Measuring performance overheads of software memory management using functional-first simulators

Yves Vandriessche*, Wim Heirman*, Ed Nutting†, Jeremy Birch†, Judah Daniels†, Mae Hood† and Pascal Costanza*
*Intel Corporation
†VyperCore

I. Abstract

Measuring the overhead of memory management in software applications is a hard problem due to the lack of a baseline; it cannot be turned off, only replaced with a different strategy with different tradeoffs. We present a straightforward technique to approximate a baseline that relies on function-first simulation and demonstrate its application with a Python memory management overhead analysis.

II. Memory Management overhead analysis

The performance impact of userspace software memory management is an ongoing concern and feeds a rich research field around manual allocators [1], and automatic memory management such as reference counting [2], [3] and Garbage Collection (GC) [4]. These works rely on comparative studies, as even estimating the absolute cost of software memory management techniques is famously difficult [5]. Understanding the true cost of the direct, and indeed indirect, impact of memory management techniques is crucial to correctly focus the development of software and hardware improvements to the areas that would make the greatest impact.

Memory managers introduce easy to measure overheads in places where the application hands over control. For example, a call to libc's `malloc()` or `free()`, or the language runtime pausing the application for a stop-the-world GC memory reclamation. However, memory management strategies also introduce "death by $1,000$ cuts" overheads in the form of meta-data and bookkeeping across the entire application's execution. These types of operations have an first-order effect of extra instructions and increased memory footprint, but also introduce second order effects such as cache and branch predictor pollution. The first order effects of these "death by 1,000 cuts" overheads are already challenging to directly measure in sampling based measurement approaches (Intel EMON, linux perf), but can be directly measured in simulation if all such bookkeeping regions are appropriately identified and marked. The second-order effects could be estimated by omitting these bookkeeping operations. However, excluding the bookkeeping operations is not possible in functional simulation or native execution without changing the program functionality and correctness.

III. Approach

Our approach relies on functional-first simulators separating the functional and timing concerns into two distinct simulation models. Given an application annotated with memory management region markers, the functional model executes the full application as usual, but the instructions in marked regions are excluded from the timing model. A benefit of this straightforward approach is that it does not require functional changes to the runtime or application code.

Collecting the metrics of instructions inside the marked regions measures only the first-order effects of memory management. To capture the second-order effects as well, we perform two simulations: with and without marked region exclusion. Taking the difference yields a lower bound measurement of the memory management overhead for a variety of performance metrics.

A. Limitations

We do not claim that this simulation-exclusion approach is the definitive accurate measurement of memory management overhead. Rather, this is a useful tool in the toolbox to estimating memory management overheads. Directly measuring memory management overheads is intractably hard due to the absence of a clear zero-cost baseline. Memory allocation strategies constantly trade off higher management overhead against overall application performance gains, this applies to both manual and automatic memory management strategies [6].

This simple approach does not modify the runtime or its data layout. As such, the objects will still reside in memory locations determined by the original memory manager and the object representation will still include the reserved space for metadata, although accesses to it will not be observed by the performance model. For example, a reference counted object will still contain the counter slot, only its access instructions are excluded from the timing model's instruction stream. What this approach measures is therefore still a lower bound estimate.

IV. Demonstrator

A. Setup

We demonstrate the feasibility of our approach by applying it to a set of Python benchmarks from the pyperformance suite. The following results are only preliminary, a deeper and broader analysis will be the topic of a future publication.

Our functional-first simulation setup uses an in-house simulator derived from Sniper [7] with a 4th-gen Intel Xeon Scalable Processor (Sapphire Rapids) timing model. No scale-down simulation is used here, meaning that the single-threaded python benchmarks can make full exclusive use of the 105MB LLC.

The functional model is the Pin-based Intel SDE emulator. System calls are not emulated, however no such calls were recorded in the benchmark's region of interest.

The region exclusion feature is implemented in a straightforward way, filtering out instructions between region-exclusion markers and taking into account region nesting. Python operates a variety of allocation and reclamation strategies with various bookkeeping functionality spread throughout the interpreter code. The markers allows all of these to be taken into account. Around 100 regions were identified and marked in the CPython v3.12 interpreter, encompassing reference counting operations, tracing garbage collection, zero-initialization, arena allocation and calls to the underlying libc allocator.

Four benchmarks with distinct workloads are selected from the pyperformance suite: *deepcopy*, *chaos*, *django template* and *comprehensions*. They are executed unmodified with a custom runner script that injects the simulator's region of interest simulation markers for cache warmup and the detailed simulation phases. This excludes Python startup, module importing, etc…from the performance simulation, letting it run the benchmark workload in a fully warmed up state.

B. Preliminary results

Generally, these early results on a handful of benchmarks are not sufficient to make broad estimates about the memory management overheads in Python. Nevertheless, the trend across the analysed workloads points to a clear reduction in total runtime.

Across all analysed workloads, the instruction count drops significantly when excluding the memory management regions, with the differences ranging from -15.8% to -20.9% with a geometric mean of -20.2%. This reduction is in line with expectations in earlier functional-only emulation experiments. The timing model reports an equivalent reduction in runtime of geomean -19.2%. Depending on the workload, the change in instructions per cycle (IPC) swings between -2.6% and $+1.7\%$. Looking closer to the internal IPC bottleneck metrics, we do observe a small shift from cycles spent waiting for data caches (-1.6%) to waiting for dependent instructions $+1.2\%$.

It is interesting to note that the change in most performance metrics varies wildly across the different workloads. For instance, the *deepcopy* benchmark sees a significant -80.9% geomean reduction in L2 and last-level cache miss rates, whereas the *comprehension* and *django template* benchmarks see an increase of respectively 18.1% and 23.1%. As the excluded instructions are a common part of the runtime, removing them from execution results in the inherent workload performance profile to be magnified. The region-exclusion simulation helped reveal the workload's baseline performance profile, an essential tool in the development of memory management improvements.

References

[1] D. Leijen, B. Zorn, and L. de Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems*, A. W. Lin, Ed. Cham: Springer International Publishing, 2019, pp. 244–265.

[2] R. Shahriyar, S. M. Blackburn, X. Yang, and K. S. McKinley, "Taking off the gloves with reference counting immix," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 93–110. [Online]. Available: https://doi.org/10.1145/2509136.2509527

[3] S. Gross, "Making the global interpreter lock optional in cpython," PEP 703, 2023. [Online]. Available: https://peps.python.org/pep-0703/#reference-counting

[4] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2023.

[5] Z. Cai, S. M. Blackburn, M. D. Bond, and M. Maas, "Distilling the real cost of production garbage collectors," in *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2022, pp. 46–57.

[6] D. Grunwald, B. Zorn, and R. Henderson, "Improving the cache locality of memory allocation," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 177–186. [Online]. Available: https://doi.org/10.1145/155090.155107

[7] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, Aug. 2014. [Online]. Available: https://doi.org/10.1145/2629677