GreenMalloc: Allocator Optimisation for Industrial Workloads

Aidan Dakhama[1[0009−0002−7318−7964]], W.B. Langdon[2[0000−0002−6388−4160]], Hector D. Menendez[1[0000−0002−6314−3725]], and Karine Even-Mendoza[1[0000−0002−3099−1189]]

[1] King's College London, London, UK. {aidan.dakhama, karine.even_mendoza, hector.menendez}@kcl.ac.uk
[2] University College London, London, UK. w.langdon@ucl.ac.uk

Abstract. We present GreenMalloc, a multi-objective search-based framework for automatically configuring memory allocators. Our approach uses NSGA-II and rand_malloc as a lightweight proxy benchmarking tool, we efficiently explore allocator parameters from execution traces and transfer the best configurations to gem5, a large system simulator, in a case study on two allocators: the GNU C/C++ compiler's glibc malloc and Google's TCMalloc. Across diverse workloads, our empirical results show up to 4.1% reduction in average heap usage without loss of runtime efficiency, indeed we get a 0.25% reduction.

1 Introduction

Efficient memory management remains a challenge in modern computing, with empirical studies showing that allocator choice and configuration significantly affect performance and resource use [3,12], and consequently influence energy consumption [11]. Memory allocators such as the GNU allocator (glibc malloc) [9] and Google's TCMalloc [6] are widely deployed, but tuning their runtime parameters, which allows fine-grained control of their operation, is difficult since optimal settings vary substantially with workload and system behaviour. As a result, production systems often rely on default configurations, which can waste memory, increase energy use, and degrade performance.

The gem5 simulator has become a de facto standard for hardware–software co-design evaluation in academia and industry [1]. gem5 is a large, complex codebase with unusual allocation patterns. Simulations are notoriously slow, often lasting minutes to days, magnifying the impact of allocator efficiency while making manual parameter tuning infeasible. Even modest improvements in heap usage or allocation overhead can shorten runtimes, reduce computational costs, and lower the energy footprint of repeated experiments. Among the many potential optimisation targets, allocator parameters represent a promising but underexplored opportunity to improve runtime and memory consumption in gem5. Yet, allocator optimisation in realistic workloads is challenging: exploration is costly due to long execution times and high memory demands, and results shall be relevant to system behaviour in the wild.

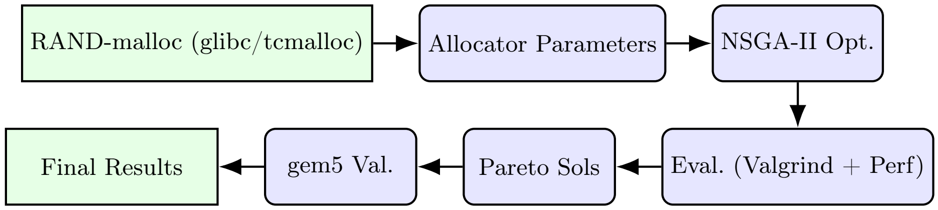

Fig. 1: General GreenMalloc workflow: starting with rand_malloc optimisation to identify efficient allocation parameters, ended by validation on gem5 to assess improvements in memory usage and runtime.

We show it is possible to automatically tune memory allocator parameters to reduce heap usage and energy consumption in industrial workloads. By targeting both memory and runtime, we identify allocator configurations that yield practical improvements across different contexts. We employ a search-based optimisation approach to tune allocator parameters and evaluate their impact on heap usage and runtime in gem5's System Emulation (SE) mode [1]. To avoid the prohibitive cost of optimising directly on full simulations, we use rand_malloc [8] as a lightweight proxy benchmark, enabling efficient exploration of the parameter space before deploying promising configurations in gem5.

Experimental results demonstrate that while peak heap usage remained largely unchanged, glibc malloc tuning yields consistent reductions in average memory (≈4% improvement), a big improvement in free rate (≈2.4× faster release of memory), and a small but measurable reduction in instructions (≈0.25% less). TCMalloc shows more modest improvements in stability and predictability, with small gains in memory free rate and instruction count. These findings suggest that automated allocator tuning significantly reduces computational and energy footprints of long-running industrial simulations keeping reliability and performance. This paper makes the following key contributions:

- A novel search-based optimisation methodology for configuring memory allocators to improve performance and energy efficiency.
- GreenMalloc, a prototype implementation of this approach, designed to be generalisable beyond the allocators studied in this paper.
- A systematic study of allocator tuning using GreenMalloc, applied to two widely used allocators, glibc malloc and TCMalloc, in the context of gem5, a complex and industrially relevant system.

Availability. GreenMalloc and all artifacts are at DOI 10.5281/zenodo.17182847.

2 Methodology

Figure 1 shows the whole system workflow. First, we optimise over a synthetic benchmark rand_malloc [8] to explore the parameter search space efficiently; next, we run the resulting optimal parameters on gem5 simulator to evaluate the transferability to the real system under complex running conditions.

2.1 Heap Memory Allocator Parameters

Both allocators expose several tunable parameters that govern allocation behaviour. For GLIBC MALLOC, parameters including thresholds for switching between `mmap` and `sbrk` allocation, trimming and padding behaviour, and arena limits, all of which were extracted from the GLIBC MALLOC manual [9]. For TCMalloc, there is a range of tunable parameters, such as release rates, thread cache size, page size overrides, and heap limits and were extracted from Google's TCMalloc documentation [5]. These parameters form a high-dimensional, mixed discrete-continuous search space with non-trivial interactions, making manual tuning impractical. A summary of these parameters is in our artifact [7].

2.2 Genetic Algorithm Optimisation with `pymoo`

We employ the genetic algorithm (GA), `NSGA-II`, implemented using `pymoo`, selected for its effectiveness in multi-objective optimisation. Each candidate encodes a *configuration of allocator parameters*, which are passed as environment variables to the allocator implementation (`glibc malloc` or `tcmalloc`). Standard GA operators (mutation, crossover, and elitism) are applied to evolve the population towards better-performing configurations.

We formulate the optimisation as a multi-objective problem, jointly targeting *peak heap usage* and *execution time* to balance sustainability and performance:

- *Green Allocation.* Peak heap usage, measured using `valgrind`'s `massif` tool to minimise the total memory consumed by GEM5 at any given point. Lowering peak memory has two sustainability benefits: it reduces hardware requirements, and enables better workload co-location. Whilst average heap usage may correlate with runtime energy consumption, peak heap determines the minimal system configuration required. Our evaluation measures both metrics.
- *Performance.* Execution time, measured with the `time` utility to balance memory efficiency against performance. This ensures that improvements do not come at the cost of excessively slow executions.

For each generation, `NSGA-II` evaluates candidate parameter vectors by executing RAND_MALLOC and recording the two optimisation objectives. This yields a Pareto front of non-dominated solutions balancing memory usage and runtime. From this front, we select representative candidates: the one minimising runtime, the one minimising memory, and a balanced solution near the centre of the trade-off curve.

2.3 Case Study: Allocator Optimisation for GEM5

Allocator optimisation in realistic workloads is challenging: exploration is expensive, and results must align with system behaviour in the wild. We investigate the potential of memory allocator optimisation through a case study on GEM5, a large and complex system simulator.

Synthetic Benchmarking with RAND_MALLOC. Direct optimisation against the full GEM5 system would be prohibitively expensive, as a single run may take hours or days. We therefore employ RAND_MALLOC [8] as a proxy benchmark to effectively explore the allocator parameter space. RAND_MALLOC provides a synthetic workload, generated from a seed trace of the real system, to exercise the memory allocation behaviour while remaining representative of the memory behaviour observed in GEM5. This can also be used to measure and emulate the workload patterns of other systems, in order to provide a proxy to optimise over.

Evaluation on GEM5. To assess the effectiveness of our approach, we transfer the best-performing parameter configurations to GEM5. We evaluated GEM5's system emulation (SE) mode. For the SE mode, we rely on benchmark programs from previous work on fuzz testing of GEM5 [2,4].

3 Evaluation

We evaluated GreenMalloc on GEM5 using 50 C test programs from SearchSYS's datasets [2,4]. The study analyses GreenMalloc's impact on peak and average heap size, memory release rate, and instruction counts under four configurations:

- **(1) Def `glibc`:** unmodified GLIBC MALLOC (baseline)
- **(2) Opt `glibc`:** GLIBC MALLOC tuned with GreenMalloc
- **(3) Def `tcmalloc`:** unmodified TCMalloc (baseline)
- **(4) Opt `tcmalloc`:** TCMalloc tuned with GreenMalloc

We then ask the following research questions (RQ):

RQ1: *What are the trade-offs between runtime performance and memory efficiency along the Pareto front of configurations discovered by GreenMalloc?*
RQ2: *How effective is GreenMalloc's optimisation in reducing memory consumption and execution time in industrial simulation workloads?*

For each case study, we compiled our generated C test inputs for use in GEM5's SE mode, executed them under the four configurations, and measured memory usage with Valgrind [10] and instruction counts with perf. We use a population size of 24, and run for 500 generations before stopping across all repetitions. We used Valgrind 3.18.1 and perf 5.15.184, running on an Intel Xeon D-1548 (2.0 GHz, 8 cores) with 64 GB RAM, 8 GB swap, and Ubuntu 22.04.5 LTS (x86_64).

Results: RQ1. The hypervolume and Pareto front reveal trade-offs between the two allocators. For glibc, we achieve a mean hypervolume of 9.37×10^{16}. The Pareto fronts average 3 solutions per run, across 0.095% of the instruction space and 0.162% of the peak-heap space relative to the best-performing configuration on the front, with a trade-off slope of -0.216. While TCMalloc achieves a mean hypervolume of 5.46×10^{16}, with smaller Pareto fronts averaging 1.6 solutions per run. While its instruction span is constrained to just 0.005%, it offers greater potential with a 0.259% peak-heap span; however, it also has a much steeper trade-off of -3.17. This demonstrates glibc malloc's default configuration allows for more gradual trade-offs, while TCMalloc operates closer to optimal performance boundaries, constraining the search space but requiring more aggressive trade-offs between objectives. Full RQ1 results are provided in the zip file at [7].

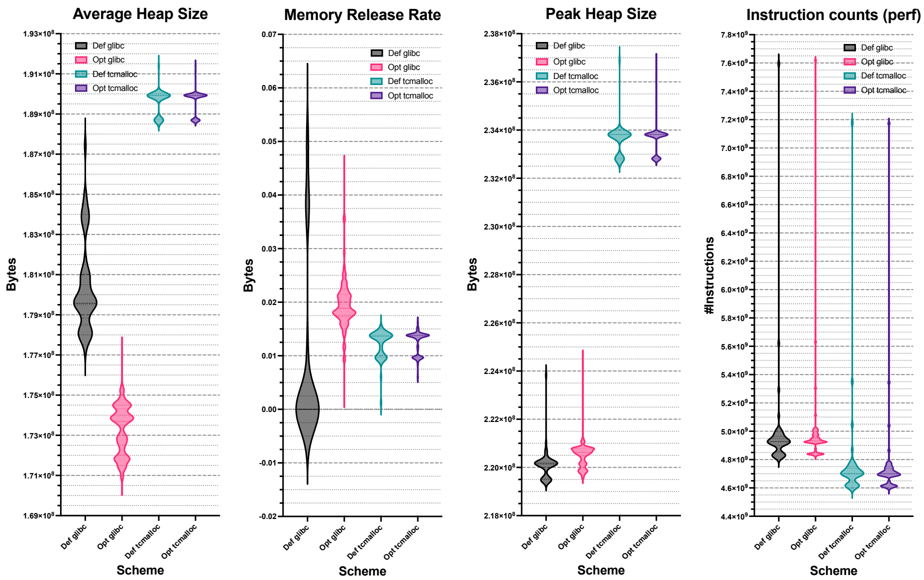

Fig. 2: Comparison of default and GreenMalloc-optimised configurations of glibc malloc (`glibc`) and TCMalloc (`tcmalloc`). From left to right: average heap size, Memory release Rate, peak heap size, and instruction counts, as measured with Valgrind, perf, and gem5. Values are all pareto optimal values.

Results: RQ2. For **average heap usage**, `glibc` shows a clear improvement: Tuning reduced the mean from 180 428 315 to 173 293 862, with a tighter standard deviation (2.2M to 1.6M). This indicates both greener execution – due to a lower memory footprint – and greater performance stability. For `tcmalloc`, the mean remains nearly unchanged, though the density distribution in Figure 2 shows less variance around hotspots, improving predictability. Neither allocator shows reductions in **peak heap usage**. `glibc`'s average slightly increases (220 104 049 to 220 523 190), while `tcmalloc` remains stable. This shows **peak heap usage** is close to its minimum. `glibc` benefits significantly in **memory release rate**: The free rate rises from 0.0080 to 0.0196, reducing retention. `tcmalloc` also improves, with both average release and minimum release increasing. For **instructions executed**, `glibc` improves from 4.992×10^9 to 4.990×10^9, with standard deviation tightened. `tcmalloc` shows a clearer benefit: reducing average instructions from 4.77×10^9 to 4.76×10^9, with variance shrinking as well. These reductions translate into greener execution and improved efficiency.

In best-case runs, we found a single instance of TCMalloc achieved a significant improvement over the unoptimised baseline: specifically, a 4.65% reduction in instruction count as well as a simultaneous 2.06% reduction in peak heap usage. In contrast, the best case run for `glibc` demonstrated a tradeoff between the two optimisation parameters, though it achieved a larger improvement in each metric individually. This suggests that while the transferability from the synthetic benchmark is sufficient, it could benefit from further tuning.

Overall, allocator tuning reduces memory usage and instruction count in `glibc`, while improving consistency in both `glibc` and `tcmalloc`, demonstrating tangible gains in both sustainability and performance. The improvements found by the synthetic benchmark could effectively translate to real workloads. In `glibc`, the tuned parameters reduced average memory and instructions and increased release rate significantly – matching synthetic predictions of greener, more efficient behaviour. In `tcmalloc`, while mean values remained largely stable, the improved consistency of both heap usage and instruction counts mirrors the synthetic outcomes. Full RQ2 results are provided in the zip file at [7].

4 Conclusion

We introduced GreenMalloc, a search-based framework for memory allocator parameter optimisation using lightweight benchmarking. With gem5 we observed reductions in heap usage and instruction counts for both glibc and TCMalloc, highlighting malloc parameter optimisation as a practical approach for efficient and greener systems. The combination of search-based optimisation with lightweight benchmarking opens the door to investigating other aspects of complex software using this strategy, such as gem5's full system (FS) mode, and broader targets including VMs, simulators, emulators, and interpreters.

References

1. Binkert, N., et al.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011). https://doi.org/10.1145/2024716.2024718
2. Dakhama, A., et al.: Enhancing search-based testing with LLMs for finding bugs in system simulators. Automated Software Engineering **32**(2) (2025)
3. Durner, D., et al.: On the impact of memory allocation on high-performance query processing. In: DaMoN 2019. https://doi.org/10.1145/3329785.3329918
4. Even-Mendoza, et al.: Search+LLM-based testing for ARM simulators. In: ICSE-SEIP 2025. pp. 469–480. https://doi.org/10.1109/ICSE-SEIP66354.2025.00047
5. Ghemawat, S.: TCMalloc: Thread-caching malloc (2024), https://gperftools.github.io/gperftools/tcmalloc.html, accessed: Sep. 2025
6. Google: TCMalloc. https://github.com/google/tcmalloc, accessed: Sep. 2025
7. GreenMalloc: This paper's artifact (2025). https://doi.org/10.5281/zenodo.17171047
8. Langdon, W.B.: A genetic improvement parameter benchmark: rand_malloc.c. In: UKCI (2025), https://gpbib.cs.ucl.ac.uk/gp-html/Langdon_2025_UKCI.html
9. Loosemore, S., et al.: The GNU C Library Reference Manual. GNU Project, https://sourceware.org/glibc/manual/latest/pdf/libc.pdf, accessed: Sep. 2025
10. Nethercote, N., et al.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: PLDI 2007. p. 89–100
11. Pereira, R., et al.: Energy efficiency across programming languages: how do energy, time, and memory relate? In: SLE 2017. p. 256–267. ACM
12. Zhou, Z., et al.: Characterizing a memory allocator at warehouse scale. In: ASPLOS 2024. p. 192–206. ACM. https://doi.org/10.1145/3620666.3651350