A Low-Latency Garbage Collector for GHC

Ben Gamari
Well-Typed LLP
London, U.K.
ben@well-typed.com

Laura Dietz
University of New Hampshire
Durham, NH, U.S.A.
dietz@cs.unh.edu

Abstract

GHC 8.10.1 offers a new latency-oriented garbage collector to complement the existing throughput-oriented copying collector. This demonstration discusses the pros and cons of the latency-optimized GC design, briefly discusses the technical trade-offs made by the design, and describes the sorts of application for which the collector is suitable. We include a brief quantitative evaluation on a typical large-heap server workload.

CCS Concepts: • **Software and its engineering → General programming languages**.

Keywords: garbage collection implementations

1 Introduction

The past several years have seen a significant increase in Haskell adoption in industry, particularly in server-side applications as components of larger distributed systems.

In such systems, the worst-case response time is dictated by the worst-case response time of any constituent system component. Consequently developers of distributed systems must maintain tight control over the tail of each component's response time distribution to maintain overall system performance.

Stop-the-world garbage collection represents a major source of uncontrolled pauses in many server applications. In particular, GHC's moving garbage collector incurs a maximum pause time proportional to the amount of live heap data. With even moderate-sized heaps, these pauses can span many hundreds of milliseconds.

As a result, authors of Haskell server applications must resort to the use of off-heap data structures, compact normal forms, multiple server processes, and other techniques to avoid unacceptably-long pause times.

In GHC 8.10.1 we introduced[1] a new garbage collection scheme targetting latency-sensitive applications with moderate-to-large working-sets.

2 Approach

We use a non-moving collection scheme inspired by Ueno, *et al.* [3] to manage old objects, while relying on a bounded-size moving nursery for new and young objects. Collection of the non-moving heap is achieved via concurrent mark & sweep.

Our collector relies on a snapshot-at-the-beginning collection strategy, supported by a variant of the heap structure described by Ueno, *et al.* [4]. The collector preserves fast allocation and avoids the need for recompilation by relying on the existing moving collector and nursery for mutator allocation. The collector is generational, with one or more moving young generations, which are evacuated into a single non-moving old generation.

The concurrent non-moving collector imposes few additional write barrier obligations on the mutator, taking advantage of the fact that the most mutations in Haskell programs are due to thunk updates. Consequently, nearly no changes are necessary in generated mutator code.

Note that unlike the earlier work of Ueno, *et al.* [3], our collector is not a fully-concurrent design: all mutator threads must be stopped for at the beginning and end of every major collection. However, as most marking work is done concurrently, these pauses are typically short (on the order of milliseconds). Moreover, full-concurrency would offer relatively few advantages without also addressing the problem of stop-the-world minor collections (e.g. Marlow, *et al.* [2]).

3 Usage

The non-moving garbage collector can be selected by enabling the `+RTS --nonmoving-gc` runtime system flag. For concurrent collection the only requirement is that the program is linked against GHC's threaded runtime system; no further recompilation is necessary. Diagnostic output characterising heap fragmentation, collection lifecycle, and timings can be enabled via the `+RTS -s` flag and the usual GHC eventlog.

4 Should You Use This Collector?

Garbage collector design is a field fraught with trade-offs. In this demonstration, we provide some insight into the trade-offs made by our collector and their implications on users of GHC. In short, the new collector is right for applications that:

- struggle with long pause times: the new collector is a concurrent collector, requiring only two short pauses at the beginning and end of the collector.
- have large heap: under the moving collector, applications with very large heaps (e.g. tens of gigabytes) can

[1] More details on the design, implementation, and performance characteristics of the collector can be found in our ISMM 2020 paper [1]

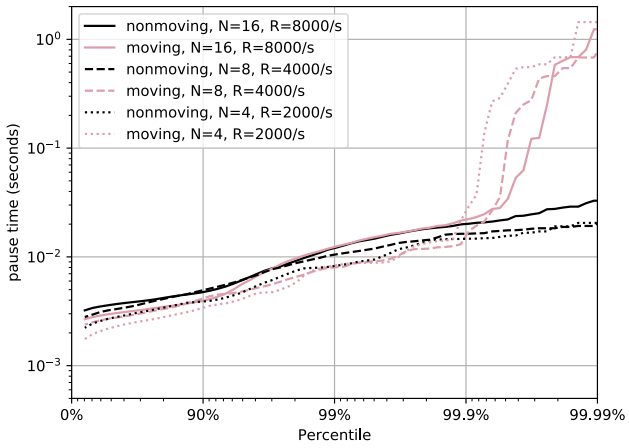

Figure 2. A latency histogram showing the GC pause time distribution (including both major and minor collection pauses) from the same application shown in Figure 1.

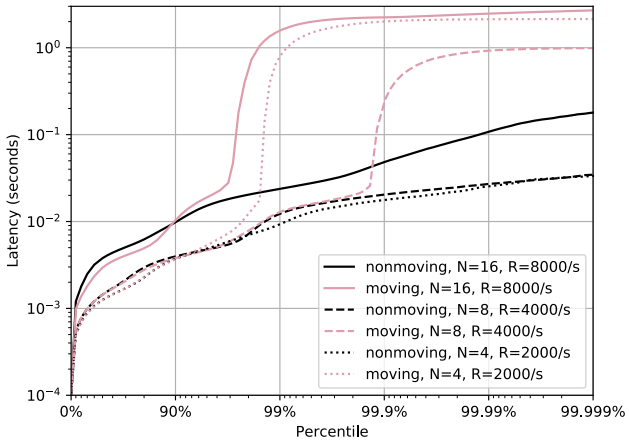

Figure 1. A latency histogram showing end-to-end response time of a simple server application emulating a large caching HTTP server. Four different server and load conditions are shown: N denotes the server's core count where R refers to the request rate as provided by the wrk2 load generator run on eight cores.

produce pauses of many seconds. This is particularly relevant for long-lived server applications with large working sets.

- prefer low-latency over high throughput: in particular, interactive applications and distributed systems

By contrast, applications with the following properties would likely be better served by the moving collector:

- high-throughput demands: batch data processing, compilation
- small heap sizes: these cases will see very little benefit from the concurrent collection provided by the non-moving collector.

The moving collector is quite well-suited for throughput-oriented applications. Its compacting nature provides excellent data locality for the mutator while providing efficient collection. Its Achilles heel is the requirement to pause the mutator for the entirety of the copying process.

The new collector comes at the price of slower allocation and loss of compaction, leading to a reduction in mutator performance. However, it brings the advantage of being able to perform the majority of marking work concurrently with mutator execution.

In addition, the non-moving collector incurs a small additional cost on the mutator while marking is underway in the form of a write barrier.

5 Evaluation

Figures 2 and 1 display pause-time and response-time statistics for a set of measurements on a model client/server application with 20 gigabyte heap.

6 Demonstration

In our demonstration we will discuss the motivations for our design, describe the sorts of applications for which our collector is especially suitable, describe the performance characteristics users can expect from these applications and show a few examples of the such applications in action, comparing with GHC's existing copying collector.

In addition, we will describe a few opportunities for future improvement and how these improvements might further grow the collector's domain of applicability.

Acknowledgments

We would like to thank Ömer Sinan Ağacan, whose code, skillful debugging, and helpful discussions during the implementation part of the project were invaluable in bringing it into its current state.

We would also like to acknowledge the contributions of Pepe Iborra and Atze Djikstra for their assistance in testing and characterising the collector.

References

[1] Ben Gamari and Laura Dietz. 2020. Alligator collector: a latency-optimized garbage collector for functional programming language. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*.

[2] Simon Marlow and Simon Peyton Jones. 2011. Multicore garbage collection with local heaps. In *Proceedings of the 10th International Symposium on Memory Management*. ACM New York, NY, USA.

[3] Katsuhiro Ueno and Atsushi Ohori. 2016. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 421–433.

[4] Katsuhiro Ueno, Atsushi Ohori, and Toshiaki Otomo. 2011. An efficient non-moving garbage collector for functional languages. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ACM New York, NY, USA, 196–208.