24th UK Workshop on Computational Intelligence (UKCI 2025) Emma Hart, Sarah L. Thomson, Zhiyuan Tan, Tomas Horvath, Neil Urquhart, Kehinde Babaagba *Eds.*,

A Genetic Improvement Parameter Benchmark: rand_malloc.c

William B. Langdon

Department of Computer Science, University College London, UK
w.langdon@cs.ucl.ac.uk
http://www.cs.ucl.ac.uk/staff/W.Langdon/

Abstract. We describe a new benchmark, `https://github.com/wblangdon/rand_malloc`, for GI via parameter tuning, and demonstrate it using heap performance data collected via our modified Valgrind DHAT from Python/C++ simulation tool gem5 with the GNU gcc compiler's glibc malloc. As well as GI applications, `rand_malloc` might help develop, stress test, tune, and benchmark other dynamic memory managers.

Keywords: automatic programming · software optimisation · data structures · dh_view · M_MMAP_THRESHOLD · evolutionary computing · fitness landscapes · SBSE · Magpie · CMA-ES

1 Why a new benchmark

It is often said that for a subject to develop rapidly it needs well designed benchmarks [31]. For example, genetic algorithms prospered following De Jong's PhD thesis [12] and Ackley's work [1]. This has been further spurred by the Black Box Optimization Benchmarking (BBOB) competition (e.g. [3]). Genetic Improvement [2,15,26] uses search techniques such as genetic programming [28,18] and increasingly Artificial Intelligence [6,14] to improve existing programs, e.g. to increase performance [19], reduce energy consumption [7] or for security [25,23,24].

Genetic Improvement [26] has progressed since the introduction of standard tools such as Gin [30] and MAGPIE [4]. Mostly Genetic Improvement, including Automatic Program Repair [22], has concentrated upon improving existing software by changing its source code [26], however software can sometimes be improved by tuning parameters [21,10]. For example, Magpie can not only operate on multiple programming languages (C/C++, Java, Python, Ruby, etc.) but can also tune parameters. Indeed it can do both simultaneously [27].

gem5 [9] https://www.gem5.org/, is a state of the art discrete time simulator for VLSI digital circuits, such as computer CPUs, GPUs, FPGAs, memory chips and digital caches. It is open source and widely used both in university research and in industry [13]. gem5 is written in a combination of C++ and Python and contains more than a million lines of C++. It is usually compiled with the GNU g++ compiler. As it actively supported and is open source, we have used it for research in automated bug finding [11,13], robustness [17] and genetic

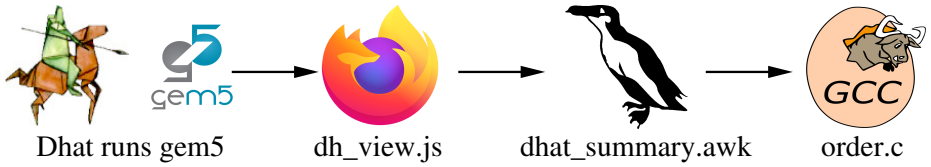

Fig. 1. The target program is run by Valgrind's DHAT to record every malloc operation. (Here slow down caused by DHAT is 160 fold.) DHAT's output is processed by our modified dh_view, in which all significance thresholds, e.g. `sig: 0.005`, are set to zero. This means dh_view reports every use of malloc, not just the frequent ones. The output of dh_view is converted into a C data structure #include file order.c, which is compiled into rand_malloc

improvement [8]. Indeed by tuning GNU heap memory parameters we reduced its heap by 11% [16]. Enthused by this initial success we tried tuning the heap parameters of four more large C/C++ programs: Microsoft's theorem prover Z3 (600 000 lines of C++), the GNU compiler itself, the Clang LLVM C++ compiler and Redis Ltd.'s key-value store (150 000 LOC C++ compiled with the GNU compiler and using the GNU heap). The results were mixed [20], which prompted this similarly mixed more detailed study of the GNU C runtime library. (The GNU C and C++ compiler's dynamic memory use a common run time library, which we will refer to simply as malloc.)

2 rand_malloc.c

rand_malloc is an abstraction of gem5/Python's use of the C++ heap. gem5 was run by Valgrind's DHAT (version 3.16.1) and gem5's creation of dynamic memory blocks, subsequent deletion and life time on the heap were recorded, see Figure 1. DHAT's significance threshold was set to zero, so that rand_malloc's include file, order.c, covers 100% of gem5's use of the C++ heap. For simplicity structures from different classes which have the same average size are combined and, in Table 1 except for the smallest (average 1 bytes) and the largest, only average sizes with more than 10 000 examples are shown. Notice in gem5, the huge range of dynamic memory use, both in terms of the size of data structures (1 byte to 3 MB) and their life times. The shortest lived data item lasts on average only 55 instructions (0% of program duration) before it is deleted. Whilst the longest, lasts 4 248 956 352 instructions (99.03% of program duration). By default rand_malloc applies a linear time rescaling, so that each of its simulation time steps corresponds to 500 instructions. Thus data with a lifetime of 55 instructions are created and then immediately deleted. Whilst the longest lived are deleted about 8.5 million simulation steps after they were created.

Although gem5 is (almost) deterministic, it is simulating events (such as fetching data from memory) which, at a high level, appear to be in a random order and to take a random length of time. Most of gem5's use of the C++ heap is dominated by how it models events. For each event gem5 creates one or more fixed data structures on the heap. These remain on the heap until gem5 has finish processing the event whereupon they are deleted. This random allocation/deallocation would appear to be a reasonable model of many program's

use of dynamic memory. Therefor rand_malloc abstracts the details by assuming memory is allocated from one of a small number of fixed classes at random. Each class has an associated fixed heap block size and either a fixed life time or a random life time of a given average length (see Table 1).

Like gem5, rand_malloc is a discrete time simulation. Since all it does is call malloc and free, time is also simplified, so that each time step a new randomly chosen item is added to the heap by calling malloc and the time when it will be removed is recorded. After the block is added, if any blocks have reached their time limit rand_malloc calls free for each of them. For simplicity each allocation is modeled as one instruction. This is probably unrealistic and may lead to more overlap of short lifetime blocks than actually happens in gem5. At the end of the simulation, rand_malloc continues calling free in time order until all the blocks it has added to the heap are removed.

3 Results

The GNU C runtime library, including malloc, has been under development for several decades. We will concentrate upon two versions GLIBC 2.17 and GLIBC 2.34 which differ in their level of support for 64 bit pointers, see Figure 2. Notice in Figure 2 the one dimensional slice of the malloc parameter fitness is remarkably flat but shows some scope for improvement. For example with glibc 2.17 a saving of 188 KB is possible. Indeed this is confirmed by ten runs of Magpie and ten runs of CMA-ES on the whole fitness space, i.e., all three relevant malloc parameters (whereas Figure 2 shows a projection of the whole search space). Magpie local search and CMA-ES were run with their default parameters, except they were both allowed up to 1000 fitness evaluations. Details are given in [16]. Although improvements seen here are small, rand_malloc might be a useful testing and benchmarking tool for C, C++ and languages, such as Python, which interface to C.

4 Summary

Although software has been optimised by Genetic Improvement parameter tuning e.g. [21], rand_malloc is the first purposed designed benchmark for GI parameter tuning [5]. Use of Valgrind's DHAT allows integration of real heap memory statistics. Here although rand_malloc approximates gem5's use of dynamic memory, we did not capture the more than 10% heap savings that both Magpie and CMA-ES achieved when tuning gem5 directly [16]. Indeed results are more like other large C++ programs [20]. We hope the availability of a simple GI memory management benchmark will simulate further progress and so have made rand_malloc available via https://github.com/wblangdon/rand_malloc. Also it may be useful for both functional and performance testing of existing and future compilers and their dynamic memory managers.

Acknowledgment My thanks to Nicholas Nethercote and the referees.

Table 1. Summary of the size of gem5 dynamic data structures. Last four columns relate to the number of rand_malloc simulation steps before the structure is deleted.

			life time		
Average size (bytes)	number	fraction	mean	std	min–max
min 1	9379	0.1%	2789287	3827145	1–8496280
6	34819	0.5%	72004	763265	13–8496440
7	12043	0.2%	280036	1476808	13–8496498
8	138560	2.0%	108025	898063	0–8496436
10	24889	0.4%	2558509	325107	3–8496442
16	232591	3.3%	74028	666639	0–8496667
27	204866	2.9%	3559	170690	0–8496442
32	736320	10.5%	72413	747645	0–8496603
38	18107	0.3%	4697406	879031	0–8496438
48	304795	4.4%	694097	1790222	0–8496665
64	283653	4.0%	82438	613085	0–8492004
65	239296	3.4%	5647	212188	0–8492599
72	205862	2.9%	15093	324314	0–8496278
96	205610	2.9%	21837	420900	1–8496028
120	58384	0.8%	2163	130496	0–8080120
168	224602	3.2%	36052	375617	10–8492689
272	230124	3.3%	2901	113898	1–8496082
512	208190	3.0%	1996	91643	1–8496533
max 3145728	1	0.0%	8487315	0	–
others	129986	1.9%	2320468	3318545	0–8497914

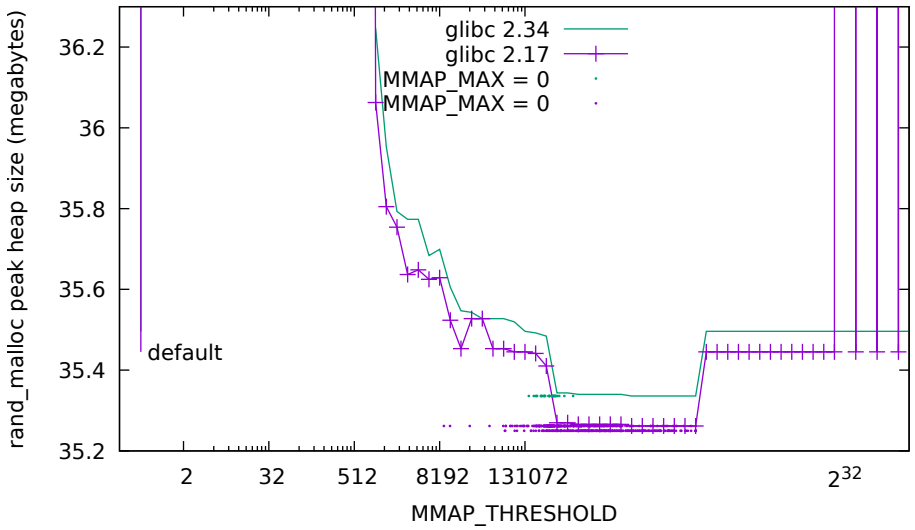

Fig. 2. Lines show variation of maximum heap used by rand_malloc for two implementations of glibc with one of malloc's tuning parameters (MMAP_THRESHOLD, the other 6 are left at their default values). + data above 800 sampled at powers of $\sqrt{2}$. Dots show CMA-ES sometimes found marginal improvement by zeroing one of the other parameters. Notice the older glibc version does not deal correctly with this parameter above 2^{32}. Non-linear horizontal scale.

References

1. Ackley, D.H.: An empirical study of bit vector function optimization. In: Genetic Algorithms and Simulated Annealing, chap. 13, pp. 170–204. Pittman (1987)
2. Arcuri, A., White, D.R., Clark, J., Yao, X.: Multi-objective improvement of software using co-evolution and smart seeding. In: SEAL 2008. LNCS, vol. 5361, pp. 61–70. Springer, Melbourne, Australia (2008), http://dx.doi.org/10.1007/978-3-540-89694-4_7
3. Auger, A.: Benchmarking the (1+1) evolution strategy with one-fifth success rule on the bbob-2009 function testbed. In: GECCO 2009. pp. 2447–2452. ACM, Montreal, Canada (2009). https://doi.org/http://dx.doi.org/10.1145/1570256.1570342
4. Blot, A., Petke, J.: MAGPIE: Machine automated general performance improvement via evolution of software. arXiv (2022), http://dx.doi.org/10.48550/arxiv.2208.02811
5. Blot, A., Petke, J.: A comprehensive survey of benchmarks for improvement of software's non-functional properties. ACM Computing Surveys **57**(7), Article no. 168 (2025), http://dx.doi.org/10.1145/3711119
6. Bouras, D.S., Petke, J., Mechtaev, S.: LLM-assisted crossover in genetic improvement of software. In: GI@ICSE 2025. pp. 19–26. Ottawa (2025), http://dx.doi.org/10.1109/GI66624.2025.00012, best presentation
7. Bruce, B.R.: The Blind Software Engineer: Improving the Non-Functional Properties of Software by Means of Genetic Improvement. Ph.D. thesis, Computer Science, University College, London, UK (2018), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/bruce_bobby_r_thesis.pdf
8. Bruce, B.R.: Automatically exploring computer system design spaces. In: GI@GECCO 2022. pp. 1926–1927. Association for Computing Machinery, Boston, USA (2022), http://dx.doi.org/10.1145/3520304.3534021
9. Bruce, B.R., Akram, A., Nguyen, H., Roarty, K., Samani, M., Fariborz, M., Reddy, T., Sinclair, M.D., Lowe-Power, J.: Enabling reproducible and agile full-system simulation. In: ISPASS 2021. pp. 183–193. , Stony Brook, NY, USA (2021), http://dx.doi.org/10.1109/ISPASS51385.2021.00035
10. Chi Ho Chan, Nita, S.: A three-stage genetic algorithm for compiler flag and library version selection to minimize execution time. In: GI@ICSE 2025. pp. 1–2. Ottawa (2025), http://dx.doi.org/10.1109/GI66624.2025.00009
11. Dakhama, A., Even-Mendoza, K., Langdon, W., Menendez, H.D., Petke, J.: Enhancing search-based testing with LLMs for finding bugs in system simulators. Automated Software Engineering p. Article:63 (2025), http://dx.doi.org/10.1007/s10515-025-00531-7
12. De Jong, K.A.: An Analysis of the Behavior of a Class of Genetic Adaptive Systems. Ph.D. thesis, Computer and Communications Sciences, University of Michigan, USA (1975)
13. Even-Mendoza, K., Menendez, H.D., Langdon, W., Dakhama, A., Petke, J., Bruce, B.R.: Search+LLM-based testing for ARM simulators. In: ICSE 2025 (SEIP). pp. 469–480. Ottawa, Canada (2025), https://solar.cs.ucl.ac.uk/pdf/ICSE_SEIP_2025___SearchSYS_and_ARM.pdf
14. Jingyuan Wang, Hanna, C., Petke, J.: Large language model based code completion is an effective genetic improvement mutation. In: GI@ICSE 2025. pp. 11–18. Ottawa (2025), http://dx.doi.org/10.1109/GI66624.2025.00011, best paper
15. Langdon, W.B.: Genetic improvement of programs. In: MENDEL 2012. Brno University of Technology, Brno, Czech Republic (2012), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2012_mendel.pdf, invited keynote

16. Langdon, W.B., Bruce, B.R.: The gem5 C++ glibc heap fitness landscape. In: GI@ICSE 2025. pp. 3–10. Ottawa (2025), http://dx.doi.org/10.1109/GI66624.2025.00010
17. Langdon, W.B., Clark, D.: Deep mutations have little impact. In: GI@ICSE 2024. pp. 1–8. ACM, Lisbon (2024), http://dx.doi.org/10.1145/3643692.3648259, best paper
18. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In: EuroGP 2014. LNCS, vol. 8599, pp. 87–99. Springer, Granada, Spain (2014), http://dx.doi.org/10.1007/978-3-662-44303-3_8
19. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Transactions on Evolutionary Computation **19**(1), 118–135 (2015), http://dx.doi.org/10.1109/TEVC.2013.2281544
20. Langdon, W.B., Petke, J., Clark, D.: gem5/z3/gcc/clang/redis glibc heap fitness landscapes. In: Evostar 2025 Late breaking abstracts. Trieste (2025), http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/Langdon_2025_evostarLBA.pdf
21. Langdon, W.B., Petke, J., Lorenz, R.: Evolving better RNAfold structure prediction. In: EuroGP 2018. LNCS, vol. 10781, pp. 220–236. Springer Verlag, Parma, Italy (2018), http://dx.doi.org/10.1007/978-3-319-77553-1_14
22. Le Goues, C., Pradel, M., Roychoudhury, A.: Automated program repair. Communications of the ACM **62**(12), 56–65 (2019), http://dx.doi.org/10.1145/3318162
23. Leach, K., Dougherty, R., Spensky, C., Forrest, S., Weimer, W.: Evolutionary computation for improving malware analysis. In: GI@ICSE 2019. pp. 18–19. IEEE, Montreal (2019), http://dx.doi.org/10.1109/GI.2019.00013, best presentation
24. Mesecan, I., Blackwell, D., Clark, D., Cohen, M.B., Petke, J.: Keeping secrets: Multi-objective genetic improvement for detecting and reducing information leakage. In: ASE 2022. p. Article no. 61. Michigan, USA (2022), http://dx.doi.org/10.1145/3551349.3556947
25. Petke, J.: Genetic improvement for code obfuscation. In: GI-2016. pp. 1135–1136. ACM, Denver (2016), http://dx.doi.org/10.1145/2908961.2931689
26. Petke, J., Haraldsson, S.O., Harman, M., Langdon, W.B., White, D.R., Woodward, J.R.: Genetic improvement of software: a comprehensive survey. IEEE Transactions on Evolutionary Computation **22**(3), 415–432 (2018), http://dx.doi.org/doi:10.1109/TEVC.2017.2693219
27. Songpetchmongkol, T., Blot, A., Petke, J.: Empirical comparison of runtime improvement approaches: Genetic improvement, parameter tuning, and their combination. In: GI@ICSE 2025. pp. 35–42. Ottawa (2025), http://dx.doi.org/10.1109/GI66624.2025.00014
28. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ICSE 2009. pp. 364–374. Vancouver (2009), http://dx.doi.org/10.1109/ICSE.2009.5070536, winner ACM SIGSOFT Distinguished Paper Award. Gold medal at 2009 HUMIES. Ten-Year Most Influential Paper [29]
29. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: It does what you say, not what you mean: Lessons from a decade of program repair. ICSE 2019 (2019), https://conf.researchr.org/profile/icpc-2019/westleyweimer
30. White, D.R.: GI in no time. In: GI-2017. pp. 1549–1550. ACM, Berlin (2017), http://dx.doi.org/doi:10.1145/3067695.3082515
31. White, D.R., et al.: Better GP benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines **14**(1), 3–29 (2013), http://dx.doi.org/10.1007/s10710-012-9177-2