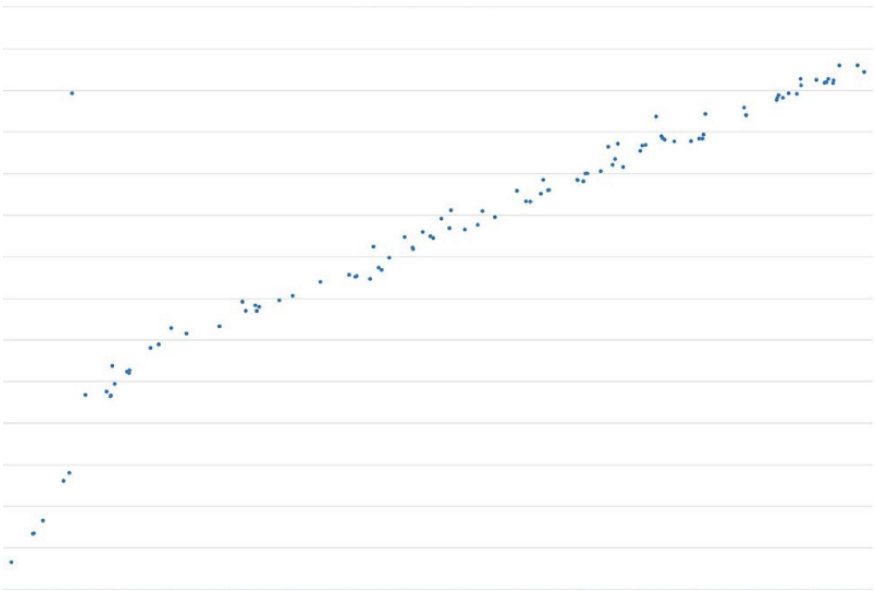
Instagram Engineering
Dec 21, 2017 · 4 min read

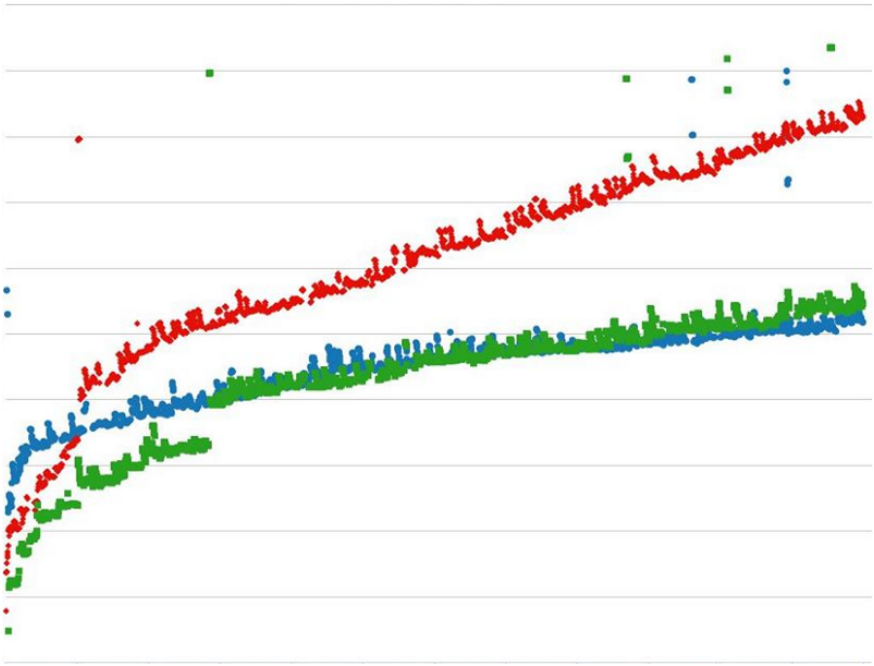
Copy-on-write friendly Python garbage collection

At Instagram, we have the world's largest deployment of the Django web framework, which is written entirely in Python. We began using Python early on because of its simplicity, but we've had to do many hacks over the years to keep it simple as we've scaled. Last year we tried dismissing the Python garbage collection (GC) mechanism (which reclaims memory by collecting and freeing unused data), and gained 10% more capacity. However, as our engineering team and number of features have continued to grow, so has memory usage. Eventually, we started losing the gains we had achieved by disabling GC.

Here's a graph that shows how our memory grew with the number of requests. After 3,000 requests, the process used ~600MB more memory. More importantly, the trend was linear.



From our load test, we could see that memory usage was becoming our bottleneck. Enabling GC could alleviate this problem and slow down the memory growth, but undesired Copy-on-write (COW) would still increase the overall memory footprint. So we decided to see if we could make Python GC work without COW, and hence, the memory overhead.



Red: without GC; Blue: calling GC collect explicitly; Green: default Python GC enabled

First try: redesign the GC head data structure

If you read our last GC post carefully, you'll notice the culprit of the COW was ahead of each python object:

```
/* GC information is stored BEFORE the object
structure. */
typedef union _gc_head
{
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        Py_ssize_t gc_refs;
    } gc;
    long double dummy; /* force worst-case alignment
*/
} PyGC_Head;
```

The theory was that every time we did a collection, it would update the gc_refs with ob_refcnt for all tracked objects—but unfortunately this write operation caused memory pages to be COW-ed. A next obvious solution was to move all the head to another chunk of memory and store densely.

We implemented a version where the pointer in the gc_head struct didn't change during collection:

```
typedef union _gc_head_ptr
{
    struct {
        union _gc_head *head;
    } gc_ptr;
    double dummy; /* force worst-case alignment */
} PyGC_Head_Ptr;
```

Did it work? We used the following script to allocate the memory and fork a child process to test it:

```
lists = []
strs = []

for i in range(16000):
    lists.append([])
    for j in range(40):
        strs.append(' ' * 8)
```

With the old gc_head struct, the child process's RSS memory usage increased by ~60MB. Under the new data structure with the additional pointer, it only increased by ~0.9 MB. So it worked!

However, you may have noticed the additional pointer in the proposed data structure introduced memory overhead (16 bytes—two pointers). It seems like a small number, but if you consider it applied to every collectable Python object (and we usually have millions of objects in one process, with ~70 processes per host), it could be a fairly big memory overhead on each server.

> *16 bytes* 1,000,000 * 70 = ~1 GB*

Second try: hiding shared objects from GC

Even though the new gc_head data structure showed promising gains on memory size, its overhead was not ideal. We wanted to find a solution that could enable the GC without noticeable performance impacts. Since our problem is really only on the shared objects that are created in the master process before the child processes are forked, we

tried letting Python GC treat those shared objects differently. In other words, if we could hide the shared objects from the GC mechanism so they wouldn't be examined in the GC collection cycle, our problem would be solved.

For that purpose, we added a simple API as gc.freeze() into the Python GC module to remove the objects from the Python GC generation list that's maintained by Python internal for tracking objects for collection. We have upstreamed this change to Python and the new API will be available in the Python3.7 release (https://github.com/python/cpython/pull/3705).

```
static PyObject *
gc_freeze_impl(PyObject *module)
{
    for (int i = 0; i < NUM_GENERATIONS; ++i) {
        gc_list_merge(GEN_HEAD(i),
&_PyRuntime.gc.permanent_generation.head);
        _PyRuntime.gc.generations[i].count = 0;
    }
    Py_RETURN_NONE;
}
```

Success!

We deployed this change into our production and this time it worked as expected: COW no longer happened and shared memory stayed the same, while average memory growth per request dropped **~50%.** The plot below shows how enabling GC helped the memory growth by stopping the linear growth and making each process live longer.

Blue: is no-GC; Red: auto-GC

Credits

Thanks to Jiahao Li, Matt Page, David Callahan, Carl S. Shapiro, and Chenyang Wu for their discussions and contributions to the COW-friendly Python garbage collection.

Zekun Li is an infrastructure engineer at Instagram.