Who Ate My Memory? Towards Attribution in Memory Management

Gunnar Kudrjavets, Ayushi Rastogi
University of Groningen
9712 CP Groningen, Netherlands
g.kudrjavets@rug.nl, a.rastogi@rug.nl

Jeff Thomas, Nachiappan Nagappan
Meta Platforms, Inc.
Menlo Park, CA 94025, USA
jeffdthomas@meta.com, nnachi@meta.com

***Abstract*—To understand applications' memory usage details, engineers use instrumented builds and profiling tools. Both approaches are impractical for use in production environments or deployed mobile applications. As a result, developers can gather only high-level memory-related statistics for deployed software. In our experience, the lack of granular field data makes fixing performance and reliability-related defects complex and time-consuming. The software industry needs lightweight solutions to collect detailed data about applications' memory usage to increase developer productivity. Current research into memory attribution-related data structures, techniques, and tools is in the early stages and enables several new research avenues.**

***Index Terms*—Allocator, memory attribution, memory tagging**

I. Background and motivation

This paper is motivated by our industry experience while working on problems related to optimizing and tracking memory usage in both commercial and open-source software products. We observe the presence of a homogeneous set of issues related to *attribution* ("How much memory is used by a particular component?") and *accountability* ("What component is exceeding their memory budget?"). Answering those questions correctly and continuously is necessary to ensure the application's performance and reliability.

Software *performance* is associated with user satisfaction and how actively users engage with an application [1]. One criterion that impacts an application's performance is its *memory usage* [2]. Characteristics such as the total size of allocations, allocation rate, or the allocated memory type serve as metrics describing how efficiently an application uses memory. Memory is one of the most precious system resources on mobile platforms that do not support paging, such as iOS [3]. A suggested design pattern for environments where memory is limited is to make each component responsible for accounting for its own memory usage [4].

A detailed understanding of memory usage is also required to increase *reliability*. A variety of modern kernels and operating systems based on them, such as Android, FreeBSD, iOS, or Linux, use a system component called an *out-of-memory killer* [5], [6], [7]. An out-of-memory killer is responsible for terminating processes when their memory usage exceeds a certain quota, or excessive memory usage puts the stability of an entire operating system in danger. *Engineers need to understand the application's memory footprint in detail to avoid premature termination*.

In industry, we note that organizations develop their in-house ecosystems to solve company-specific memory attribution issues. A similar trend is present with open-source software where projects such as Mozilla Firefox have a product-specific extensive framework to track and attribute memory usage [8]. We observe the shortcomings in memory attribution capabilities and tools even in popular kernels such as Linux that has three decades of real-world usage and an active developer community [9].

Most of the research related to memory management focuses on increasing the performance of custom allocators [10], [11], [12], [13], [14] or ensuring their correctness [15], [16]. The limited existing research into memory attribution is in the early stages, involves invasive profiling techniques, and is specific to the Message Passing Interface [17].

II. State of affairs

"...how it is that memory can be allocated without the kernel knowing where it went. The problem is that the tracking infrastructure just isn't there."

— 2022 Linux Storage, Filesystem, MM and BPF Summit

Modern operating systems enable tracking memory usage at the process level [18], [19], [20], [21]. However, the *granularity at the process level is insufficient for engineers to perform efficient debugging or performance engineering* [2]. A common task that engineers encounter in practice is *determining what portion of the allocated amount of memory is consumed by a specific component, scenario, or a subsystem*. In the context of performance or reliability engineering that can mean a dynamic library, feature, function, or thread. Data about memory consumption is necessary to determine (a) what components to optimize, (b) if performance regressions are present, and (c) what performance tuning techniques to use. It is also necessary to understand how memory usage changes over time. For example, after a new commit, updates to the toolset, such as a compiler, change in dependent libraries, or change in some application's configuration parameters.

The standard approach to acquiring details about an application's memory usage is to use a *profiler*. When an

application is executed under the profiler, such as Visual Studio Profiler [22] or Xcode [23], then the profiler tracks each allocation and its source. An engineer can later filter, query, and visualize the resulting dataset. For example, the profiler may record the complete stack information coupled with the allocation size and originating dynamic library. Gathering the profiler data is time-consuming, can require the application to be compiled with specific flags (in case an instrumentation framework such as Valgrind [24] is used), and requires a significant amount of disk storage to store the entire history of allocations. Above all, *this approach is not practical outside the application's development environment*. The overhead caused by instrumentation can cause an application to execute an order of magnitude slower.

A. Existing attribution techniques

Two main approaches to attribute memory usage in source code exist: annotating each call to allocate memory and annotating each scope. We describe the existing usage patterns for each technique.

In kernel mode, Windows uses *pool tagging* [19]. Each driver can specify a pool tag when it requests to allocate memory [25], [26], [27]. Depending on a size of a driver, it can use one or more pool tags to differentiate between various subsystems. Pool tags also have another function in Windows—kernel crashes if the driver does not release all the allocations with a specific tag when the driver is unloaded [19].

On macOS, a caller can use a function such as `OSMalloc` and associate each allocation with an opaque tag. However, that tag is used only for reference counting [28]. The tag count is increased by one each time a specific tag is allocated. In user mode, macOS also enables passing custom tags generated by `VM_MAKE_TAG` macro to functions such as `vm_allocate` [28]. Listing 1 shows a sample usage pattern when a custom tag is associated with an allocation.

Listing 1
TAGGED ALLOCATION IN MACOS.

```
/* An allocation billed to networking. */
err = vm_allocate(..., VM_MEMORY_LIBNETWORK);
```

One of the FreeBSD ports is a basic heap memory accounting system `libpdel` [29] that similarly requires each caller to specify a "memory type" in a form of a string.

As a result of annotations, the application can during the runtime enumerate its virtual memory, and gather the distribution and size of allocations per a different memory tag or a type.

Hierarchically tracking memory usage by annotating source code is another option. Developers need to attribute each scope with a specific tag. All the allocator activity in that scope and its children will be attributed to that tag. Listing 2 displays how all the allocations in the function `bar()` and its children will be "billed" to the tag `foo` using the example of `TfMallocTag` tagging system [30].

Listing 2
ALLOCATION TRACKING USING THE TFMALLOCTAG SYSTEM.

```
void bar() {
    TfAutoMallocTag tag("foo");

    funcA();
    funcB();
}
```

B. Limitations of current techniques

All these approaches have constraints because they require (a) annotation of each allocation or scope, (b) complete source code to be available, and (c) hierarchical tracking needs to intercept all the allocations in the current process. However, a standard application has dependencies, such as system or third-party libraries. Without modifying the dependencies, the ability to track allocations in detail is limited to the application's "own code."

We are unaware of any operating systems, languages, or tools that enable engineers to query and keep track of memory attribution at a granular level *without sacrificing the application's performance*. The exploration of possible solutions is still in the early stages. The most recent proposal for memory allocation tracking in Linux is from August 2022 [31]. The lack of these facilities negatively impacts each non-trivial software project. Understanding the application's memory usage in detail is realistic only in the development environment. However, in our experience, *predicting or debugging an application's behavior in the production environment based on the data from the development environment is ineffective*.

III. FUTURE RESEARCH DIRECTIONS

The choice of the abstraction layers and a variety in the problem space enables multiple research avenues. Ideally, the support for memory attribution will be integrated throughout the operating system, memory allocator, and a runtime library such as the GNU C Library (glibc) [32]. Support for programming languages used for systems programming (e.g., C, C++, Rust) is imperative.

The desired solution to improve engineers' ability to attribute memory (a) can be enabled and disabled on demand, (b) has a minor performance overhead and will be usable in the production environment, (c) enables querying the memory attribution during runtime, and (d) has a well-designed set of APIs.

One potential practical approach we envision in user mode is the *usage of custom memory allocators [14], [33], [34], [13] to assist with the attribution*. Custom allocators such as jemalloc intercept each allocation request made in the context of a process. Therefore, the intercept mechanism can track all the metadata such as allocation size, current thread, timestamp, or specific flags passed to the function. The intercept mechanism can use either data from the application that specifies the current attribution scope, classify callers based on sampling the call stack, or use some other techniques.

References

[1] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A Survey of Performance Optimization for Mobile Applications," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2879–2904, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3071193

[2] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2nd ed., ser. Addison-Wesley professional computing series. Boston, MA, USA: Addison-Wesley, 2020.

[3] J. Levin, **OS internals. Volume 1: User space*, 2nd ed. Edison, NJ, USA: Technologeeks.com, 2017.

[4] J. Noble and C. Weir, *Small Memory Software*, ser. Software Patterns Series. Boston, MA, USA: Addison Wesley, Nov. 2000.

[5] Android OS Documentation. (2022, Oct.) Low Memory Killer Daemon. [Online]. Available: https://source.android.com/devices/tech/perf/lmkd

[6] D. Xu. (2018, Jul.) Open-sourcing oomd, a new approach to handling OOMs. Meta Platforms, Inc. [Online]. Available: https://engineering.fb.com/2018/07/19/production-engineering/oomd/

[7] Eklektix, Inc. (2022) OOM killer. [Online]. Available: https://lwn.net/Kernel/Index/#OOM_killer

[8] Mozilla Foundation. (2022) Performance—Firefox Source Docs documentation. [Online]. Available: https://firefox-source-docs.mozilla.org/performance/index.html

[9] J. Corbet. (2022, May) Better tools for out-of-memory debugging. [Online]. Available: https://lwn.net/Articles/894546/

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

[11] M. Jansson. (2017) rpmalloc—General Purpose Memory Allocator. [Online]. Available: https://github.com/mjansson/rpmalloc

[12] 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

[13] 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

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

[15] T. Ball, S. Chaki, and S. K. Rajamani, "Parameterized Verification of Multithreaded Software Libraries," in *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS 2001. Berlin, Heidelberg: Springer-Verlag, 2001, p. 158–173. [Online]. Available: https://doi.org/10.1007/3-540-45319-9_12

[16] A. W. Appel and D. A. Naumann, "Verified Sequential Malloc/Free," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 48–59. [Online]. Available: https://doi.org/10.1145/3381898.3397211

[17] S. K. Gutiérrez, D. C. Arnold, K. Davis, and P. McCormick, "On the Memory Attribution Problem: A Solution and Case Study Using MPI," *Journal on Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e5159, Feb. 2019. [Online]. Available: https://doi.org/10.1002/cpe.5159

[18] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2001.

[19] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals*, 6th ed. Redmond, WA, USA: Microsoft Press, 2012, OCLC: ocn753301527.

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

[21] W. Stallings, *Operating Systems: Internals and Design Principles*, 6th ed. Upper Saddle River, NJ, USA: Pearson/Prentice Hall, 2009.

[22] Microsoft Corporation. (2022) Measure performance in Visual Studio—Visual Studio (Windows). [Online]. Available: https://docs.microsoft.com/en-us/visualstudio/profiling/

[23] Apple Inc. (2022) Xcode—Features. [Online]. Available: https://developer.apple.com/xcode/features/

[24] Valgrind™ Developers. (2022) Valgrind Home. [Online]. Available: https://valgrind.org/

[25] W. Oney, *Programming the Microsoft Windows Driver Model*, ser. Microsoft programming series. Redmond, WA, USA: Microsoft Press, Oct. 1999.

[26] E. N. Dekker and J. M. Newcomer, *Developing Windows NT device drivers*. Boston, MA, USA: Addison-Wesley Educational, Mar. 1999.

[27] P. Orwick and G. Smith, *Developing drivers with the Windows Driver Foundation*. Redmond, WA, USA: Microsoft Press, Apr. 2007.

[28] A. Singh, *Mac OS X Internals: a Systems Approach*. Boston, MA, USA: Addison-Wesley Professional, 2016, OCLC: 1005337597.

[29] A. Cobbs and Packet Design, LLC. (2002, Apr.) typed_mem—heap memory accounting system. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=FREE&manpath=FreeBSD+13.1-RELEASE+and+Ports

[30] Pixar. (2022, Jul.) Universal Scene Description: The TfMallocTag Memory Tagging System. [Online]. Available: https://graphics.pixar.com/usd/dev/api/page_tf__malloc_tag.html

[31] S. Baghdasaryan. (2022, Aug.) Code tagging framework and applications. [Online]. Available: https://lore.kernel.org/all/20220830214919.53220-1-surenb@google.com/

[32] Free Software Foundation. (2022) The GNU C Library. [Online]. Available: https://www.gnu.org/software/libc/

[33] 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

[34] ——. (2011, Jan.) Scalable memory allocation using jemalloc. [Online]. Available: https://engineering.fb.com/2011/01/03/core-data/scalable-memory-allocation-using-jemalloc/