Instagram Engineering Follow
Jan 18, 2017 · 8 min read

Dismissing Python Garbage Collection at Instagram

By dismissing the Python garbage collection (GC) mechanism, which reclaims memory by collecting and freeing unused data, Instagram can run 10% more efficiently. Yes, you heard it right! By disabling GC, we can reduce the memory footprint and improve the CPU LLC cache hit ratio. If you're interested in knowing why, buckle up!

How We Run Our Web Server

Instagram's web server runs on Django in a multi-process mode with a master process that forks itself to create dozens of worker processes that take incoming user requests. For the application server, we use uWSGI with pre-fork mode to leverage memory sharing between master and worker processes.

In order to prevent the Django server from running into OOM, the uWSGI master process provides a mechanism to restart the worker processes when its RSS memory exceeds the predefined limits.

Understanding Memory

We started by looking into why worker RSS memory grows so fast right after it is spawned by the master process. One observation is that even though the RSS memory starts with 250MB, its shared memory drops very quickly—from 250MB to about 140MB within a few seconds (shared memory size can be read from `/proc/PID/smaps`). The numbers here are uninteresting because they change all the time, but the scale of shared memory dropping is very interesting—about 1/3 of the total memory. Next we wanted to understand why this shared memory becomes private memory per process at the beginning of the worker spawning.

Our theory: Copy-on-Read

Linux kernel has a mechanism called Copy-on-Write (CoW) that serves as an optimization for forked processes. A child process starts by sharing every memory page with its parent. A page copied to the child's memory space only when the page is written to (for more details refer to the wiki https://en.wikipedia.org/wiki/Copy-on-write).

But in Python land, because of reference counting, things get interesting. Every time we read a Python object, the interpreter will increase its refcount, which is essentially a write to its underlying data structure. This causes CoW. So with Python, we're doing Copy-on-Read (CoR)!

```
#define PyObject_HEAD                   \
    _PyObject_HEAD_EXTRA                \
    Py_ssize_t ob_refcnt;               \
    struct _typeobject *ob_type;

...

typedef struct _object {
    PyObject_HEAD
} PyObject;
```

So the question is: are we copy-on-writing immutable objects such as the code objects? Given `PyCodeObject` is indeed a "sub-class" of `PyObject`, apparently yes. Our first thought was to disable the reference counting on `PyCodeObject`.

Attempt 1: Disable reference count on code objects

At Instagram, we do the simple thing first. Given that this was an experiment, we made some small but hacky changes to CPython interpreter, verified the reference count number didn't change on code object, and then shipped that CPython to one of our production servers.

The result was disappointing because there was no change on shared memory. When we tried to figure out why, we realized we couldn't find any reliable metrics to prove our hack worked, nor could we prove the connection between the shared memory and the copy of code objects. Apparently, something was missing here. Lesson learned: prove your theory before going for it.

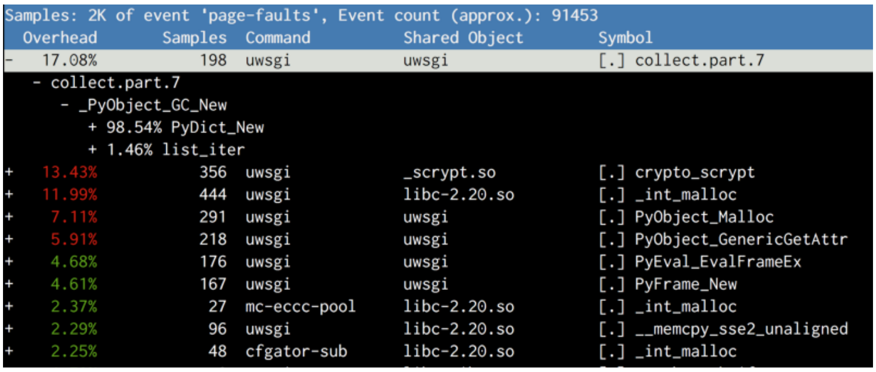
Profiling page faults

After some googling on Copy-on-Write, we learned Copy-on-Write is associated with page faults in the system. Each CoW triggers a page fault in the process. Perf tools that come with Linux allow recording hardware/software system events, including page faults, and can even provide stack trace when possible!

So we went to a prod server, restarted the server, waited for it to fork, got a worker process PID, and then ran the following command.

```
perf record -e page-faults -g -p <PID>
```

Then, we got an idea about when page faults happen in the process with stack trace.



The results were different than our expectations. Rather than copying the code object, the top suspect is `collect`, which belongs to `gcmodule.c`, and is called when a garbage collection is triggered. After reading how GC works in CPython, we have the following theory:

CPython's GC is triggered deterministically based on the threshold. The default threshold is very low, so it kicks in at a very early stage. It maintains linked lists of generations of objects, and during GC, the linked lists are shuffled. Because the linked list structure lives with the object itself (just like `ob_refcount`), shuffling these objects in the linked lists will cause the pages to be CoWed, which is an unfortunate side effect.

```
/* 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;
```

Attempt 2: Let's try disabling GC

Well, since GC is backstabbing us, let's disable it!

We added a gc.disable() call to our bootstrapping script. We restarted the server, but again, no luck! If we look at perf again, we'll see `gc.collect` is still called, and the memory is still copied. With some debugging with GDB, we found that apparently one of the third-party libraries we used (msgpack) calls `gc.enable()` to bring it back, so gc.disable() at bootstrapping was washed.

Patching msgpack is the last thing we would do because it leaves the door for other libraries to do the same thing in the future without us noticing. First, we need to prove disabling GC actually helps. The answer again lives in `gcmodule.c.` As an alternative to `gc.disable`, we did `gc.set_threshold(0)`, and this time, no libraries brought it back.

With that, we successfully raised the shared memory of each worker process from 140MB to 225MB, and the total memory usage on the host dropped by 8GB per machine. This saved 25% RAM for the whole Django fleet. With such big head room, we're capable of running a lot more processes or running with a much higher RSS memory threshold. In effect, this improves the throughput of Django tier by more than 10%.

Attempt 3: Completely shutdown GC takes churns

After we experimented with a bunch of settings, we decided to try it on a larger scale: a cluster. The feedback was pretty quick, and our continuous deployment broke because restarting our web server became much slower with GC disabled. Usually restarting takes less than 10 seconds, but with GC disabled, it sometimes took more than 60 seconds.

```
2016-05-02_21:46:05.57499 WSGI app 0 (mountpoint='')
ready in 115 seconds on interpreter 0x92f480 pid:
4024654 (default app)
```

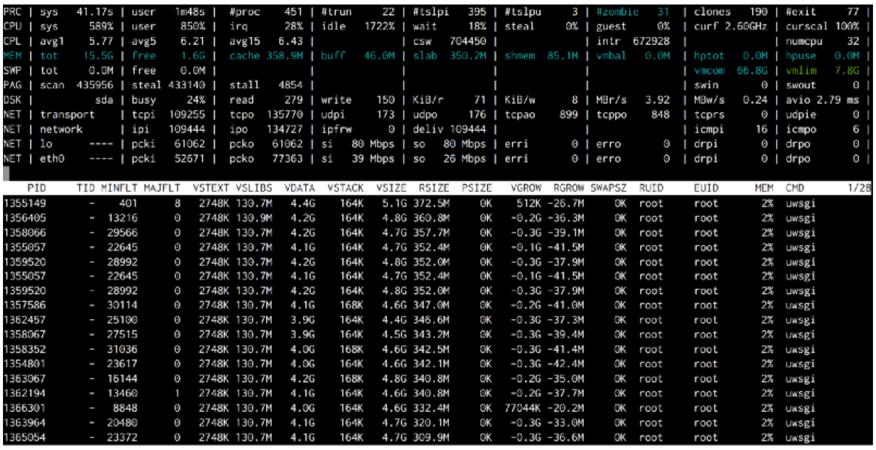
It was very painful to re-produce this bug because it's not deterministic. After a lot of experiments, a real re-pro shows in atop. When this happened, the free memory on that host dropped to nearly zero and jumped back, forcing out all of the cached memory. Then came the moment where all the code/data needed to be read from disk (DSK 100%), and everything was slow.

This rung a bell that Python would do a final GC before the interpreter shut down, which would cause a huge jump in memory usage in a very short period of time. Again, I wanted to prove it first, then figure out how to deal with it properly. So, I commented out the call to `Py_Finalize` in uWSGI's python plugin, and the problem disappeared.

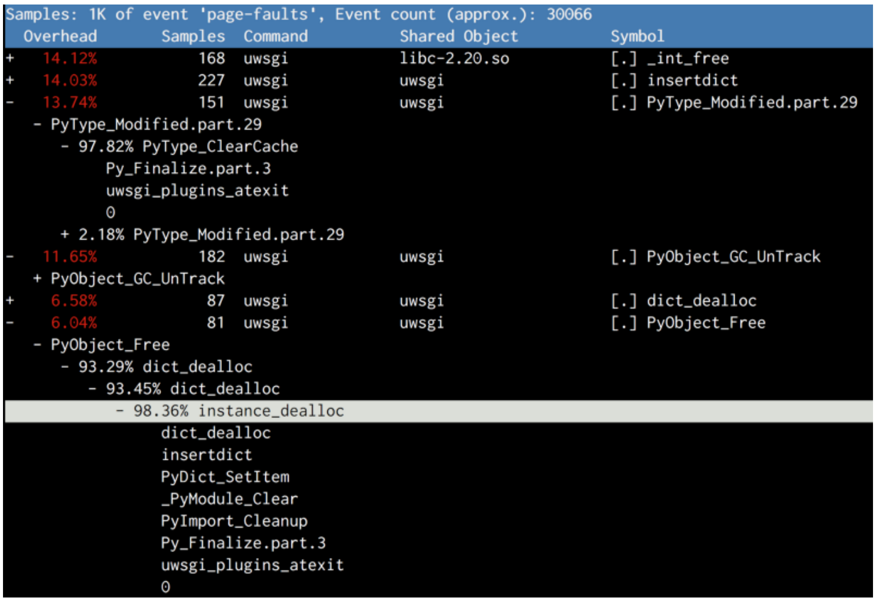
But apparently we couldn't just disable `Py_Finalize` as it was. We had a bunch of important cleanups using atexit hooks that relied on it. What we ended up doing is adding a runtime flag to CPython that would disable GC completely.

Finally, we got to roll it out to a larger scale. We tried our entire fleet after this, but the continuous deployment broke again. However, this time it only broke on machines with old CPU models (Sandybridge), and was even harder to re-pro. Lesson learned: always test the old clients/models because they're often the easiest ones to break.

Because our continuous deployment is a fairly fast procedure, to really catch what happened, I added a separate `atop` to our rollout command. We're able to catch a moment where cache memory goes really low, and all of uWSGI processes trigger a lot of MINFLT (minor page faults).



Again, by perf profiling, we saw `Py_Finalize` again. Upon shutdown, other than the final GC, Python did a bunch of cleanup operations, like destroying type objects and unloading modules. Again, this hurt shared memory.



Attempt 4: Final step for shutting down GC: No cleanup

Why do we need to clean up at all? The process is going to die, and we're going to get another replacement for it. What we really care about is our atexit hooks that do cleanup for our apps. As to Python's cleanup, we don't have to do it. This is what we ended up with in our bootstrapping script:

```
# gc.disable() doesn't work, because some random 3rd-
party library will
# enable it back implicitly.
gc.set_threshold(0)
# Suicide immediately after other atexit functions
finishes.
# CPython will do a bunch of cleanups in Py_Finalize
which
# will again cause Copy-on-Write, including a final GC
atexit.register(os._exit, 0)
```

This is based on that fact atexit functions run in the reverse order of registry. The atexit function finishes the other cleanups, then calls `os._exit(0`) to exit the current process in the last step.

With this two-line change, we finally finished rolling it out to our entire fleet. After carefully adjusting the memory threshold, we got a 10% global capacity win!

Looking back

In reviewing this performance win, we had two questions:

First, without garbage collection, wasn't the Python memory going to blow up, as all memory allocation wouldn't be freed ever? (Remember, there is no real stack in Python memory because all objects are allocated on heap.)

Fortunately, this was not true. The primary mechanism in Python to free objects is still reference count. When an object is de-referenced (calling `Py_DECREF`), Python runtime always checks if its reference count drops to zero. In such cases, the deallocator of the objects will be called. The main purpose of garbage collection is to break the reference cycles where reference count does not work.

```
#define Py_DECREF(op)
\
    do {
\
        if (_Py_DEC_REFTOTAL  _Py_REF_DEBUG_COMMA
\
        --((PyObject*)(op))->ob_refcnt != 0)
\
            _Py_CHECK_REFCNT(op)
\
        else
```

```
\
        _Py_Dealloc((PyObject *)(op));
\
    } while (0)
```

Breaking down the gains

Second question: where did the gain come from?

The gain of disabling GC was two fold:

- We freed up about 8GB RAM for each server we used to create more worker processes for memory-bound server generation, or lower the worker respawn rate for CPU-bound server generation;
- CPU throughput also improved as CPU instructions per cycle (IPC) increased by about 10%.

```
# perf stat -a -e cache-misses,cache-references --
sleep 10

 Performance counter stats for 'system wide':

       268,195,790      cache-misses              #
12.240 % of all cache refs     [100.00%]
     2,191,115,722      cache-references

      10.019172636 seconds time elapsed
```

With GC disabled, there was a 2–3% of cache-miss rate drop, which was the main reason behind the 10% IPC improvement. CPU cache miss is expensive because it stalls CPU pipeline. Small improvements on the CPU cache hit rate can usually improve IPC significantly. With less CoW, more CPU cache lines with different virtual addresses (in different worker processes) point to the same physical memory address, leading to better cache hit rate.

As we can see, not every component worked as expected, and sometimes, the results can be very surprising. So keep digging and sniffing around, and you'll be amazed how things really work!

Chenyang Wu is a software engineer and Min Ni is an engineering manager at Instagram .