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

Andrew Nelson
CSC 550

March 23, 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 with multiple forked workers and profile its memory usage with various Garbage Collection strategies. We use PyPy, a Python runtime environment with a modular codebase, and modify the Garbage Collector to improve its use of copy-on-write memory. Specifically, we take the Garbage Collector state which was previously stored inside the objects and instead store that state in separate memory pages. That way when a collection event occurs, those pages which were shared between Python processes can remain untouched during collection. Our analysis found that the modification decreases private memory usage in worker processes by up to 20% and preserves shared memory between workers.

Contents

1 Introduction

Advances in interpreters for high level languages have enabled their use 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. In order for these popular websites to continue to scale up, the tools on which they are developed need to get more efficient. Many of these systems are memory bound and so in order to make them more efficient we must make them use less memory.

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. In a web-server environment where worker processes are often forked repeatedly during operation, this situation is very common. Workers forked from the master process initially share some memory with the master and so the memory usage of the system as a whole is directly related to how much memory can be shared between processes. This is the type of environment that many of today's largest websites like Instagram operate with[16].

Oftentimes in environments like this, the Garbage Collector will disturb pages during collection by writing to them. When doing a Mark-and-Sweep operation, the Garbage Collector traverses the entire object-space and marks the objects which are live. This marking disturbs the page regardless of whether any objects within it are actually garbage. While Garbage Collection has the intention of reducing the number of pages used by a process, by disturbing shared pages in forked workers, the Garbage Collector ends up increasing memory usage in the group as a whole [16].

This paper examines an optimization in PyPy's Generational Garbage Collector[14] which is designed to reduce the number of shared pages that get disturbed during a major collection. To accomplish this, we take the Garbage Collector state which was previously stored within each object and we move that state into a dense index stored in its own page in memory. This prevents the Garbage Collector from needing to write to shared memory during collection. We then provide experimental evidence showing that this optimization does in fact reduce memory usage of a group of processes.

Section 2 explains various topics relating to the optimization and builds up to a demonstration intended to create some intuition into how this optimization is designed to work. Section 3 explains the details of the optimization and builds intuition as to which data structures need to be changed. Section 4 summarizes the source changes made to PyPy's Garbage Collector in order to implement this. Section 5 talks about how these changes were tested and section 6 examines the results of these tests and explains them. Finally, section 8 talks about other optimizations that have been made to Garbage Collectors to optimize for Copy-On-write behavior and explains how they differ from this research.

2 Background

Before presenting the modifications made to PyPy's Garbage Collector, we first motivate the need for Copy-on-Write-Sensitive Garbage Collection by explaining how shared memory gets wasted. First we review the Linux memory model and Copy-on-Write semantics. Next we review Reference Counting Garbage Collection as well as Mark-and-Sweep Garbage Collection and Compacting Generational Garbage Collection. Finally we put this information together to demonstrate how these algorithms when used in conjunction can end up unnecessarily wasting memory in a shared-memory environment.

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 all the physical pages in the parent process, 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 original physical page is no longer being shared (for example if all the other processes already made their own copies) 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 entire word 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 the object is deleted in place. When an object is collected, all the objects that it referred to are updated 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 in-degree of its associated node. When there are no edges into a node, the algorithm knows that this 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. Additionally, because objects are collected in place, this collection strategy is subject to fragmentation over time [4].

This algorithm has been used extensively in CPython since Python2 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 tagged 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].

Much like with RCGC, 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. Unlike the RCGC, because a complete Mark-and-Sweep collection needs to traverse the entire object-graph, there is a significant pause while collection occurs [4].

This algorithm has been used in conjunction with Reference Counting in CPython since Python2. 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. 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].

In many cases there are 3 or 4 successively larger generations. The same rules for moving objects from

generation 0 to generation 1 apply. 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 order to correctly account for inter-generational references, each generation must keep track of references from other generations into itself. 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].

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 simply never collect it. This implementation is not technically correct because mature objects which become garbage will never be collected but it performs well in practice.

The PyPy Python environment uses CGGC with two generations: a "nursery generation" for young objects and a "mature generation" for mature objets. On occasion, the mature generation is collected using the MSGC. Memory fragmentation in the mature generation can be filled when objects are copied from younger generations into mature [14].

2.5 Tragedy of the commons: Dirtying Shared Pages During GC

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 [16].

As a thought experiment, consider a forking webserver written in Python running atop a standard CPython interpreter. 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 metadata that the Garbage Collector stores on the page has changed. In this way, RCGC disturbs 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, MSGC disturbs shared pages which can increase memory used by the system.

This problem is somewhat alleviated when using CGGC such as is included in PyPy because we can copy objects out of a shared page without writing to it, however PyPy's GC is also subject to the dirtying of pages that happens when the mature generation is Mark-and-Swept.

While these algorithms are successful at reducing the memory usage of an individual process, when examining a collection of related processes they do poorly at reducing total combined memory usage. The next section is dedicated to modifying PyPy's CGGC to prevent the Mark-and-Sweep phase from disturbing shared pages.

3 Design

In this paper we modify the Garbage Collector in the PyPy Python runtime environment. We choose PyPy because it is written in a dialect of Python which makes the interpreter easy to modify for experimental purposes. Also, because PyPy's Garbage Collector is a relatively simple CGGC, the modification and analysis is relatively straightforward. PyPy's Garbage Collector is a CGGC with generations: a "nursery generation" for young objects and a "mature generation" for mature objets. On occasion, the mature generation is collected using the MSGC. Copying Generational Garbage Collection events are referred to as "minor collections" while Mark-and-Sweep Garbage Collection events are referred to as "major collections" [14]. The modification described here is intended only to affect major collections.

To prevent the Garbage Collector from touching objects in place, we add a hashtable to the Garbage Collector that contains the set of all the elements that have been visited. When a Mark-and-Sweep occurs in the mature generation, rather than writing

to the objects in place, the Garbage Collector inserts the object's id into the hashtable. When the Mark-and-Sweep finishes, that hashtable is cleared. This hashtable will be stored in different physical pages from the objects which it describes. By doing this we create two categories of pages: those that contain objects and those that contain object metadata. The GC usually only needs to modify the pages that contain object metadata which means that the pages containing objects can remain undisturbed.

4 Implementation

PyPy has several Garbage Collectors but the one we chose to patch for the purposes of this paper is called "minimark". Minimark is a CGGC with two generations [14]. New objects are created in the "nursery" and are moved to the "mature" generation on minor collection events. When the mature generation exceeds a certain size, a major collection starts and the mature generation is Mark-and-Swept. During this MSGC, each object in the mature generation is visited and a bit is set in-place which tags that object as "safe" during this collection.

To implement this change, we added a member variable to the MiniMarkGC class of type *rpython.memory.support.AddressDict*. PyPy's memory library does not provide an AddressSet type so this was chosen as an approximation. Objects which are in the mapping and which map to themselves are considered visited and objects which either are not in the mapping or which map to *rpython.memory.Null* are considered unvisited.

The following code snippet was added to MiniMarkGC and demonstrates the new interface for determining an object's visited state.

```
def obj_visited(self, obj):
  return (
    self.visited_objs.contains(obj) and
    self.visited_objs.get(obj) == obj)

def mark_obj_visited(self, obj):
  self.visited_objs.setitem(obj, obj)

def unmark_obj_visited(self, obj):
  self.visited_objs.setitem(obj,
        llmemory.NULL)
```

We treat *visited_objs* like a set. At the start of collection, *visited_objs* is initialized to be an empty set. During the Mark phase, objects are inserted into this set. During the sweep phase, the entire object-space is scanned again. When objects are traversed, they are either collected if they were not visited during the Mark phase or are unmarked if they were visited in the Mark phase. Afterwards, if running in debug mode, PyPy traverses the entire object-space a third time to ensure consistency. In a clasic MSGC, the third pass is unnecessary, but PyPy does it to simplify development.

5 Methodologies

This paper focuses on a web-server workload and so to validate our patch, we chose to create the closest approximation of a production webserver possible. Our model was Instagram's web-tier servers which are briefly described in their external blog. Instagram is a massive Django instance running atop CPython2.7 using Ngnx as a front-end server and using uWSGI to manage Django worker processes [16]. uWSGI is running in pre-fork mode meaning that a worker is started and then forked several times to create more workers. When a worker dies, an existing worker is forked to replace it.

With this test environment we create 2 experimental groups. The groups are as follows:

1. Minimark GC without patch
2. Minimark GC with patch

5.1 Test Environment

In order to replicate this environment, we created a docker image that had Ngnx, uWSGI, Django, and PyPy installed on it. We ran uWSGI in pre-fork mode with 4 worker processes. The docker image was capped at 8GB of memory and 3 cpu cores. All benchmarks were ron on a ThinkPad T460 with 12GB ram and an i5-6300 2.4GHz quad-core processor. Because the docker image was capped at well below the host's resource capacity, we believe benchmarks are reasonably isolated from resource contention with host processes.

Because Instagram is closed-source, proprietary software, we were unable to procure its source code for use in testing. Instead we created a Django application which was created specifically to emulate a production web-server load. Specifically, we created a Django application which allocates a large chunk of memory at startup and which allocates an additional chunk of memory for each request. Below are the snippets of code used to create test-garbage:

```
class LargeObject():
  # Use a bunch of memory
  def __init__(self):
    self.lists = []
    self.strs = []
    for i in range(1600):
      self.lists.append(["text"] * i)
      for j in range(40):
        self.strs.append("str" * 8)

def on_startup():
  global static_memory
  static_memory = [
    LargeObject()
    for x
    in range(5)]

def on_page_load():
  global glob_a, glob_b, glob_c
  glob_a, glob_b, glob_c = (
    LargeObject(), glob_a, glob_b)
```

We believe that this mix of memory allocation somewhat approximates what might be seen in a production web-server.

5.2 Stress-Testing the Deployment

To generate a test load we used httperf which makes a series of http requests to a given http endpoint. We chose to track two specific metrics for each experimental group: The time it took to handle N requests and the memory used after handling N requests.

To determine the time to handle N requests, we use the httperf tool to make N requests and record the wall-time it took for all requests to get a response. We run this experiment with N values of 5k, 10k, 15k, and 20k.

To track memory usage as the number of requests increases, we make a burst of 100 requests and then check the memory usage of each uWSGI worker process. We repeat this process until we've made 50 bursts of requests or 5000 requests in total. At each step we check memory usage by scraping that process's $/proc/PID/smaps$ file. The private memory used by that process is the sum of all lines starting with "Private_clean" and "Private_dirty" and the shared memory used by that process is the sum of all lines starting with "Shared_clean" and "Shared_dirty".

6 Analysis

Experimental results show that this approach was successful in preserving shared memory in each worker process. During our verification step we measured two types of performance: memory usage and latency. In the first section here we see that the Patched version of the minimark GC uses less memory overall than the Standard version by preserving shared memory. In the second section here we see that the Standard version serves more requests per second up until a certain point after which it underperforms because of paging. The latency in the Patched version can be attributed to the runtime overhead of looking up objects in the visited-objects set. The observed latency implies that depending on the setup, the cost of the modification can be outweighed by the savings in memory.

6.1 Memory Usage

Figures 1 and 2 clearly show that the Patched version of the Garbage Collector preserves more shared memory than the Standard version. In the upper portion of each figure we see the shared memory for each case. In the Standard GC group, we see that the amount of shared memory trails off as the number of requests grows because when each of the worker processes runs major Garbage Collection they taint many of the pages. On the other hand, the shared memory chart for the Patched GC is almost entirely flat meaning that those shared pages are untouched and preserved. In the lower chart we can see that after 5000 requests, the Patched worker processes take up somewhere in the realm of 800Mb of space per worker while the Standard worker processes take up 1000 to 1200Mb of space per worker. This accounts for a reduction of around 20%.

6.2 Latency

The table below shows the time per request for our deployment with and without the optimization. Figure 3 shows the same information in a graphical format. We can see that for a small amount of traffic, the standard minimark GC responds to more requests per second than our patched version. However, once the number of requests per second exceeded around 13k, the total memory of the system exceeded the 8GB allotted and the deployment started paging aggressively. On the other hand the patched GC continues to settles down for larger numbers of requests. The upward trend in response rate is a result of PyPy's tracing Just-in-Time compilation - over the course

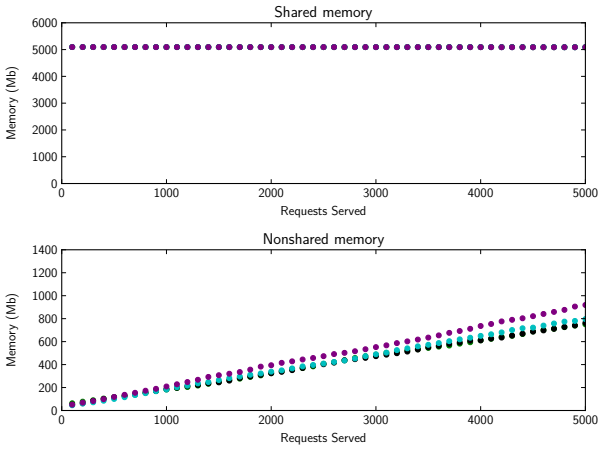

Figure 1: Memory usage of patched Garbage Collector running 4 workers after X requests. Each dot represents a worker's memory consumption after X requests.

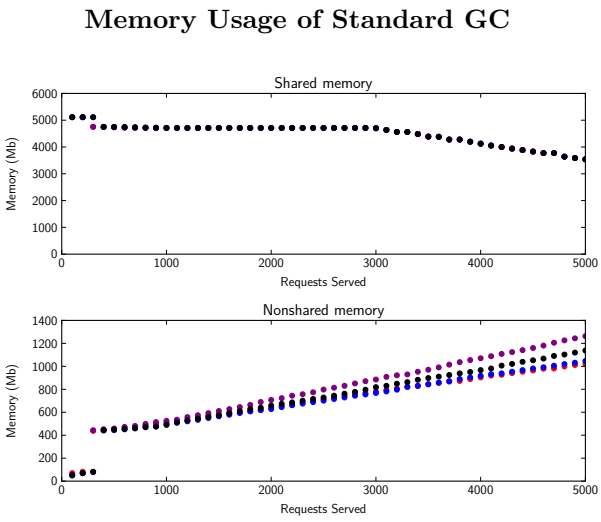

Figure 2: Memory usage of standard Garbage Collector running 4 workers after X requests. Each dot represents a worker's memory consumption after X requests.

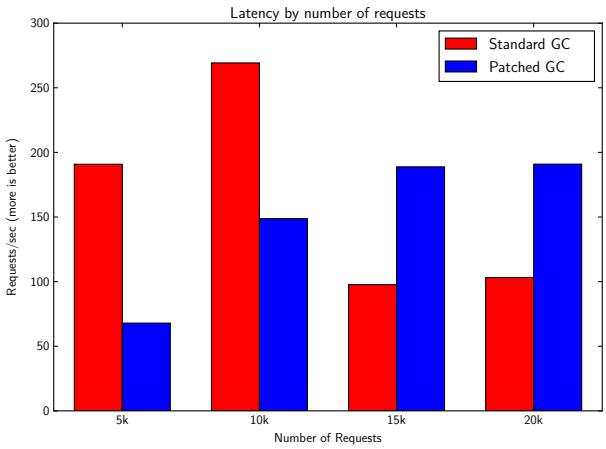

Figure 3: Request Rate for different Request Quantities. Note Standard GC started paging after 10k requests

of a program's lifetime PyPy will compile commonly executed codepaths. When the application is freshly started, PyPy interprets the source code and spends time compiling codepaths. After the application has been running for a while, PyPy spends less time interpreting and compiling and more time executing native code [14]

	Standard	**Patched**
5k Requests	190.8 conn/sec	67.9 conn/sec
10k Requests	269.2 conn/sec	148.7 conn/sec
15k Requests	97.7 conn/sec	188.8 conn/sec
20k Requests	103.2 conn/sec	190.9 conn/sec

From this data we can see that there is time overhead in our optimization. If we had allocated more than 8GB of memory to the test, the Standard GC process would likely not have paged and would have responded to more requests per second than the patched version. However the patched version used less memory overall and thus in our setup did not spend any time paging.

7 Future Work

While the optimization explored in this paper account for noticeably memory savings, there is much more that can be done in the future.

In a CGGC like PyPy, when objects are copied from the nursery into the mature generation, all the mature objects which pointed to the new object in the nursery must be updated to refer to that objects new post-copy location. This results in writes to mature objects that can be avoided. If instead of updat-

ing the pointers to young objects in place in the old objects, there were a reference table so that all references from old to young objects were indirect, then when young objects were copied only the reference table would need to be updated. This lets the mature objects remain untouched even when the object which they are referencing moves.

Additionally, the benchmark used in this paper was concocted specifically for testing purposes. The results from using this benchmark may or may not accurately reflect the reality of real-world Django deployments. Future work would likely take the findings in this paper and apply them to actual existing production websites. Finding an open source website that runs on Django that is relatively easy to deploy would be paramount in bringing this research to real-world relevance.

Lastly it would be interesting to see how the preservation of shared memory affects the cache miss rate of the worker processes. In theory, if the worker processes are sharing physical memory then a single line-item in the cache can be shared amongst worker processes. Whether this happens in would be worth looking into.

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 a similar update to the GC that improved CoW performance. Ruby's GC is a classic CGGC with a young generation and an old generation that is occasionally Mark-and-Swept. 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 [12].

The optimization described in this paper is very similar to this, however we are specifically testing this optimization in the context of Python for a webserver workload.

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.

In their article, the Instagram engineering team explored multiple ways of reducing memory usage. The first modification they tried was taking the reference counts stored in every Python object and moving them to a separate page. The intent was that when reference counts get updated, only the reference count page needed to be changed and the page storing the objects could remain shared. In practice this modification did not reduce memory usage [16].

The modification Instagram ended up implementing 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 Garbage Collector state embedded within them is not touched. Because Instagram's specific use case involves a lot of static objects that are shared between processes, this worked effectively [11].

9 Conclusion

The results in this paper clearly show that by moving the Garbage Collector state out of the objects and into isolated Garbage Collector memory, we can prevent unnecessary memory writes and we can preserve shared memory between forked processes. While the changes described in this paper have some runtime overhead in terms of mapping objects in memory to their associated Garbage Collector state, we found that for our deployment the savings in memory far outweighed the runtime overhead.

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 outgoing 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/latest

[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.