GC Collect report

Overview

The “gc_collect” benchmark is used to test the performance of CPython garbage collector. The idea is to measure how Python handles memory management, specifically garbage collection, to understand how much overhead it causes.

Garbage collection is a method of automatic memory management. The garbage collector runs and checks if memory previously allocated by the program is no longer in use. If so, it is considered garbage and is freed. This sort of method is popular in high-level languages as it allows the programmer to not directly manage memory.

From the article cited in the sources (“A Hardware Accelerator for Tracing Garbage Collection”): applications that use languages utilizing GC can spend 10-35% of their CPU cycles and an average of 10% of the total energy consumption on garbage collection.

Python primarily uses two methods of garbage collection:

1. Reference counting: For every object in Python exists a reference count. It is updated when an object is referenced or dereferenced. When an object's reference count drops to zero, it is considered garbage and Python deallocated its memory.
2. Generational garbage collection: This method is used less often and is mostly employed to solve cyclic references. It splits objects into several groups and checks the “younger” objects more often.

Reference counting is fairly efficient, but cannot handle cyclic references. Cyclic references occur when two or more objects reference each other, forming a cycle.

Generational garbage collection in Python is non-copying and follows the mark-and-sweep method. **Mark Stage**: The garbage collector traverses the object graph to identify objects that are part of a circular reference. It temporarily decrements the reference count of objects as it walks the graph. Any objects that have a non-zero reference count after this process are part of a cycle. **Sweep Stage**: Once the unreachable cycles are identified, the garbage collector "sweeps" through the marked objects and frees their memory.

This benchmark aims to stress the Python garbage collector on both methods. In this benchmark, several linked lists (“CYCLES”=100) are created, each containing several nodes (“LINKS”=21). For each list, the last node references the first one, creating a cycle.

It is important to mention that the CPython garbage collector often runs concurrently with the Python program as part of the CPython interpreter. In this benchmark, the garbage collector is explicitly called (“gc.collect()” line 53).

During research for this project, we encountered several warnings against such use of the garbage collector. The warnings are issued as the CPython garbage collector is a “stop-the-world” action, creating a pause; as such, it is important to use it carefully and only with good reason.

Only the “gc” and “pyperf” libraries are used in this benchmark. Only the “Node” class is defined.

Initial Analysis

During initial analysis, we inspected the benchmark code, as well as learned about the different Python interpreters and their inner workings. We looked at the CPython garbage collector code that is written in C.

It is important to note that in this benchmark, the garbage collection of the nodes isn't done in the background or while the program runs something else, as often happens. This, combined with the fact that the Hardware has only one core, significantly limits the hardware-aware software changes we can use, as concurrency isn't native to this benchmark.

This benchmark's performance is bound by the time it takes to go over the allocated objects.

An example of a common speedup is adjusting the gc.threshold() so that the GC runs more frequently but for shorter periods. This is not relevant for this benchmark as the GC is manually triggered.

We also investigated the memory use, analysing how much space each "node" and thus each "cycle" takes.

While on the surface each node only uses 48 Bytes of space, upon further inspection, we found out it actually uses 280 Bytes per node. This significant difference comes from the attribute dictionary (__dict__). Python, by default, allows adding more attributes to a class at runtime. To allow that, each such object stores its attributes in a dictionary, creating overhead. The 48 Bytes measured are the "base size" while 280 Bytes includes the memory consumed by the attributes dictionary etc. We will use this fact to optimize performance.

Using perf record, we created a flamegraph. We can see the two main GC stages: mark as "**deduce_unreachable**" and sweep as "**delete_garbage**". The mark stage takes 11x as long as the sweep stage.

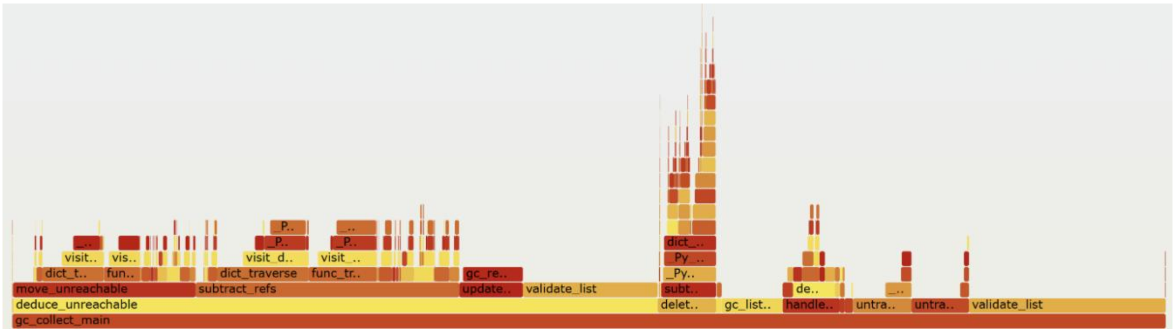

Figure 1 gc_collect FlameGraph

Another interesting thing clear from the flamegraph is that **dict_traverse** (as well as other dictionary handling functions) takes a significant amount of time in the mark phase.

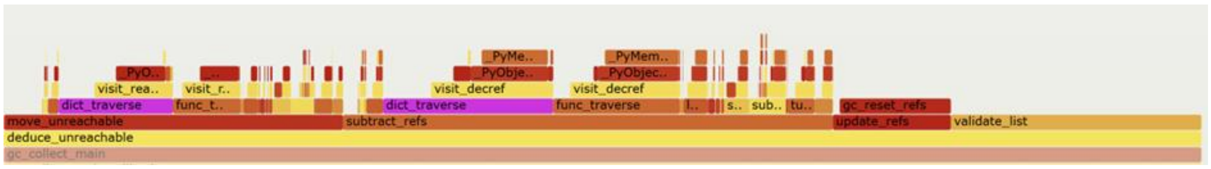

Figure 2 gc_collect FlameGraph – emphasis on deduce_unreachable

The original benchmark ran for 7.3 milliseconds.

Optimizations

Since this benchmark is memory-bound, our optimization attempts focused on optimizing memory access. Both optimizations decrease the amount of memory the garbage collector must examine, thus allowing it to run faster.

We implemented 2 optimizations on this benchmark.
Both are very readable and simple to implement during development.

Using gc.freeze()

We added the line gc.freeze() after initialization is done and before the test begins:

```
def benchamark_collection(loops, cycles, links):
    total_time = 0
    gc.freeze()
    for _ in range(loops):
        gc.collect()
        all_cycles = create_gc_cycles(cycles, links)

        # Main loop to measure
        del all_cycles
        t0 = pyperf.perf_counter()

        collected =gc.collect()
        total_time += pyperf.perf_counter() - t0

        assert collected is None or collected >= cycles * (links + 1)

    return total_time
```

Figure 3 gc_collect_opt.py code, adding the gc.freeze()

From the Python documentation: “gc.freeze(): Freeze all the objects tracked by the garbage collector; move them to a permanent generation and ignore them in all the future collections.”

Smart use of this function allows us to tell the GC which objects may interest it, and which do not. Before adding it, each call to the GC triggered an inspection of all the objects in the application’s memory. After the change, the only objects that the GC checks are our cycles, allowing a shorter run.

We can confirm that by running **“len(gc.get_objects())”** right before gc.collect().

	Number of objects tracked by gc
Using gc.freeze()	2100 == number of nodes allocated
No use of gc.freeze()	28808

Table 1 Amount of objects tracked by GC, with and without freeze

Using the “__slots__” attribute

Adding the __**slots**__ class attribute to the “nodes” class.

```
class Node:
    __slots__ = ('next', 'prev')
    def __init__(self):
        self.next = None
        self.prev = None

    def link_next(self, next):
        self.next = next
        self.next.prev = self
```

Figure 4 gc_collect_opt, the node class

As discussed before, Python objects by default store their attributes in a dictionary structure to enable dynamically adding attributes at runtime.

In this benchmark, as in many applications, this functionality isn’t used, thus we can use the __slots__ attribute.

The __slots__ attribute allows explicit declaration of the class’s attributes, allowing to get reed of the dictionary and save the class in a more memory-efficient way.

When using this attribute, each node shrunk from 280 bytes total to only 64 Bytes! (Measurements taken using “pympler import **asizeof**”)

This attribute allows for 2 things:

1. Memory efficiency: as the nodes grow smaller, more of them can be stored in L1. This allows for faster access.
2. Faster attribute Access: when looking for the object's attributes, there is no need to do a dictionary lookup, as we know ahead of time where everything is stored.

This optimization does not affect the logical correctness or functioning of the code and is fairly easy to implement and very readable to a programmer.

Performance Comparison

Each of the optimizations alone gave us moderate acceleration. Using both together, we got a significant speedup of about $10x$.

	Run time	Speed up (base/modified)
Base	7.30 ms +- 0.03 ms	-
Using __slots__	6.00 ms +- 0.02 ms	1.2x
Using gc.freeze()	3.93 ms +- 1.04 ms	1.8x
Using both	751 us +- 28 us	9.7x

Table 2 Performance comparison with different improvements

We can clearly see that these two mods work in synergy, lowering the amount of memory the Python Garbage collector must traverse to finish its job.

The performance improvements are also clear on the flamegraph, we can notice that the portion of time spent on the mark stage is now smaller than the sweep phase, while on the base benchmark, it was 11x longer. We can also notice that "dict_traverse" does not appear at all on the flamegraph now.

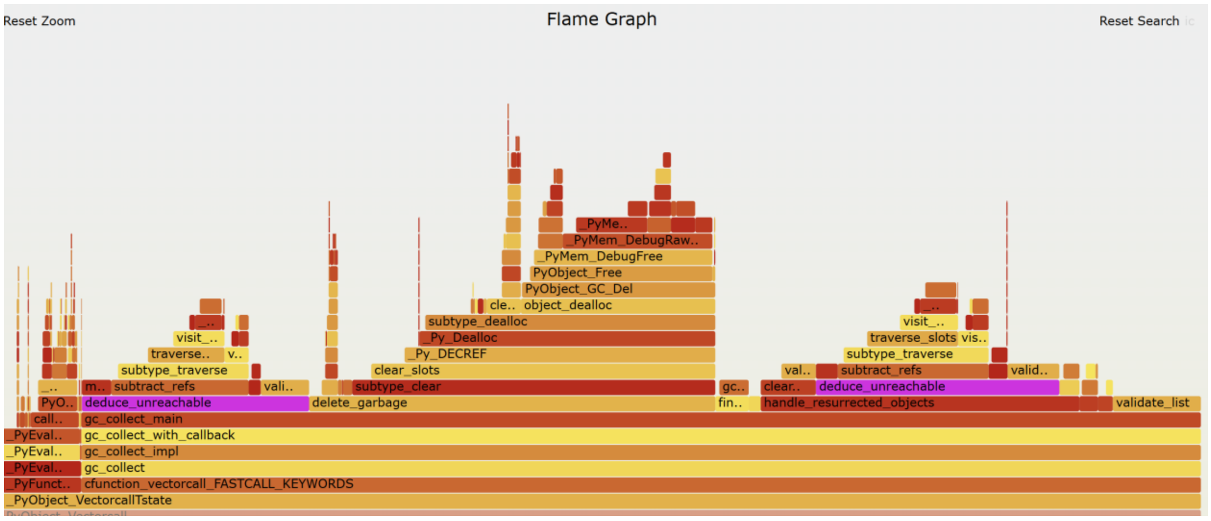

Figure 5 gc_collect_optimized Flamegraph

This change is reflected in the perf reports as well. Functions that took up most of the runtime now barely appear in the report.

Specific events inspection

The use of perf stats and basic perf record proved unhelpful in this project, as the pyperformance setup and preparation for the test itself added a lot of noise to the output data.

Instead, we created scripts named gc_profiler.py and gc_opt_profiler.py, each runs the benchmark 1000 times without the assistance of the Pyperformance suite. We then used "perf record -F 99 -e **event** -g python3-dbg file.py", capturing data about 4 different events, to create "perf report" that gives us specific data about what function caused the event. It allows us not only to measure the collective number of L1-load-hits (for example) but also to know which function caused them, allowing us to further filter out the irrelevant data.

The table below shows the data focusing on the function "visit_reachable", representing the mark stage of the gc.collect(), and L1-cache events, to gain a better understanding of the memory use. The data was recorded using scripts that run the "benchmark_collection" 1000 times.

The number of events was calculated using: $(Event\ count\ approx.) \cdot (\%\ of\ events\ in\ visit\_reachable)$

	Base	**Optimized**
L1-hit	12.28% of all L1 hits in run	1% of all L1 hits in run
	$1370 \cdot 10^6$	$49.2 \cdot 10^6$
L1-misses	14.32% of all L1 misses in run	0.46% of all L1 misses in run
	$139 \cdot 10^6$	$0.34 \cdot 10^6$
Total number of L1 accesses (hits+misses)	$1509 \cdot 10^6$	$49.5 \cdot 10^6$
% of L1 hits during marking	90%	99%

Table 3 visit_reachable L1 perf report data

We can notice a $30x$ improvement in the number of L1 requests after the optimizations.
As well as an improvement in the percentage of successful L1 requests.

Focusing on validate_list, another part of the gc_collect application, also shows good results. On both profiler runs, this was the function that caused the most L1-misses. L1-miss is inherently a performance bottleneck and causes memory delays.
On the un-optimized version, we observed $270 \cdot 10^6$ such misses inside "validate_list" while on the optimized version, we only $25 \cdot 10^6$ were recorded, a $10x$ decrease in L1 misses.

A similar improvement in both the number of calls and percentage of successful calls was observed for 'dTLB-loads' events as well. In the base profiler 20.84% (ie around $11 \cdot 10^6$) of the TLB-misses occur in the "visit_reachable" function. After our changes, only 0.15% occur in it (around 1000).
Besides 6.47% used to occur in "dict_traverse", after optimization, no samples of TLB misses inside "dict_travers" were recorded.

The measurable changes in L1 access and success rates, as well as the changes in TLB misses, show that the performance improvements we observe are indeed due to faster memory access.

Hardware Acceleration Proposal

Although several hardware accelerations can be beneficial to CPython garbage collection, it is important to note that not all of them would cause performance improvements on this benchmark due to its very sequential nature. For example, a second core used for running the GC in the background, coupled with CPython support, would speed up general GC work but not on this benchmark.

During the creation of this proposal, we took inspiration from a design that offloads the garbage collection (GC) from the CPU to a dedicated hardware accelerator, as is detailed in the article "A Hardware Accelerator for Tracing Garbage Collection" that is linked in the sources section.

A dedicated Garbage Collection hardware accelerator

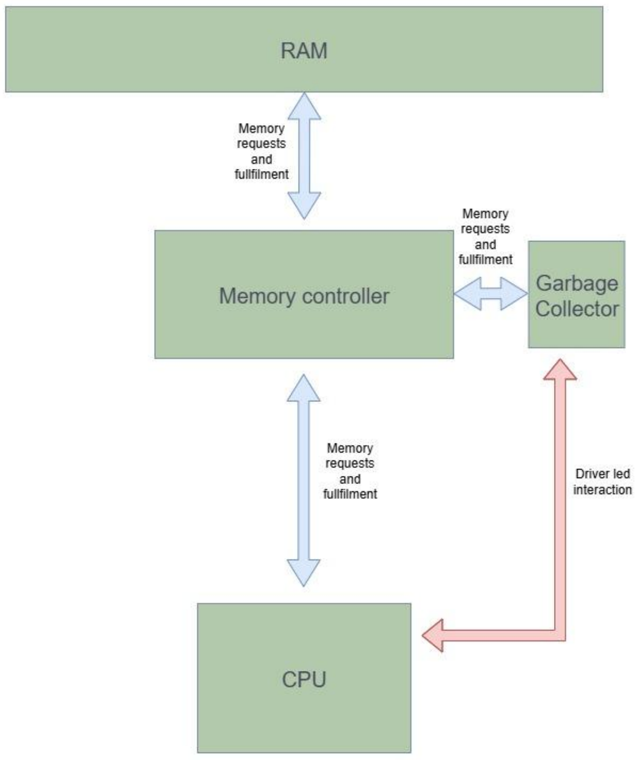
RAM
Memory requests and fullfilment
Memory requests and fullfilment
Memory controller
Garbage Collector
Memory requests and fullfilment
Driver led interaction
CPU

Figure 6 gc_collect hardware acceleration block diagram

As discussed above, the Mark and Sweep GC process makes poor use of the CPU and memory capabilities, not using locality. During the run, it uses the memory very heavily, creating a bottleneck while tracing the heap. This leads to a substantial portion of the CPU cycles being wasted on idle.

Our suggestion is a dedicated, simple core designed to handle the GC workload. This accelerator would be placed close to the memory controller to minimize data latency.

Upon calling gc.collect(), the CPU would provide the accelerator with a list of the root pointers to the tracked objects, heap memory bounds, and address space details.
The accelerator would then execute the mark stage, autonomously navigating the object graph, and mark all reachable objects. Then the sweep stage runs, scanning the heap and identifying unmarked objects, deallocating them, and freeing the memory.

To integrate this into a working system, a custom driver will be needed, interfacing the operating system and HW. The driver is the one in charge of passing relevant data to the acceleration unit. For CPython, the run-time also needs to be modified to be able to call this driver instead of running gc.collect() on the CPU. Internal TLB is also advised to reduce time spent on virtual-address to physical-address translation.

A possible problem that can accrue is “Bus contention”. The GC-accelerator is meant to perform a lot of memory accesses very fast, which can cause a bandwidth shortage and a “traffic jam”. For our intended use, garbage collection is a “stop-the-world” action, so this is less likely to happen. Several methods to deal with this exist, some prioritizing CPU memory calls and some the GC memory actions. The relevant method must be chosen based on the use case. An example is “Dynamic throttling”- the GC accelerator watches the memory bus and only uses it when the CPU is idle, ensuring the accelerator doesn't interfere with the main application.

Some other HW choices could help general gc.collect actions, but not necessarily in this benchmark: Another more invasive change is adding a few CPU instructions that would allow the CPU to directly initiate a GC run, pass parameters, and poll for completion status without needing to context switch into kernel mode. The driver would still be in charge of setup and error handling. Any such use, concurrent with the application, must consider synchronization considerations and carefully implement relevant locks and systems.

The small size and task-specific design would help save both time and energy.
When concurrency is allowed, these modifications would further improve performance, allowing the main CPU to run other tasks while the unit handles garbage collection.

Conclusions

This was a hard benchmark to optimize. We couldn’t use SIMD techniques, nor could we mask the gc.collect() runtime by running it in parallel with other things.

The gc.collect() function, when called directly, is memory-bound as the GC has to touch all the relevant objects.

While Python is a popular and rapidly growing language, it comes with the price of significant overhead. Managing the memory automatically while allowing the developer to focus on other things may be a powerful tool in some situations, but it comes with the cost of extra runtime and energy consumption.

Even when using Python, smart and efficient memory allocation, coupled with manually lowering the number of objects tracked by the GC, can dramatically improve performance, in this case by around 10x.

Faster memory access assisted by hardware can also improve performance.

While HW changes are not always easy, cheap, or possible, our implemented software changes are easy to understand, write, and read, making them ideal for development.

An interesting idea for further study is to create a software static code analysis tool that adds the __slots__ attribute to relevant classes to improve memory efficiency.

Sources

Python GC explanation:

https://www.datacamp.com/tutorial/python-garbage-collection

Object life cycle explanation:

https://docs.python.org/3.15/c-api/lifecycle.html

GC accelerator paper:

https://adept.eecs.berkeley.edu/wp-content/uploads/2018/06/A-Hardware-Accellerator-for-tracing-garbage-Collection-Maas-6-2018.pdf

Repository

YuvalMandel/PyperformancBenchmarkOptimization: Final Project of the Technion course 00460882 - Improving the performance of the pyaescrypto & gc_collect