CS263 Python Garbage Collection

Chinmay Sonar and Andrew Zhang

1 Abstract

An essential part of any runtime system is *memory management*. Traditionally, the programmer was supposed to take care of heap allocations and deallocations during program execution. Since doing such memory management is a tedious task, modern programming languages such as Java, Python automated this procedure using garbage collectors to produce accurate management. As garbage collector has to take care of allocations/deallocations on the heap *during* the execution of the program, it inadvertently introduces an overhead. Hence, designing an efficient garbage collector is of paramount importance for the overall program performance. Designing such efficient collectors keeping in mind specific constructs in a particular programming language has been a topic of study for a long time. In this project, we study the garbage collectors for Python. In particular, we understand the internal mechanism of garbage collectors for CPython and PyPy, and we compare them with internal parameter tuning and benchmark programs.

2 Introduction/Outline

The objective of this project was to explore and learn the details of garbage collection, specifically with Python. We wanted to see how the concepts we learned in class were used in a real runtime system. In addition, we wanted to profile Python's garbage collector on different programs, and observe the effect of modifying the garbage collector's tunable parameters. After watching a PyCon presentation titled "PyPy - The Hero We All Deserve" [1], which praised PyPy's garbage collector, we wanted to do similar profiling with PyPy's garbage collector. With this data, we would be able to see some of CPython's garbage collector's and PyPy's garbage collector's strengths and weaknesses.

In order to profile the time spent for garbage collection, we created a visualization tool that aggregates and plots times for individual collections. This data is provided by both CPython's and PyPy's *gc* modules through built-in debug statements and callbacks respectively. The produced graphs contain points for each collection representing its time, as well as a line representing the cumulative time.

In this report we detail what we learned about the Python garbage collectors and the tests we conducted. In Sections 3 and 4, we explain the internal working of garbage collectors in CPython and PyPy respectively. Next, Section 5 – Parameter Comparison for CPython and PyPy, describes the tests we conducted relating to modifying the garbage collectors' parameters. Furthermore, in Section 6, we compare CPython and PyPy GC's on benchmark programs from Pyperformance benchmark suite. We provide a division of benchmark according to their performance, and describe a few program traits favorable for each garbage collector.

3 Cpython Garbage Collector

CPython uses two methods for garbage collection. The first is reference counting. When an object has a reference count of zero, it is considered unreachable and safe to collect. However, reference counting is not sufficient for all cases, specifically when reference cycles are present.

To collect reference cycles, CPython uses a second garbage collection method that can be interfaced using Python's *gc* module. This uses an algorithm based on reference counts to determine whether a reference cycle is safe to collect or not [2]. The garbage collector tracks all container objects, or objects that can contain references to other objects, including lists, dictionaries, and classes, which can potentially create reference cycles. When a collection is triggered, reference counts are copied to prevent unwanted changes, and updated as if the objects are collected. The algorithm iterates over every container object, and decrements the reference counts of the objects it contains, simulating its collection. After this iteration, a second pass is conducted to determine if the cycles are unreachable. If unreachable, the reference count of every object within a cycle should be zero, as its only references were from other objects in the cycle. (The reference counts will not actually be zero, as these objects are being tracked by the collector.) If any object has a reference count greater than zero, then it must be reachable from outside the reference cycle, and therefore the entire cycle is not safe to collect.

The CPython garbage collector is also generational, using the idea that objects frequently die young. By default, CPython uses three generations. If an object survives a collection within a generation, it moves up to the next generation. Collections within each generation are triggered when a corresponding threshold is met. For example, a generation 0 collection is triggered when the number of object allocations minus the number of object deallocations is greater than the generation 0 threshold. These threshold values can be changed through the *gc* module, which will be discussed in a later section.

4 PyPy Garbage Collector

PyPy is written in a restricted subset of python called RPython [3]. Currently, RPython is primarily translated to low-lever language C. Since PyPy is written in RPython, the interpreter is abstracted from low-level details like GC. Such an implementation offers a variety of advantages over CPython where the implementation is littered with reference counting artifacts. PyPy has a *pluggable* garbage collector i.e. we can choose the garbage collector at runtime. PyPy used several garbage collectors over the years [4]. The current collector is an incremental version of Mark and Sweep collector known as Incminimark. Next, we briefly describe the working of Incminimark garbage collector.

Incminimark GC

One major drawback of mark and sweep collector is its "Stop the World" approach i.e. GC stops the program execution during the collection process. *CPython suffers through this issue of large intermittent delays* which can be deal-breaker for interactive servers or real-time video processing. Improving on CPython, *PyPy uses an incremental garbage collection policy* which promises not to stop the program for more than a few milliseconds each time. PyPy achieves this by dividing the work of a full (major) collection to be performed at different times. We describe Incminimark in details, and provide comparison of GC routines in Section 8.1.

5 Parameter Comparison for CPython and PyPy

CPython Threshold Values

One way to interact with CPython's garbage collector and gain more control of how it functions is to change the generational threshold values through the *gc* module. We explored the effects of changing these threshold values, specifically the generation 0 threshold, on the total time spend in garbage collection. The default value of the generation 0 threshold is 700, so a generation 0 collection is triggered once the number of object allocations minus deallocations since the last generation 0 collection is greater than 700.
We ran the go.py from Pyperformance, varying the generation 0 threshold, and recorded the total time spent in the garbage collector. Figure 1 shows the average times spent over 16 trials for each threshold

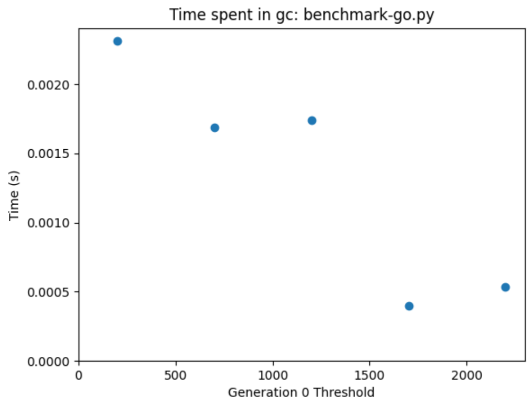

Figure 1: This graph shows the time spent in GC for various gen0 threshold values.

value. There is a general decreasing trend in the total time as the threshold values increase, up until around 1700. This can most likely be attributed to the decrease in the number of collections being conducted. With a threshold of 200, 14 collections were conducted, and with thresholds of 2200 and 2700, only 1 collection was conducted. As a result, we determine that increasing the threshold such that the number of collections is minimized, while observing limits of the physical machine, allows for the least amount of time spent in the GC. In addition, increasing the size any larger, without decreasing the number of collections, will not provide any benefits and will increase the time taken for collection. However, this will not be the best solution for any program, as other considerations should be made. For example, having a larger threshold usually leads to larger pauses for collections, which may be unacceptable for certain low-latency programs.

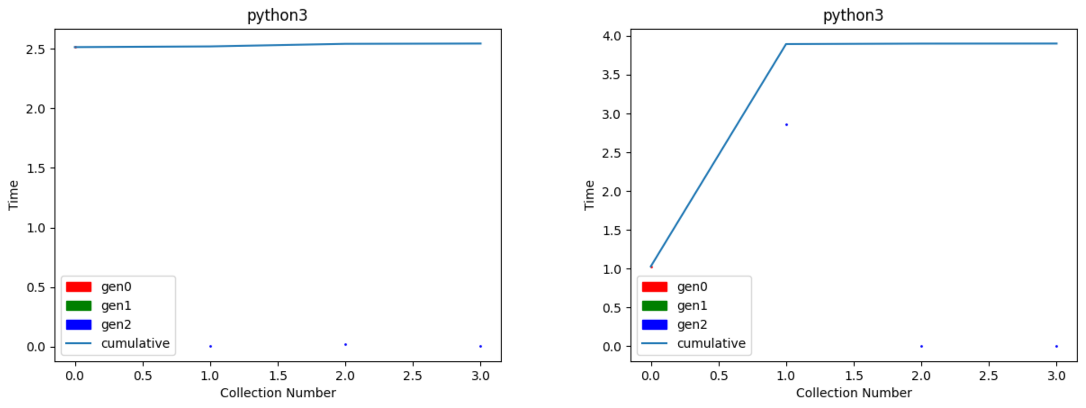

Figure 2: The left graph shows the time spent in GC with a gen0 threshold of 10,000,001. The right graph shows the time spent in GC for the same program with a gen0 threshold of 10,000,000.

Though the difference seems pretty small in this benchmark program, a longer running program may run many more collections and small differences can add up to larger effects. To highlight this effect, we created our own adversarial example program that results in a large change in time due to a small change in threshold value. The program simply allocates many self-referencing objects, creating many reference cycles that must collected. Specifically, there are around 10,000,000 objects total that should be collected. As seen in Figure 2, The total time increases from about 2.5 seconds to about 4 seconds despite the threshold decreasing by only one, from 10,000,001 to 10,000,000. Though these threshold values and program may

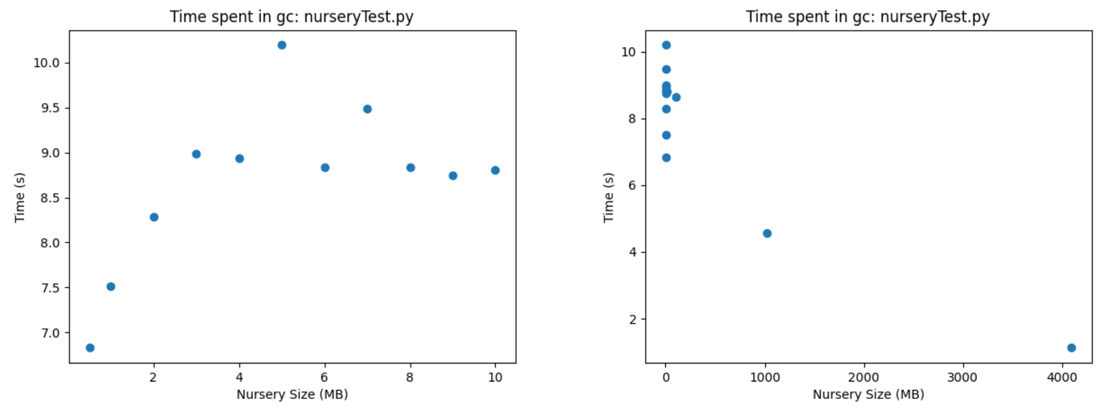

Figure 3: The left graph shows the time spent in GC for our custom test for nursery sizes from 0.5MB to 10MB. The right graph includes additional larger nursery sizes up to 1024MB.

Nursery Size (MB)	0.5	1	2	3	4	5	10	100	1024
# Minor Collections	48040	23892	11906	7938	5943	4755	2403	244	24
# Major Collections	19650	9770	5880	4057	3515	2744	1312	71	1

Table 1: Each column contains the number of minor collections and major collections for that nursery size.

be unreasonable, a similar effect can reasonably appear in actual programs on a smaller scale, which can build up as the program continues running. This shows the importance of understanding and profiling a program, as well as the machine the program is running on, before setting custom threshold values. Blindly setting them can have a negative effect on the program's time efficiency.

PyPy Nursery Size

The size of PyPy's GC nursery parallels CPython's GC threshold values. Like the threshold values, the nursery size can be modified and will determine when a collection is triggered. The PyPy "gc" documentation states that by default it will be half the size of the machine's last-level cache. Because of this, we explored if and how cache misses affected the total time spent in garbage collection.

We created a custom test program that allocates and deallocates many objects, with some living longer than others. We recorded the total time spent in the GC while varying the nursery size by setting the PYPY_GC_NURSERY environment variable. We ran the program 16 times for each nursery size, ranging from 0.5MB to 1000MB. Seen in Figure 3, for nursery sizes 10MB and smaller, the smallest nursery size resulted in the shortest amount of time spent in garbage collection. As nursery size increases, collection time also increases, until a certain size is reached. After that, the collection time decreases, as seen as the 100MB and 1024MB nursery sizes.

We believe this pattern can be attributed to cache misses. The last-level cache on my computer is 3MB, and as the nursery size approaches and passes 3MB, the total time increases, likely because caches misses occur more often. These cache misses may be very costly to collection time. At 0.5MB, even though more collections are triggered, the smaller nursery size results in fewer cache misses. However, once the nursery size was increased such that the number of collections decreased significantly, the time taken also drastically decreased. The number of collections for each nursery size is shown in Table 1. We concluded that with a significant number of collections, the effect of cache misses outweighs the number of collections made, while with only a few collections, the number of collections has a greater effect than cache misses. Similar tests were also conducted on Pyperformance benchmarks, which is discussed in Section 8.2.

Like with CPython's threshold values, PyPy's nursery size should not be modified blindly. Though it may seem like choosing a large nursery size will guarantee less time spent in collection, this is not true for some cases. For example, for a program that runs for a very long time with many collections, even large nursery sizes can result in a significant number of collections such that cache misses will result in inefficient collection time. As a result, it is once again important to profile and understand the program that is running before setting a custom nursery size.

6 Performance on Benchmark Programs

In this section, we compare the performance of PyPy and CPython garbage collector. The objective is to find the specific traits of the program favorable for CPython and PyPy. We use the visualizer we developed to compare the time spent in garbage collection for CPython 3.6 and PyPy 7.3.1 for 10 benchmark programs from Pyperformance benchmark suite [5].

CPython	**PyPy**
nqueens	float
pidigits	deltablue
fannkuch	hexiom
go	pickle
raytrace	simple_cycle

Table 2: First column contains a set of benchmark programs in which CPython GC outperforms PyPy, and vice versa for the second column.

We next describe our ***observations** on benchmark programs*: **(i)** PyPy performs well when the number of objects allocated is relatively small, and in the cases when the objects are long lived. **(ii)** CPython performs well when program has a lot of short lived objects. **(iii)** PyPy also performs well on a simple code snippets. We believe this is because on a simple code with localized references PyPy JIT can optimize a lot better.

We further discuss our detailed observations on a few benchmark programs in Section 8.3.

7 Conclusions

Overall, no definitive conclusion on which Python implementation's garbage collector is better in terms of time. In addition, no threshold values or nursery size is better than another in all cases. In order to maximize efficiency for a given program, requirements like latency and memory available must be considered, and memory allocations must be profiled and understood. Only then can an informed decision on which garbage collector should be used and how parameters should be set.

Our analysis suffers from one issue – for CPython, our visualizer does not take into account the time spent in reference counting. We only consider the time spent in collecting objects with reference cycles. Since reference counting instructions are embedded in the program, the way to account for that is not clear to us. One way would be count number of times the references are updated, and multiply that by average time required to execute reference update.

References

[1] PyPy - The Hero We All Deserve
`https://www.youtube.com/watch?v=zQVytExlnEk` 1

[2] Time to take out the rubbish: garbage collector
`https://www.youtube.com/watch?v=CLW5Lyc1FN8` 2

[3] PyPy introduction,
`https://doc.pypy.org/en/latest/introduction.html` 2

[4] PyPy Garbage Collectors,
`https://doc.pypy.org/en/release-1.9/garbage_collection.html` 2

[5] Pyperformance Benchmark Suite,
`https://github.com/python/pyperformance/tree/master/pyperformance/benchmarks` 5

[6] Knowing your garbage collector,
`https://www.youtube.com/watch?v=sSMPiQZTyrI&t=1869s` 7

8 Appendix

8.1 Incminimark Garbage Collector

PyPy uses a 2-generational GC. The younger generation is known as *Nursery* whilst the older one's *Arena*. All new (young) objects are allocated in the Nursery (except for very large objects which are allocated in a special space outside Nursery) [6]. The collections in Nursery and Arena are known as Minor and Major collections respectively. Objects surviving minor collections are moved to the Arena. Since PyPy uses a generational collector, it takes an advantage of "High Infant Mortality". Also, as it only uses only two generations, every object is copied at most once, and hence, the overhead of copying is not much (this is known as "Pig in the snake problem").

PyPy divides a major collection in several minor collections (GC spends sometime in major collection during each minor collection). Each collection consists of mark and sweep phase. A major challenge is to perform marking (exploring reachability graph of each object) incrementally since the code executed in-between two minor collections can add/remove the references. PyPy addresses this issue with a *special Write Barrier (WB)* data structure which is essentially a queue. Traditionally, WB is used only to keep track of references from Old to Young objects which is useful in every minor collection (since we do not want to delete any Young object referred by an Old object). Now, with incremental marking phase, we want to keep track of both Old to Young, and newly added Old to Old references during intermediate code execution between minor collections (as marking should be cognizant of the latest reference graph). Hence, at the end of each minor collection, we add two kinds of objects to WB: Young objects which survived the minor collection, and Old objects for which references are updated. During incremental marking, we go through each object in WB, and explore reachability graph from that object. Intuitively, WB is the *state* of incremental marking. A traditional sweep algorithm works just fine during incremental sweeping. This concludes the description of PyPy GC.

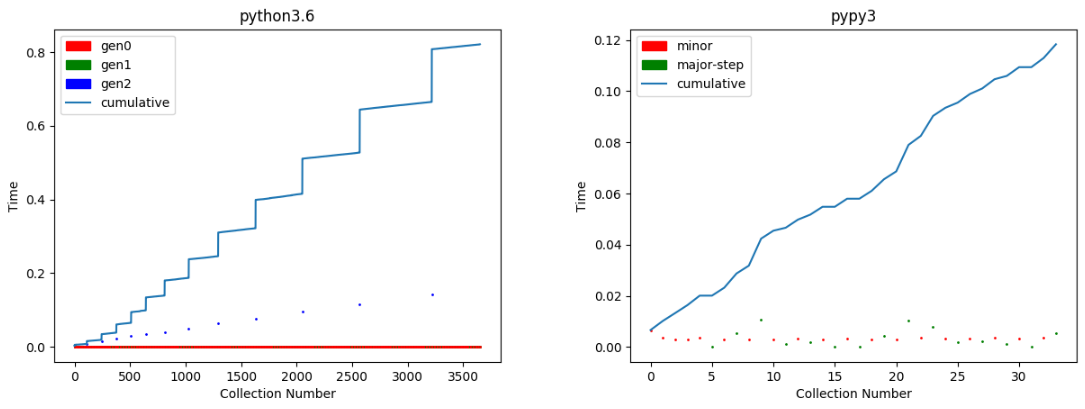

Figure 4: The left, right graphs show the time spent in GC for our custom test for reference cycle for CPython and PyPy respectively. Notice that CPython GC induce long pauses whilst PyPy does not have to suffer through long pauses.

8.2 Nursery Size Profiling on Benchmark Programs

We ran the same procedure as our custom nursery test on a couple benchmark programs, including go.py from Pyperformance, as seen in Figure 5. The trend we saw here is that time taken decreased as nursery

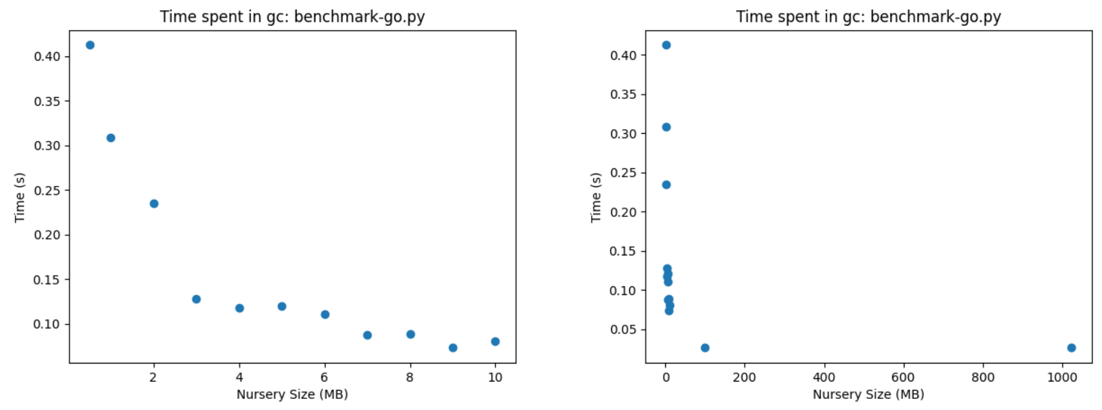

Figure 5: The left graph shows the time spent in GC for go.py for nursery sizes from 0.5MB to 10MB. The right graph includes additional larger nursery sizes up to 1024MB.

size increased. This follows our prior reasoning from our custom test. These benchmarks ran much fewer garbage collections than our custom test, so there were not enough collections for cache misses to play a large role. Also, as seen in the graphs that include 100MB and 1024MB, increasing the nursery size further does not give much or any benefit. This is because only around one collection is run at nursery sizes greater than 10MB, so there is not any room to further decrease time.

8.3 Performance on Benchmark Programs

float.py

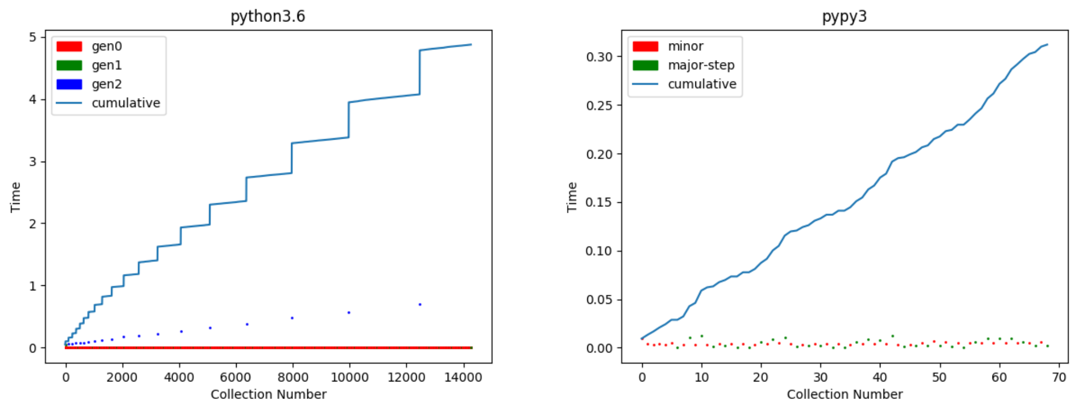

Figure 6: The left, right graphs show the time spent in GC for the benchmark program float.py. Here, PyPy performs 16x faster than CPython.

The program float.py initially creates n-floating point numbers. The rest of the program involves manipulation on these floating point numbers. Program allocates a fixed set of long lived objects. In CPython, since there are three generations, the long lived objects have to be copied several times, and the references have to be updated accordingly. We believe this might be a major factor in slow performance of CPython.

nqueens.py

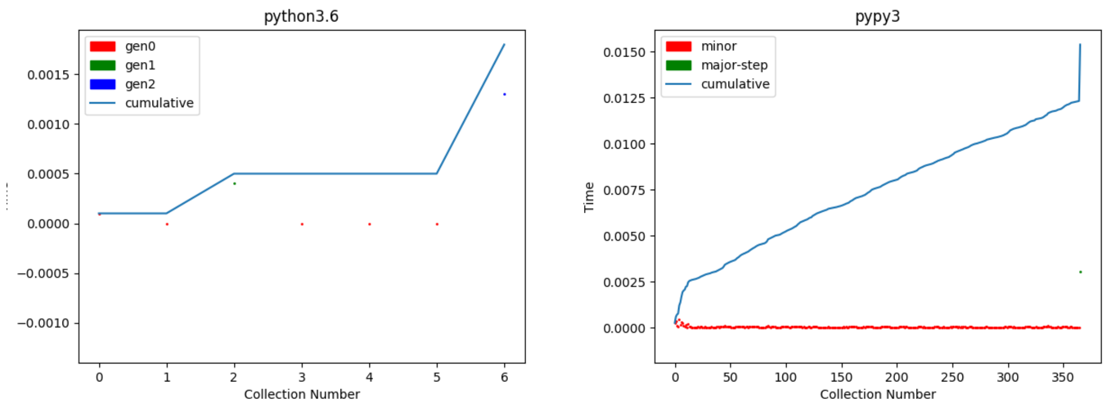

Figure 7: The left, right graphs show the time spent in GC for the benchmark program nqueens.py. Here, CPython runs 10x faster than PyPy.

For nqueens, the task is place a set of nqueens on an $n \times n$ chessboard such that no pair of queens attack each other. The program iterates through $n!$ possible positions, and returns a correct answer. Here, since each iteration of the loop tries a new position, the number of objects introduced is large but the objects are short lived. We believe that since CPython uses reference counting for objects without reference cycles, it efficiently gets rid of short lived objects. On the other hand, for PyPy, these objects will be cleaned only at a minor collection.

8.4 Machine Specifications

Andrew Zhang: Windows 10 with Intel Core i5-7200U CPU, 8GB RAM, and 3MB L3 cache.

Chinmay Sonar: Ubuntu 16.04 with Intel Core i7-6700T CPU, 16GB RAM, and 8MB L3 cache.