Multithreaded Python without the GIL

(abridged from Sam Gross’s doc: https://docs.google.com/document/d/18CXhDb1ygxg-YXNBJNzfzZsDFosB5e6BfnXLlejd9l0/edit?tab=t.0#heading=h.kcngwrty1lv)

The goal of this project is to remove the global interpreter lock (GIL) from CPython to enable multiple threads to execute Python code in parallel. The purpose of this document and associated [code](https://github.com/colesbury/nogil) is to convince you of a few things:

1. That it is feasible to remove the GIL from Python.
2. That the tradeoffs are manageable and the effort to remove the GIL is worthwhile.
3. That the main technical ideas of this project (reference counting, allocator changes, and thread-safety scheme) should serve as a basis of such an effort.

# Why remove the GIL?

The GIL is a major obstacle to concurrency. For scientific computing tasks, this lack of concurrency is often a bigger issue than speed of executing Python code, since most of the processor cycles are spent in optimized CPU or GPU kernels. The GIL introduces a global bottleneck that can prevent other threads from making progress if they call any Python code.

# Design

## Reference counting

CPython’s reference counting system is not thread-safe without the GIL. The simplest change would be to replace *non-atomic* reference count operations with their *atomic* equivalents. However, atomic instructions are more expensive than their non-atomic counterparts. Replacing Py\_INCREF and Py\_DECREF with atomic variants would result in a 60% average slowdown on the *pyperformance* benchmark suite.

This project uses “**biased reference counting**” ([Choi, 2018](https://dl.acm.org/citation.cfm?id=3243195)). Each object is associated with an owning thread (the thread that created it). Reference counting operations from the owning thread use non-atomic instructions to modify a “local” reference count. Other threads use atomic instructions to modify a “shared” reference count. The owning thread can combine the two fields and give up ownership when the local reference count goes to zero.

### Immortalization

Some objects, such as interned strings, small integers, statically allocated PyTypeObjects, and the True, False, and None objects stay alive for the lifetime of the program. These objects are marked as **immortal** by setting the least-significant bit of the local reference count field (bit 0). The Py\_INCREF and Py\_DECREF macros are no-ops for these objects. This avoids contention on the reference count fields of these objects when they are accessed concurrently by multiple threads.

CPython does not currently support immortalization for statically allocated objects because the cost of supporting it (extra branches) is slightly larger than the savings from avoiding reference count operations in a predominantly single-threaded environment[[1]](#footnote-2). The tradeoff changes in a multi-threaded/multi-core environment: avoiding contention on reference count fields becomes much more important.

### Deferred reference counting

A few types of objects, such as top-level functions, code objects, and modules tend to be frequently accessed by many threads concurrently, but don’t necessarily live for the lifetime of the program, so immortalization is not a good fit. This project uses a variant of **deferred reference counting** to avoid contention on the reference count fields of these objects in multi-threaded programs. (For a general description of deferred reference counting see [this](https://www.memorymanagement.org/glossary/d.html" \l ":~:text=Deferred%20reference%20counting).)

## Performance

As mentioned above, the no-GIL proof-of-concept interpreter is about 10% faster[[2]](#footnote-3) than CPython 3.9[[3]](#footnote-4) (and 3.10) on the pyperformance benchmark suite. It gets about the same average performance as the “main” branch of CPython 3.11 as of early September 2021.

Stripping out some of the GIL-removal related changes would result in even faster performance on the single-threaded pyperformance benchmark suite. Of course, the resulting interpreter can no longer safely run without the GIL, but this exercise provides an estimate for how much the GIL-removal changes “cost” compared to a hypothetical CPython interpreter optimized solely for single-threaded performance. I’ve attempted to remove the features that have a substantial negative impact on single-threaded performance, but are necessary for thread-safety without the GIL.

The resulting interpreter is about 9% faster than the no-GIL proof-of-concept (or ~19% faster than CPython 3.9.0a3). That 9% difference between the “nogil” interpreter and the stripped-down “nogil” interpreter can be thought of as the “cost” of the major GIL-removal changes. The approximate breakdown by feature is:

* 1.5% - per-object mutexes in collections (dict, list, queue)
* 4% - biased reference counting and deferred reference counting
* 2% - global free-lists (mostly tuple and float free-lists)
* 1% - immortalization

Note that the above breakdown is very much an approximation, and that the features interact in complicated ways. Changing the order in which features are added (or removed) would change the breakdown. Additionally, the reference counting changes would be more expensive to implement directly in CPython without the other interpreter changes described in the previous section.

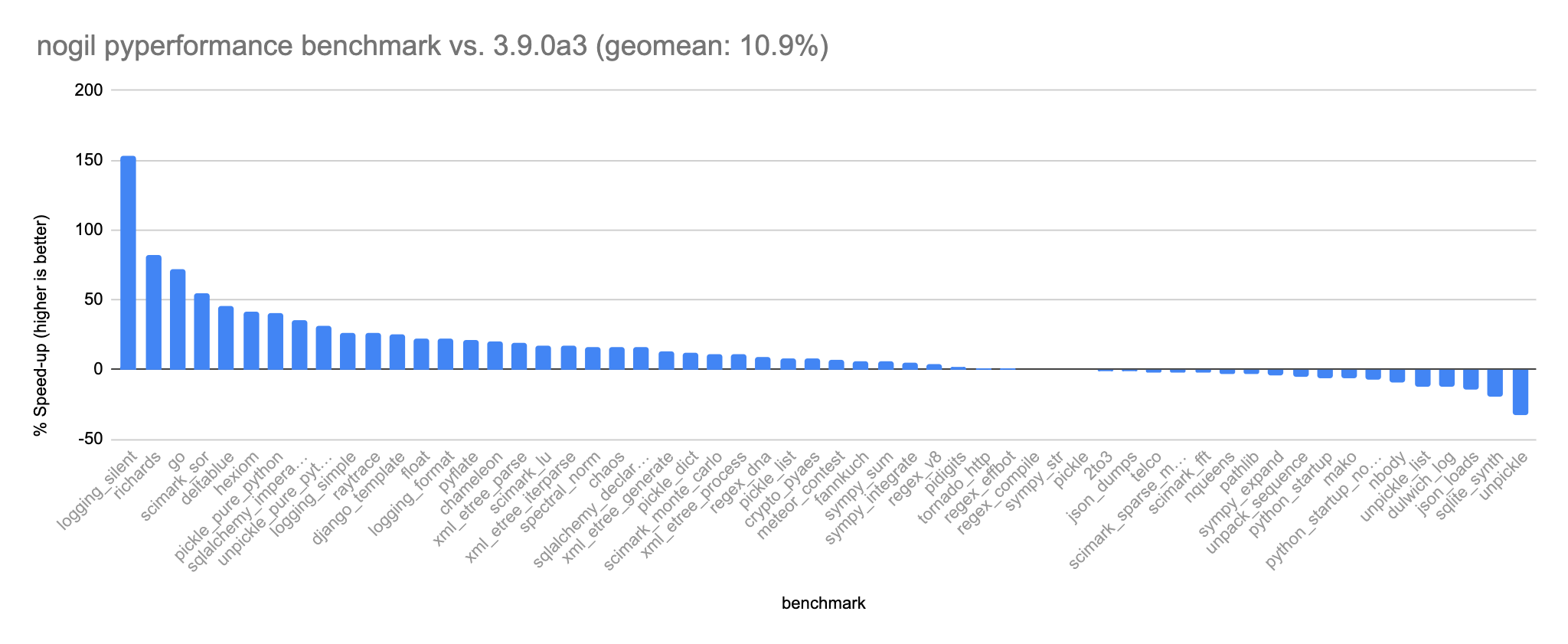


Figure 1: nogil speed-up on the pyperformance benchmark suite compared to Python 3.9.0a3. The geometric mean of the speed-up is 10.9% across 58 benchmarks. The fastest speed-up is on “logging\_silent” due to faster function calls. The largest regression (-32%) is on the unpickle benchmark.

Compilation used GCC 9 with PGO enabled.

1. Cinder (Facebook’s fork of CPython) supports immortalization to avoid copy-on-writes in a fork-based multi-process environment. [↑](#footnote-ref-2)
2. Geometric average cross 58 benchmarks [↑](#footnote-ref-3)
3. The proof-of-concept is based on CPython 3.9.0a3 [↑](#footnote-ref-4)