Comparative Study of Memory Allocators

Arun Kumar
114708780

Namrata Balwani
115192977

Prajjawal Agarwal
115134849

Abstract

Memory allocation is a critical component of system performance and resource management. With the increasing usage of memory by high-workload applications, there is a need for efficient memory allocation by the operating systems in order to improve program performance and reduce memory usage.

In this paper, we evaluate the performance of several popular memory allocators, on some benchmarks. We measure the performance of each allocator in terms of memory usage, fragmentation, and latency, and compare the results across different benchmarks and configurations. In addition, we use hardware performance monitoring tools such as valgrind and Linux perf to gather statistics on cache misses and memory heaps and use flame graphs to visualize the results. Our findings show that the choice of memory allocator can have a significant impact on system performance and that certain allocators may be better suited for specific workloads.

CCS Concepts: • **Memory Management → Performance Analysis**; *Debugging and profiling*; Empiricial Studies.

Keywords: dynamic memory allocators, benchmarks, linux perf tool, valgrind, memcheck, massif, flamegraphs

1 Introduction

In order to successfully assign memory to running processes, an operating system must effectively manage the system's available memory. Efficient memory allocation can help prevent memory shortages, which can cause programs to crash or the operating system to become unstable. Dynamic memory allocators are used to allocate memory dynamically during the execution of a program, as opposed to allocating memory statically at compile-time.

There are several dynamic memory allocators currently available, each of which was made for different use cases such as CPU-intensive, memory-intensive, and multithreaded applications. There have been few works done in this area to compare different memory allocators for different benchmark applications [2]. Inspired by those works we have tried comparing some general-purpose memory allocators for a different set of benchmarks. We have used five different memory allocators which are described in the next sub-sections:

1.1 jemalloc

jemalloc is a general-purpose malloc(3) implementation that emphasizes fragmentation avoidance and scalable concurrency support. It was first created as a FreeBSD-specific library, but it has now been ported to a variety of other systems and is freely accessible[3].

1.2 SuperMalloc

SuperMalloc is an implementation of malloc(3) originally designed for X86 Hardware Transactional Memory (HTM)[7]. This is a highly scalable allocator beneficial for parallel programs running on multicore processors. SuperMalloc's capacity to manage several threads without experiencing significant contention or lock overhead is one of its main benefits.

1.3 ptmalloc

ptmalloc is the standard memory allocator used in libc library. It aims to minimize cache misses by grouping together small objects that are likely to be allocated together. When compared to other memory allocators, one of the important characteristics of ptmalloc is its capacity to decrease lock contention in multi-threaded applications.

1.4 Hoard

Hoard is a Scalable Memory Allocator for Multithreaded Applications primarily focussing on improving performance through speed, scalability, false sharing avoidance, and low fragmentation [1] It's capacity to manage numerous threads without experiencing severe lock contention or synchronization costs is one of its main advantages.

1.5 TCMalloc

TCMalloc is Google's customized implementation of C's malloc() and C++'s operator newly used for memory allocation within our C and C++ code. It is a fast, multi-threaded malloc implementation and is made to handle numerous threads concurrently without slowing down.

2 Methodologies

In this paper, we are trying to evaluate the performance of various popularly used dynamic memory allocation algorithms such as jemalloc, SuperMalloc, hoard, TCMalloc, and ptmalloc. We use six dynamic memory management intensive benchmarks from various sources to evaluate the memory allocators.

We have chosen benchmarks that are essentially allocation-intensive and have a large amount of live data for which space cost will matter the most and hence will enable us to differentiate between the various allocator's performances. In addition, some of our measurements of memory usage may introduce errors of up to 4 or 5 KB in bad cases; we wanted

to ensure that these errors were generally small relative to the actual memory usage and fragmentation.

We tried to obtain a variety of traces, including several that are widely used as well as CPU and memory-intensive[6]. We also tried to evaluate the performance on the basis of single and multithreaded applications to get an insight into how the allocators perform when there is a careful consideration of parallel processing, concurrent accesses, and scalability in general for the multithreaded environment versus the single-threaded one.

The next subsections contain the benchmarks that were used and the details about our experimental study.

2.1 Single threaded benchmarks

2.1.1 espresso. espresso is a widely used optimizer for programmable logic arrays, described by Grunwald, Zorn, and Henderson[5]. The benchmark is used in the context of cache-aware memory allocation.

2.1.2 cfrac. cfrac by Dave Barrett, is an implementation to factor large integers using the continued fraction factorization method. The original application uses explicit reference counting to reclaim storage[4].

2.1.3 gs. Ghostscript is a widely-used portable interpreter for PostScript[6] and PDF files. We used a benchmark that makes Ghostscript process the entire Intel Software Developer's Manual PDF, which is around 5000 pages.

2.1.4 lua. Lua is a lightweight scripting language that is often used for embedded systems and other resource-constrained environments. The benchmark compiles the lua interpreter with the goal to measure the allocator's performance in terms of speed and memory usage.

2.1.5 redis. We are running redis-benchmark, with 1 million requests pushing 10 new list elements and then requesting the head 10 elements, and measuring the requests handled per second. This benchmark simulates a real-world workload.

2.2 Multi threaded benchmarks

2.2.1 Larson. Larson simulates a server workload using 100 separate threads which each allocate and free many objects but leaves some objects to be freed by other threads. Also, it uses sized deallocation calls which have a fast path in some allocators [8]. This benchmark is used to evaluate memory allocation metrics for long-running server applications and scalability on SMP systems.

2.3 Experimental study

In this experimental study, we used the Mimalloc-bench, an open-source suite for benchmarking malloc implementations [9], which is a collection of various benchmarks from the academic literature and memory allocators. We wrote various scripts to run hardware monitoring tools such as Linux perf to collect CPU data during the execution for a given set of benchmarks and memory allocators and visualized this data using flame graphs to validate our observations. Additionally, we modified a few scripts in the test suite for fetching the data from the Valgrind tool to get the memory allocation and fragmentation details for each allocator on various benchmarks. This data was then converted into normalized graphical representations using some additional scripts.

3 Experimental Setup

An x86_64-architectured computer system that could run in both 32-bit and 64-bit CPU op-modes was used. The machine had 16GB of RAM and a dual-core CPU that could handle two threads running in parallel on each core. A 2.80GHz Intel(R) Xeon(R) Platinum 8370C CPU was chosen as the specific CPU model. Through the VT-x technology, the system also included virtualization capabilities, with Microsoft serving as the hypervisor vendor. The type of virtualization employed was full.

4 Result

This investigation provides several insights when it comes to time and memory optimization. All the figures and results that we presented in this investigation are normalized on our standard ptmalloc allocator which is standard on most Linux machines.

We present a comparison of the runtimes of our benchmark applications normalized to the original sys or ptmalloc allocator. We have represented both sys (or the kernel) time and the total time taken by these benchmarks as shown in Figure 3 and Figure 4 respectively.

We can see that the average total time taken by all allocators is more or less similar for all benchmarks except we see hoard, SuperMalloc, and TCMalloc performing better on multithreading workloads like Larson. With the kernel runtime, we see that the SuperMalloc performs extremely well for the Larson benchmark with low page faults (see Figure 2), almost showing 70% better performance compared to the others, because it uses hardware transactional memory to speed up parallel operations which is useful in the multithreaded Larson application.

For the two extremely allocation-intensive benchmarks, cfrac and lua, we also observed that SuperMalloc performs worst on both cfrac and lua compared to other allocators, which we suspect is because SuperMalloc is optimized for low memory fragmentation and thus incurs additional overhead when from extra virtual memory allocations [7].

An interesting observation was that the number of bytes allocated (see Figure 7) was the same among jemalloc, hoard, SuperMalloc for benchmarks: cfrac, espresso, gs. For Larson benchmark number of bytes allocated vary for different allocators, ptmalloc assigns almost twice of jemalloc and

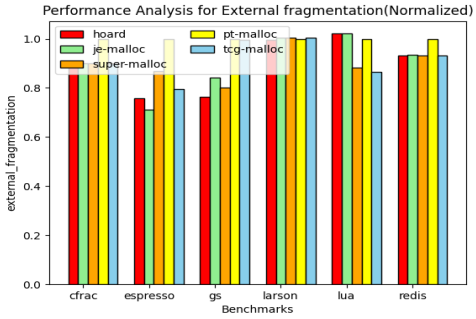

Figure 1. External fragmentation (Normalized)

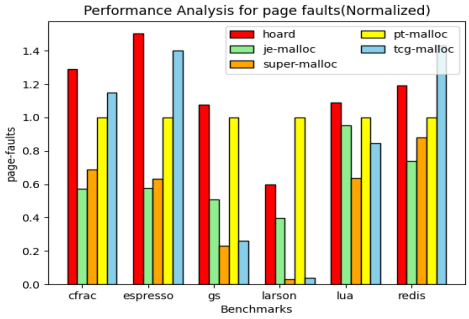

Figure 2. Page Faults (Normalized)

SuperMalloc which has the least. We observed that for multi-threaded benchmarks, scalable allocators like hoard and SuperMalloc have performed better in almost all the metrics compared to our standard pt-malloc.

When it comes to external fragmentation (see Figure 1), we observed SuperMalloc and hoard have fared better compared to all other allocators. Though we think our methodology to measure external fragments has some gap, it does account for padding present inside the allocated structure, and the general trend is the same as we expected to observe.

We use Redis to test whether the allocators can take on large numbers of requests and still maintain a high workload. For almost all the allocators, we observed similar performances when it comes to time and external fragmentation. However, we observed significantly fewer page-faults for jemalloc and SuperMalloc nearly 15% and 25% compared to our standard allocator ptmalloc.

We also built the flame graphs (see Figure 6) using the perf tool and were able to correlate our results with the data presented there. But due to space constraints, a larger discussion cannot be done there.

We also calculated other metrics context switches, resident set size, no of free, and number of allocations. However, due to space constraints, we are not able to discuss everything, but both the normalized as well as the experimental data are available at our GitHub repo here https://github.com/prajjawal05/comparing-mallocs.

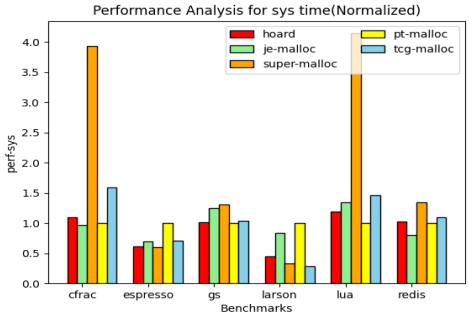

Figure 3. Kernel run-time (Normalized)

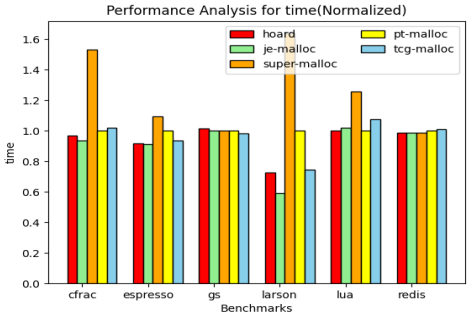

Figure 4. Total time (Normalized)

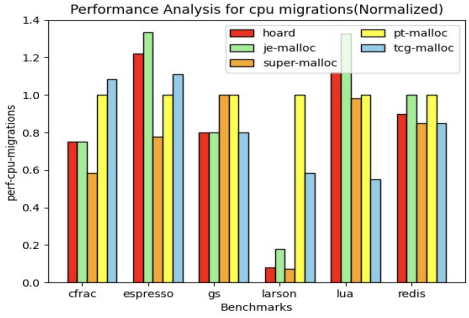

Figure 5. CPU Migrations (Normalized)

5 Conclusions

Dynamic memory management is a critical part of many important applications for which performance is crucial. Organizations like Facebook and Google, in an effort to avoid the overhead of standard allocators like ptmalloc provided in Linux, often write their own custom allocation implementations in an effort to increase performance further. Our experiments with different benchmarks suggest allocators performed best when they are designed with the type of load they are expected to encounter. Most of the allocator trade-offs include extra memory overhead or allocations

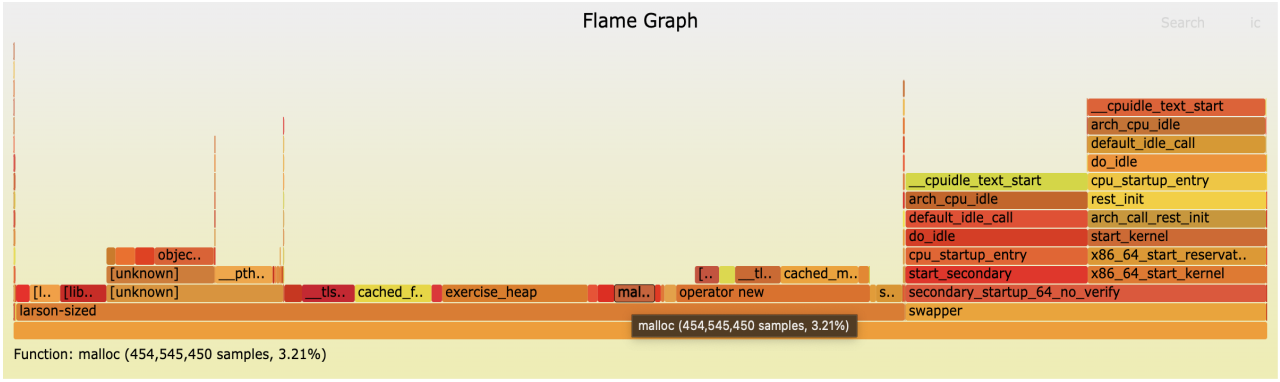

Figure 6. Flamegraph for Larson Benchmark with SuperMalloc

	Bytes Allocated					No of Allocations				
Benchmark	**Hoard**	**je-malloc**	**super-malloc**	**pt-malloc**	**tcg-malloc**	**Hoard**	**je-malloc**	**super-malloc**	**pt-malloc**	**tcg-malloc**
cfrac	1,831,917,180	1,831,917,180	1,831,917,180	1,831,844,476	1,831,917,182	91,530,285	91,530,285	91,530,285	91,530,284	91,530,287
espresso	3,726,668,584	3,726,668,584	3,726,668,584	3,726,595,880	3,726,668,586	33,510,241	33,510,241	33,510,241	33,510,240	33,510,243
gs	4,149,824,291	4,149,824,291	4,149,824,291	4,149,556,363	4,149,519,893	125,100	125,100	125,100	124,711	125,152
larson	36,810,491,086	38,201,791,724	41,378,361,911	60,990,019,712	48,977,905,881	73,125,313	75,887,665	82,200,177	121,162,758	97,300,210
lua	759,628	759,637	759,670	686,853	759,687	12,302	12,302	12,302	12,306	12,304
redis	25,745,084	25,745,084	25,745,084	25,745,084	25,745,084	113,868	113,868	113,868	113,868	113,868

Figure 7. Table showing the number of bytes allocated and the number of allocations for each benchmark application and a memory allocator.

and performance. Strategies like per thread or processor allocations tend to provide performance benefits with extra memory overhead. Scalable allocators like Hoard and super malloc tend to perform better when they experience a multithreaded workload.

During executing our experiments we encountered some issues. While trying to collect TLB misses data using Valgrind we faced some errors due to incompatible architectures. As we were running our test suite on a guest VM through a hypervisor, capturing this data was difficult and did not give appropriate results. There was a similar LLVM test suite which was officially provided by Microsoft, that we were not able to run because of the system compatibility issues with cmake tool. While running the Larson benchmark to collect data from Valgrind, it was taking a long time approx 9+ hrs to get the full data for 16 processes, we modified the script to capture the data without the memory leaks.

We believe our experimental results are explanatory, however, due to the high complexity in present applications it is difficult for us to pin-point the exact reasons for differences in performances. Future work should be directed more toward increasing the explainability of software performance and increasing the scope of performance metrics.

6 Acknowledgement

We would like to express our sincere gratitude to Professor Dongyoon Lee for his invaluable guidance and support throughout the duration of the project and course. His extensive knowledge and expertise in the field have been instrumental in shaping our understanding and helping us to achieve our goals.

References

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

[2] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. 2001. Composing high-performance memory allocators. *ACM SIGPLAN Notices* 36, 5 (2001), 114–124.

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

[4] David Gay and Alex Aiken. 1998. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation.* 313–323.

[5] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. 1993. Improving the cache locality of memory allocation. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation.* 177–186.

[6] Mark S Johnstone and Paul R Wilson. 1998. The memory fragmentation problem: Solved? *ACM Sigplan Notices* 34, 3 (1998), 26–36.

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

[8] Per-Åke Larson and Murali Krishnan. 1998. Memory allocation for long-running server applications. *ACM SIGPLAN Notices* 34, 3 (1998), 176–185.

[9] Daan Leijen Microsoft Corporation. 2019. *Mimalloc-bench.* https://github.com/daanx/mimalloc-bench/tree/master