The Quest for Memory Allocation Efficiency: A Benchmarking Analysis of Popular Allocators

Mohammad Khakhariawala
Stony Brook University

Abstract

Efficient memory allocation is crucial for modern systems. This project delves into six popular memory allocators: jemalloc, tcmalloc, mimalloc, snmalloc, hoard, and lockfree-malloc. I compare their performance across diverse workloads (cfrac, espresso, barnes, etc.) using metrics like execution time and memory usage. My goal is to identify the inherent trade-offs associated with each allocator and guide developers towards the optimal choice for their specific needs. This project expands on existing research by encompassing a wider range of allocators and benchmarks, offering valuable insights for developers seeking the best memory allocation solution.

1 Introduction

Modern applications heavily rely on dynamic memory allocation, making efficient memory management crucial for optimal performance and stability. At the heart of this management lie memory allocators – software components responsible for allocating and deallocating memory blocks as programs require. A diverse landscape of memory allocators exists, each with unique strengths and trade-offs.

As the complexity of software systems grows, the demand for memory allocation efficiency intensifies. Researchers and developers are constantly innovating, designing new algorithms and data structures to enhance allocator performance. Evaluating the effectiveness of these advancements necessitates rigorous benchmarking and performance analysis. However, benchmarking memory allocators presents a challenge, as results can be significantly impacted by the specific application and underlying hardware architecture.

This project embarks on a journey to explore the landscape of memory allocation efficiency. It begins by establishing its motivation and introducing the research question surrounding memory allocation efficiency. Following this I delve into six popular allocators: jemalloc, tcmalloc, mimalloc, snmalloc, hoard, and lockfree-malloc. My investigation begins by providing an overview of these allocators and the motivations behind their selection. Next, I explore the chosen benchmarks, which encompass two categories: real-world applications and a set of stress tests, to assess their performance. Finally, I present the results of these benchmarks, drawing conclusions about the strengths and limitations of each allocator. Through this analysis, I aim to offer valuable insights for developers seeking the optimal memory allocation solution for their specific needs. All benchmarking results are included here https://github.com/Mohamk1234/os_project24. Project recording is here https://drive.google.com/drive/folders/15nInYmG6iJgg5Ctr8G78LqOB02G9uk_B?usp=sharing

2 Motivation

Modern software applications heavily utilize dynamic memory allocation. This allows for flexible memory usage during runtime, adapting to the program's needs. However, efficient management of this dynamic memory pool is critical for optimal performance and stability.

Key Challenges and Opportunities:

Performance Bottleneck Memory allocation overhead can become a bottleneck in applications with frequent memory requests. Inefficient allocators can lead to increased execution time and resource consumption.

Fragmentation Over time, repeated allocation and deallocation can lead to memory fragmentation, reducing the availability of contiguous memory blocks for larger allocations. This can significantly impact performance as the allocator searches for suitable free spaces.

Scalability Concerns As software systems grow in complexity and core count, memory allocation strategies need to adapt. Some allocators might not scale well with increased parallelism, leading to performance degradation in multi-threaded environments.

Justification for Project Focus:

This project focuses on exploring memory allocation efficiency due to the significant impact it has on application performance, resource usage, and overall system stability. By evaluating the performance characteristics of various memory allocators through a comprehensive benchmarking process, I aim to:

Identify the Trade-Offs Different allocators prioritize distinct aspects like speed, memory footprint, or thread safety. Our analysis will highlight the strengths and weaknesses of each allocator, enabling developers to make informed decisions based on their specific needs.

Guide Developers Understanding the performance characteristics of popular allocators empowers developers to choose the most suitable option for their applications, leading to optimized code and better resource utilization.

3 Memory Allocators

The selection of these six allocators was deliberate. They represent a diverse range of design philosophies and optimization goals. This diverse set allows us to explore the effectiveness of different algorithms and data structures in memory allocation. By analyzing their performance across various benchmarks, we can understand which approaches excel in specific scenarios and identify potential trade-offs between speed, scalability, fragmentation control, and thread safety.

3.1 Jemalloc

Jemalloc [6] prioritizes a trifecta of efficiency, scalability, and ease of use. It manages memory in chunks of variable sizes, catering to diverse application needs. Free memory is organized efficiently using a combination of bins and runs. Bins hold small chunks of similar sizes, while runs store larger contiguous memory regions subdivided into chunks. This tiered structure allows jemalloc to quickly locate appropriate free memory based on the allocation size. For small objects, jemalloc employs slab allocation, pre-allocating a fixed number of identically sized objects to minimize fragmentation caused by frequent allocations and deallocations. Additionally, jemalloc utilizes arenas, which are dedicated memory regions within the overall heap. This allows for better memory isolation and control in multi-threaded environments. Overall, jemalloc's design and implementation make it a highly effective choice for high-performance systems.

3.2 Tcmalloc

Tcmalloc [7], developed by Google, focuses on achieving peak performance and scalability, particularly in multi-threaded environments. It prioritizes low latency and high throughput for memory operations. Tcmalloc employs thread-caches for frequently accessed memory and central page heaps to manage larger allocations. Threads have private caches for recently used memory blocks, reducing the need to access the central heap and improving performance. This design offers excellent thread safety features, making it well-suited for latency-sensitive workloads and applications with a high degree of parallelism.

3.3 Lockfree-malloc

Lockfree-malloc [5] prioritizes thread safety and real-time performance. It utilizes lock-free data structures to ensure safe concurrent access. Unlike traditional locking approaches that can introduce bottlenecks, lock-free data structures rely on atomic operations and retries to avoid contention. While guaranteeing thread safety, this approach can potentially incur higher overhead compared to other allocators. This tradeoff between speed and safety is precisely what makes lockfree-malloc an interesting subject for study. It allows us to evaluate the performance cost associated with lock-free techniques and understand their suitability for real-time systems and multi-threaded environments where thread safety is critical but real-time constraints exist.

3.4 Mimalloc

Mimalloc [4], developed by Microsoft, prioritizes fast allocation and deallocation operations while also offering robust thread safety. It leverages a technique called free list sharding, which improves both performance and thread locality. Memory is divided into multiple, thread-private free lists, reducing contention for locks and improving scalability. Additionally, each free list is further subdivided into sub-lists based on object size, allowing threads to quickly find free memory blocks of the desired size within their local cache. Mimalloc is also designed to work well with reference counting languages, making it a strong contender for applications that utilize this memory management approach.

3.5 Snmalloc

Snmalloc [12] takes a different approach, optimized for concurrent message passing systems. Its primary focus is minimizing the overhead associated with memory allocation, particularly for applications with frequent memory operations. Snmalloc utilizes a central pool with lock-free mechanisms to facilitate concurrent access from multiple threads. Unlike traditional locking approaches that can introduce bottlenecks, lock-free mechanisms rely on atomic operations and retries to ensure thread safety without significant performance overhead. This design makes it ideal for message-passing systems where low allocation overhead is paramount.

3.6 Hoard

Developed by Emery Berger, Hoard [3] is a memory allocator built for speed in multithreaded applications. It achieves this by dividing the memory pool into chunks and keeping a personal stash of these chunks for each thread. Threads prioritize using their own stash when requesting memory, minimizing competition for the central memory pool. To further reduce competition, Hoard allocates large blocks (superblocks) containing multiple chunks at once. Additionally, Hoard monitors memory usage patterns to optimize allocation based on frequently requested object sizes. This allows Hoard to create dedicated caches for commonly used sizes, streamlining the memory allocation process. In summary, Hoard prioritizes reducing competition and optimizing memory usage to deliver top performance for multithreaded programs.

4 Benchmarks

Real world program benchmarks:

cfrac This benchmark implements continued fraction factorization, a process that involves iteratively dividing numbers. It heavily relies on frequent allocations and deallocations of small memory blocks, similar to the memory access patterns anticipated for software like Koka and Lean.

espresso [8] This benchmark, focusing on programmable logic arrays (PLAs), provides insights into how memory allocators handle cache locality. PLAs have specific memory access patterns that can be influenced by caching strategies.

larsonN [11] The larsonN benchmark simulates a multithreaded server workload using 100 separate threads. Threads mimic server tasks by allocating and deallocating memory objects frequently. This benchmark incorporates 'bleeding' behavior, where some objects are deliberately left for other threads to free, reflecting real-world server environments where memory management can involve multiple processes accessing and releasing memory concurrently.

barnes [10] The barnes benchmark utilizes a hierarchical n-body particle solver to simulate gravitational forces acting on a significant number of particles. This solver is optimized for this specific problem. In contrast to benchmarks like cfrac and espresso, barnes exhibits a lower memory allocation footprint.

Stress test benchmarks:

malloc-large [2] This test focuses on exercising memory allocators with large allocations (several megabytes).

sh6benchN [1] The sh6benchN stress test allocates and deallocates memory objects in a controlled manner using N threads. It introduces non-uniform memory access patterns by employing a mix of last-in, first-out (LIFO) and reverse-order deallocation strategies. This benchmark helps evaluate the allocator's performance and behavior under these specific memory access conditions.

sh8benchN Building on sh6bench, this benchmark extends the test to multi-threaded environments with N threads. It introduces complexities similar to the Larson benchmark, where objects can be freed by different threads and even in reverse order. This allows us to assess an allocator's ability to handle concurrent memory access and maintain thread safety under demanding memory access patterns.

mstressN This benchmark simulates memory usage patterns in real-world servers. It utilizes N threads, and allocations follow a power-of-two size distribution. Objects can move between threads, and some have extended lifespans. The workload is not evenly distributed across threads. After each phase, all threads are terminated and recreated, with some objects persisting between phases.

alloc-test1 and alloc-testN [9] The alloc-testN is a modern benchmark developed by OLogN Technologies AG (ITHare.com) specifically designed to evaluate allocator performance under heavy workloads. It simulates real-world scenarios with a Pareto size distribution, meaning most allocations are small with a few larger allocations mixed in. This benchmark runs on N cores, with each thread performing 100 million allocations of objects up to 1 kilobyte (KB) in size.

5 Evaluation

5.1 Setup

To gain a comprehensive understanding of the strengths and weaknesses of various memory allocators, I conducted a rigorous performance evaluation on Ubuntu 22.04.3 LTS. This analysis compared the behavior of six popular allocators: jemalloc, tcmalloc, mimalloc, lockfree-malloc, snmalloc, and hoard. This system is equipped with a powerful 14-core Intel® Core™ i7-12700H processor, featuring a combination of 6 performance-cores and 8 efficient-cores. Hyperthreading technology further expands processing capabilities by providing 20 logical processors. The processor has 1.2 MB of L1 cache, 11.5 MB of L2 cache, and 24.0 MB of L3 cache.

5.2 Methodology and Metrics

I ran all the other benchmarks (real-world applications and stress tests) on all 20 cores. Finally, I compared the performance of a retuned version of jemalloc against the default configuration. Each test was executed five times, and the results

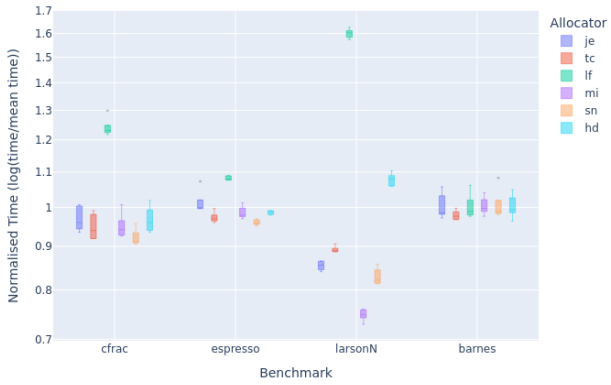

Figure 1: Normalised time for real world programs

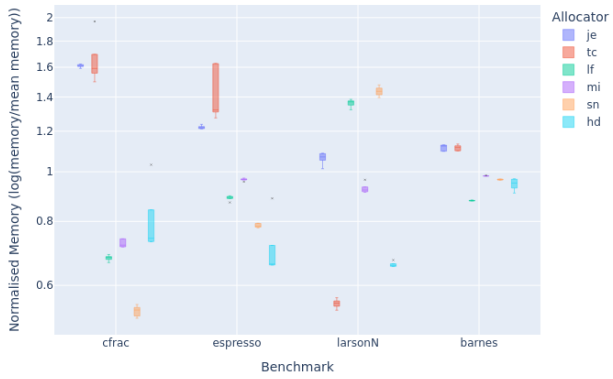

Figure 2: Normalised memory for real world programs

are visualized using box plots. The graphs will present normalized data (using the mean time/memory for each benchmark) on a logarithmic scale for effective comparison across varying workloads and configurations. I analyzed a range of performance metrics beyond just execution time, including Resident Set Size (RSS) to gauge memory footprint, page faults to understand memory access patterns, and page reclaims to assess memory management overhead. This multifaceted evaluation, encompassing diverse test scenarios and detailed performance metrics, aims to provide valuable insights into the strengths and weaknesses of each allocator. Ultimately, this will guide developers in selecting the optimal choice for their specific application needs.

5.3 Observations

Figure 1 and 2 provides the results of cfrac, espresso, barnes and larsonN. Figure 1 reveal that most allocators (except lockfree-alloc) exhibit similar performance across the benchmarks. Snmalloc leads in cfrac and espresso by a narrow margin, while mimalloc excels in larsonN, demonstrating roughly 10% faster performance compared to its competitors. However, memory efficiency varies significantly. Jemalloc and tcmalloc show considerably higher memory usage, particularly in the cfrac and espresso benchmarks. Interestingly, lockfree-alloc, despite its focus on efficiency, doesn't demonstrate the best memory characteristics. Some notable results include snmalloc's significant 20% improvement in memory efficiency compared to others in the cfrac benchmark. Additionally, the larsonN benchmark reveals an unexpected redemption for tcmalloc, showing surprisingly good performance. Finally, Hoard demonstrates consistent performance across all benchmarks. No single allocator dominates all tests. However, mimalloc and snmalloc strike a commendable balance, offering both fast allocation speeds and efficient memory usage.

Figure 3 and 4 provides the results of the stress tests. Performance-wise, mimalloc shines across all benchmarks. Notably, all allocators perform similarly on the alloctests with lockfree-alloc being about 10% slower. Tcmalloc keeps pace with mimalloc in most cases, except on sh6bench and sh8bench where it falls behind. Overall, hoard and lockfree-alloc lag the most in performance, while snmalloc and jemalloc sit in the middle. Memory efficiency presents a different picture. Hoard excels on mstress and both alloctests, demonstrating nearly 6 times better efficiency than jemalloc. Lockfree-alloc takes the crown for malloc-large, sh6bench, and sh8bench, using roughly half the memory compared to other allocators in these tests. Jemalloc struggles on the alloctests, consuming almost 3 times more memory than its competitors. However, it performs well on malloc-large and mstress. Conversely, snmalloc shows poor performance on malloc-large but excels on the alloctests. Tcmalloc delivers consistent performance across all benchmarks. The stress tests reveal a clear trade-off between performance and memory efficiency. Tcmalloc demonstrates a near-perfect balance, except for its performance on sh8bench.

Table 1 presents the average performance of the stock jemalloc allocator over 5 runs on each benchmark. Table 2 shows the performance of a tuned jemalloc allocator with bakcground thread enabled which generally improves the tail latency for application threads, since unused memory purging is shifted to the dedicated background threads. In addition, unintended purging delay caused by application inactivity is avoided with background threads. Support fro huge pages enabled which usually reduces TLB misses significantly, especially for programs with large memory footprint and frequent allocation / deallocation activities. Metadata memory usage may increase due to the use of huge pages. And finally dirty decay and muzzy decay set very high which determines how fast jemalloc returns unused pages back to the operating system, and therefore provides a fairly straightforward trade-off

Benchmark	Time(s)	RSS(KB)	Usertime(s)	System time(s)	Page-faults	Page-reclaims
cfrac	4.24	10024	4.24	0	0	245
espresso	5.12	9744	5.11	0.01	0	302
larsonN	2.908	216964	97.43	0.87	0	87712
barnes	2.31	76432	2.3	0	0	2519

Table 1: Results for untuned jemalloc

Benchmark	Time(s)	RSS(KB)	Usertime(s)	System time(s)	Page-faults	Page-reclaims
cfrac	3.27	9868	3.26	0	0	240
espresso	4.03	10024	4.03	0	0	263
larsonN	3.431	210940	98.05	0.77	0	72495
barnes	1.87	76616	1.86	0	0	2522

Table 2: Results for tuned jemalloc

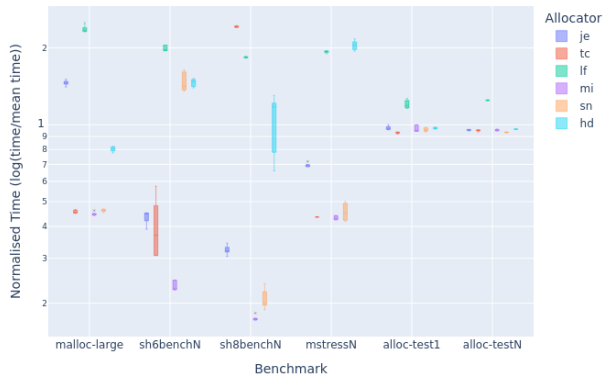

Figure 3: Normalised time for stress tests

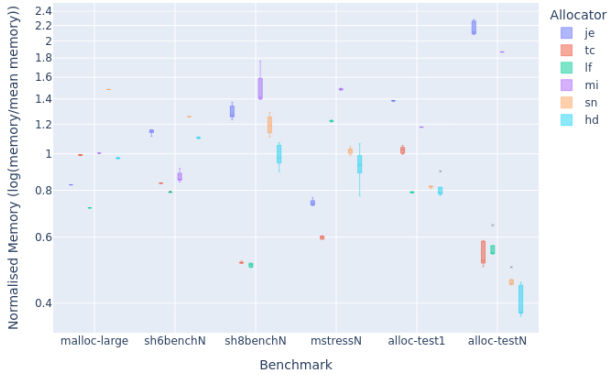

Figure 4: Normalised memory for stress tests

between CPU and memory usage. Shorter decay time purges unused pages faster to reduces memory usage (usually at the cost of more CPU cycles spent on purging), and vice versa. The retuning of jemalloc has an overall positive performance impact without comsuming too much memory. Except for larsonN the retuned version performs well in all the benchmarking tests.

6 Conclusion

This paper explores the role of memory allocators in optimizing memory usage for modern computing systems. I evaluated various allocators through a custom benchmarking framework designed to simulate diverse program behaviors. The benchmarks revealed a trade-off between speed and memory usage for memory allocators. Mimalloc and snmalloc offered a commendable balance of speed and memory efficiency, while Hoard excelled in memory usage, and mimalloc led in performance across most benchmarks. Tuning jemalloc yielded significant improvements, highlighting its potential. This project empowers developers to make informed decisions by understanding the trade-offs between allocators and their suitability for specific application needs. All benchmarking results used in the report(along with some additional ones) can be viewed in the github repository provided in the introduction section.

References

[1] http://www.microquill.com. sh6bench available at http://www.microquill.com/ 2006.

[2] https://github.com/daanx/mimalloc-bench. Jun 2019.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating*

Systems, ASPLOS IX, page 117–128, New York, NY, USA, 2000. Association for Computing Machinery.

[4] leijen daan. https://github.com/microsoft/mimalloc. Jun 2019.

[5] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 163–174, New York, NY, USA, 2002. Association for Computing Machinery.

[6] Jason Evans. In proceedings of the 2006 bsdcan conference. bsdcan'06'. ottowa,ca. may 2006. http://people. freebsd.org/ jasone/jemalloc/bsdcan2006/jemalloc.pdf. 01 2006.

[7] Google. https://github.com/gperftools/gperftools. 01 2014.

[8] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. *SIGPLAN Not.*, 28(6):177–186, jun 1993.

[9] OLogN Technologies AG (ITHare.com). "testing memory allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc, while trying to simulate real-world loads." jul. 2018. http://ithare.com/testing-memory-allocators-ptmalloc2- tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/. test available at https://github. com/node-dot-cpp/alloc-test. 2018.

[10] P. Hut J. Barnes. A hierarchical o(n log n) force-calculation algorithm. 1986.

[11] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. *SIGPLAN Not.*, 34(3):176–185, oct 1998.

[12] 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*, ISMM 2019, page 122–135, New York, NY, USA, 2019. Association for Computing Machinery.