Multithreaded Python without the GIL

Sam Gross

[colesbury@gmail.com](mailto:colesbury@gmail.com) / [sgross@fb.com](mailto:sgross@fb.com)

Source code:

<https://github.com/colesbury/nogil>

Questions? Comments? Please post in:

<https://github.com/colesbury/nogil/discussions>

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.

The associated code constitutes a working proof-of-concept of a CPython-based interpreter that supports running multiple threads without the GIL. The main purpose of the code is to enable discussions about removing the GIL by answering questions about compatibility and performance implications, but it may also be useful as a basis for eventual changes to CPython.

Removing the GIL will require making trade-offs and taking on substantial risk:

1. Removing the GIL will be a large, multi-year undertaking with increased risk for introducing bugs and regressions.
2. Removing the GIL means making trade-offs between single and multi-threaded performance. I believe we can make a future CPython without the GIL that is faster for both single and multi-threaded workloads than current CPython, but if we keep the GIL and focus solely on single-threaded workloads, we would achieve even better single-threaded performance. (See the “[Performance](#_6i2dwx2chdq9)” section for more details.)
3. Most Python programs are not optimized for multi-threaded performance; they will likely require changes to take advantage of running without the GIL.
4. Multithreaded programs are prone to concurrency bugs. Although Python already supports threads (with the GIL), successfully removing the GIL may increase the use of threads and consequently Python programmers exposure to concurrency bugs.

Currently, Python supports a number of ways to enable parallelism, but they all come with significant limitations. These techniques, which include the existing threading library, multiprocessing, and moving code to C, are described in the “Existing Techniques” section.

There are other efforts to remove or replace the GIL. The Gilectomy project also aimed to remove the GIL, but introduced substantial performance regressions. PEP 554 (multiple interpreters) aims to replace the global interpreter lock with a per-interpreter lock. Multiple interpreters are less flexible than multiple threads because of the strict limits on sharing Python data structures between interpreters.

The project aims for a concurrency model that matches the threads + shared memory model implemented in common operating systems and in programming languages like Java, C++, and Swift[[1]](#footnote-0). We also aim for similar safety guarantees to Java and Go -- the language doesn’t prevent data races, but data races in user code do not corrupt the VM state. (Notably, Swift and C++ do not have this property: data races in those languages may corrupt VM state leading to segfaults). This strikes a balance between safety and performance. Weaker guarantees, like in Swift and C++, can make debugging difficult, while much stronger guarantees can limit flexibility or hamper performance.

The shared-memory model is notoriously difficult to program correctly, but provides a good base to build better abstractions[[2]](#footnote-1) becauses it closely matches the primitives provided by the operating system (e.g. threads, mutexes).

# 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.

In the “Why Swift for Tensorflow” doc, Chris Lattner writes “the GIL increases complexity throughout the stack in large scale settings: it prevents model authors from achieving reasonable performance for some things (without themselves writing C++/CUDA), and prevents natural expression of concurrent algorithms in a model.”[[3]](#footnote-2)

A lot of engineering effort has gone into working around the GIL. One of the consequences of this is that it’s unlikely that simply removing the GIL will magically speed-up existing code. However, I believe that removing the GIL will make it easier to take advantage of parallelism, simplify development of new libraries, and allow us to improve existing libraries.

# Existing Techniques for Parallelism in CPython

## Multiprocessing

The multiprocessing library allows Python programs to start and communicate with Python sub-processes. This allows for parallelism because each sub-process has its own Python interpreter (i.e there’s a GIL per-process). Communication between processes is limited and generally more expensive than communication between threads. Objects generally need to be serialized or copied to shared memory. Starting a sub-process is also more expensive than starting a thread, especially with the “spawn” implementation. Starting a thread takes ~100 µs, while spawning a sub-process takes ~50 ms (50,000 µs) due to Python re-initialization.

The multiprocessing library has a number of downsides. The “fork” implementation is prone to deadlocks. Multiple processes are harder to debug than multiple threads. Sharing data between processes can be a performance bottleneck. For example, in PyTorch, which uses multiprocessing to parallelize data loading, copying Tensors to inter-process shared-memory is sometimes the most expensive operation in the data loading pipeline. Additionally, many libraries don’t work well with multiple processes: for example CUDA doesn’t support “fork” and has limited IPC support.

## Threading

Python provides a threading library that maps to OS threads[[4]](#footnote-3). It also provides locks (mutexes), condition variables, and thread-local data. Parallelism is limited due to the GIL, since only one thread may execute Python code at a time. It’s still sometimes useful for I/O and compute-bound operations implemented in C that can release the GIL.

## Internal parallelization in C code

Functions implemented in C may use multiple threads internally. For example, Intel’s NumPy distribution, PyTorch, and TensorFlow all use this technique to internally parallelize individual operations. This works well when the basic operations are large enough to be parallelized efficiently, but not when there are many small operations or if the operations depend on some Python code. Calling into Python from C requires acquiring the GIL -- even short snippets of Python code can inhibit scaling. Addressing this issue sometimes requires rewriting large portions of Python code in C/C++ to actually achieve the desired parallelism. For example, PyTorch rewrote all of the core automatic differentiation from Python to C++ to avoid acquiring the GIL in the backward pass.

# 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.

Biased reference counting is performant for accesses to same-thread objects. Most accesses fall into this category. However, a few categories of objects are frequently accessed by multiple threads. This project uses two techniques: **immortalization** and **deferred reference counting** to improve the performance of these accesses. The implementations of these techniques make use of the two least significant bits of the local reference count field (bit 0 and bit 1)[[5]](#footnote-4). This means that Py\_INCREF and Py\_DECREF operations adjust the reference count field by 4, but that’s invisible to code that doesn’t access the fields directly[[6]](#footnote-5).

### 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[[7]](#footnote-6). 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#:~:text=Deferred%20reference%20counting).)

Deferred reference counting is indicated by the second-most least-significant bit in the local reference count field. Loads of these objects to the interpreter stack do not modify the objects’ reference counts. These objects are only deallocated during garbage collection. That might be problematic for most objects, but deferred reference counting is only used for objects that typically would only be deallocated during garbage collection anyways. (For example, top-level functions and modules form a reference cycle with their globals dictionary.)

Handling of deferred reference counting is limited to a few places, like the interpreter and the garbage collection system. Code that is unaware of deferred references treats these objects just like other objects: calls to Py\_INCREF and Py\_DECREF behave normally on deferred reference counted objects. This is important for compatibility. Even though these objects are only collected during GC cycles, there may be references to the objects outside the interpreter that are invisible to the GC. The Py\_INCREF calls outside the interpreter remain necessary to ensure that the objects are not freed too soon.

## Memory allocator

Python’s built-in “pymalloc” memory allocator is not thread-safe. This project replaces pymalloc with **mimalloc** [[link](https://github.com/microsoft/mimalloc)], a memory allocator developed by Daan Leijen at Microsoft. Mimalloc generally has better performance than pymalloc and is thread-safe. Additionally, mimalloc’s layout of objects in the heap allows for finding all GC-tracked objects without maintaining an explicit list, which isn’t possible to do efficiently with *pymalloc* or other allocators built on top of *malloc*. Finally, the design of *mimalloc* enables a thread-safe implementation of dictionaries and other collections that avoids locking during most non-mutating access.

This design mostly precludes swapping allocators with the **PyMem\_SetAllocator** API. For example, you can’t replace *mimalloc* with another custom allocator without breaking the garbage collection and thread-safe collection implementations. However, allocators that “layer” on top of the underlying allocator, like tracemalloc and the debug allocator still work.

### Garbage collection

The garbage collector uses a single-threaded, stop-the-world implementation. The garbage collector uses the **eval\_breaker** mechanism to signal and wait for other threads to reach and pause at a safe point. (Any bytecode boundary constitutes a safe-point). The existing GIL APIs are re-purposed to support the garbage collector. The garbage collector doesn’t wait on threads that call **PyEval\_ReleaseThread** (ones that would have released the GIL). This is important so that threads blocked on I/O don’t prevent garbage collection progress.

CPython uses a global linked-list of objects tracked by the garbage collector. It’s difficult to efficiently maintain a shared linked-list in a thread-safe manner, so this project does away with the linked-list. Instead, GC tracked objects are maintained in a separate *mimalloc* heap. (*mimalloc* heaps are lightweight.) The garbage collector scans each block in the heap to find GC-tracked objects. Unallocated blocks and GC objects that are currently untracked have the least-significant-bit (LSB) of the first word set to zero; GC-tracked objects have the LSB set to one. To reduce changes to the garbage collection code, the implementation currently constructs the linked-list during the scan of the heap. With additional changes, the linked-list could probably be entirely removed to improve performance.

## Method resolution on types

The method resolution order (MRO) determines the order in which Python searches the type hierarchy to resolve a method invocation to a concrete function. CPython caches the results of these lookups in a global hash table (the MRO cache). The cache is not thread-safe and multiple threads may try to perform MRO lookups concurrently. This project makes the cache per-thread (accessible by PyThreadState). Invalidations (**PyType\_Modified**) are guarded by a global (in PyRuntime) mutex. The cache adds about 96 KiB of overhead per-thread. (For comparison, the typical stack size is about 2 MiB per-thread).

## Collection thread-safety

This section describes the implementation of thread-safe collections that avoid locks and expensive atomic operations in the typical “fast path” for read accesses. This design is implemented for the **list** and **dict** classes. Although the design is thread-safe, it’s optimized primarily for single-threaded access because these workloads are most common. Collections that require frequent concurrent modifications would be better addressed by a different design.

The design uses a per-collection lock to protect against concurrent modifications. The cost of a lock for modifications is relatively small; modifications are less frequent than read-only accesses (a typical ratio is roughly 1:10) and more expensive (so the lock acquisition cost is a relatively smaller part of the operation).

Avoiding the lock for read-only accesses is tricky due to Python’s reference counting system. Reference counting poses additional hazards for concurrent read and write operations. The steps for retrieving an item (object) from a collection can be generalized as:

1. Load the address of the item
2. Increment the reference count of the item
3. Return the address of the item

The additional hazard is that if steps (1) and (2) are not performed together as an atomic action (e.g. because of the GIL) then a concurrent write operation may replace the item in-between steps (1) and (2), decrement the reference count and potentially free or reuse the underlying memory for another allocation. When the retrieval operation continues with step (2), it may crash the program or corrupt memory when it tries to increment the reference count of an object that no longer exists. Note that atomic reference counting doesn’t sufficiently address this issue; it’s necessary for the entire retrieval operation to behave as-if it were atomic. In CPython, the GIL provides this behavior. It’s also possible to get the same behavior by acquiring a per-collection lock around *both* read and write operations (but in this design we’d like to avoid the lock around read operations).

The design is described as a series of incremental changes to the above algorithm to address the above hazard.

First, we change the second step to a *conditional* incref, to handle the case where the reference count is zero:

1. Load the address of the item
2. Increment the reference count of the item, if it is non-zero (otherwise retry the entire operation)
3. Return the address of the item

(We typically acquire the per-collection mutex when retrying operations to avoid potential reader starvation issues.[[8]](#footnote-7))

This still leaves unresolved the issues due to the underlying memory being freed and potentially re-used or returned to the operation system. To address this, we add an additional constraint to the system: the memory underlying Python objects (PyObject) can only be re-used for other Python objects of the same size (and the memory can’t be immediately returned to the operating system). This ensures that the reference count field remains valid, even when the memory is re-used for a new Python object. We also add an additional step to the algorithm to avoid returning objects reusing the same memory, but not in the collection.

1. Load the address of the item
2. Increment the reference count of the item, if it is non-zero (otherwise retry)
3. Verify that the item still exists at the same location in the collection (otherwise retry)
4. Return the address of the item

*Constraint:* *the memory allocator can only re-use memory for Python objects of the same size (for the duration of the read operation)*

This constraint (a variation of *type stable memory management*[[9]](#footnote-8)) is less onerous than it might sound. First, even though re-use is constrained, destructors are still called immediately: the behavior visible to the Python programmer is not delayed. Second, *mimalloc* already enforces similar constraints for performance reasons. The allocator already avoids returning memory to the operating system immediately because *mmap* and *munmap* calls (and the equivalent calls on Windows) are expensive. *Mimalloc* also typically re-uses memory for allocations of the same *size class[[10]](#footnote-9)* because free lists are organized by size. Only once all allocations in a *mimalloc “page” (“superblock”)[[11]](#footnote-10)* are freed can the “page” be re-used for allocations of a different size class. This project modifies mimalloc to tag “pages” (“superblocks”) when they become empty and to only re-use “pages” from a different size class when it’s tag is old enough to indicate that there are no stale pointers to Python objects previously contained in the page.[[12]](#footnote-11)

An additional wrinkle is that Python collections are resizable and store their items through an indirection: lists store their items in an array of Python object pointers and dicts store their items in a PyDictKeysObject. These “backing arrays” aren’t Python objects and may be freed when the collection is resized. This requires adding additional steps and a constraint to the retrieval algorithm:

1. Load the version counter from the collection
2. Load the “backing array” from the collection
3. Load the address of the item (from the “backing array”)
4. Increment the reference count of the item, if it is non-zero (otherwise retry)
5. Verify that the item still exists at the same location in the collection (otherwise retry)
6. Verify that the version counter did not change (otherwise retry)
7. Return the address of the item

*Constraint:* *the memory allocator can only re-use memory for Python objects of the same size (for the duration of the read operation)*

*Constraint:* *the memory allocator can only re-use memory for backing arrays of the same size and collection type (for the duration of the read operation)*

Additionally, operations that reallocate the backing array (resize operations) must increment the version counter *after* storing the new backing array, but *before* freeing the previous backing array.[[13]](#footnote-12)

Finally, there is an optimization to the above algorithm with biased reference counting. If the item is “immortal” or “local” to the reading thread (the most common case), then the reader can skip step 5 (checking that the item still exists in the collection) because no other thread can deallocate the item while it remains “local” to the reading thread.

The actual implementation in the **list** and **dict** objects differs slightly from the above generalized algorithm. For example, the list implementation relies on monotonically increasing sizes instead of version counters, but this can (and probably should) be changed.

## Thread states and GIL API

The GIL APIs (e.g. PyEval\_AcquireThread/ReleaseThread) are still useful, even though they no longer acquire a global lock. Threads can be in an “attached” or “detached” state. PyEval\_AcquireThread marks a thread as “attached” and PyEval\_ReleaseThread marks a thread as “detached”. The garbage collector uses these states to avoid waiting on threads in the “detached” state which may be blocked on I/O. Additionally, the thread-safety scheme takes advantage of these states because threads in the “detached” state cannot have any dangling pointers to Python objects.

These reinterpretations of the GIL APIs do not require any changes to C-API extensions, but they also *do not relax existing restrictions*! For example, C-API extensions still must call PyGILState\_Ensure before calling into the C API from a new thread and it’s still good practice to call PyEval\_ReleaseThread (or equivalent) before making a potentially blocking call in C code.

## Interpreter and bytecode modifications

This project substantially modifies the Python bytecode interpreter (ceval.c). The goals of the modifications were to improve single threaded performance, reduce the performance impact of the reference counting changes, and improve scalability when running with multiple threads.

Originally, this project contained fewer changes to the core Python interpreter (ceval.c), but it proved difficult to integrate deferred reference counting. Deferred reference counting was limited to a few new opcodes -- some patterns that could not make use of these opcodes scaled poorly. Additionally, the reference counting changes introduced single-threaded performance regressions and some existing optimizations (like the LOAD\_GLOBAL caching) were not thread-safe. I eventually concluded that it was worth trying a substantial rewrite to the bytecode interpreter to address these challenges.

The new interpreter (together with the GIL changes) is about 10% faster than CPython 3.9 on the single-threaded pyperformance benchmarks. A few caveats: First, the performance improvement is an average (geometric mean across ~60 benchmarks)*.* Some benchmarks run much faster and a few run slower than upstream. Second, the new interpreter stripped of the GIL changes would be even faster on single-threaded workloads[[14]](#footnote-13). Finally, there are still some use cases that don’t scale well, but I think these cases are easier to address with the new design.

CPython contains a concurrency benchmark “ccbench”, with three multithreaded tasks: computing the digits of pi, regular expression search, and bz2 compression. The new interpreter scales well on all these tasks. For example, with 20 threads, the “pi” task runs 18.2x faster, the “regex” task is 19.5x faster, and “bz2” is 19.4x faster. The “bz2” task scales well in upstream CPython because it releases the GIL when performing the compression, but the other two tasks do not scale beyond a single thread in upstream CPython.

The new interpreter uses a “register machine with accumulator register” model (roughly based on V8’s ignition interpreter design). The purpose of this design choice (particularly the “accumulator” register) was to make the frequent reference counting operations in the interpreter efficient and enable the compiler to make efficient use of CPU registers (by keeping the number of live variables across external function calls small). It’s hard to measure the impact of this design choice and is probably worth revisiting at some point in the future.

A brief digression: there is not a tremendous difference between a “register machine” like the new interpreter and a “stack machine” as implemented in CPython. In both cases there is an array that holds a function’s local variables and non-visible temporary “variables” (f\_localsplus in CPython, “regs” in the new interpreter). In both cases, the temporary variables naturally form a “stack” (due to the recursive nature of expression evaluation). The difference is that in CPython (stack machine), the top-of-stack is maintained explicitly by the interpreter and implicitly indexed by bytecodes, while the new interpreter does *not* maintain a top-of-stack variable and bytecodes explicitly index into the “stack”/“registers”.

### Tagged object representation

The interpreter operates on tagged Python object pointers. Each interpreter “register”, if it’s not empty, stores a Python object pointer tagged with least-significant-bit set to indicate when the reference count should be *not* decremented when the register is cleared. The tag is useful because the deferred reference counting implementation means that operations can produce “borrowed” references (with +0 reference count) or “new” references (with +1 reference count). The borrowed references are marked with the least-significant-bit set to distinguish them from “new” references.

An alternative design might require that *all* references in the interpreter to objects that support deferred reference counting are borrowed. This would avoid the need for tagging references, but I think this would be more expensive overall. Many references are produced outside the interpreter (without knowledge of deferred reference counting) and would require “fix-up” by the interpreter adjusting reference counts of objects before storing their references into interpreter “registers”.

### Optimized function calls

To speed-up function calls, the interpreter uses a linear, resizable stack to store function call frames, an idea taken from LuaJIT. The stack stores the interpreter registers (local variables + space for temporaries) plus some extra information per-function call. This avoids the need for allocating PyFrameObjects for each call. For compatibility, the PyFrameObject type still exists, but they are created lazily as-needed (such as for exception handling and for sys.\_getframe).

The optimized function calls have about an order of magnitude less overhead than CPython 3.10. A similar change is already implemented in the CPython 3.11 main branch.

The change also simplifies the use of deferred reference counting with the data that is stored per-call like the function object. The interpreter can usually avoid incrementing the reference count of the function object during a call. Like other objects on the stack, a borrowed reference to the function is indicated by setting the least-significant-bit.

### Thread-safe metadata for LOAD\_ATTR, LOAD\_METHOD, LOAD\_GLOBAL

The new interpreter contains a thread-safe replacement for CPython’s LOAD\_GLOBAL caching that is also extended to support LOAD\_ATTR and LOAD\_METHOD opcodes. Each of these opcodes has an associated *metadata* slot in their containing code objects. The metadata is either an *offset hint* or a *cached negative result* containing a dict version tag for missing keys. For *offset hints*, the interpreter loads the key at the specified offset from the relevant dict and compares it to the name specified by the opcode. If they match, as is typically the case, the interpreter loads the value at the same offset. If they don’t match, the interpreter falls back to the standard, slower operation. For *cached negative results*, the interpreter compares the stored version tag to the dict’s current version tag. If the version tags match, then the interpreter continues assuming the key does not exist in the dict.

The design is thread-safe because it operates on single-word metadata slots and is robust to stale metadata. It’s sufficient for the interpreter to use C’s *relaxed atomics,* which typically compile to plain load and store instructions.

The above algorithm has more steps than CPython’s LOAD\_GLOBAL caching, but still runs faster due to a more efficient implementation.

## Performance

As mentioned above, the no-GIL proof-of-concept interpreter is about 10% faster[[15]](#footnote-14) than CPython 3.9[[16]](#footnote-15) (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.

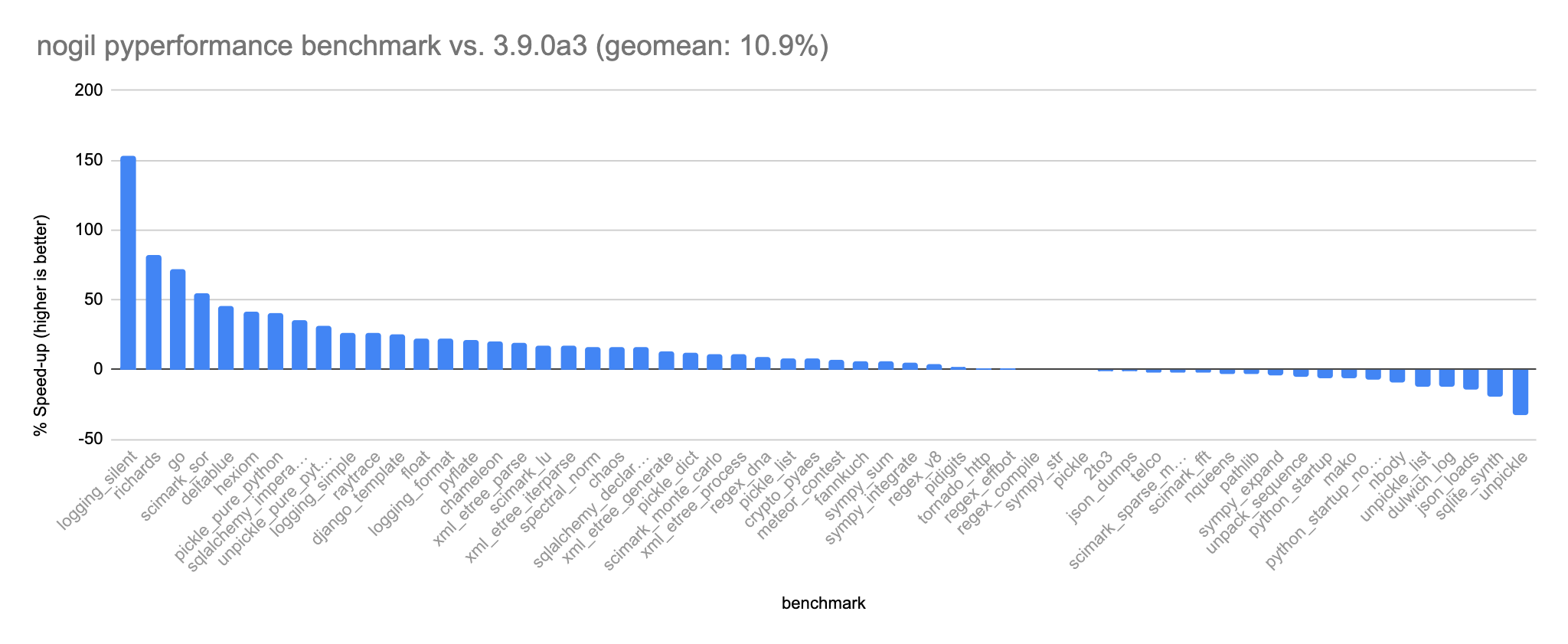


Figure 1: nogil speed-up on the pyperformance benchmark suite compared to Python 3.9.0a3. The geometric mean of the speed-up is 10.9% across 58 benchmarks. The fastest speed-up is on “logging\_silent” due to faster function calls. The largest regression (-32%) is on the unpickle benchmark.

Compilation used GCC 9 with PGO enabled.

## Compatibility

The vast majority of Python programs should not require any changes to Python code to continue working. Most of the issues I have encountered so far are due to being based on 3.9.0a3 and libraries using features from the 3.9 final release[[17]](#footnote-16). A few of the issues have been due to changes to *code* object internals or the bytecode format, but so far I have been able to address those by making changes to the interpreter to improve backwards compatibility.

All extensions that use the Python C API will require re-compilation, even the few that are limited to the stable ABI. (The reference counting changes do not preserve the stable ABI).

To compile successfully, some C API extensions will need minor modifications, typically replacement of direct access of PyObject reference count fields with the Py\_REFCNT macro.

To run successfully, some C API extensions need modifications to protect global data structures in C code that were previously protected by the GIL. For an example, see these two [patches to NumPy](https://github.com/colesbury/numpy/commits/v1.19.3-nogil).

To mitigate compatibility issues and improve debugging, the proof of concept can run with the GIL enabled or disabled controlled at runtime by an environment variable (or command-line option). If CPython adopts some form of the GIL changes, I’d expect this runtime control to be useful for at least a few releases to address [*flag day*](https://en.wikipedia.org/wiki/Flag_day_(computing)) issues. In my experience, it was not too difficult to ensure that the dozens of packages used by a given project work without the GIL; I’d expect this to be easier with community involvement. But I’d still think it would be nearly impossible to ensure that all tens of thousands of Python packages work correctly or are updated across a single Python release.

Other compatibility issues:

* PyMem\_SetAllocator only works for *delegating* allocators (like tracemalloc), not wholesale replacement of the allocator
* The bytecode format has changed; most of the opcodes have the same name and same general behavior, but the output of *dis* is different.
* Slight differences in profiling and tracing behavior due to different opcodes
* Local variables are not captured by tracebacks (breaks cgitb, possible to fix)
* Local variables on frame objects aren’t always up-to-date if the frame is no longer “alive” (similar to the traceback issue)
* Generator objects do not have a gi\_frame attribute
* Extensions must use Py\_REFCNT and Py\_SET\_REFCNT instead of directly accessing reference count fields

1. Go provides uses both a shared-memory and message passing model, but message passing is the preferred approach [↑](#footnote-ref-0)
2. For example async/await and message passing (Actors/CSP) [↑](#footnote-ref-1)
3. <https://github.com/tensorflow/swift/blob/master/docs/WhySwiftForTensorFlow.md> [↑](#footnote-ref-2)
4. pthreads for Linux, Mac OS, and other Unix-like OS’s [↑](#footnote-ref-3)
5. The two least significant bits on the shared reference count field also serve special purposes. Bit zero indicates if an object has a “merged” reference count field. Bit one indicates if an object was “queued” to be merged later by the owning thread. [↑](#footnote-ref-4)
6. For example, Py\_REFCNT divides the local and shared fields by 4 and sums the result. [↑](#footnote-ref-5)
7. Cinder (Facebook’s fork of CPython) supports immortalization to avoid copy-on-writes in a fork-based multi-process environment. [↑](#footnote-ref-6)
8. Modifications to a dict can cause the read operation to fail and need to be retried (the reverse is not true). A thread that incessantly modifies a dict might perpetually cause read operations to fail. Acquiring the per-collection mutex when retrying operations avoids this issue. [↑](#footnote-ref-7)
9. See https://www.usenix.org/legacy/publications/library/proceedings/osdi96/full\_papers/greenwald/node2.html [↑](#footnote-ref-8)
10. Size classes are effectively “rounded sizes''. It’s safe to substitute “size class” for “size” in the above constraint. [↑](#footnote-ref-9)
11. Not an OS “page”! Mimalloc “pages” are groupings of allocations (“blocks”) of the same size class. Also called “superblock” in other allocators. [↑](#footnote-ref-10)
12. The scheme for tagging blocks is based on FreeBSD’s “GUS”. Joel Fernandez has a useful [overview of GUS](https://people.kernel.org/joelfernandes/gus-vs-rcu). [↑](#footnote-ref-11)
13. The accesses to the version counter require C11 acquire-release semantics (or stronger) to ensure proper ordering. This is “free” on x86-64: these accesses are compiled to plain load and store instructions and relatively inexpensive on recent ARMv8 processors. [↑](#footnote-ref-12)
14. I don’t yet have a solid estimate for how much single-threaded performance could be improved by taking the new interpreter and stripping out the GIL related changes. [↑](#footnote-ref-13)
15. Geometric average cross 58 benchmarks [↑](#footnote-ref-14)
16. The proof-of-concept is based on CPython 3.9.0a3 [↑](#footnote-ref-15)
17. I am in the process of rebasing on v3.9.7 [↑](#footnote-ref-16)