Old is Gold: Optimizing Single-threaded Applications with ExGen-Malloc

Ruihao Li, *Student Member, IEEE*, Lizy K. John, *Fellow, IEEE*, and Neeraja J. Yadwadkar

***Abstract*—Memory allocators, though constituting a small portion of the entire program code, can significantly impact application performance by affecting global factors such as cache behaviors. Moreover, memory allocators are often regarded as a "datacenter tax" inherent to all programs. Even a 1% improvement in performance can lead to significant cost and energy savings when scaled across an entire datacenter fleet. Modern memory allocators are designed to optimize allocation speed and memory fragmentation in multi-threaded environments, relying on complex metadata and control logic to achieve high performance. However, the overhead introduced by this complexity prompts a reevaluation of allocator design. Notably, such overhead can be avoided in single-threaded scenarios, which continue to be widely used across diverse application domains. In this paper, we present *ExGen-Malloc*, a memory allocator specifically optimized for single-threaded applications. We prototyped *ExGen-Malloc* on a real system and demonstrated that it achieves a geometric mean speedup of $1.19\times$ over dlmalloc and $1.03\times$ over mimalloc, a modern multi-threaded allocator developed by Microsoft, on the SPEC CPU2017 benchmark suite.**

***Index Terms*—Memory Allocator, Multi-Threading.**

I. Introduction

With the increasing prevalence of multi-threaded applications that exploit parallelism for performance gains, modern memory allocators are typically architected to support concurrent allocation and deallocation requests across multiple threads, minimizing contention and maximizing scalability. To meet the demands of multi-threaded applications, memory allocators have themselves shifted from single-threaded to multi-threaded library designs. Fig. 1 shows the evolution of memory allocators over time. During the mid-1990s, single-threaded memory allocators, such as the Win32 allocator and dlmalloc [1], [2], [3] (also referred to as Windows XP memory allocator and Lea allocator), were predominant, reflecting the widespread use of single-core processors. By the late 1990s and early 2000s, with the rise of multi-core processors, LKMalloc [4] became the first multi-threaded memory allocator. Advancements introduced techniques like tiered metadata and better metadata management with improved control flow, culminating in the development of allocators such as Hoard [5], tcmalloc [6] from Google, jemalloc [7] from Meta, and mimalloc [8] from Microsoft.

These modern memory allocators, are complex pieces of code that need to excel at many tasks, including (a) performing fast allocation and deallocation of objects, (b) handling objects of various sizes efficiently with minimal fragmentation, (c) returning freed space quickly so that further allocations can be made, (d) being scalable to many threads and cores, and so on. To achieve these tasks, modern memory allocators rely on complex metadata [5], [7], [8], [9], [10].

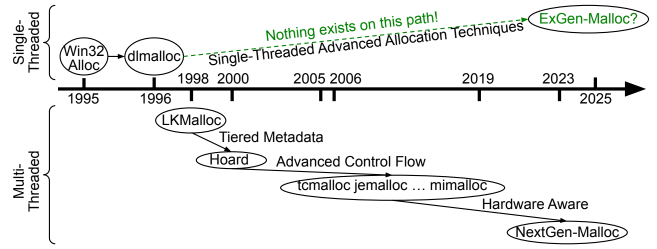


Fig. 1. Timeline of memory allocators (for C/C++). Since LKMalloc, efforts have primarily focused on multi-threaded allocators, leaving single-threaded allocators largely overlooked.

Since most developments focused mainly on multi-threaded applications, modern allocators have silently ignored single-threaded applications. However, single-threaded applications remain widely popular across various domains today. For example, *serverless* workloads are often single-threaded [11], most *data compression* algorithms operate in a single-threaded manner [12], and *Redis*, one of the most popular in-memory databases, is also single-threaded [13]. The complexities of modern multi-threaded memory allocators are shown to lead to performance degradation for applications, mainly due to the cache pollution [14] induced by allocator metadata evicting cache lines containing user program data. In contrast, single-threaded applications generally impose less demanding requirements on memory allocators compared to their multi-threaded counterparts. However, the widespread use of allocators optimized for multi-threaded workloads—even in single-threaded contexts—introduces unnecessary overheads such as synchronization primitives (e.g., locks) and additional metadata management. These overheads, while essential for thread safety in multi-threaded environments, become superfluous in single-threaded settings and can lead to measurable performance degradation.

We argue that if a program is statically known to be single-threaded, a specialized memory allocator optimized for single-threaded execution can be safely selected at link time. This approach is practical, as determining the threading model of a program—whether single-threaded or multi-threaded—is feasible during compile-time or link-time analysis [15] (details in § III). We revisit legacy single-threaded memory allocators, such as dlmalloc [2], which inherently avoid synchronization and metadata management overheads. However, to remain effective in modern systems, *ExGen-Malloc* must also accommodate increasingly complex allocation patterns, data locality behaviors, and object size distributions [16]. To this end, we propose *ExGen-Malloc*, a hybrid allocator that maintains a single-threaded design while incorporating key principles from modern multi-threaded allocators. Specifically, *ExGen-Malloc* adopts techniques such as aggregated metadata for

Manuscript received May 16, 2025; revised June 18, 2025; accepted July 7, 2025.

Ruihao Li, Lizy K. John, and Neeraja J. Yadwadkar are with The University of Texas at Austin, Austin, TX 78712, USA (e-mail: liruihao@utexas.edu, ljohn@ece.utexas.edu, neeraja@austin.utexas.edu).

efficient indexing and rapid reuse of freed memory blocks, as exemplified by mimalloc [8]. We implement a prototype of *ExGen-Malloc* on a real system and evaluate its practicality using the SPEC CPU2017 benchmark suite. *ExGen-Malloc* achieves a 1.19× geometric mean speedup over dlmalloc and a 1.03× speedup over mimalloc, a modern multi-threaded allocator developed by Microsoft.

II. A CASE FOR *ExGen-Malloc*

We first provide an overview of memory allocators (§ II-A) and discuss the evolution from single-threaded to multi-threaded designs (§ II-B). We then highlight the limitations of universally applying multi-threaded allocators to all applications (§ II-C) and motivate the need for single-threaded allocators tailored to single-threaded programs (§ II-D).

A. Background: Memory Allocators

Memory allocators manage heap memory for a process using functions like malloc() and free(), often through libraries such as mimalloc or tcmalloc (in C/C++). To track heap usage, allocators maintain internal book-keeping structures, called allocator metadata. For example, allocators use linked lists to record available memory blocks of various sizes. Coarser size blocks enable faster indexing during allocation but lead to higher memory fragmentation, whereas fine-grained size blocks reduce fragmentation at the cost of increased metadata complexity and latency of allocation. Achieving a balance between allocation speed and memory fragmentation in memory allocators remains a challenging problem that continues to attract research interest from both software and hardware communities [5], [8], [10], [17], [11], [16], [18].

B. Transition from Single- to Multi-threaded Allocators

Modern multi-threaded applications issue memory allocations concurrently, prompting a shift from single-threaded allocators like dlmalloc [2] to multi-threaded designs such as LKMalloc [4], Hoard [5], and many others developed in both industry and academia [3], [7], [19], [8], [10], [9], [20], [21]. These multi-threaded allocators are more complex and must balance allocation speed with memory fragmentation.

To support concurrent allocation, multi-threaded allocators typically adopt either (a) per-thread private memory pools or (b) a centralized shared memory region. Per-thread pools reduce contention but can cause memory *blowup*—a linear increase in memory usage with thread count [5]. In contrast, centralized pools conserve memory but incur cross-core synchronization overhead due to lock contention. Modern allocators address this trade-off with a hybrid design: thread-local caches handle fast, small allocations, while a global memory pool balances memory across threads. As illustrated in Fig. 2 (left), each *malloc()* request first consults the thread-local cache and falls back to the shared pool when needed. Periodic synchronization ensures balanced distribution of free memory among threads.

Modern memory allocators efficiently support concurrent allocation and deallocation, but their tiered metadata designs introduce two key overheads:

- **Cache pollution**: accessing allocator metadata can evict useful user data from CPU caches [14]. In multi-threaded workloads, this effect is amplified, as metadata accesses may interfere with cache lines used by other cores, causing false sharing [5] (independent variables modified by different threads reside in the same cache line).
- **Synchronization**: allocators use synchronization primitives (e.g., locks) to maintain metadata consistency, but the resulting communication overhead grows with core count, degrading system performance [22].

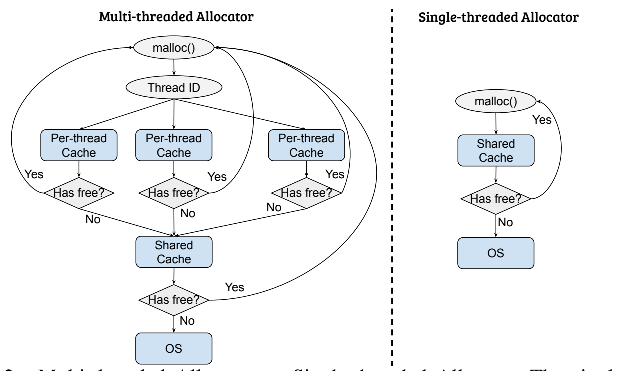


Fig. 2. Multi-threaded Allocator vs Single-threaded Allocator. The single-threaded allocator uses single-layer metadata and simplifies the control logic.

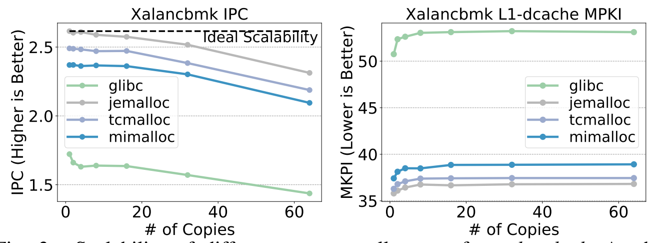


Fig. 3. Scalability of different memory allocators for *xalancbmk*. As the number of copies increases, the IPC decreases due to higher overhead caused by an increase in L1-dcache misses.

C. Multi-threaded Allocators in Single-threaded Programs

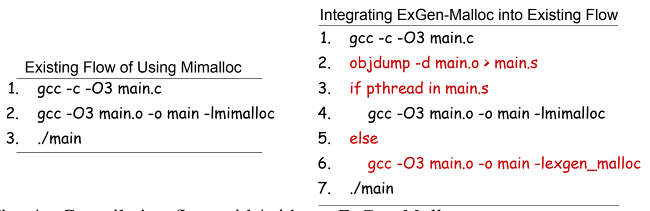
When multi-threaded memory allocators are used for single-threaded programs, they impose unnecessary performance overhead. The potential overhead includes:

- Additional tiered metadata data structure. As the *blowup* issue does not exist in single-threaded programs, a single metadata layer is sufficient to manage the available memory blocks.
- Additional synchronization and control logic. As single-tiered metadata is enough for single-threaded programs, allocators do not need complex control logic, e.g., mapping each allocation request to each thread-local cache.

Fig. 3 gives an example of the scalability of using multi-threaded allocators for *single-threaded* application *xalancbmk* (a representative workload from SPEC CPU2017 [23]). Each copy is isolated and runs on its own core, where private L1-dcache misses would typically remain stable. However, these misses still increase when using multi-threaded allocators, which is the result of metadata shared between cores.

D. A Case for Single-Threaded Allocators

The performance overhead of multi-threaded allocators suggests they may not be suitable for all applications. Notably, after LKMalloc [4] and Hoard in the early 2000s [5], research has predominantly focused on multi-threaded scenarios [3],


Existing Flow of Using Mimalloc

```
gcc -c -O3 main.c
gcc -O3 main.o -o main -lmimalloc
./main
```

Integrating ExGen-Malloc into Existing Flow

```
gcc -c -O3 main.c
objdump -d main.o > main.s
if pthread in main.s
    gcc -O3 main.o -o main -lmimalloc
else
    gcc -O3 main.o -o main -lexgen_malloc
./main
```

Fig. 4. Compilation flow with/without *ExGen-Malloc*.

[7], [19], [24], [6], [8], [9], [10], [16]. However, single-threaded applications remain popular as they are often simpler to develop, test, and debug, making them an attractive choice for developers who prioritize simplicity and reliability. Moreover, the overhead associated with thread management in multi-threaded applications can sometimes negate the performance benefits for certain types of workloads. Many important workloads remain single-threaded, including *serverless* functions [11], *data compression* algorithms [12], and in-memory databases such as *Redis* [13]. In energy-constrained edge environments, single-threaded applications are favored for their simplicity and energy efficiency [25]. Additionally, single-threaded designs offer cost benefits [26], [27], making them a practical choice in cloud computing platforms [28]. These trends motivate the exploration of memory allocators tailored to single-threaded applications.

III. Our Proposal: *ExGen-Malloc*

In this paper, we argue that it is time to resurrect single-threaded memory allocators, especially given that single-threaded applications continue to be popular in the cloud and the datacenters [11], [12], [13], [29], [30]. However, we cannot rely solely on traditional single-threaded allocators such as dlmalloc [2], as contemporary single-threaded applications depict increasingly complex data locality, allocation patterns, and object size distributions characteristic of modern software [16]. To meet these demands, we envision *ExGen-Malloc* that incorporates modern design principles inspired by multi-threaded allocators, including (but not limited to):

- Using aggregated metadata for efficient indexing and rapid reuse of freed memory blocks.
- Using bitmaps or other data structures to optimize common allocation and free paths, enabling constant-time (O(1)) execution.
- Deferring merging of adjacent free blocks to avoid runtime overhead.
- Using fine-grained size classes to reduce internal fragmentation, especially for small object allocations.
- Leveraging inline functions to reduce the overhead of allocation function calls.

By eliminating unnecessary metadata and simplifying control logic, while retaining key techniques from modern multi-threaded allocators, *ExGen-Malloc* can achieve high efficiency in single-threaded applications.

To ensure *ExGen-Malloc* is a practical solution, it must integrate seamlessly with existing compilation workflows. This requires an understanding of how current systems incorporate custom memory allocators (e.g., mimalloc), typically by linking the allocator library and overriding the default memory allocation functions. For example, mimalloc can be integrated either by linking its static library or by preloading its shared library as a drop-in replacement. As illustrated in Fig. 4 (left), a common approach on POSIX-based systems using GCC is to preload mimalloc in place of the default Glibc allocator. This workflow serves as a foundation for integrating *ExGen-Malloc* into the compilation process.

Integrating *ExGen-Malloc* into existing compilation workflows requires an additional step to determine whether a program is single- or multi-threaded. One way is to introduce an extra compilation pass. As shown in Fig. 4 (right), on a POSIX-based system using GCC, the object file $main.o$ is disassembled after compilation to detect references to threading function calls, such as $pthread\_create()$ in C, before linking a memory allocator. This detection can be automated using shell scripts (or scripts in other languages) and incorporated into the build system (e.g., $Makefile$). Importantly, developers can continue using standard memory allocation functions (e.g., `malloc()`/`free()`) without modifying their applications.

IV. Prototyping *ExGen-Malloc*

To validate that using single-threaded memory allocators for single-threaded programs can lead to performance improvements, we **prototyped** *ExGen-Malloc* on a real system (AMD EPYC 7763 with 64 cores and 8× 16GB 3200 MT/s DDR4, running Linux 5.4). We developed *ExGen-Malloc* by building atop mimalloc, removing multi-thread-related metadata and control logic to better cater to the needs of single-threaded applications. We compared *ExGen-Malloc* against the single-threaded allocator dlmalloc [2] and three state-of-the-art multi-threaded allocators—jemalloc [7], tcmalloc [6], and mimalloc [8]—using SPEC CPU2017 intrate workloads (running 64 copies, compiled with gcc-13.2.0 -O3).

As shown in Fig. 5, although jemalloc [7][1], tcmalloc [8], and mimalloc [8] are multi-threaded, they outperform the single-threaded dlmalloc [2] due to applying advanced design principles in modern allocators (§ III). By adopting modern allocator design principles while maintaining a single-threaded architecture, *ExGen-Malloc* consistently outperforms existing allocators, achieving a geometric mean speedup of $1.19\times$ over dlmalloc $1.03\times$ over mimalloc on SPEC CPU2017.

For allocation-intensive workloads with diverse object sizes such as *xalancbmk* [31], [14], *ExGen-Malloc* delivers substantial performance improvements over modern industry allocators ($1.13\times$ and $1.19\times$ speedups over jemalloc and mimalloc). However, *ExGen-Malloc* is not universally effective across all workloads, as most SPEC CPU benchmarks exhibit minimal dynamic memory activity during steady-state execution [16][2]. To further understand the source of the performance gains by using *ExGen-Malloc*, we profiled the cache behavior of *xalancbmk* under different memory allocators. As shown in Table I, *ExGen-Malloc* reduces L1-dcache MPKI by 1.25%

[1]We compile jemalloc using the lazy_lock flag, enabling it to operate in single-threaded mode without incurring mutex locking overhead. While this reduces synchronization costs, the underlying metadata structures remain as complex as in the multi-threaded configuration.

[2]*ExGen-Malloc* is designed to target real-world applications with more complex and sustained allocation patterns, rather than benchmark suites such as SPEC CPU workloads. On average, datacenter applications yield $15\times$ more time spent on memory allocation than SPEC CPU [16], [18].

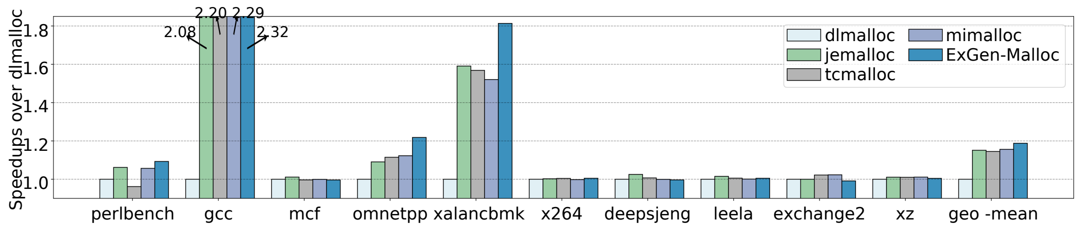


Fig. 5. *ExGen-Malloc* vs other allocators (speedups over dlmalloc, the larger the better) on SPEC CPU2017 intrate running 64 copies.

TABLE I
ExGen-Malloc REDUCES L1 AND L2 CACHE MPKI COMPARED TO OTHER ALLOCATORS ON XALANCBMK.

Metrics	dlmalloc	jemalloc	tcmalloc	mimalloc	*ExGen-Malloc*
L1-dcache MPKI	39.36	34.80	36.15	36.18	34.36
L2-cache MPKI	6.15	2.43	2.45	2.73	1.72

and 5.03%, and L2-cache MPKI (data misses) by 29.10% and 36.93%, compared to jemalloc and mimalloc, respectively.

V. Future of *ExGen-Malloc*

Beyond optimizing memory allocation with *ExGen-Malloc*, it is valuable to reassess the overhead introduced by other multi-threaded system libraries, particularly those used in common for microservices, which are often considered part of the "datacenter tax" [32], [33], [34], [16]. Even a minimal percentage of improvement in these libraries can translate to substantial power and energy savings, potentially amounting to megawatts at datacenter scale. Following the direction of *ExGen-Malloc*, we believe *ExGen-Xs* (X represents system libraries beyond memory allocators) are also on their way.

Challenges: Our comparison of dlmalloc, *ExGen-Malloc*, and other modern allocators (Fig. 5) demonstrates that reverting to older single-threaded libraries does not inherently yield performance improvements without targeted optimizations. This highlights the need for root cause analysis of each library to effectively isolate the performance impact of transitioning from a multi-threaded to a single-threaded version.

VI. Conclusion

Modern memory allocators are designed to support multi-threaded applications. To do so, these memory allocators often rely on complex metadata structures and control logic. While this complexity enables them to handle diverse and dynamic allocation patterns in modern applications, it also introduces non-trivial overhead. In contrast, such overhead can be avoided in single-threaded contexts, which remain prevalent across a broad range of applications. To address this gap, we propose *ExGen-Malloc*, a hybrid allocator that retains a single-threaded design while incorporating key principles from modern multi-threaded allocators. Prototypes on a real system show that *ExGen-Malloc* achieves a geometric mean speedup of 1.19× over dlmalloc and 1.03× over mimalloc, a modern multi-threaded memory allocator, on the SPEC CPU2017 benchmark suite. We conclude with an outline of the next research avenues using *ExGen-Malloc*: its design principles can be extended to other system libraries, such as network packet processing libraries, offering both opportunities and challenges.

References

[1] J. M. Richter, *Advanced Windows: the developer's guide to the Win32 API for Windows NT 3.5 and Windows 95*. Microsoft Press, 1995.
[2] D. Lea *et al.*, "A memory allocator," 1996.
[3] E. D. Berger *et al.*, "Reconsidering custom memory allocation," in *OOPSLA*, 2002.
[4] P.-Å. Larson *et al.*, "Memory allocation for long-running server applications," *ISMM*, 1998.
[5] E. D. Berger *et al.*, "Hoard: A scalable memory allocator for multithreaded applications," *ASPLOS*, 2000.
[6] Google, "Tcmalloc," https://github.com/google/tcmalloc/, 2025.
[7] J. Evans, "A scalable concurrent malloc (3) implementation for freebsd," in *Proc. of the bsdcan conference, ottawa, canada*, 2006.
[8] D. Leijen *et al.*, "Mimalloc: Free list sharding in action," Microsoft, Tech. Rep. MSR-TR-2019-18, June 2019.
[9] P. Liétar *et al.*, "Snmalloc: a message passing allocator," in *ISMM*, 2019.
[10] A. Hunter *et al.*, "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator," in *OSDI*, 2021.
[11] Z. Wang *et al.*, "Memento: architectural support for ephemeral memory management in serverless environments," in *MICRO*, 2023.
[12] S. Karandikar *et al.*, "Cdpu: Co-designing compression and decompression processing units for hyperscale systems," in *ISCA*, 2023.
[13] "Redis," https://redis.io/, 2025.
[14] R. Li *et al.*, "Nextgen-malloc: Giving memory allocator its own room in the house," in *HotOS*, 2023.
[15] H.-W. Tseng *et al.*, "Cdtt: Compiler-generated data-triggered threads," in *HPCA*, 2014.
[16] Z. Zhou *et al.*, "Characterizing a memory allocator at warehouse scale," in *ASPLOS*, 2024.
[17] S. Kanev *et al.*, "Mallacc: Accelerating memory allocation," in *ASPLOS*, 2017.
[18] S. Apostolakis *et al.*, "Necro-reaper: Pruning away dead memory traffic in warehouse-scale computers," in *ASPLOS*, 2025.
[19] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.
[20] H. Yang *et al.*, "Numalloc: A faster numa memory allocator," in *ISMM*, 2023.
[21] A. Reitz *et al.*, "Starmalloc: Verifying a modern, hardened memory allocator," in *OOPSLA*, 2024.
[22] A. Asgharzadeh *et al.*, "Free atomics: hardware atomic operations without fences." in *ISCA*, 2022.
[23] "Spec cpu 2017," https://www.spec.org/cpu2017/.
[24] W. Gloger, ""wolfram gloger's malloc homepage"," http://www.malloc.de/en/, 2025.
[25] A. Crotty *et al.*, "The case for in-memory olap on" wimpy" nodes," in *ICDE*, 2021.
[26] F. McSherry *et al.*, "Scalability! but at what {COST}?" in *HotOS*, 2015.
[27] M. Balduini *et al.*, "Cost-aware streaming data analysis: Distributed vs single-thread," in *DEBS*, 2018.
[28] "Aws ec2 instances," https://aws.amazon.com/ec2/instance-types/, 2025.
[29] J. Kim *et al.*, "Functionbench: A suite of workloads for serverless cloud function service," in *CLOUD*, 2019.
[30] Y. Gan *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019.
[31] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *USENIX Security*, 2010.
[32] S. Kanev *et al.*, "Profiling a warehouse-scale computer," in *ISCA*, 2015.
[33] A. Sriraman *et al.*, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in *ASPLOS*, 2020.
[34] A. Gonzalez *et al.*, "Profiling hyperscale big data processing," in *ISCA*, 2023.