	100%	60%
Realized fragmentation (normalized)	100%	100.08%

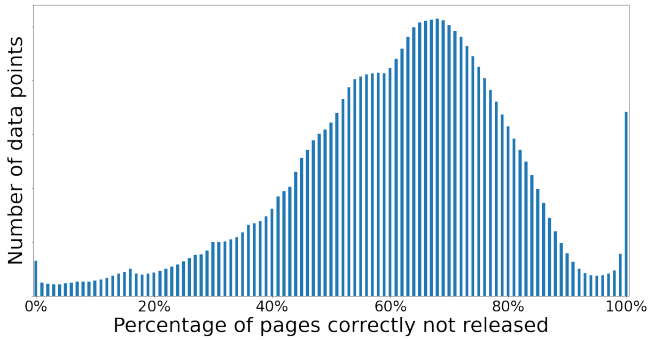


Figure 10. Fraction of correct release decisions across a random sample of workloads from one day in the fleet. We can see that decisions are mostly correct. Note that applications that never subrelease are always correct, which is part of the reason for the spike at 100%.

5.3 Fleet-Wide Evaluation

Following our initial evaluation, we rolled out the service fleet-wide and ran a large scale experiment where a small fraction of all servers in the fleet enabled the new mechanism and were then compared to a carefully chosen control group with the same workload mix and other workload properties. Application throughput is measured through end-to-end metrics such as QPS or CPU time, depending on the workload. Table 1 shows that adaptive huge page subrelease improves overall application throughput by 0.77% for latency-sensitive workloads and 1.44% for latency-tolerant workloads, over 1% in total. Note that these improvements are over a highly optimized baseline, and some workloads have already fine-tuned TCMalloc's release rate for their application.

These improvements go hand-in-hand with improvements in low-level metrics. Huge page coverage improves dramatically and the number of huge pages broken since the start of execution drops by a significant amount. Note that realized fragmentation does slightly increase. This is by design – we allow the application to hold on to more memory to avoid breaking up huge pages. However, the intention is that most of this memory is backed for less than 5 minutes, since such memory cannot be reused. Using our new telemetry, we see a 0.08% increase in memory consumption (within the noise). However, this measurement ignores any workload that has subrelease disabled, a small but not negligible set (the throughput numbers are fleet-wide). As such, we believe these memory overheads to be negligible.

5.4 Accuracy of Subrelease Decisions

We wanted to understand how often the mechanism makes the correct decision. To this end, we collected the fraction of correctly released pages across an entire day and found that a high fraction of these decisions was retroactively deemed correct (Figure 10).

5.5 Sensitivity to Configuration Parameters

Finally, we want to understand how setting the configuration parameters of the mechanism affects its performance. Adaptive huge page subrelease is affected by two parameters: TCMalloc's release rate and Δ.

To measure the effect of these parameters, we created a small microbenchmark that varies memory usage over time by emulating arbitrarily chosen spikes of memory usage of varying size and lifetime. We ran this benchmark with a range of different values for the two parameters. The results are shown in Figure 11. We observe the following:

- As expected, a release rate of 0 results in no subrelease and hence perfect huge page coverage. However, this comes at the cost of increased fragmentation (including realized fragmentation). Note that a "correct subrelease fraction" of 0 in this case simply means that there were no subrelease decisions made.
- Similarly, if Δ is 0 (i.e., adaptive huge page release is disabled), no subrelease decisions are made and higher release rates lead to lower huge page coverage.
- Increasing the release rate decreases both fragmentation and huge page coverage. It also leads to more subreleases being skipped, which is expected since there are more candidates for skipping.
- Increasing Δ results in more subreleases being skipped but also in a higher fraction of correct subrelease decisions. The reason is that our correctness checks only capture false positives, not false negatives.
- High huge page coverage and low fragmentation are often mutually exclusive. The design points that have high huge page coverage are those with large fragmentation and vice-versa.

These numbers show that both release rate and Δ have an impact on the overall huge page coverage. It is therefore necessary to tune both of them together. By looking at both the heatmaps for huge page coverage and realized fragmentation, we can pick a design point that achieves an acceptable trade-off between the two.

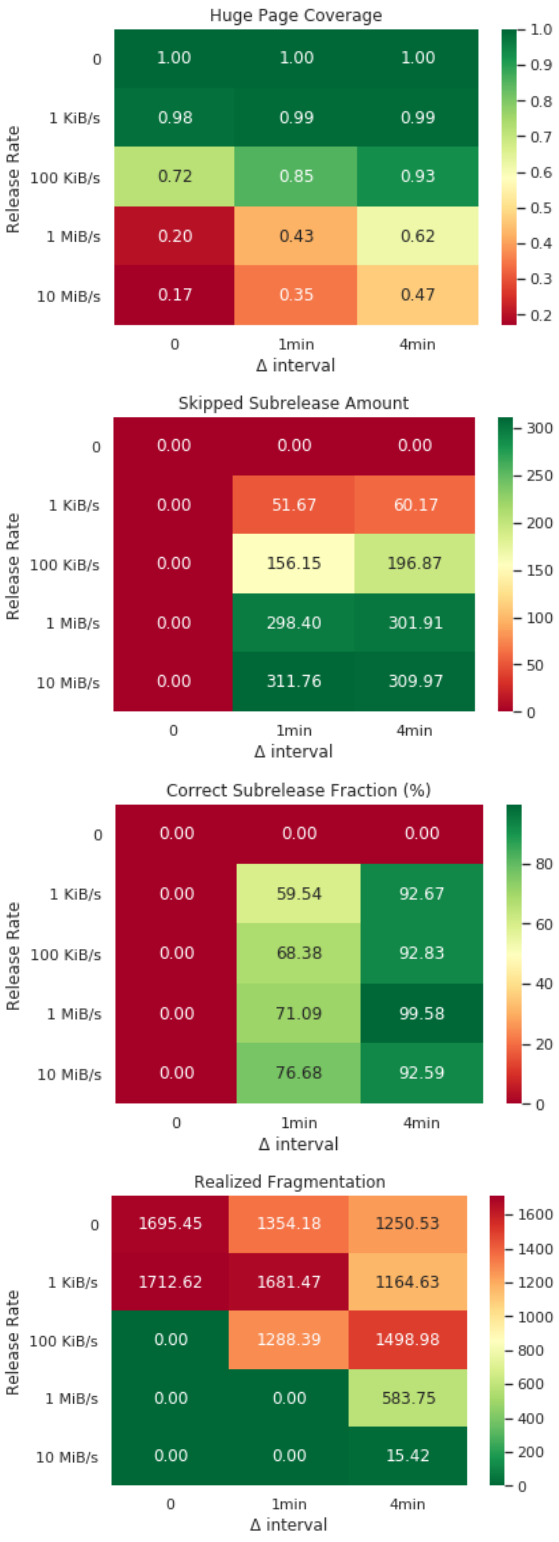


Figure 11. Varying both the memory release rate and the Δ interval. When Δ = 0, adaptive huge page release is disabled. When the memory release rate is 0, no pages are subreleased. All memory amounts are in MB.

6 Related Work

Fragmentation in C/C++ non-moving memory allocators has seen a significant amount of work over the years. While the fragmentation problem is considered mostly solved with conventional 4 KB pages [1, 11], managing user-level memory at the granularity of huge pages has been shown to introduce new fragmentation problems [14]. One recent memory allocator, LLAMA [14], addresses this problem by predicting object lifetimes using a machine learning model and using these lifetimes to ensure that huge pages become empty at similar points in time. This approach requires a trained lifetime model and a new hierarchical region memory allocator, which differs substantially from the free-lists implemented in most other explicit memory allocators. Another solution, Mesh [17], combats fragmentation through compaction, by mapping multiple pages on top of each other. However, this approach requires a new randomized allocation approach. In contrast, our approach is non-invasive and can easily be integrated into existing free-list allocators.

Our approach has similarities to recent work on huge page management at the OS level. For example, Ingens [13] proposes to treat huge page availability as a first-class resource that it aims to preserve – this is similar to how our approach treats huge pages at the user level. In contrast to our work, however, kernel-level work often focuses on improving compaction techniques [15, 16], an option that we do not have in a non-moving application-side allocator.

Garbage collection algorithms that move objects, such as those that are commonly used in Java Virtual Machines, face similar fragmentation problems, but they can move objects. They thus copy objects to limit the kind of fragmentation that requires subrelease in a non-moving environment. Examples of algorithms that successfully minimize fragmentation include Immix [2], G1 [4], C4 [20], ZGC, and Shenandoah [6]. As far as we are aware however, the integration and impact of huge pages on these approaches has not been closely examined. While moving objects is another way to sidestep the problem we are solving in this paper, this solution is not viable in a C++ environment since objects are non-movable.

7 Conclusion

We demonstrated that the fragmentation in warehouse-scale computers should not only take into account the average or maximum free memory in an individual workload but instead consider whether the memory returned to the OS is free for long enough that it could be reused by other workloads. We introduced *realized fragmentation*, a new fragmentation metric that accounts for this effect, and demonstrated that using average fragmentation, which is standard, significantly overestimates fragmentation. We then demonstrated a technique to adaptively handle huge page subrelease decisions, leading to over 1% throughput improvement in fleet-wide deployment, in production data centers.

Acknowledgments

We want to thank Ethan Lin for helping us gather and analyze performance results. We also want to thank Chandu Thekkath, James Laudon, and the anonymous reviewers for feedback on drafts of this paper.

References

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, Massachusetts, USA) *(ASPLOS IX)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[2] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: A Mark-region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. 22–32. https://doi.org/10.1145/1375581.1375586

[3] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGPLAN Not.* 49, 4 (Feb. 2014), 127–144. https://doi.org/10.1145/2644865.2541941

[4] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*. 37–48.

[5] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference* (Ottawa, Canada).

[6] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. ACM, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[7] Wolfram Gloger. 1997. Dynamic memory allocator implementations in Linux system libraries. http://www.malloc.de/papers/malloc-slides.html

[8] Google. 2020. TCMalloc. https://github.com/google/tcmalloc

[9] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 295–308.

[10] Andrew Hunter, Chris Kennelly, Darryl Gove, Parthasarathy Ranganathan, Paul Turner, and Tipp Moseley. 2021. Beyond malloc Efficiency to Fleet Efficiency: A Hugepage-Aware Memory Allocator. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation (OSDI'21)*. USENIX Association, USA.

[11] Mark S. Johnstone and Paul R. Wilson. 1998. The Memory Fragmentation Problem: Solved?. In *Proceedings of the 1st International Symposium on Memory Management* (Vancouver, British Columbia, Canada) *(ISMM '98)*. ACM, New York, NY, USA, 26–36. https://doi.org/10.1145/286860.286864

[12] Bradley C. Kuszmaul. 2015. SuperMalloc: a super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA*. 41–55. https://doi.org/10.1145/2754169.2754178

[13] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 705–721. http://dl.acm.org/citation.cfm?id=3026877.3026931

[14] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[15] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-Grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 347–360. https://doi.org/10.1145/3297858.3304064

[16] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. ACM, New York, NY, USA, 679–692. https://doi.org/10.1145/3173162.3173203

[17] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. 2019. Mesh: Compacting memory management for C/C++ applications. In *ACM Conference on Programming Language Design and Implementation(PLDI)*. 333–346. https://doi.org/10.1145/3314221.3314582

[18] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* (2010), 65–79. http://www.computer.org/portal/web/csdl/doi/10.1109/MM.2010.68

[19] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. 2013. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 351–364. https://doi.org/10.1145/2465351.2465386

[20] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the international symposium on Memory management*. 79–88.

[21] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.