There Ain't No Such Thing as a Free Custom Memory Allocator

Kudrjavets, Gunnar; Thomas, Jeffrey; Kumar, Aditya; Nagappan, Nachiappan ; Rastogi, Ayushi

Published in:
2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)

DOI:
10.1109/ICSME55016.2022.00079

2022 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 978-1-6654-7956-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSME55016.2022.00079

There Ain't No Such Thing as a Free Custom Memory Allocator

Gunnar Kudrjavets
University of Groningen
9712 CP Groningen, Netherlands
g.kudrjavets@rug.nl

Jeff Thomas
Meta Platforms, Inc.
Menlo Park, CA 94025, USA
jeffdthomas@fb.com

Aditya Kumar
Snap, Inc.
Santa Monica, CA 90405, USA
adityak@snap.com

Nachiappan Nagappan
Meta Platforms, Inc.
Menlo Park, CA 94025, USA
nnachi@fb.com

Ayushi Rastogi
University of Groningen
9712 CP Groningen, Netherlands
a.rastogi@rug.nl

***Abstract*—Using custom memory allocators is an efficient performance optimization technique. However, dependency on a custom allocator can introduce several maintenance-related issues. We present lessons learned from the industry and provide critical guidance for using custom memory allocators and enumerate various challenges associated with integrating them. These recommendations are based on years of experience incorporating custom allocators into different industrial software projects.**

***Index Terms*—Memory, allocator, software maintenance.**

I. Introduction

A memory allocator is responsible for handling memory management requests coming from an application. The typical operations that an application can request an allocator to perform are related to allocating and releasing a specific amount of memory. The entirety of memory available to the operating system is managed by a kernel memory allocator. A typical application runs in a user mode. In this paper, we focus on user mode memory allocators.

A *default allocator*, such as one from GNU C Library [1], is typically provided by an operating system or a runtime library of a specific programming language. One approach to improve the application's performance is to *replace the default memory allocator with a custom one*. Various case studies [2], [3], [4], [5] document the benefits of using custom memory allocators in the industry. However, the maintenance issues (see Section III) associated with integrating a custom memory allocator with the application's code base have not been studied. Based on our experience with integrating and using custom memory allocators for several years, we present a structured enumeration of various issues associated with maintaining custom memory allocators.

II. Background

The performance characteristics of an application are important criteria for its success. Improving performance is also one of the main reasons to use custom memory allocators. Applications like database engines, stock trading systems, or web browsers use custom memory allocators [4], [5]. A variety of allocators exist: *jemalloc* [6], *mimalloc* [7], *rpmalloc* [8], *snmalloc* [9], and *tcmalloc* [10]. We observe that both industry and open-source software projects tend to use either *jemalloc* (currently mainly developed by Facebook) or *tcmalloc* (developed by Google). The *mimalloc* project developed by Microsoft is another industrial-strength allocator.

Some allocators were developed for the sole purpose of increasing the performance of a specific application. For example, the introduction of *PartitionAlloc* into Chromium resulted in up to 22% memory savings on Windows [11].

Using custom memory allocators has multiple benefits:

- **Ability to tune allocator's behavior to meet the needs of a specific application**. Default allocators are optimized using the one-size-fits-all approach to work at an equally acceptable level for *all the applications*. With custom allocators, such as *jemalloc*, several configuration options can be used to optimize the allocator's performance both during the initialization or during the runtime.
- **Improved debugging and tracing facilities**. Memory-related defects compose a sizeable portion of post-release defects. For example, 70% of post-release security issues Microsoft has to fix annually are related to memory management [12]. Typical memory-related problems are double-frees, heap corruptions, and memory leaks. Custom allocators provide an enhanced debugging experience and help engineers fix the defects faster. Using an allocator with improved debugging capabilities is an efficient way to find defects that the default allocator cannot detect.
- **Access to complete source code**. In closed-source operating systems, engineers do not have access to the allocator's source code. That makes debugging complex issues challenging. Using an open-source allocator introduces several benefits. Engineers can add new features or remove unnecessary code to further optimize applications performance. Availability of source code also enables engineers to have an improved debugging experience.

2576-3148/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSME55016.2022.00079

We have observed that the benefits provided by custom allocators come with the significant cost related to maintenance of the allocator's code base alongside the application's code base. Existing research [2], [3], [4], [5] highlight the benefits, but they do not document the drawbacks and potential problems related to using custom allocator code. Using a custom allocator does not guarantee that the application's performance will improve. The behavior of a custom allocator on the same platform varies significantly [13].

In practice, replacing various components with custom versions is a standard performance optimization technique. For example, engineers can use a different library to parse JSON, or use an implementation of data structures that are different from those provided by the language runtime. With those components, engineers typically explicitly call one function instead of another. Replacing the memory allocator has one crucial difference—*it impacts all the code interacting with the allocator that is loaded and executed in the same process address space*.

One way to replace a default allocator is to provide custom implementations for a fixed set of functions that relate to memory management. Their interface does not change but the implementation does. Typical functions that memory allocators intercept and replace [14] are shown in Listing 1.

Listing 1
POSIX COMPLIANT MEMORY ALLOCATION FUNCTIONS.

```
void free(void* ptr);
void* malloc(size_t size);
void* calloc(size_t num, size_t size);
void* realloc(void* ptr, size_t size);
```

Replacing allocators is a risky proposition. Application code, all of its dependencies, and system libraries will have the component responsible for memory management replaced underneath them. There is no indication that the switch has happened. An issue related to memory management (either exposed or introduced by a custom allocator) has catastrophic consequences for an entire application. Corrupting a single location in the heap of the current process means that the application will be in an inconsistent state, and its future behavior is undetermined.

III. MAINTENANCE CHALLENGES

A. Source code management

1) Build support: Custom memory allocators are mostly developed as open-source software. Various build systems such as Ant, Babel, Buck, CMake, and Ninja exist. Authors of the custom allocator support only a limited set of build systems. However, an application that intends to use a custom allocator may utilize a build system that is unsupported by the provider of an allocator. For example, a company can use an internal build system that they do not expose to the public. Similar problems are related to the supported set of compilers and their versions.

As a first task, engineers may spend a nontrivial amount of time (days, weeks) to build a working version of the allocator. That version needs to work with the compiler, compiler switches used to develop a particular application, and a specific build system. In some cases, the support for a target operating system itself may be experimental. As a result, engineers themselves are responsible for porting the allocator to a new platform. For example, as of April 2022, *jemalloc* is still not officially supported for a popular platform like iOS.[1]

2) Version updates: Like any other dependency exposed in the form of source code, periodic updates to the dependency's source code may need to be applied. Typical reasons for integrating new versions include the availability of desired features, fixes for critical defects, or performance improvements. Engineers will need to allocate some resources for periodic updates (e.g., weekly, monthly, or release basis) or continuously monitor the dependency's code base. Each update may again have problems described in Section III-A1. For example, new code changes in allocator may use language features that are not supported by the compiler used to build the application consuming the allocator.

B. Quality

1) Defect management: Commercial operating systems, such as Apple's Darwin or Microsoft Windows, are developed and tested by a relatively large team of engineers [15], [16] compared to open-source software. The commercial allocator's code benefit from going through an extensive testing process to ensure that all the applications deployed with the operating system are functioning correctly. If a user discovers critical bugs after the operating system's release, there are usually channels to request fixes. For example, a premium support contract or a service-level agreement may guarantee a particular defect resolution time.

For systems that demand reliability, like stock trading infrastructure, having this level of access to support is critical. In the case of custom allocators released as open-source software, such formal agreements do not exist. Projects like *jemalloc* or *mimalloc* have only a handful of core contributors. There are no guarantees that any discovered issue will get a resolution during a specific timeframe. The support channel may consist of just filing an issue on GitHub with the hope that someone will respond to it and provide an actionable resolution.

2) Toolchain support: During their daily work, engineers use developer productivity tools such as debuggers and profilers. These tools help to analyze the application's memory usage and debug issues related to memory management. Custom allocators use various techniques to replace the default allocator. Those techniques include the usage of callbacks [17, p. 977], hooks [18], or preloading the shim [19]. The productivity tools may also need to replace a default allocator to track the memory usage. The intercept mechanism may be identical that used by a custom allocator. No support exists for more than one entity to intercept allocations in parallel. As a result, the productivity tools are not usable with a custom allocator, thus reducing the techniques and tools engineers have at their disposal to detect and fix defects.

[1]https://github.com/jemalloc/jemalloc/issues/2086

We have witnessed anecdotal instances of productivity tools assuming the presence of a default allocator. In those cases, the tools may rely on undocumented internal implementation details such as the presence of certain linker symbols, patterns signifying how the invalid memory is identified, or the location of log files.

Using a custom allocator breaks those assumptions. If engineers observe a memory corruption or a leak, they cannot use productivity tools with the custom memory allocator. Efficient debugging and profiling are possible only with the default allocator. This limitation makes it challenging to determine if the root cause of the incorrect behavior is the application, the memory allocator itself, or a combination of both. Limited reproducibility means that engineers may first have to validate all memory-related defects with a default allocator.

3) Defect exposure: There exist cases where the allocator may hide certain types of errors from users or try to avoid crashing or corrupting the process heap in case of invalid input. For example, in C++, it is recommended to use either operator `new` or `std::make_unique` to allocate the storage for an object. The expectation is that if the caller allocates memory via operator `new`, the caller will use operator `delete` to release it. However, a caller could use the `free()` function instead. A default allocator may allow releasing memory allocated using a "mismatched function" but the custom allocator will not. Such differences in behavior may expose genuine defects in the application.

Another category of problems is related to unwritten contracts when allocating memory. For example, some allocators may block the call to `malloc()` until there is available memory, and some may not. Some allocators may subscribe to the low-memory notifications from the operating system to preemptively either release the pre-allocated memory or perform an internal memory compaction process. This phenomenon of depending on observable behaviors is known as the "Hyrum's Law."[2]

4) Support for multiple allocators: There are various reasons why the application needs to support multiple allocators in parallel. In our experience, the most common reasons are:

- *Ability to revert to the default allocator*. A custom memory allocator may contain defects that cannot be fixed in the time allocated. The engineers may have to revert the application to using a default allocator to unblock the development process. Alternatively, the latest operating system update may not be compatible with how the custom allocator works. For engineers to revert the dependency usage efficiently, they must make sure that the application still builds and works with various allocators. That requires the introduction of an additional build flavor that adds to the daily maintenance costs.
- *Ability to compare the performance characteristics*. As the application's code base evolves, its behavior changes. Those changes can significantly impact the application's memory usage patterns. The acceptable performance results from weeks or months ago may no longer be optimal. Using a custom allocator may result in performance degradation instead of improvements [20]. Therefore, engineers must periodically run performance and stress tests using multiple allocators.
- *Preemptive detection of compatibility*. For open-source software, the success of a project or a company may depend on software's adoption rate. Adaptors of software involved in tasks that require a significant amount of resources (e.g., database engine, image processing) may use custom allocators that the authors of the component do not support. If the authors cannot make the component function with a custom allocator, the consumer may switch to a different product.

Additional reasons we have observed in the past include discontinuing of custom allocator development, software licensing changes, or the release of a new allocator that provides better performance.

C. Performance

1) Side-effects of optimizations: When an application requests memory, the allocator rounds up the allocation size to a specific value (e.g., next power-of-two) depending on the algorithm it uses. That value may be specific to a page size of an operating system or how a particular allocator manages memory internally. For example, when a caller allocates 33 bytes on macOS, the actual size of allocation as reported by `malloc_size()` is instead 48 bytes. The memory allocated is 31% larger than asked for.

A standard performance optimization technique is to customize the layout of data structures. Data structures are aligned and combined to reach a specific size to reduce waste. The application then explicitly makes requests rounded to a specific size. However, using a custom memory allocator may increase the request size internally because it must account for its private metadata. Therefore, allocations assumed to be of optimal size may use significantly more memory and increase the application's resource usage. We have observed this problem repeatedly during commercial software development.

2) Cost of generalization: The open-source software allocators are intended for multiple operating systems. Each operating system has its kernel allocator and memory management strategy that is different from others. For example,

- iOS uses a concept called *compressed memory* [21]. Releasing memory on iOS can require uncompressing the affected memory region into the available free memory to be released. Ironically, freeing memory on iOS may cause an application to run out of memory and possibly be costlier than just allocating memory.
- Linux kernel uses a concept called *overcommitting*. It means that during the allocation, memory is not reserved [22], [23]. The memory is allocated only when the first write to the allocated region is made. On the other hand, Windows explicitly tries to avoid this situation [24].

Understanding the nuances of memory management for each operating system is an involved task. Engineers re-

[2]https://www.hyrumslaw.com/

sponsible for developing custom allocators may not have the resources to *become experts for all the operating systems*. To increase the allocator's portability, the allocators limit the interface they use to interact with the operating system's memory management routines. For example, *mimalloc* makes a conscious choice to use only `mmap()` to request memory [7].

Authors of an allocator can choose to optimize code paths for each operating system separately. Supporting code specific to an operating system means the usage of many `#ifdef` type of statements across the code base. Ensuring that the allocator builds without errors may require a different build for each preprocessor directive specific to an operating system. That implies increased costs for software maintenance.

IV. Conclusions and recommendations

Existing research on allocators enumerate either the technical nuances necessary to produce one or the benefits of utilizing a custom allocator. This paper documents several complications that result from the usage of custom allocators. Those issues can increase the maintenance cost of the software, expose existing bugs, and result in inadvertent performance regressions. *Most of the maintenance challenges we enumerate are also relevant to software components other than allocators*. We recommend that a project using a custom allocator:

1) *Invest in developing the in-house expertise regarding a particular allocator*. Engineers may need that knowledge to fix the defects in the allocator's code, tune its performance, and efficiently upgrade the source code with each new release. Both the initial and ongoing maintenance investments are necessary to integrate a custom memory allocator successfully
2) *Maintain working builds of the application with both the default and custom allocators*. A project may have to be ready to switch back to a default allocator at a moment's notice if defects cannot be fixed immediately or the custom allocator causes significant performance regressions. Engineers may have to use the application built with the default allocator to use specific tools such as memory leak detectors or profilers.
3) *Cultivate a good relationship with the community and authors of the allocator*. If using the custom memory allocator is critical to the product's success, then making a conscious investment into establishing good relations with the community producing the allocator is important. Engineers should report all the defects found, try to improve the documentation, and contribute back any improvements made to the allocator's code base.

References

[1] R. McGrath. The GNU C Library. [Online]. Available: https://www.gnu.org/software/libc/

[2] J. Evans, "A scalable concurrent malloc(3) implementation for FreeBSD," in *Proceedings of the BSDCan Conference*. Ottawa, Canada: University of Ottawa, May 2006. [Online]. Available: https://papers.freebsd.org/2006/bsdcan/evans-jemalloc/

[3] ——, "Scalable memory allocation using jemalloc," Jan. 2011. [Online]. Available: https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/

[4] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Composing high-performance memory allocators," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 114–124. [Online]. Available: https://doi.org/10.1145/378795.378821

[5] D. Durner, V. Leis, and T. Neumann, "On the impact of memory allocation on high-performance query processing," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3329785.3329918

[6] J. Evans, "Tick tock, malloc needs a clock," in *Applicative 2015*, ser. Applicative 2015. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2742580.2742807

[7] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," Microsoft, Tech. Rep. MSR-TR-2019-18, June 2019. [Online]. Available: https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/

[8] M. Jansson. rpmalloc - General Purpose Memory Allocator. [Online]. Available: https://github.com/mjansson/rpmalloc

[9] P. Liétar, T. Butler, S. Clebsch, S. Drossopoulou, J. Franco, M. J. Parkinson, A. Shamis, C. M. Wintersteiger, and D. Chisnall, "Snmalloc: A message passing allocator," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 122–135. [Online]. Available: https://doi.org/10.1145/3315573.3329980

[10] S. Lee, T. Johnson, and E. Raman, "Feedback directed optimization of TCMalloc," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2618128.2618131

[11] B. Lizé and B. Nowierski. (2021, Apr) Efficient And Safe Allocations Everywhere! [Online]. Available: https://blog.chromium.org/2021/04/efficient-and-safe-allocations-everywhere.html

[12] M. Matt, "Trends and challenges in the vulnerability mitigation landscape," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA, USA: USENIX Association, Aug. 2019.

[13] S. Patel, "Behavioural study of memory allocators for Android platform," in *2017 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, 2017, pp. 52–55. [Online]. Available: https://doi.org/10.1109/ICCE-ASIA.2017.8309320

[14] IEEE and The Open Group. (2018) The Open Group Base Specifications Issue 7, 2018 edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

[15] M. Lucovsky, "Windows—a software engineering odyssey," in *4th USENIX Windows Systems Symposium*, Redmond, WA, USA, Aug 2000. [Online]. Available: https://www.usenix.org/legacy/events/usenix-win2000/invitedtalks/lucovsky_html/

[16] V. Maraia, *The Build Master: Microsoft's Software Configuration Management Best Practices*. Boston, MA, USA: Addison-Wesley Professional, 2005.

[17] A. Singh, *Mac OS X Internals: a Systems Approach*. Boston, MA, USA: Addison-Wesley Professional, Jun 2006.

[18] Hooks for Malloc (The GNU C Library). [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html

[19] T. Kobayashi, "Tips of malloc & free. Making your own malloc library for troubleshooting," San Francisco, CA, USA, Feb. 2013. [Online]. Available: https://elinux.org/images/b/b5/Elc2013_Kobayashi.pdf

[20] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Reconsidering custom memory allocation," *SIGPLAN Not.*, vol. 37, no. 11, p. 1–12, Nov 2002. [Online]. Available: https://doi.org/10.1145/583854.582421

[21] J. Levin, **OS Internals. Volume 1: User Space*, 2nd ed. Edison, NJ, USA: Technologeeks.com, 2017.

[22] R. Love, *Linux Kernel Development*, 2nd ed. Indianapolis, IN, USA: Novell Press, 2005.

[23] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Jan 2003.

[24] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 2*, 6th ed. Redmond, WA, USA: Microsoft Press, Sep 2012.