Free list sharding for Improved jemalloc Performance

Mohammad Khakhariawala
Stony Brook University

Abstract

Current memory allocators like jemalloc excel in scalability, but struggle with workloads dominated by small, short-lived allocations. Mimalloc tackles this with free list sharding, using multiple free lists and techniques like deffered free. This project explores incorporating such concepts into jemalloc, aiming to create a hybrid allocator with the strengths of both. We'll use a comprehensive benchmark suite including cfrac, espresso, and real-world workloads like Lean compilation to assess performance improvements. This research has the potential to significantly improve memory allocation performance for various applications.

1 About jemalloc

Jemalloc, a high-performance memory allocator, excels in managing memory requests efficiently. It achieves this through several key functionalities.

FreeBSD allows dynamic configuration of jemalloc, enabling performance tweaks and debugging. A core concept is arenas, fixed-size memory pools. The number of arenas scales with processors to minimize allocation contention. Thread assignment to arenas is round-robin, ensuring fair workload distribution.

Memory from the kernel (via sbrk or mmap) is managed in chunks, multiples of a pre-defined size, simplifying allocation calculations within a chunk. Chunks are typically 2MB and associated with specific arenas.

jemalloc categorizes allocations into three main classes: small, large, and huge. Small allocations are further divided based on alignment and size. Large allocations fall between half a page and half a chunk. Huge allocations, the largest, are directly backed by dedicated chunks.

Allocation strategies differ based on size. Huge allocations are managed by a single red-black tree for efficiency. Small and large allocations use the binary buddy system to carve chunks into page runs. Runs can be split or merged for different allocation sizes. Information about run states is stored in a page map at the beginning of each chunk.

Small allocations are segregated, with each run managing a single size class. A region bitmap within each run efficiently identifies free regions. This minimizes internal fragmentation (unused space within allocations) but can lead to external fragmentation (unusable memory gaps). Multi-page runs are used for larger size classes to limit external fragmentation.

Each size class has a "current" run for allocations. Runs are categorized by fullness (used regions). Runs in the lowest fullness category are never destroyed, while those in higher categories can be for efficiency. Fullness categories also guide selection of new current runs, prioritizing partially full runs.

In essence, jemalloc employs a sophisticated strategy for memory management. It balances efficiency, scalability, and fragmentation control through careful memory allocation in arenas and chunks, categorized allocation sizes, and run management techniques.

2 Proposed changes to jemalloc's design

While jemalloc excels in scalability and fragmentation reduction, its performance can be hampered by workloads dominated by small, short-lived allocations. This project proposes incorporating a concepts like free list sharding inspired by mimalloc.

Thread-local free lists provide a readily available pool of memory for each thread. Threads can quickly service allocation and deallocation requests from their local pool, minimizing the overhead associated with requesting memory from the central allocator. This approach is particularly beneficial for workloads with frequent memory churn, as threads can efficiently manage their own pool of recently freed memory.

We anticipate this modification to significantly improve performance for applications heavily reliant on small allocations. Benchmarks encompassing diverse allocation patterns, including cfrac and real-world scenarios like Lean compilation, will be used to quantify the impact. Additionally, stress tests will assess performance under extreme allocation scenarios.

By combining the strengths of jemalloc's core design with the efficiency of free list sharding, we aim to create a hy-

brid allocator offering superior performance across a wider range of applications. This research has the potential to significantly improve memory allocation performance, particularly for workloads with frequent small object creation and destruction.

3 Evaluation methodologies

To assess the effectiveness of incorporating free list sharding into jemalloc, we'll leverage a comprehensive benchmark suite encompassing real-world programs and synthetic stress tests. This suite is provided by the mimalloc team.

cfrac This benchmark, simulating continued fraction factorization, heavily relies on short-lived small allocations, a scenario well-suited for thread-local free lists.

espresso This benchmark, analyzing programmable logic arrays, emphasizes cache-aware memory allocation, allowing us to evaluate the interaction of thread-local free lists with cache behavior.

barnes This benchmark simulates a hierarchical n-body solver, representing a workload with moderate allocation demands. It serves as a point of comparison for workloads less impacted by small allocations.

leanN This benchmark measures the performance of the Lean compiler compiling its standard library concurrently. This real-world scenario involves intensive allocation, allowing us to gauge the impact on a complex workflow.

redis We'll analyze the requests handled per second by the redis server under high load. This benchmark assesses the impact on a server application heavily reliant on memory management.

larsonN This benchmark simulates a server workload with complex memory access patterns, including "bleeding" where objects are freed by other threads. It allows us to evaluate how thread-local free lists handle inter-thread memory management.

alloc-testN This test simulates intensive allocation workloads with a Pareto size distribution, pushing allocators to their limits under high memory churn.

sh6bench and sh8benchN These stress tests evaluate how the allocator handles scenarios where objects are freed in both LIFO (last-in-first-out) and reverse order, with the multi-threaded version (sh8benchN) introducing inter-thread deallocation.

xmalloc-testN This test simulates an extreme producer-consumer pattern with threads purely allocating or deallocating memory of various sizes. This asymmetric workload challenges thread-local cache management within the allocator.

cache-scratch This test assesses potential passive false sharing of cache lines. It evaluates how the allocator's memory management interacts with cache behavior.

By analyzing the results from this comprehensive benchmark suite, we can quantify the performance improvements achieved by incorporating free list sharding into jemalloc. This data will shed light on the effectiveness of this modification across various allocation patterns and workload intensities, building upon the insights gained from mimalloc's evaluation.

4 Related Work

jemalloc The focus of our proposed modification, jemalloc is a widely-used, general-purpose memory allocator known for its efficiency, scalability, and fragmentation control. It employs a variety of techniques like arenas, chunks, and size classes to cater to diverse allocation patterns. Our work aims to enhance jemalloc's performance for workloads dominated by small allocations by incorporating thread-local free lists.

mimalloc The inspiration for our proposed modification, mimalloc is a high-performance memory allocator specifically designed for efficiency in handling small object allocations. It leverages thread-local free lists to minimize overhead and improve performance for workloads with frequent short-lived object creation and destruction.

tcmalloc Developed by Google, tcmalloc is another popular memory allocator known for its scalability and performance. It utilizes thread-caching techniques similar to jemalloc, but with a focus on centralizing cache management.

Hoard Designed for multithreaded applications on multiprocessor systems, Hoard is another high-performance memory allocator known for its efficient handling of concurrent memory requests.