Optimizing Pypy's Garbage Collector for Copy On Write Performance given a forky Web-Server Workload

Andrew Nelson
CSC 550

February 28, 2018

Abstract

Most research into garbage collection is focused on analyzing a single process in isolation. This paper focuses on the garbage collection of many processes with shared memory. We examine a deployment of a Django webserver (written in python) running in multi process mode and profile its memory usage with various garbage collection algorithms. We then modify pypy's generational garbage collector to improve its use of copy-on-write memory. Specifically, we make references between objects into indirect references and store those in densely packed reference tables. That way when an object is relocated from a younger into an older generation, only the reference table needs to be updated to reflect the object's new location. This prevents the garbage collector from unnecessarily writing to shared memory.

Contents

1 Introduction

Advances in interpretors for high level languages have enabled the use of high level languages in performance critical applications. Some of the most visited websites on the Internet as of 2018 are written in high level languages such as Python, Ruby, and PHP. A common problem with these deployments is memory usage: garbage collected languages tend to use more memory than their manually-memory-managed counterparts. Most research into garbage collection focuses on the process in isolation. This paper, however, examines garbage collection in a group of processes which share memory in a Copy-On-Write fashion.

1. We are looking at web servers. Web servers fork a lot.
2. Forked processes share CoW memory... garbage collection is messy
3. We change garbage collector to optimize CoW by making mature pointers all indirect so mature objects don't get touched during minor collection

2 Background

This section provides some background information that may help to contextualize the methods used in this paper. Section 2.1 is a quick review of the Linux memory model as well as an explanation of Copy-on-Write semantics. Sections 2.2 and 2.3 are a quick review of the Reference-Counting and Mark-and-Sweep algorithms used together in CPython's Garbage Collector. Section 2.4 is an overview of the Compacting Generational Garbage Collection algorithm used in PyPy and racket. Finally, section 2.5 is an overview of how these algorithms can work against each other to increase memory used by a collection of related processes. Those familiar with the algorithms covered in sections 2.1, 2.3, and 2.4 may wish to skip over those sections. Section 2.5 is more than simple review and thus should not be skipped.

2.1 Copy-on-Write (CoW) semantics

When a process in Linux forks, a new process is spawned with an identical address space. In order to save memory, the Linux kernel does not copy the entire address space, but rather it copies the page table of the parent process and marks all physical pages as shared and read-only [8]. If the physical page was already marked shared, the reference count of the page is incremented. When either the parent or child process tries to read from a location in memory, the virtual address referenced by the process gets translated to a physical address using the page table for that process. This happens transparently from the perspective of the process [8]. Since both processes can share the same physical pages in memory, the combined memory consumption of these two processes is less than the sum of the individual memory consumptions.

When either the parent or the child process attempt to write to a shared page in memory, a minor page fault occurs. Control is shifted to the kernel which determines how to proceed. If the physical page is still being shared, the kernel allocates a new physical page, copies the contents of the shared page into the new physical page, decrements the reference count in the shared page, and inserts the new physical page into the faulted process's page table [8]. If the physical page is no longer being shared (for example if the child process already wrote to the physical page in question and made their own copy) then the page is converted from a read-only shared page to a read-write private page [8]. In this way, memory is allocated and the process's page-table is updated in an extremely lazy manor which has proven to be very efficient in practice.

In the event that a process forks multiple times, the same mechanism still works. A single address space can be shared among N processes. Since each page has an unsigned int dedicated to reference count, quite a few pages can refer to a single page and copying will happen as needed [7, 8].

2.2 Reference-Counting Garbage Collection (RCGC)

Under a Reference-Counting Garbage Collection regime, every object has an internal reference count attribute. When new references to this object are created, the reference count attribute is incremented. When references to this object are updated or when objects which refer to this object are collected, the reference count attribute is decremented. When the reference count of an attribute reaches zero, it is garbage collected and all the objects to which it refers are visited accordingly [4].

This garbage collection algorithm can be visualized as a directed graph where every node is an object in memory and where every edge is a reference from one object to another. The reference count for each object is the the in-degree of its associated node. When there are no edges into a node, the algorithm knows that that node (and all its out-edges) can be collected. This scheme has advantages in that many objects can be collected as soon as they become unreachable. However, if a group of unreachable objects form a reference cycle, their reference counts will never reach zero and so they will never be collected by a strictly reference-counted system.

This algorithm has been used extensively in CPython since python 2 and accounts for the majority of object collection in most running systems [6]. Whenever a reference is created, updated, or deleted, CPython visits the objects being referenced and updates their reference counts appropriately. To account for cycles in the object graph, CPython does occasional collections using the Mark-And-Sweep algorithm.

2.3 Mark-And-Sweep Garbage Collection (MSGC)

Mark-and-Sweep Garbage Collection can be modeled as a traversal of the object graph starting with a set of ”accessible roots”. These accessible roots are objects that are being directly referenced by the program (e.g. globals, variables on the stack, intermediate values of a computation, etc.). Starting with these nodes, the algorithm recursively visits adjacent nodes marking them as ”accessible” when they are visited. At the end of this first sweep, the algorithm has divided all objects into two sets: those which are accessible to the program (nodes tags as ”accessible”) and those which are not accessible to the program (nodes not tagged as ”accessible”). To do the collection, the algorithm makes a pass over every object in the system. Those not tagged as ”accessible” are deleted in place [4].

A common problem with this algorithm is that objects are deleted in-place and so the physical memory where objects reside tends to become fragmented over time. This increases the number of physical pages in use and increases the cache miss rate.

This algorithm has been used in conjunction with Reference Counting in CPython since python 2. While the reference counter is always running in the background during CPython execution, CPython will initiate a MSGC (usually called a ”major collection event”) collection once memory usage exceeds a certain threshold. In combination, these algorithms eliminate most garbage in a timely manor while also correctly collecting cycles that form on occasion [6].

2.4 Compacting Generational Garbage Collection (CGGC)

In Compacting Generational Garbage Collection, objects are divided into multiple ”generations” which are each located on separate physical pages of memory. In many cases there are 3 or 4 successively larger generations. New objects are inserted into generation 0 (often called the ”nursery”). Once generation 0 reaches a certain size, the objects in generation 0 are traversed starting with the accessible roots. Objects in generation 0 which are still accessible are copied to generation 1. When all the accessible objects in generation 0 have been relocated, generation 0 is cleared [4].

When generation 1 gets larger than some threshold, it is collected in the same way and accessible objects are copied to generation 2. Indeed when generation N gets large enough, its contents are traversed, accessible objects are copied to generation $N + 1$, and generation N is cleared.

In most implementations there is a final generation (often called the ”mature” generation) for objects that have survived all the prior collections. There are several ways to handle this final generation - the most common being to let the garbage build up. This implementation is not technically correct but it performs well in practice. In PyPy, the mature generation is collected using the MSGC. Memory fragmentation which may occur in PyPy's mature generation are possible filled when objects are copied from younger generations into mature [14].

> ”Many generational GC's are not comprehensive–they don't successfully remove all the garbage (long-lived garbage, in particular, may never get collected). But it's the fresh garbage that smells the worst. :)”
>
> – C2 Wiki[1]

2.4.1 Inter-Generational References: Always Use Protection!

The CGGC algorithm as described above is incomplete. It's possible for objects in one generation to refer to objects in another generation and doing a naive ”accessible-roots” seeded traversal will not see these references which may cause some objects to get collected prematurely. To prevent this we introduce a protective write barrier.

Every generation keeps a list of cross-generational references into that generation. When an object in generation M is updated to reference an object in generation N, that object is added to generation N's cross-reference list. When generation N is collected, the garbage collector adds references in the cross-reference list to the root accessible set and those references are used when tracing which objects are in use. When an object in generation M is copied to generation $M+1$, all the updates to that object must be updated to refer to this new location in memory [4].

2.5 Tragedy of the commons: Dirtying Shared Pages in CGGC

Each of these algorithms alone serve to minimize the memory used by an individual process as it runs. CoW allows related processes to share physical memory, and all three garbage collection algorithms reduce the number of physical pages needed by each process. However, when used in conjunction we often see that the memory performance of a collection of related processes suffers.

As a thought experiment, consider a forking webserver written in python running atop a standard CPython interpretor. When a request comes in, the process forks to spawn a worker thread which begins processing the request. The parent process has much in memory the child process can reference during its computation.

As the child process continues, it will surely create or update a reference to some existing constant data structure. When this happens, the reference count field in the referenced object must be updated. This triggers a copy-on-write because even though none of of the objects stored on the page changed, the meta-data that the garbage collector stores on the page has changed. In this way, reference counting breaks shared pages which can increase memory used by the system.

When a CPython major collection event occurs, the MSGC algorithm will start a full heap traversal that marks objects as "accessible" or "not-accessible". Because the garbage collector stores this information adjacent to the objects in memory, the shared pages containing these objects get written to which triggers a copy-on-write and the page sharing breaks. In this way, a single major garbage collection event can greatly increase memory used by the system.

This problem is somewhat alleviated when using a generational garbage collector such as is included in PyPy because we can copy objects out of a shared page without writing to it, however when we do the copy, those inter-generational references must be updated. Doing a minor collection of generation 1 may force writes to generations 2, 3, 4, etc, which breaks the sharing of those pages.

While these algorithms are successful at reducing the memory usage of an individual process, when examining a collection or related processes they do poorly at reducing total combined memory usage. The next section is dedicated to modifying CGGC to solve this problem.

3 Design

The default garbage collector in PyPy is called "minimark" and it is a classic Generational Garbage Collector with 2 generations [14]. New objects are created in the nursery and are moved to gen1 on minor collection events.

1. Currently when mature objects refer to old objects, that reference is stored in a special "write barrier" so their references are counted
2. When young object moves from young to mature generation, the objects which refer to it get updated in-place
3. We are going to put all references between objects in a reference table... every lookup will be a double lookup but every update will be a single edit to the table

4 Implementation

1. Give a summary of which structures and functions actually changed
2. Where did I store this information? How is it accessed in code?
3. maybe some pseudocode snippets

5 Methodologies

1. Ngnx + uWSGI + Django + pypy in multi-process mode
2. Use thebluealliance.com as a test load becaues it caches a lot of data and does some non-trivial computation
3. Generate a bunch of fake traffic using httperf
4. Monitor running code
   (a) How long did it take to handle 10,000 requests
   (b) What was the average time-to-last-byte?

(c) Total memory of all processes

(d) Average memory of each process

(e) shared memory per process

(f) num page faults

(g) cache miss rate?

(h) pages served per second (did it go up or down?)

(i) How many times did major and minor garbage collection happen?

(j) How many times did this inter-generation optimization actually occur?

6 Analysis

1. Did we actually save any memory? How much on average? Were there any patterns?
2. How often did old objects actually refer to young objects?
3. How often did minor collection happen? Major collection?
4. How much CPU overhead was there in following that extra pointer every time? Can I even measure this?
5. How many pages did we serve per second? Did it go up or down?

7 Future Work

1. What other changes could I make to improve CoW performance? Lemme think.

8 Related work

This paper is very much inspired by recent updates in CPython's and Ruby MRI's garbage collectors which were updated in 2017 and 2014 respectively.

8.1 Copy-On-Write optimization in Ruby MRI's MSGC

Ruby 2.0 contains an update to the GC that improved CoW performance. Ruby's GC is a classic CGGC with a young generation and an old generation. In the old algorithm, every object contained a "visited" bit that would note whether the object had visited by the tracer. In the new GC algorithm, these "visited" bits are stored outside the objects being traced. In this way, the objects themselves don't need to be updated during a GC event.

The optimization described in this paper is different in that we focus on preserving old objects when objects are copied from eden into the older generation.

8.2 Copy-On-Write Friendly Garbage Collection in CPython at Instagram

A 2017 article from Instagram's Engineering series describes changes to CPython's garbage collector made at instagram aimed at improving memory performance in their web-tier servers. The experimentation section of this paper is heavily inspired by the test setup used at Instagram.

The optimization implemented at instagram was to have the programmer manually mark certain objects as "GC-Frozen". Objects that are GC-Frozen are not touched by the garbage collector and so the data in and around them is not touched. Because Instagram's specific use case involves a lot of static objects that are shared between processes, this worked effectively.

References

[1] “C2 wiki.” [Online]. Available: http://wiki.c2.com/

Contains witty quotes about almost every computing topic under the sun

[2] “Pypy source,” 2018. [Online]. Available: https://bitbucket.org/pypy/pypy/

[3] S. Auhagan, L. Bergstrom, M. Fluet, and J. Reppy, “Garbage collection for multicore numa machines,” *ACM Sigplan Workshop on Memory system Performance and Correctness*, June 2011.

These folks set forth a goal to write a garbage collector that scales to 48 NUMA cores. Part of the challenge with NUMA is that inter-processor communication is very expensive so they added a variety of optimizations to a traditional mark-and-sweep garbage collector so that the cores had to interact as little as possible. Many of these optimizations look like they would strongly benefit Copy on write performance.

[4] Bacon, Cheng, and Rajan, “A unified theory of garbage collection,” *OOPSLA*, 2004.

Very exhaustive survey of garbage collection that provides a theoretical spectrum in which to place every popular modern garbage collection scheme.

[5] T. M. Chilimbi and J. R. Larus, “Using generational garbage collection to implement cache-conscious data placement,” *ACM Sigplan*, vol. 34, March 1999.

A group modifies a copying generational garbage collector to monitor object reference frequency. On copy, the garbage collector co-locates frequently used objects. This reduces cache miss rate significantly. Copying generational garbage collector outperforms traditional mark and sweep in cache performance.

[6] G. et. al., *The Python3 Standard Library*, The Python Software Foundation. [Online]. Available: https://docs.python.org/3/library/gc.html

[7] L. T. et. al., “Linux kernel,” 2018. [Online]. Available: https://github.com/torvalds/linux

[8] M. Gorman, *Understanding the Linux Virtual Memory Manager*, 2004.

Reference manual for the 2.5 Linux kernel Virtual Memory Manager. Kernel 2.5 is pretty old so much of the code is different today but the high level algorithms should have stayed mostly the same.

[9] M. Hertz, Y. Feng, and E. D. Berger, “Garbage collection without paging,” *ACM Sigplan*, vol. 40, June 2005.

A group modifies a mark-and-sweep garbage collector to compactly store all the in- and out-going references in a page of objects when it is swapped out of memory. This prevents memory from needing to be swapped in just to scan during a major garbage collection event.

[10] R. Hudson and E. Moss, “Incremental collection of mature objects.” Springer, Berlin, Heidelberg.

Most generational garbage collectors just do mark and sweep on the oldest generation so as to avoid copying all that memory over again, but this can lead to fragmentation. This paper demonstrates a semispacial mature generation for a generational garbage collector and explains how to make mature collections incremental. It guarantees a cap on collection duration by sacrificing completeness. There is a very interesting analysis section explaining how they reached a good tradeoff between speed and completeness in this garbage collector.

[11] Z. Li. (2017, December) Copy-on-write friendly python garbage collection.

A team at Instagram edit CPython’s mark-and-sweep garbage collector so that it disturbs less memory during garbage collection in order to improve shared memory

[12] K. Sasada. (2014) Incremental gc for ruby interpretor.

Presentation at RubyConf2014 with very detailed description of Ruby's garbage collector and what changes they made to improve the performance in ruby's mark-and-sweep style garbage collector

[13] J. Seligmann and S. Grarup, "Incremental mature garbage collection using the train algorithm," April 2000.

First paper that actually implements the "train" garbage collector. My understanding is that it's like a generational garbage collector except it chunks up the old generation into many train-carts and copies each object several times through the series. Seems sub-optimal but there is a lot of buzz around it.

[14] *Welcome To PyPy*, Software Freedom Conservancy. [Online]. Available: http://doc.pypy.org/en/release-2.4.x

[15] B. Veal and A. Foong, "Performance scalability of a multi-core web server," *ACM/IEEE Symposium on Architecture for networking and communications systems*, December 2007.

Analysis of what system resources are the bottleneck in high-load web-servers. This document found that one of the biggest bottlenecks is on the address bus. Memory in cache was very quick to access, but anything in main memory needed to be fetched and awaited.

[16] C. Wu and M. Ni. (2017, January) Dismissing python garbage collection at instagram.

A team at Instagram Engineering discovered that they can reduce the memory usage of their web tier servers (running Django on CPython) by disabling garbage collection.

[17] B. Zorn, "Comparing mark-and-sweep and stop-and-copy garbage collection," *ACM Conference on LISP and functional programming*, 1990.

In depth comparison of Mark and Sweep vs Generational garbage collection. The paper found that mark-and-sweep has slightly higher cpu overhead but that generational garbage collectors require slightly less memory. This paper didn't reference the problem of memory fragmentation that occurs with a mark-and-sweep garbage collector.