# Characterizing a Memory Allocator at Warehouse Scale

Zhuangzhuang Zhou*
Cornell University

Vaibhav Gogte
Google

Nilay Vaish
Google

Chris Kennelly
Google

Patrick Xia
Google

Svilen Kanev
Google

Tipp Moseley
Google

Christina Delimitrou
MIT

Parthasarathy Ranganathan
Google

## Abstract

Memory allocation constitutes a substantial component of warehouse-scale computation. Optimizing the memory allocator not only reduces the datacenter tax, but also improves application performance, leading to significant cost savings.

We present the first comprehensive characterization study of TCMalloc, a memory allocator used by warehouse-scale applications in Google's production fleet. Our characterization reveals a profound diversity in the memory allocation patterns, allocated object sizes and lifetimes, for large-scale datacenter workloads, as well as in their performance on heterogeneous hardware platforms. Based on these insights, we optimize TCMalloc for warehouse-scale environments. Specifically, we propose optimizations for each level of its cache hierarchy that include usage-based dynamic sizing of allocator caches, leveraging hardware topology to mitigate inter-core communication overhead, and improving allocation packing algorithms based on statistical data. We evaluate these design choices using benchmarks and fleet-wide A/B experiments in our production fleet, resulting in a 1.4% improvement in throughput and a 3.4% reduction in RAM usage for the entire fleet. For the applications with the highest memory allocation usage, we observe up to 8.1% and 6.3% improvement in throughput and memory usage respectively. At our scale, even a single percent CPU or memory improvement translates to significant savings in server costs.

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

*Work performed while at Google.



*ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA*

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651350

***Keywords:*** Datacenter, Warehouse-Scale Computing, Memory Allocator, Memory Management

**ACM Reference Format:**
Zhuangzhuang Zhou, Vaibhav Gogte, Nilay Vaish, Chris Kennelly, Patrick Xia, Svilen Kanev, Tipp Moseley, Christina Delimitrou, and Parthasarathy Ranganathan. 2024. Characterizing a Memory Allocator at Warehouse Scale. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651350

## 1 Introduction

The datacenter tax [29, 34, 59, 62] within a warehouse-scale computer (WSC) represents the cumulative time spent on common service overheads, including serialization, remote procedure calls, compression, hashing, data movement, and memory allocation. The diversity of WSC workloads [34] implies that optimizing a single application may not yield substantial improvements in system efficiency for the entire fleet, as the costs are distributed across numerous independent workloads. On the contrary, optimizing components of datacenter tax can significantly improve the efficiency and performance of the fleet, since entire classes of WSC applications benefit from the improvements made.

In this paper, we focus on improving the memory allocation and deallocation process, which constitute a substantial component of warehouse-scale computation [34]. We focus on memory allocator optimizations that maximize the productivity of WSCs by doing more useful work with the same or fewer hardware resources. Memory allocators directly affect the data locality of allocated objects and provide significant opportunities to optimize application performance. In comparison, optimizing the amount of time spent in the allocator itself is less important. Prior work profiles the CPU usage of memory allocators in datacenters [29, 34, 63] or measures the allocator performance using sets of benchmarks [10, 15, 25, 30, 42, 45, 46, 55, 67]. However, these studies solely focus on the time spent in the allocator or use benchmarks with limited memory allocation patterns, and

thus provide only a narrow understanding of the performance characteristics and memory allocation behavior of WSC workloads. To fill this gap, we present the first comprehensive characterization study of TCMalloc [10], a memory allocator used by WSC applications in Google's global datacenter fleet. We collect fleet-wide statistics from production workloads, and perform a detailed quantitative analysis of general memory allocator properties and memory allocation behaviors of WSC workloads: the latency of allocation for different levels of allocator caches, the CPU cycles and memory fragmentation breakdown, and the distribution of allocated object sizes and their lifetimes. We also take an in-depth look at each component in the TCMalloc cache hierarchy, from the front-end per-CPU cache to the back-end pageheap [33], to identify performance bottlenecks resulting from the diverse allocation characteristics of warehouse-scale applications running on a fleet of heterogeneous servers.

Our characterization reveals profound diversity in memory allocation patterns, hardware platforms, as well as allocated object sizes and lifetimes for WSC workloads. We observe that workloads are often co-located, and constrained to run on a subset of CPUs by the control plane. The dynamic input load causes the number of worker threads of a WSC application to fluctuate constantly, resulting in significant variation in the cache miss ratio across the allocator's front-end caches. We show that the heterogeneity in our server fleet (e.g., differences in cache topologies of hardware platforms) can lead to varied data transfer overheads and increased cache pressure. We also observe that the distribution of object lifetimes varies across different sizes, which makes it challenging to make allocation packing decisions at different granularities (e.g., spans, hugepages) to reduce memory fragmentation and improve hugepage coverage [33, 48, 49, 70].

Based on our characterization, we derive unique insights and use them to design a memory allocator for WSC applications. In particular, such an allocator needs to (1) adapt to the dynamic resource usage of WSC applications, (2) be aware of heterogeneity in hardware platforms, and (3) utilize diverse lifetime information to make memory packing decisions. While our characterization study centers on TCMalloc, most modern memory allocators (e.g., jemalloc [25], mimalloc [42]) share a similar hierarchical system architecture and cache memory allocations in multiple tiers, making these insights universal to memory allocators used in WSCs. Based on these insights, we redesign and tune each component in the TCMalloc cache hierarchy for WSC environments, including enabling usage-based dynamic sizing of per-CPU caches, leveraging the hardware topology to mitigate the inter-core communication overhead in the transfer cache, and improving the allocation packing algorithms based on statistical data in the central free list and the pageheap. We evaluate these design choices with fleet-wide A/B experiments and longitudinal rollout in our production WSCs. Evaluation results show that by redesigning the memory allocator for warehouse-scale environments, we achieve a significant improvement in fleet productivity.

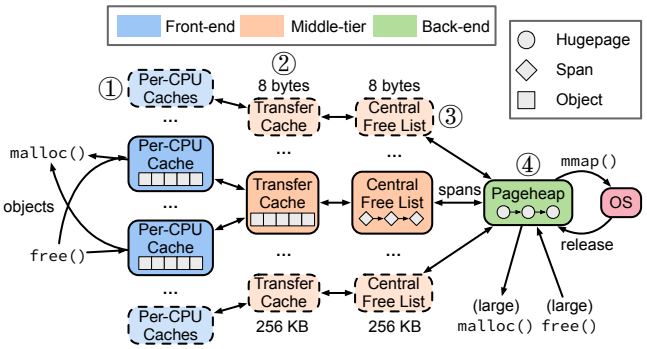



**Figure 1.** System architecture of TCMalloc. It has a tiered cache structure that aids fast allocations and deallocations.

This paper makes the following main contributions:

- The first comprehensive characterization study of TCMalloc, a memory allocator used in warehouse-scale environments. Based on profiling data collected from production WSC workloads, we characterize the general memory allocator properties, and delve into each tier of the TCMalloc cache hierarchy.
- Based on our characterization, we uncover insights into designing a memory allocator for WSC applications, including adapting to dynamic application resource usage, being aware of server heterogeneity, and leveraging object lifetime information to improve allocation placement decisions.
- We redesign and tune each component in the TCMalloc cache hierarchy for warehouse-scale environments and evaluate their performance impact through fleet-wide experiments, resulting in a 1.4% throughput improvement and a 3.4% memory reduction across the fleet. For the applications with the highest `malloc` usage, we observe up to 8.1% and 6.3% improvement in throughput and memory usage respectively. At our scale, a single percent CPU or memory improvement translates to significant resource savings.

## 2 Background and Methodology

TCMalloc [10] is a memory allocator used in warehouse-scale environments. It is a fast, multi-threaded `malloc` implementation that has shown robust performance in large-scale production services [33, 35, 48, 49].

### 2.1 TCMalloc System Architecture

Figure 1 shows the hierarchical system architecture of TCMalloc. In TCMalloc, allocations of small objects (i.e., ≤ 256 KB) are rounded up to one of 80–90 size classes. Objects of each size class are cached by multiple *per-CPU caches* (①),

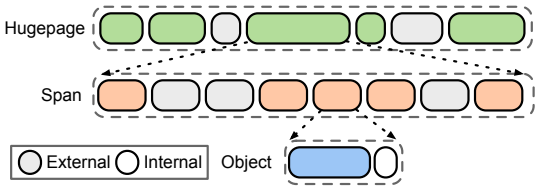



**Figure 2.** Memory organization layout and fragmentation in TCMalloc. Hugepages are divided into spans of various sizes, and spans are sub-divided into objects of fixed size classes.

a *transfer cache* (②), and a *central free list* (③). Large objects that exceed the threshold are directly allocated from the *pageheap* (④), without being cached by the front-end or middle-tier caches.

The front-end contains per-CPU caches (①) that provide fast memory allocation and deallocation for the application. The per-CPU caches store *objects* in a large contiguous block of memory that is divided between CPUs, and each CPU uses a portion of the block to store metadata and pointers to the available objects. Each per-CPU cache can only be accessed by a single thread at a time. Therefore, no locks are required and most operations are fast. When the front-end is empty, it requests objects from the middle tier to refill the cache.

The middle tier consists of small, fast, mutex-protected transfer caches, and large, mutex-protected central free lists. The transfer cache (②) stores objects in flat arrays. It allows memory to rapidly flow between different CPUs (e.g., CPU 0 may allocate memory that is deallocated by CPU 1). If the transfer cache is unable to satisfy the memory request, or has insufficient space to hold the returned objects, it reaches out to the central free list. The central free list (③) manages *spans* in linked lists, and fulfills allocation requests by extracting objects from the spans. A span is a collection of contiguous fixed-size regions, aligned to an 8 KB TCMalloc *page* [1]. As shown in Figure 2, a span contains multiple objects of the same size class. If there are insufficient available objects in the spans, more spans are requested from the back-end.

The back-end pageheap (④) manages memory in units of hugepages [33]. The pageheap requests hugepage-aligned memory blocks from the system, which provides an opportunity for the kernel to use hugepages to cover consecutive pages in the page table [12]. A hugepage is divided into TCMalloc pages, and the pageheap extracts spans from hugepages to refill the central free list. The pageheap also periodically releases memory to the OS, either by releasing hugepages that are completely free, or by breaking partially-filled hugepages into smaller pages and subreleasing them [33, 49].

**Memory organization.** Storing and managing memory at different granularities can lead to *external* and *internal* *fragmentation* in TCMalloc, as shown in Figure 2. External fragmentation refers to the memory that is cached by the allocator, but is yet to be allocated by the application. For instance, a hugepage may contain unallocated memory regions in the spans, while a span may contain unallocated objects. In contrast, internal fragmentation in TCMalloc results from rounding allocation requests to discrete size classes. As such, it is slack between the object size requested by the application and the size class allocated by the allocator. It is critical to manage both internal and the external fragmentation to minimize memory overheads in TCMalloc.

[1]TCMalloc page size should not be confused with the system page size – the default 8 KB TCMalloc page composes of two native x86 memory pages.

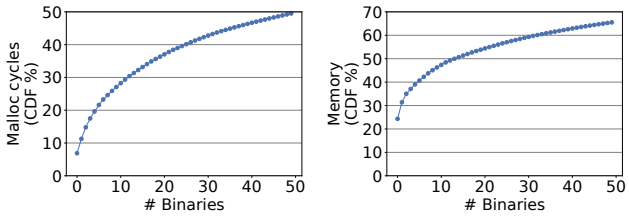
### 2.2 Characterization Methodology

We put in place telemetry for collecting fleet-wide statistics from production workloads. We use these statistics to perform a detailed characterization of TCMalloc.

**Application productivity.** Prior characterization studies focus on "datacenter tax" [29, 34, 62, 63] for common libraries in WSC applications and propose several acceleration opportunities to lower their CPU overhead. In this work, we argue that optimizing for the CPU overhead of these libraries is less important. Instead, it is more crucial to focus on improving the efficiency of these common libraries to improve fleet *productivity*, i.e., fulfilling more requests with the same or fewer hardware resources. Prior work [34, 63] analyzes processor pipeline stalls in WSC applications, attributing 20-64% stalls to back-end, primarily due to cache misses. Memory allocators directly impact the data locality of allocated objects, and thus, present significant opportunity to optimize application performance bottlenecks at scale. To this end, we focus on improving application productivity and showing how improvements to cache and dTLB locality impact WSC productivity, without directly targeting improvements to the `malloc` CPU overhead. Our fleet productivity metrics consist of per-application-defined custom throughput metrics (e.g., RPCs processed per second) that define performance for the respective applications.

**Continuous profiling.** We collect performance metrics and memory allocator telemetry from the production fleet using Google-Wide Profiling [57] (GWP), an unobtrusive profiling framework with negligible overhead. GWP randomly selects a small fraction (i.e., 1%–10%) of machines in the fleet to profile each day, and triggers profile collection remotely on each machine for a brief period of time. Continuous profiling allows us to study the memory allocation behavior of applications across the fleet at different levels of granularity and time intervals.

**Fleet experiment.** The diversity of WSC applications implies that there is no single killer application to optimize for. Figure 3 shows the fleet-wide cumulative distribution of `malloc` cycles and allocated memory, where the top 50 binaries account for over 50% `malloc` cycles and 65% of allocated memory. To measure the impact of optimizations on fleet productivity, we use an experimentation framework to A/B test




**(a)** CDF of malloc cycles. **(b)** CDF of allocated memory.

**Figure 3.** `malloc` cycle and allocated memory distribution in our fleet. The top 50 binaries in WSCs only cover ≈ 50% malloc cycles and ≈ 65% allocated memory.

implementations across our fleet at scale. For each design, the framework randomly selects 1% of the machines in the fleet as an experiment group and a separate 1% as a control group. We apply the change to all the binaries running in the experiment group and compare their performance with the control group. This lets us evaluate design choices on diverse production applications with realistic loads.

**Generalizability.** Our characterization study focuses on TCMalloc, and the design choices are closely related to TCMalloc's internal implementation. However, most modern memory allocators (e.g., jemalloc [25], mimalloc [42]) share similar hierarchical architecture and cache memory allocations in multiple tiers, which makes the insights derived from our characterization universal to memory allocators in warehouse-scale environments. For example, jemalloc has a tiered cache structure consisting of thread caches and arenas, in which it organizes memory in regions, runs and extents [25]. These memory allocators can also benefit from a comprehensive characterization of TCMalloc, and adopt similar optimizations to improve their performance at scale.

### 2.3 Production Workloads and Benchmarks

In addition to the fleet-wide metrics, we also use five production workloads in our fleet with the highest `malloc` usage for characterization and evaluation.

- **Spanner** [19] is a node in a distributed SQL database. It includes an in-memory cache of storage data, which adapts to the memory provisioned for the process.
- **Monarch** [13] is part of a scalable monitoring system that collects and stores time-series metrics for production services. It is responsible for holding stream data in memory, and participating in query evaluation.
- **Bigtable** [16] is a tablet server that hosts and serves the user data of a large-scale key-value NoSQL database. It also implements replication and coordinates compactions with external compactors.
- **F1 query** [58] is a high-performance distributed query engine. It uses RPCs to communicate with clients and data sources.

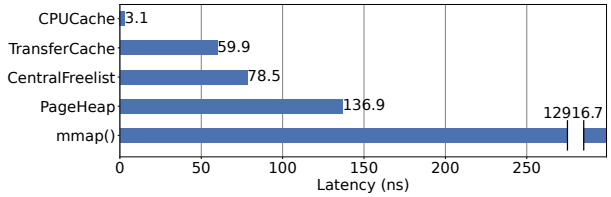



**Figure 4.** Disparity in allocation latency of hitting different tiers in the TCMalloc cache hierarchy.

- **Disk** is a low-level distributed storage system that provides RPC access to read and write files directly to a machine's local hard disk or flash memory.

We also run benchmarks on a dedicated server to demonstrate the performance impact of several optimizations.

- **Redis** [6] is an open-source in-memory key-value store (v7.0.8). We use the standard redis-benchmark, configured with 500 concurrent connections and 100K operations of 1000B as the workload generator.
- **Data processing pipeline** is a data processing workload running word count on a 1 GB file with 100M words. We run the entire computation as a single process, which creates pressure on memory allocation.
- **Image processing server** is a production server that filters and transforms images. We use a synthetic workload generator to create concurrent client requests.
- **Tensorflow** is the open-source Serving [51] framework that runs the InceptionV3 [65] image recognition model. It uses libraries (e.g. Eigen linear algebra library [2]) with complex memory allocation behavior.

## 3 General Characterization

We first conduct a general characterization to gain insights into how the memory allocator behaves in our production fleet and how its characteristics affect system performance.

**Allocation latency.** We use microbenchmarks to measure the mean allocation latency for hitting different tiers of caches. As shown in Figure 4, allocations fulfilled by the per-CPU cache have the lowest latency, since it stores objects in a contiguous block of memory, and uses a highly optimized fast path supported by restartable sequences [7, 8] to handle allocation requests. The fast-path that hits the per-CPU caches consists of ~40 hand-coded x86 instructions, with an allocation latency of 3.1 ns. Hitting the transfer cache and the central free list indicates that the front-end cache is empty, and needs to be refilled with a batch of objects. Both the transfer cache and the central free list are protected by mutex locks, which denotes an additional cost to access them. The central free list needs to extract objects from spans organized in linked lists, resulting in increased latency.

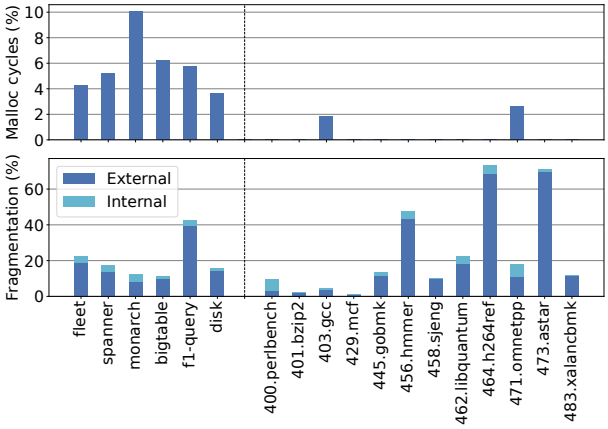



**Figure 5.** (a) Relative amount of time (% of cycles) spent in memory allocation, and (b) memory fragmentation ratio for the fleet and top 5 production workloads in two weeks. We also include SPEC CPU2006 benchmarks for comparison.

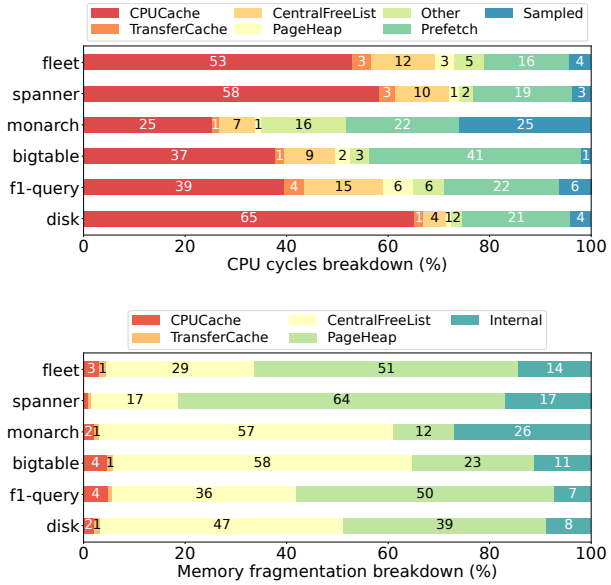



**Figure 6.** (a) Breakdown of CPU cycles consumed by TCMalloc. (b) Memory fragmentation breakdown of TCMalloc.

If both the front-end and the middle-tier caches are empty, the allocation request hits the pageheap, which holds memory in units of hugepages [33]. The pageheap tracks allocations in hugepages and results in the longest latency of over 137 ns in the cache hierarchy. We also include the latency of the `mmap` system call used in TCMalloc measured with strace [9]: when the allocation request misses all the front-end and middle-end caches, and the back-end page heap is also empty, TCMalloc requests a zero-initialized 2MB hugepage from the system using `mmap` to refill the page heap. The latency of refilling the pageheap is orders of magnitude higher than the latency of hitting the caches, highlighting the need for caching in a userspace allocator.

**Malloc CPU cycles.** Memory allocation and deallocation make up a substantial component of warehouse-scale computation. Figure 5a shows the relative amount of CPU cycles spent in allocation and deallocation functions over a two-week period, where the `malloc` overhead accounts for 4.3% fleet CPU cycles. For the top 5 applications with the highest `malloc` cycles in the fleet, the `malloc` overhead varies between 3.6%–10.1%. While understanding the `malloc` CPU overhead itself helps prioritize optimization opportunities, as we explained earlier in Section 2.2, we primarily aim to improve overall fleet productivity at scale.

We also include data collected from SPEC CPU2006 [32] benchmarks for comparison. Most of the SPEC benchmarks do not actively allocate or deallocate objects in stable state and have near-zero `malloc` cycles, which makes them unsuitable for studying memory allocation behavior.

**CPU cycles breakdown.** We classify the profiled call stack traces into several categories to further understand the breakdown of the CPU cycles used by the memory allocator. As shown in Figure 6a, TCMalloc spends most of its time (i.e., 53% of fleet-wide `malloc` cycles) in the per-CPU cache, since most of the requests hit the front-end cache. Allocation requests fulfilled by the per-CPU caches have the lowest latency (as described earlier in Figure 4), so we expect TCMalloc to spend most of its CPU cycles serving requests from front-end caches. Another 3% of the `malloc` cycles are spent in the transfer cache. The central free list accounts for 12% of the fleet `malloc` cycles, since it employs a linked-list structure to manage spans that incur higher cost to allocate objects from and deallocate objects to. Finally, TCMalloc spends 3% of its CPU cycles in the pageheap.

In the production setting, TCMalloc samples an allocation request for every 2 MB of memory allocations. *Sampled* accounts for time spent in sampled allocations, where the allocator additionally records the current call stack trace. Sampling accounts for 4% of `malloc` cycles, but in a production environment, it is invaluable for analyzing memory usage and debugging memory leaks. Some fleet applications (e.g., `monitor`) employ extensive sampling, so `Sampled` accounts for higher proportion of CPU cycles. *Other* refers to CPU cycles that were not classified into a specific category (e.g., due to allocations that require complex logic).

For each allocation request, TCMalloc prefetches the next object of the same size class that would be returned. It is too late to prefetch the current object when it is returned [41]: the user code can start using the object within a few cycles, well before prefetching from the main memory can complete. Prefetching gives time for the next object to be loaded into the cache before the next allocation request. *Prefetch* appears to be costly, taking 16% of `malloc` cycles in the fleet, but is key in reducing data cache misses.

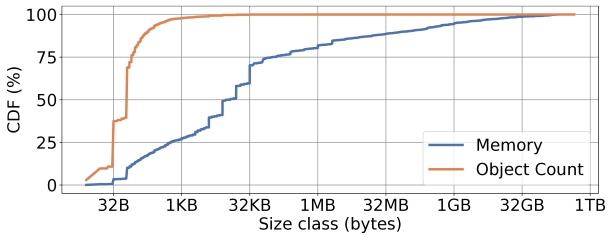



**Figure 7.** CDF of allocated objects in WSC applications.

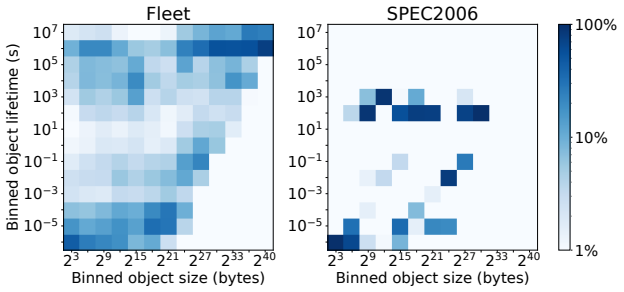



**Figure 8.** Distribution of fleet-wide object lifetime, based on object size and weighted by the number of sampled allocations. We also include the object lifetime distribution of SPEC CPU2006 benchmarks.

**Memory fragmentation.** The *fragmentation ratio* is the ratio of the fragmented memory to the live in-use memory by the application. Figure 5b shows the average memory fragmentation ratio for the fleet and the top 5 applications. The fleet-wide fragmentation ratio is 22.2% of the total application heap size, which consists of 18.8% external fragmentation (i.e., unused memory cached by the allocator) and 3.4% internal fragmentation (i.e., the slack between the allocated size class and the requested object size). For the top 5 applications with the highest allocator usage, the fragmentation ratio ranges from 11.2% to 42.5% of the respective heap size.

**Memory fragmentation breakdown.** Figure 6b decomposes external fragmentation into fragmentation within each component of the TCMalloc cache hierarchy. The major sources of fragmentation are the central free list and the pageheap, which account for 29% and 51% of the total fragmentation respectively. A span in the central free list can be returned to the pageheap only when all objects are returned to it. A single long-lived object on a span may disallow the central free list to return that span, leading to substantial memory fragmentation. The pageheap manages free spans in hugepages, and accounts for the majority of memory fragmentation in TCMalloc. It can wait for an entire hugepage to become free before releasing the memory back to the OS, or subrelease a hugepage [49] by breaking it into non-hugepage-aligned memory regions. The former preserves hugepage coverage but leaves memory idle, while the latter leads to performance degradation due to decreased hugepage coverage. TCMalloc prioritizes keeping hugepages intact [33, 49] by releasing memory gradually from the pageheap.

Internal fragmentation accounts for 15% of the fleet fragmentation, which results from the slack between the requested object sizes and the next available size class. TCMalloc may use finer-grained size classes to reduce the gap between requested memory and allocated size classes, but this also prevents reuse of memory blocks in the hierarchy. With finer-grained size classes, TCMalloc needs to manage additional per-size-class free lists in its front-end and middle-tier caches, increasing external fragmentation and reducing object reuse. Through its size class selection, TCMalloc strikes a balance between the internal fragmentation due to the number of size classes and the external fragmentation from unused memory blocks in its cache hierarchy.

**Distribution of allocated objects.** To study the distribution of allocated objects in production, we sample the memory allocations in the fleet over a two-week period. Figure 7 shows the cumulative distribution of the number of allocated objects and memory in the fleet, as a function of object size. Objects smaller than 1 KB make up 98% of the allocated objects, but occupy only 28% of the memory. However, when we focus on the total allocated memory size per size class, objects larger than 8 KB, account for 50% of the fleet memory. The largest size class in TCMalloc is 256 KB. Objects that exceed this threshold bypass TCMalloc's cache hierarchy and are directly allocated from the pageheap. They account for 22% of the allocated memory. To avoid excessive fragmentation due to these objects, the pageheap maps these allocations in a separate set of a continuous run of hugepages, as we discuss in Section 4.1. The distribution of allocated objects shows that small objects occupy only a fraction of the memory but dominate the total number of allocated objects. Therefore, the TCMalloc caches prefer optimizing available capacity towards smaller size classes to reduce overall allocation latency.

**Distribution of object lifetime.** Figure 8 shows the distribution of object lifetime, based on object size and weighted by the number of sampled allocations. We collect object lifetime profiles from servers with uptime of at least a week. We observe that objects have diverse lifetimes. For objects smaller than 16 MB, they can be long-lived (i.e., $\geq$ 7 days) or short-lived (i.e., $\leq$ 1 millisecond), or somewhere in between. Lifetimes vary greatly even for objects within the same size range. In general, smaller objects ($\leq$ 1 KB) are heavily allocated by our applications (as shown earlier in Figure 7), and also have shorter lifetimes, with 46% of objects living shorter than 1 millisecond. We also observe that large objects are likely to have longer lifetimes, where 65% of objects that are larger than 1 GB live longer than 1 day. This diversity in object lifetimes provides an interesting opportunity for the allocator to place objects with similar lifetimes together in the cache tiers (e.g., to reduce external fragmentation).

In Figure 8, we also include the object lifetime distribution sampled from SPEC CPU2006 [32] benchmarks. We run each benchmark to completion and combine the lifetime profiles together. The object lifetime distribution in SPEC benchmarks is much less diverse than what we observe in the fleet. These benchmarks do not actively allocate or deallocate objects in their stable state. Most objects are either alive as long as the program lives or only live for a short period of time (i.e., $\leq 1$ ms). This again makes SPEC benchmarks unsuitable for evaluating the performance of memory allocators.

## 4 Characterizing and Redesigning Caches

Next, we take an in-depth look at each tier in the TCMalloc cache hierarchy to uncover performance insights and potential optimization opportunities. For each tier, we perform a performance characterization, derive performance insights, propose new designs, and evaluate their performance impact.

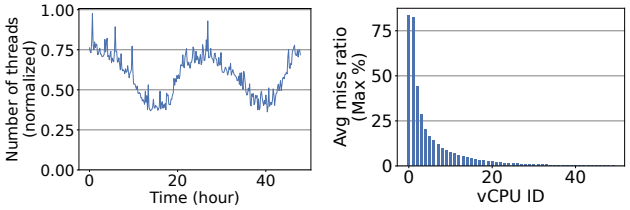
### 4.1 Per-CPU Cache

The per-CPU cache in TCMalloc is the front-end cache that provides fast allocation and deallocation of memory to the application. A per-CPU cache [2] is shared by the software threads scheduled to run on a given CPU core, which allows for more efficient use of the cache. The per-CPU cache contains a single large block of memory that is divided between CPUs. Each CPU is assigned a section of this memory to hold metadata and pointers to the available objects of a particular size class. The block size provides a bound on the capacity that TCMalloc may cache in the front-end caches.

**Virtual CPUs.** The per-CPU caches are populated for all the CPUs on which the application runs. WSC applications are often co-located on the same server, and are constrained to run on a subset of CPUs by the control plane scheduler [1, 17, 47]. For applications that use only a part of the machine, the available CPU range is excessive. In our fleet, we have observed a 4× increase in the number of hyperthreads per server system over the last five platform generations. As the number of hyperthreads increases, the per-CPU caches and the associated metadata may grow substantially between platform generations, even though the populated caches are not effectively used by the applications.

To improve scalability across platform generations, TCMalloc makes use of virtual CPU (vCPU) IDs [18], which are managed by the kernel in process-private number space. The vCPU IDs are assigned to prevent TCMalloc from initializing and maintaining per-CPU data structures for all CPU IDs available on the platform. By using a dense set of vCPU IDs to index the per-CPU cache, TCMalloc significantly reduces the number of unique CPU caches that need to be populated, and avoids populating caches on all *accessible* cores. For example, if an application runs on two CPU cores, virtual CPUs always expose IDs `0` and `1`, irrespective of which physical cores the application threads are scheduled on.

[2]TCMalloc now uses the per-CPU cache as its front-end cache by default, which is a major improvement over earlier versions that used per-thread caches. Being inaccessible to other application threads, per-thread caches strand memory when the threads become idle. The scalability becomes worse in applications with thousands of threads. Unfortunately, this also makes TCMalloc, *a thread-caching malloc*, a misnomer.




**(a)** Threads count. **(b)** Average miss ratio.

**Figure 9.** (a) The dynamic nature of WSC workloads. The number of active threads constantly fluctuates. (b) Significant variation in miss ratio of per-CPU cache for different vCPU IDs. Higher-indexed caches are inefficiently used.

**Disparity in cache usage.** By default, each per-CPU cache is statically sized to store up to 3 MB of objects. While the vCPUs reduce the number of used caches, we observe that they also bias cache usage towards the lower-indexed per-CPU caches. Figure 9a shows the number of worker threads of a middle-tier service in our search service stack. We can see that the number of worker threads constantly fluctuates, due to load spikes and diurnal usage. As such, datacenter applications typically handle dynamic loads by varying the number of CPU cores they use. A sudden burst of load may populate caches for the higher-indexed vCPUs, but the usage of these caches may subside as the load decreases.

We notice the disparity in cache usage based on the cache miss ratio. We collect the number of misses, i.e., the number of allocation and deallocation misses due to insufficient cache capacity, for all per-CPU caches with different vCPU IDs over a two-week period. Deallocation misses occur when the application frees an object, and the corresponding front-end cache is full and does not have sufficient capacity for the returned object. In such cases, the request spills over to the transfer cache. Figure 9b shows the average ratio of the number of misses encountered by each per-CPU cache to the total number of misses over all per-CPU caches. We observe that vCPU 0 suffers the highest number of misses, and the miss ratio is substantially lower for higher-indexed vCPU IDs. This clearly demonstrates that the higher-indexed per-CPU caches are infrequently used. As each per-CPU cache is only allowed to cache up to 3 MB of objects, the higher-indexed per-CPU caches use this capacity much more inefficiently than the lower-indexed per-CPU caches. This disparity suggests the need for a heterogeneous cache design.

**Heterogeneous per-CPU cache.** In contrast to statically-sized per-CPU caches that are inelastic to changing application behavior, we propose heterogeneous per-CPU caches

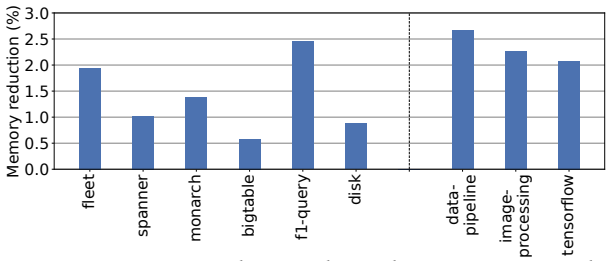



**Figure 10.** Memory reduction due to heterogeneous caches.

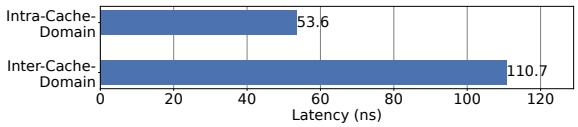



**Figure 11.** Cache to cache data transfer overhead on a platform with heterogeneous cache topology.

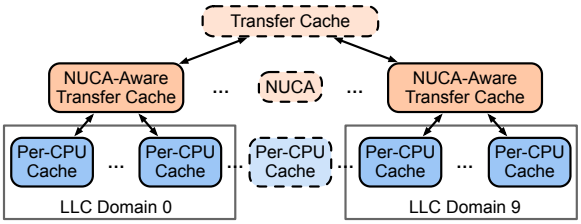



**Figure 12.** Structure of NUCA-aware transfer caches. We maintain a NUCA-aware transfer cache per cache domain.

that can be dynamically sized to balance the misses across all the populated caches. With dynamic resizing, we expect the lower-indexed per-CPU caches to have a larger capacity compared to the higher-indexed per-CPU caches in. Measuring the absolute number of requests served by each per-CPU cache can slow down the fast path. Instead, we record the total number of misses encountered by each per-CPU cache every 5 seconds, and use it as a proxy for cache utilization.

To balance the cache utilization, we employ a background thread that periodically resizes and re-allocates the capacity from lower-utilized to higher-utilized caches. During each resize interval, we identify the top five per-CPU caches with highest misses during the previous 5-second interval as the candidates we may want to grow. We then iterate through the remaining per-CPU caches in a round-robin fashion to identify the candidate per-CPU caches to steal capacity from. For each per-CPU cache we aim to shrink, we prioritize shrinking capacity for larger size classes, since the majority of allocations in our workloads are smaller objects (see Fig. 7).

**Evaluation.** As we described in Section 3, the external fragmentation overhead in TCMalloc accounts for 22.2% of the total application heap size in our fleet. Through our heterogeneous per-CPU cache design, we aim to improve this fragmentation overhead. Because the dynamic scheme improves the utilization of front-end caches, we simultaneously reduce the default size of each per-CPU cache from 3 MB to 1.5 MB. Note that, due to the reduced capacity of the front-end caches, we also observe a reduction in fragmentation in the transfer cache, central free list and pageheap, as TCMalloc ends up caching fewer objects in aggregate. Our fleet experiments reveal that lowering the capacity results in no performance impact for our applications. As shown in Figure 10, we observe a 1.94% reduction in fleet memory usage, and a 0.58% − 2.45% reduction in memory usage of the top 5 applications. For the benchmarks in Section 2.3, the memory usage of the data processing pipeline, image processing server and Tensorflow serving reduces by 2.66%, 2.27%, and 2.08%, respectively. We omit Redis because it is single-threaded, hence it uses a single per-CPU cache.

### 4.2 Transfer Cache

The transfer cache holds an array of pointers to free objects. When the per-CPU cache is depleted or full, it reaches out to the transfer cache to request or return a batch of objects. The transfer cache provides a centralized repository of objects that is shared by all the per-CPU caches. That is, an object de-allocated by one per-CPU cache may be later allocated by another per-CPU cache. As such, the transfer cache allows memory objects to flow rapidly between the per-CPU caches.

**Datacenter heterogeneity.** To achieve the desired performance, a memory allocator needs to adapt to the heterogeneity in hardware platforms. In recent years, Moore's law has slowed down [23, 24] and the cost scaling of newer silicon process nodes continues to diminish. The decline in these technology trends has given rise to datacenter designs with greater heterogeneity [21, 31, 63]. To meet the ever-growing computing demands, CPU vendors have adopted chiplet-based architectures [20, 50, 61] to improve scaling and reduce manufacturing costs. A significant portion of our fleet is composed of platforms with chiplet architectures, which provide multiple last-level cache domains within a socket, leading to Non-Uniform Cache Accesses (NUCA) [37].

**Non-uniform data transfer overhead.** To investigate the performance implications of chiplet architectures, we use Intel MLC [4] to measure the core-to-core access latency on a production platform. Figure 11 shows the access latency for data shared between the cores within the same cache domain and for different cache domains within the same socket. We observe that the inter-cache-domain latency is 2.07× of the intra-cache-domain access latency. WSC applications may span across multiple cache domains, owing to the fact that they are too large to fit within a single cache domain and/or be scheduled as such by the scheduler [1]. The disparity in access latency suggests that the memory allocator should allocate objects that are cache domain local. To this end, we propose NUCA-aware transfer caches that shard a singleton transfer cache into multiple chiplet-local caches.

| Application | Throughput change(%) | Memory change (%) | CPI change (%) | LLC Load Miss (MPKI) | |
|---|---|---|---|---|---|
| | | | | Before | After |
| fleet | 0.32 | 0.10 | -0.57 | 2.52 | 2.41 |
| spanner | 0.28 | 0.08 | -0.42 | 3.80 | 3.21 |
| monarch | 0.62 | 0.32 | -2.89 | 2.64 | 2.37 |
| bigtable | 0.47 | 0.10 | -1.28 | 2.09 | 1.96 |
| f1-query | 1.05 | 0.01 | -3.32 | 2.28 | 2.15 |
| disk | 1.72 | 0.62 | -0.52 | 4.60 | 3.99 |
| redis | / | / | / | / | / |
| data-pipeline | 2.19 | 0.08 | -2.69 | 1.82 | 1.39 |
| image-processing | 1.37 | 0.14 | -8.02 | 0.81 | 0.52 |
| tensorflow | 3.80 | 0.16 | -7.46 | 1.88 | 1.41 |

**Table 1.** Results of fleet-wide experiments and local benchmarks for enabling NUCA-aware transfer caches.

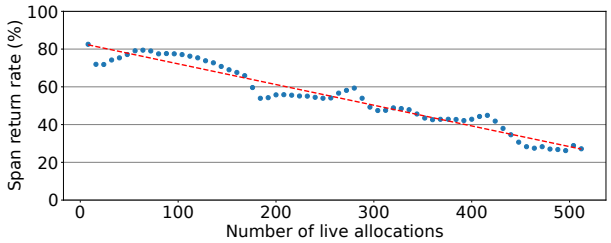



**Figure 13.** Correlation between the number of live allocations and span return rate for size class of 16 bytes.

**NUCA-aware transfer caches.** The legacy transfer cache is a centralized cache. In chiplet architectures, the legacy transfer cache may transfer memory objects between multiple cache domains. That is, objects freed by cores in one cache domain may be allocated by cores in another cache domain. Accessing such objects would require fetching data from non-local LLCs. To minimize inter-cache-domain sharing, we design NUCA-aware transfer caches that track an array of free objects local to each LLC domain. As shown in Figure 12, each NUCA-aware transfer cache only serves allocation and deallocation requests originating from its corresponding cache domain. We periodically release unused free objects in these transfer caches to prevent stranding over time. We also make sure to activate only as many NUCA-aware transfer caches as the application is scheduled on. Note that, we retain a centralized legacy transfer cache that backs NUCA-aware transfer caches, as it still offers cheaper memory allocation than the central free list.

**Evaluation.** Table 1 shows the performance improvement due to NUCA-aware transfer caches. Overall, the NUCA-aware transfer caches improve the cache locality, reducing the LLC load miss rate by 4.37%. In our fleet, we observe over 17.05% of the CPU cycles to be wasted due to back-end stalls [68]. Due to an improvement in the LLC miss rate, we achieve 0.32% improvement in application throughput in the fleet. For the top 5 applications in the production fleet, we observe a throughput improvement of 0.28%–1.72% and a reduction in the cycles per instruction (CPI) of 0.32%–3.32%. Note that, due to an additional caching layer, we also observe an increase in fragmentation by 0.10% of the fleet memory. As we discuss earlier, even with this small increase in fragmentation, we see an outsized improvement in application productivity that results in overall server resource savings. Experiments with benchmarks described in Section 2.3 also show that we can increase throughput by 1.37–3.80% with a 0.08%–0.16% increase in memory usage. We again skip Redis in this study because it is single-threaded and does not benefit from optimizations targeting multi-threaded applications.

### 4.3 Central Free List

The central free list manages memory in spans, which are collections of TCMalloc pages. It fulfills requests from the transfer cache for one or more objects by extracting free objects from spans. When the objects are returned to the central free list, each object is freed to the span it belongs to.

**Diverse object lifetime.** A span may only be released when all of the objects belonging to it are freed. However, object lifetime is extremely diverse. Even for objects of the same size class, any particular allocation might be freed instantly or may live forever (shown in Figure 8). The long-lived allocations prevent the spans from being freed, leading to increased memory fragmentation. We can potentially reduce memory fragmentation by using lifetime annotations (e.g., compiler-guided [52, 60] or application-specified) to allocate short-lived and long-lived objects on different spans. Indeed, prior work [48] uses machine learning to predict object lifetime, which can introduce significant runtime overheads.

**Live allocations and span return rate.** The central free list maintains different sets of spans to allocate objects for each size class. That is, the 8B and 16B objects are allocated from separate spans – an 8KB span may allocate up to 1024 8B objects or 512 16B objects. We use fleet-wide telemetry collected over a two-week period to study the correlation between the probability of returning a span to the page heap, and the number of live allocations on a span. Figure 13 shows the release rates for spans with different numbers of live allocations for the 16B size class, where a span can allocate up to 512 objects. As the number of live allocations increases, the probability of a span release goes down.

For each size class, the central free list organizes spans in a singly linked list. It fulfills incoming allocation requests from a span at the front of that list. As such, the objects may be allocated from spans with the fewest live allocations that are most likely to be released, just because they happen to lie in the front of the linked list. We utilize the observation from Figure 13 to propose a prioritization scheme that allocates objects from spans that are least likely to be released.

**Span prioritization.** We aim to minimize memory fragmentation in the central free list by fulfilling incoming allocations from spans that have the least likelihood of being freed, while deprioritizing spans that are expected to be freed in the near future. We restructure the central free list to manage spans in $L$ linked lists (instead of a singleton list) to track spans with varying occupancy separately. Spans with fewer

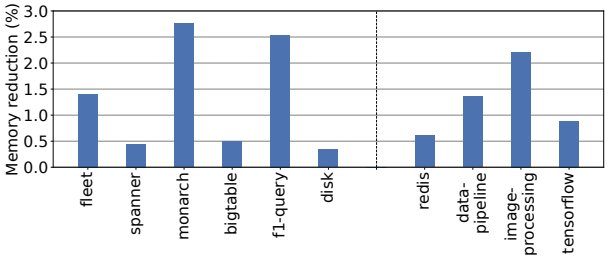



**Figure 14.** Memory reduction with span prioritization.

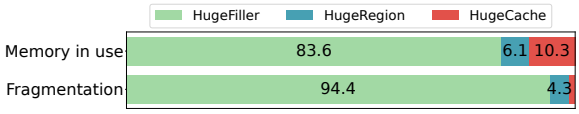



**Figure 15.** In-use memory and fragmentation (%) in the page heap. HugeFiller is the major contributor to fragmentation.

live allocations on them are mapped to higher-indexed lists. Specifically, we map a span with $A$ live allocations into a list indexed $max(0, L - \log_2(A))$. This allows us to differentiate spans with fewer allocations at a finer granularity – spans with 132 or 255 live allocations are unlikely to be released and can be mapped in the same list. Our experiments show that $L = 8$ lists are sufficient to differentiate spans.

The central free list allocates objects from spans in the list with the lowest possible index, since it contains spans with a higher number of allocated objects. We also move spans between the lists as required on each allocation and deallocation, as the number of live allocations on them change.

**Evaluation.** With span prioritization, the central free list can densely pack allocations on fewer spans. Figure 14 shows the resulting reduction in memory fragmentation. In a fleet-wide experiment, we achieve a 1.41% reduction in fleet memory usage. At our scale, this reduction leads to significant cost savings in server resources. The memory fragmentation in `monitor` reduces by 2.76%, and by 0.34%–2.54% for other fleet applications. We also confirm that the application's productivity metrics remain unchanged. For the benchmarks, we observe a memory usage reduction of 0.61%–1.36%.

### 4.4 Pageheap

TCMalloc's hugepage-aware page heap [33] manages memory in hugepage-sized chunks to take advantage of Transparent Huge Pages (THP) [12], which provides an opportunity for the kernel to cover consecutive pages using hugepages in the page table. An entire aligned hugepage (typically 2MB on x86) occupies just one TLB entry, which reduces stalls by increasing the TLB coverage and reducing TLB misses [40]. The page heap plays a critical role in efficiently managing the memory layout to maximize TLB efficiency.

The page heap [33] consists of three major components: (1) the *hugepage filler* handles allocation requests smaller than

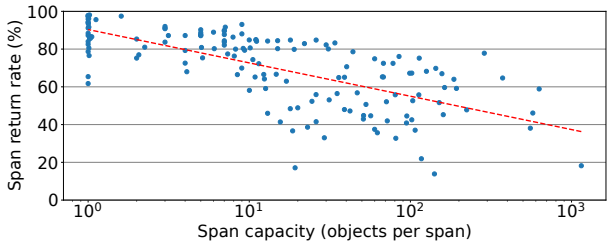



**Figure 16.** Correlation between the span capacity and span return rate for different size classes.

a hugepage. Here, spans are packed into hugepages; (2) the *hugepage region* is used for allocations that slightly exceed the size of a hugepage (e.g., 2.1MB). It packs such allocations on to a contiguous run of hugepages; (3) the *hugepage cache* also handles large allocations of at least a hugepage. Such allocations can generate slack (e.g., 1.5 MB slack from a 4.5 MB allocation), which is then donated to the hugepage filler.

**Page heap fragmentation.** While the page heap cannot control the amount of memory that the application uses, it plays a critical role in placing those allocations in hugepage-aligned memory regions to improve the TLB efficiency. The page heap is a major contributor to fragmentation, accounting for 51% of the total external fragmentation (Section 3). As shown in Figure 15, hugepage filler manages 83.6% of the total in-use memory and accounts for 94.4% of the page heap fragmentation. Given this, we focus on the hugepage filler.

**Hugepage filler.** The hugepage filler allocates spans from hugepages-aligned memory regions. It frees up a hugepage when all the spans previously allocated from it are returned by the central free list. Similar to the span prioritization mechanism that we discussed in Section 4.3, the hugepage filler prioritizes span allocations from hugepages that already have a higher number of allocations, and thus are least likely to be released. It assumes that spans themselves are independently and equally likely to become free. Instead, we analyze heuristics that can be used to identify spans that are more likely to be freed. We can then assign span allocations to different sets of hugepages based on their lifetime – we can place short-lived spans densely on fewer hugepages, thus improving TLB efficiency and fragmentation.

**Span lifetime.** We notice that spans of different size classes have diverse lifetimes. As we described in Section 2.1, TCMalloc uses a span to exclusively allocate objects of a particular size class; *span capacity* denotes the total objects for a size class that may be allocated from that span. For instance, an 8 KB span has a capacity of 1024 8B objects.

We perform a correlation study of span lifetime versus span capacity. Figure 16 shows the span capacity and its rate of returning from the central freelist to the hugepage filler for different size classes. We see a strong negative correlation (with a Spearman's correlation coefficient of −0.75) between the capacity of a span and its return rate. In Figure 16, the

| Application | Throughput change(%) | Memory change (%) | CPI change (%) | dTLB Load Walk (%) | |
|---|---|---|---|---|---|
| | | | | Before | After |
| fleet | 1.02 | -0.82 | -6.75 | 9.16 | 6.22 |
| spanner | 0.38 | -0.45 | -0.99 | 7.92 | 7.60 |
| monarch | 3.30 | -0.05 | -10.10 | 20.34 | 15.55 |
| bigtable | 2.83 | -0.13 | -4.44 | 17.25 | 15.00 |
| f1-query | 1.40 | -1.40 | -4.56 | 9.62 | 9.07 |
| disk | 6.29 | -0.38 | -17.61 | 8.42 | 6.55 |
| redis | 1.05 | -7.02 | -9.04 | 10.34 | 10.25 |
| data-pipeline | 1.43 | -1.50 | -2.76 | 5.36 | 4.97 |
| image-processing | 2.15 | -1.29 | -7.59 | 1.46 | 0.96 |
| tensorflow | 3.91 | -2.69 | -2.72 | 6.79 | 5.91 |

**Table 2.** Fleet workloads and benchmarks using the lifetime-aware hugepage filler. dTLB load walk (%) is the fraction of cycles spent in page walk, without accessing the L2 TLB.

leftmost data points show the spans allocating large size classes that can only hold one object. When the only object is returned, the span is released, resulting in a high span return rate. In contrast, spans with a significantly larger capacity are long lived and have a much lower return rate.

We use span capacity as a proxy for its lifetime to distinguish between short-lived and long-lived spans. Since a span is only released when all objects from it are freed, the object lifetime (Figure 8) is not necessarily a good measure for the span lifetime. It also requires lifetime annotations from compiler or applications that incur significant runtime overhead [48] . In contrast, span capacity is statically determined and can be used without any runtime overhead.

**Lifetime-aware hugepage filler.** We propose to make the hugepage filler aware of span lifetime to maximize the probability that a hugepage becomes totally free. That is, we aim to allocate short-lived and long-lived spans from separate sets of hugepages. To that end, we use dedicated hugepages for allocating spans with capacity $> C$ and $\leq C$. Our experiments reveal $C = 16$ as an acceptable threshold for separating span allocations. We replicate linked-list structures to track these dedicated hugepages separately. For incoming requests in each lifetime category, we prioritize allocating from hugepages that have the most allocations.

By differentiating between long-lived and short-lived spans, the lifetime-aware hugepage filler is able to densely place short-lived allocations on dedicated hugepages and release memory to the OS in complete hugepages. This improves hugepage coverage, reduces TLB misses, and boosts application performance. We also observe that smaller objects have higher access density. By separating spans with smaller size classes, we also efficiently utilize the limited TLB resources.

**Evaluation.** Table 2 shows results of enabling the lifetime-aware hugepage filler. The baseline implements the state-of-the-art hugepage-aware page heap as proposed by Hunter et al. [33]. For the fleet experiment, we achieve 1.02% improvement in throughput and 0.82% reduction in memory usage. For the top 5 applications, we improve the throughput by 0.38%–6.29% and reduce the CPI by 0.99%–17.61%. Figure 17a shows the hugepage coverage for applications. We can see that the lifetime-aware hugepage filler improves

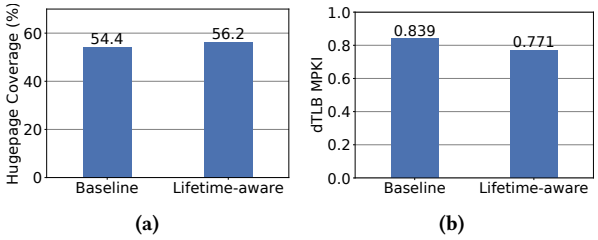



**Figure 17.** (a) Improved hugepage coverage rate and (b) reduced dTLB miss rate with lifetime-aware hugepage filler.

hugepage coverage, increasing the average percentage of heap memory backed by hugepages from 54.4% to 56.2%. Due to improved TLB efficiency, we observe a 2.94% reduction in dTLB load walk cycles and an 8.1% reduction in dTLB misses (Figure 17b) in our fleet. This again supports our argument that optimizing for application productivity provides more efficiency gains than optimizing for the `malloc` CPU overhead alone, since the memory allocator has a substantial leverage on improving data locality and maximizing TLB efficiency. Benchmark experiments on a dedicated server also show that we can achieve a 1.05–3.91% increase in throughput with a 1.29%–7.02% reduction in memory usage.

### 4.5 Putting It All Together

Our characterization study shows that a WSC memory allocator must accommodate the dynamic resource needs of WSC applications, be aware of heterogeneity in hardware platforms, and leverage the diverse lifetime characteristics to improve data locality. Based on these insights, we redesign each cache tier in TCMalloc to propose a heterogeneous per-CPU cache, a NUCA-aware transfer cache, a central freelist with span prioritization, and a lifetime-aware hugepage filler.

Redesigning the memory allocator for warehouse-scale environments helps us achieve our ultimate goal: maximize the productivity of WSC applications by using fewer server resources to complete the same or more units of work, even if it results in a lower reduction in the overall "datacenter tax". The designs in this work have been gradually rolled out to our fleet over a two-year period. Given the ever-changing nature of WSC workloads, it is non-trivial to perform a strict end-to-end evaluation of these designs. Regardless, we can estimate the aggregate performance impact of the four designs based on their relative improvement. We achieve a 1.4% increase in fleet throughput and a 3.5% reduction in fleet memory usage. For the top 5 applications, we achieve 0.7%–8.1% throughput and 1.0%–6.3% memory improvement. The source code for all the designs is open sourced and publicly available.

## 5 Discussion

In this work, we discussed certain design choices in TCMalloc as case studies following our characterization insights. Next, we discuss potential opportunities as future work.

**Room at the top** [43]. With Moore's law slowing down, performance gains need to come from improvements at the top of the computing stack [44]. In this work, we focus on the memory allocator specifically to show how leveraging fleet-wide profiling and improving common libraries [34, 62] can uncover horizontal efficiency opportunities. With diminishing returns due to technology scaling, optimizing common libraries can yield dramatic performance improvements.

**Datacenter tax and productivity metrics.** While understanding the datacenter tax can help us identify the largest building blocks of WSC applications, reducing the tax itself may yield limited efficiency gains. The efficiency improvements to the `malloc` CPU overhead may yield up to a 4.3% improvement in fleet CPU, but a larger opportunity lies in optimizing for application productivity – 20-64% of CPU cycles incur memory stalls [34, 63] and optimizing for data locality can have a larger performance upside. While we present four case studies, several opportunities remain that may improve cache efficiency and/or hugepage coverage through improved allocation placements.

**NUMA architecture and beyond.** TCMalloc has a built-in support [3] for Non-Uniform Memory Access (NUMA) architectures. In NUMA mode, it duplicates the set of size classes and page allocator for each NUMA node, ensuring that allocations always return local memory. In this work, we demonstrate the need for the memory allocator to adapt to the heterogeneity of hardware platforms. We propose NUCA-aware transfer caches that preserve cache locality in platforms with non-uniform cache domains.

**Cooperation with kernel features.** Although TCMalloc is a userspace memory allocator, it needs to cooperate with the kernel to achieve the desired performance. It uses restartable sequences [7] to optimize the fast path of the per-CPU cache, avoiding the use of locks or expensive atomic instructions when accessing per-CPU data. Virtual CPU [18] support exposes a dense set of CPU IDs to index the per-CPU cache, which helps TCMalloc reduce the memory footprint of the front-end cache. TCMalloc also makes use of transparent hugepages [12] in the back-end pageheap [33] to improve hugepage coverage and reduce dTLB misses. These optimizations illustrate the need to leverage kernel features to improve the performance of a memory allocator.

**Object lifetime and access density.** In this work, we propose a lifetime-aware allocator that uses span lifetime to improve allocation placements in hugepages. We can also use user-defined or profile-guided [52, 60] lifetime and access density annotations that can further improve TLB efficiency.

## 6 Related Work

We now discuss work related to memory allocations.

**Profiling datacenter workload.** To better understand and optimize the performance of datacenter workloads, previous studies [22, 29, 34, 38, 63] have taken the approach of profiling production workloads deployed in the datacenter. Kozyrakis et al. [38] examined Microsoft's applications, focusing on system-level metrics for balanced server design. Kanev et al. [34] profiled thousands of Google services in the datacenter fleet, identified common building blocks in datacenter computation, and proposed architectural optimizations accordingly. Sriraman et al. [63] characterized diverse microservices in Meta to show system-level and architectural acceleration opportunities. There are also efforts in academia to develop and characterize open-source benchmark suites for cloud services [27, 28, 36, 64, 69]. While these studies investigate the behavior of memory allocators in datacenter workloads, they only address CPU cycles consumed by the allocator. Our work complements these studies by focusing on the diverse allocation behavior in production workloads.

**Modern memory allocator.** To improve memory allocation performance, custom modern memory allocators [10, 14, 15, 25, 30, 39, 42, 45, 46, 55, 67] are used to replace the default `malloc` implementation (e.g., glibc[11]). Large-scale services use these allocators at scale. `jemalloc` [25] (used by Meta [26]) emphasizes fragmentation avoidance and scalable concurrency support. `mimalloc` [42] (used by Microsoft [5]) improves locality by providing users with objects from the same page. Hoard [15] focuses on reducing thread contention and false sharing. Mesh [55] uses remapping of virtual pages and randomized allocation to enable memory compaction. `snmalloc` [46] uses batched message-passing instead of per-thread caching to send batches of deallocations to the originating thread. Our in-depth characterization and optimization studies may benefit these memory allocators.

**Hugepage support.** There has been extensive work on optimizing memory management in the kernel to improve physical memory contiguity and provide better hugepage support [40, 53, 54, 56, 66, 70]. Ingens [40] manages contiguity as a first-class resource and tracks utilization and access frequency of memory pages. Contiguitas [70] proposes to separate movable allocations from unmovable ones by placing them into different memory regions. User-space memory allocators can also benefit from kernel optimizations because they rely on the system to provide contiguous memory.

## 7 Conclusion

We present the first comprehensive characterization study of TCMalloc, a memory allocator used in WSCs. We show that the memory allocator needs to adapt to the dynamic resource usage of WSC applications, be aware of heterogeneity in hardware platforms, and utilize the objects' diverse lifetime to make memory packing decisions. Based on these insights, we redesign each tier in the TCMalloc cache hierarchy for WSC environments, We evaluate these design choices using fleet-wide A/B experiments in our production fleet, resulting in a 1.4% improvement in throughput and a 3.4% reduction in RAM usage for the entire fleet.

## Acknowledgements

We sincerely thank our shepherd, Orran Krieger, and the anonymous reviewers for their time and feedback that helped us improve this paper. We also thank Darryl Gove, Evan Brown, Dmitry Vyukov, Xiaoyu Chen, Shiyu Hu, Todd Lipcon, Jeff Cox, Seong Beom Kim, Richard O'Grady, Nikita Lazarev, Varun Gohil and Mingyu Liang for their feedback. We would also like to thank our colleagues at Google, specifically the Google-wide profiling team, that built the profiling infrastructure that we used extensively in this work. This project was supported in part by NSF CAREER Award CCF-1846046, two Google Faculty Research Awards, a Sloan Foundation Research Scholarship, and an Intel Research Award.

## References

[1] Completely Fair Scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html.

[2] Eigen Linear Algebra Library. https://eigen.tuxfamily.org.

[3] Implement NUMA Awareness in TCMalloc. https://github.com/google/tcmalloc/commit/ef7a3f8d794c42705bf4327ca79fa17186904801.

[4] Intel Memory Latency Checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[5] mi-malloc. https://microsoft.github.io/mimalloc/.

[6] Redis. https://redis.io.

[7] Restartable Sequences. https://github.com/torvalds/linux/commit/d82991a8688ad128b46db1b42d5d84396487a508.

[8] Restartable Sequences. https://dynamorio.org/page_rseq.html.

[9] Strace: Linux Syscall Tracer. https://strace.io.

[10] TCMalloc. https://github.com/google/tcmalloc.

[11] The GNU C Library. https://www.gnu.org/software/libc.

[12] Transparent Hugepage Support. https://www.kernel.org/doc/html/next/admin-guide/mm/transhuge.html.

[13] Colin Adams, Luis Alonso, Ben Atkin, John P. Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George T. Talbot, Adam Jacob Tart, and Nick Taylor, editors. *Monarch: Google's Planet-Scale In-Memory Time Series Database*, 2020.

[14] Yehuda Afek, Dave Dice, and Adam Morrison. Cache index-aware memory allocation. *ACM SIGPLAN Notices*, 46(11):55–64, 2011.

[15] 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.

[16] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), jun 2008.

[17] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[18] Jonathan Corbet. Extending restartable sequences with virtual CPU IDs. https://lwn.net/Articles/885818, 2022.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's globally-distributed database. In *OSDI*, 2012.

[20] Jeff Defilippi. Why Chiplets and why now? https://community.arm.com/arm-community-blogs/b/infrastructure-solutions-blog/posts/why-chiplets-why-now.

[21] Christina Delimitrou and Christos Kozyrakis. Qos-aware scheduling in heterogeneous datacenters with paragon. *ACM Transactions on Computer Systems (TOCS)*, 31(4):1–34, 2013.

[22] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[23] Lieven Eeckhout. Is moore's law slowing down? what's next? *IEEE Micro*, 37(04):4–5, 2017.

[24] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

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

[26] Jason Evans. Scalable memory allocation using jemalloc. https://engineering.fb.com/2011/01/03/core-infra/scalable-memory-allocation-using-jemalloc/, 2011.

[27] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.

[28] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[29] Abraham Gonzalez, Aasheesh Kolli, Samira Khan, Sihang Liu, Vidushi Dadu, Sagar Karandikar, Jichuan Chang, Krste Asanovic, and Parthasarathy Ranganathan. Profiling hyperscale big data processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16, 2023.

[30] Yacine Hadjadj, Chakib Mustapha Anouar Zouaoui, Nasreddine Taleb, Sarah Mazari, Mohamed El Bahri, and Miloud Chikr El Mezouar. Vcmalloc: A virtually contiguous memory allocator. *IEEE Transactions on Computers*, 2023.

[31] Md E Haque, Yuxiong He, Sameh Elnikety, Thu D Nguyen, Ricardo Bianchini, and Kathryn S McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 625–638, 2017.

[32] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[33] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 257–273. USENIX Association, July 2021.

[34] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 158–169, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. Mallacc: Accelerating memory allocation. *SIGPLAN Not.*, 52(4):33–45, apr 2017.

[36] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[37] Changkyu Kim, Doug Burger, and Stephen W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. *SIGARCH Comput. Archit. News*, 30(5):211–222, oct 2002.

[38] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE micro*, 30(4):8–19, 2010.

[39] 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.

[40] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. Coordinated and efficient huge page management with ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 705–721, Savannah, GA, November 2016. USENIX Association.

[41] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1), mar 2012.

[42] 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.

[43] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[44] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[45] Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J Yadwadkar, and Lizy K John. Nextgen-malloc: Giving memory allocator its own room in the house. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 135–142, 2023.

[46] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J Parkinson, Alex Shamis, Christoph M Wintersteiger, and David Chisnall. Snmalloc: a message passing allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, pages 122–135, 2019.

[47] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[48] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 541–556, New York, NY, USA, 2020. Association for Computing Machinery.

[49] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S McKinley, and Paul Turner. 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*, pages 28–38, 2021.

[50] Samuel Naffziger, Noah Beck, Thomas Burd, Kevin Lepak, Gabriel H. Loh, Mahesh Subramony, and Sean White. Pioneering chiplet technology and design for the amd epyc™ and ryzen™ processor families. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 57–70. IEEE Press, 2021.

[51] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving, 2017.

[52] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 2–14. IEEE Press, 2019.

[53] Ashish Panwar, Sorav Bansal, and K Gopinath. 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*, pages 347–360, 2019.

[54] Ashish Panwar, Aravinda Prasad, and K Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 679–692, 2018.

[55] Bobby Powers, David Tench, Emery D Berger, and Andrew McGregor. Mesh: Compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 333–346, 2019.

[56] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. Trident: Harnessing architectural resources for all page sizes in x86 processors. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1106–1120, 2021.

[57] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79, 2010.

[58] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. F1 query: declarative querying at scale. *Proc. VLDB Endow.*, 11(12):1835–1848, aug 2018.

[59] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 23)*, pages 498–514, 2023.

[60] Han Shen, Krzysztof Pszeniczny, Rahman Lavaee, Snehasish Kumar, Sriraman Tallam, and Xinliang David Li. Propeller: A profile guided, relinking optimizer for warehouse-scale applications. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 617–631, New York, NY, USA, 2023. Association for Computing Machinery.

[61] Teja Singh, Sundar Rangarajan, Deepesh John, Russell Schreiber, Spence Oliver, Rajit Seahra, and Alex Schaefer. 2.1 zen 2: The amd 7nm energy-efficient high-performance x86-64 microprocessor core. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 42–44. IEEE, 2020.

[62] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[63] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. Softsku: Optimizing server architectures for microservice diversity @scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 513–526, New York, NY, USA, 2019. Association for Computing Machinery.

[64] Akshitha Sriraman and Thomas F Wenisch. $\mu$ suite: a benchmark suite for microservices. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–12. IEEE, 2018.

[65] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and*

*pattern recognition*, pages 2818–2826, 2016.

[66] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Translation ranger: Operating system support for contiguity-aware tlbs. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 698–710, 2019.

[67] Hanmei Yang, Xin Zhao, Jin Zhou, Wei Wang, Sandip Kundu, Bo Wu, Hui Guan, and Tongping Liu. Numalloc: A faster numa memory allocator. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management*, pages 97–110, 2023.

[68] Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[69] Ahmad Yasin, Yosi Ben-Asher, and Avi Mendelson. Deep-dive analysis of the data analytics workload in cloudsuite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 202–211. IEEE, 2014.

[70] Kaiyang Zhao, Kaiwen Xue, Ziqi Wang, Dan Schatzberg, Leon Yang, Antonis Manousis, Johannes Weiner, Rik Van Riel, Bikash Sharma, Chunqiang Tang, et al. Contiguitas: The pursuit of physical memory contiguity in datacenters. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.