# lgmalloc: Predictive Memory Allocation

Luca M. Goddijn
luca.goddijn@gmail.com
Independent Researcher
Amsterdam, Netherlands

## Abstract

Memory allocation represents a critical bottleneck in multi-threaded applications. Traditional allocators, like `tcmalloc`, suffer from central contention points that limit scalability, while thread-local implementations require substantial per-thread memory overhead. `lgmalloc` shifts the demand-reactive paradigm that standard allocation algorithms use, to a preemptive and predictive heuristic approach. The approach leverages the fundamental observation that call sites demonstrate predictable allocation patterns. By parsing the executable's `ELF` format before the program's entry point, `lgmalloc` performs static analysis of call sites to reveal allocation patterns and generate confidence scores for size classes. Based on the confidence scores, the system utilizes optimized code paths to the backend allocation API, reducing allocation latency. When confidence scores are low, the system falls back to reliable, standard allocation strategies. The thread-local design eliminates lock contention while substantially reducing per-thread memory overhead by optimizing the pre-allocated size classes each thread requires.

`lgmalloc` is available at: https://github.com/Arty3/lgmalloc

## Keywords

memory allocation, heap, malloc, allocator, static analysis, predictive optimization, thread-local, heuristics

## 1 Introduction

As performance requirements scale, memory allocators face a rising trade-off barrier between memory efficiency and thread scalability. Existing and standardized allocators demonstrate a clear division: optimizing for performance or minimizing memory overhead. Both strategies significantly compromise in the opposing dimension. `tcmalloc` [2] is Google's custom memory allocator, designed for high-performance infrastructure applications, while `mimalloc` [5] is Microsoft's general-purpose allocator which focuses on minimizing memory overhead. Both implementations clearly illustrate the fundamental trade-offs that limit current allocation strategies.

These trade-offs manifest in concrete design choices. Performance-focused allocators like `tcmalloc` employ fundamental strategies like thread-local caches and per-size-class free lists to minimize lock contention, which comes at the cost of substantial memory overhead required to maintain these structures across threads. Conversely, memory-efficient implementations reduce per-thread allocation and utilize compact data structures. These sacrifice performance by bottlenecking critical contention points, which scale poorly at higher thread counts. The fundamental issue underlying these trade-offs is that the core principles allocators rely on are founded on a demand-reactive paradigm. This paradigm entails larger-than-needed allocations, organized in core data structures which are plagued by thread contention, or poor memory efficiency at scale.

`lgmalloc` is a general-purpose `malloc(3)` implementation that addresses this paradigm by leveraging the fundamental observation that allocation call sites exhibit significant predictability. This observation introduces three key innovations: Static analysis, confidence-based routing and a thread-local design whose effectiveness emerges through its integration with the other two approaches. `lgmalloc` statically analyzes the executable's structure before the program's main entry point, predicting allocation patterns and creating optimized code paths for allocation logic. This approach offers significant optimization potential for the key trade-off elements.

Consider an ideal scenario: a multi-threaded server where each thread allocates a compile-time constant size to process incoming data. The compile-time constant yields maximum confidence, constituting a perfectly optimized allocation system, where each thread only allocates exactly what it requires. The confidence-based routing system can now perform a direct jump to the respective backend allocation API function, which follows linear and predictable logic, offering minimal side effects. In contrast, existing allocators face bottlenecks at scale, for example `tcmalloc` experiences high contention on its global free list [4] and `jemalloc` [1] experiences significant memory overhead [3]

The key contributions of this work are:

- A novel static analysis system for predicting allocation patterns
- A confidence-based routing mechanism for optimized allocation paths
- A thread-local design that minimizes both contention and memory overhead
- Comprehensive evaluation showing [...]

The remainder of this paper is organized as follows: Section 2 surveys related work, Section 3 presents the `lgmalloc` architecture, Section 4 details key implementation aspects, Section 5 evaluates performance, Section 6 discusses results and implications, and Section 7 concludes.


ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

## References


[1] Jason Evans. 2006. A scalable concurrent malloc(3) implementation for FreeBSD. In *Proceedings of the BSDCan Conference*. Ottawa, Canada.

[2] Google. [n. d.]. TCMalloc: Thread-Caching Malloc. https://google.github.io/tcmalloc/

[3] Rivalino Matias Jr., Autran Macedo, Luciano C. Serafim, Rodrigo M. Rocha, and Felipe M. G. França. 2011. An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*. IEEE, 92–98. doi:10.1109/PDCAT.2011.53 Comparative study showing jemalloc's thread-cache behavior and memory consumption patterns in multi-threaded scenarios.

[4] Sangho Lee, Teresa L. Johnson, and Easwaran Raman. 2014. Feedback Directed Optimization of TCMalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*. ACM, New York, NY, USA. doi:10.1145/2618128.2618131 Study demonstrating performance bottlenecks in TCMalloc's central cache management and thread-local cache interactions.

[5] Daan Leijen. 2019. mimalloc: Free List Sharding in Action. In *ISMM*.