So You Want To Remove The GVL?

Jan 29, 2025

I want to write a post about Pitchfork, explaining where it comes from, why it is like it is, and how I see its future. But before I can get to that, I think I need to share my mental model on a few things, in this case, Ruby's GVL.

For quite a long time, it has been said that Rails applications are mostly IO-bound, hence Ruby's GVL isn't that big of a deal and that has influenced the design of some cornerstone pieces of Ruby infrastructure like Puma and Sidekiq. As I explained in a previous post, I don't think it's quite true for most Rails applications. Regardless, the existence of the GVL still requires these threaded systems to use `fork(2)` in order to exploit all the cores of a server: one process per core. To avoid all this, some people have been calling for the GVL to simply be removed.

But is it that simple?

GVL and Thread Safety

If you read posts about the GVL, you may have heard that it's not there to protect your code from race conditions, but to protect the Ruby VM from your code. Put another way, GVL or not, your code can be subject to race conditions, and this is absolutely true.

But that doesn't mean the GVL isn't an important component of the thread safety of the Ruby code in your applications. Let's use a simple code sample to illustrate:

```ruby
QUOTED_COLUMN_NAMES = {}

def quote_column_name(name)
  QUOTED_COLUMN_NAMES[name] ||= quote(name)
end
```

Would you say this code is thread-safe? Or not?

Well, if you answered "It's thread-safe", you're not quite correct. But if you answered "It's not thread safe", you're not quite correct either.

The actual answer is: "It depends".

First, it depends on how strict of a definition of thread-safe you are thinking of, then it depends on whether that `quote` method is idempotent and finally, it depends on which implementation of Ruby you are using.

Let me explain.

First `||=` is syntax sugar that is hiding a bit how this code actually works, so let's desugar it:

```ruby
QUOTED_COLUMN_NAMES = {}

def quote_column_name(name)
  quoted = QUOTED_COLUMN_NAMES[name]

  # Ruby could switch threads here

  if quoted
    quoted
  else
    QUOTED_COLUMN_NAMES[name] = quote(name)
  end
end
```

In this form it's easier to see that `||=` isn't a single operation but multiple, so even on MRI[1], with a GVL, it's technically possible that Ruby would preempt a thread after evaluating `quoted = ...`, and resume another thread that will enter the same method with the same argument.

In other words, this code is subject to race conditions, even with a GVL. To be even more precise, it's subject to a *check-then-act* race condition.

If it's subject to race conditions, you can logically deduce that it's not thread-safe. But here again, it depends. If `quote(name)` is idempotent, then yes there's technically a race-condition, but it has no real negative impact. The `name` will be quoted twice instead of once, and one of the resulting strings will be discarded, who cares? That is why in my opinion the above code is effectively thread-safe regardless.

And we can verify this experimentally by using a few threads:

```ruby
QUOTED_COLUMN_NAMES = 20.times.to_h { |i| [i, i] }

def quote_column_name(name)
  QUOTED_COLUMN_NAMES[name] ||= "`#{name.to_s.gsub('`', '``')}`".freeze
end

threads = 4.times.map do
  Thread.new do
    10_000.times do
      if quote_column_name("foo") != "`foo`"
        raise "There was a bug"
      end
      QUOTED_COLUMN_NAMES.delete("foo")
    end
```

```
  end
end

threads.each(&:join)
```

If you run this script with MRI, it will work fine, it won't crash, and `quote_column_name` will always return what you expect.

However, if you try to run it with either TruffleRuby or JRuby, which are alternative implementations of Ruby that don't have a GVL, you'll get about 300 lines of errors:

```
$ ruby -v /tmp/quoted.rb
truffleruby 24.1.2, like ruby 3.2.4, Oracle GraalVM Native [arm64-darwin20]
java.lang.RuntimeException: Ruby Thread id=51 from /tmp/quoted.rb:20 termina
    at org.truffleruby.core.thread.ThreadManager.printInternalError(ThreadMa
    ... 20 more
Caused by: java.lang.NullPointerException
    at org.truffleruby.core.hash.library.PackedHashStoreLibrary.getHashed(Pa
    ... 120 more
java.lang.RuntimeException: Ruby Thread id=52 from /tmp/quoted.rb:20 termina
    at org.truffleruby.core.thread.ThreadManager.printInternalError(ThreadMa
    ... 20 more
... etc
```

The error isn't always exactly the same, and sometimes it seems worse than others. But in general, it crashes deep inside the TruffleRuby or JRuby interpreters because the concurrent access to the same hash causes them to hit a `NullPointerException`.

So we can say this code is thread-safe on the reference implementation of Ruby, but not on all implementations of Ruby.

The reason it is that way is that on MRI, the thread scheduler can only switch the running thread when executing pure Ruby code. Whenever you call into a builtin method that is implemented in C, you are implicitly protected by the GVL. Hence all methods implemented in C are essentially "atomic" unless they explicitly release the GVL. But generally speaking, only IO methods will release it.

That's why the real version of this code, that I took from Active Record, doesn't use a `Hash`, but a `Concurrent::Map`. On MRI that class is pretty much just an alias for `Hash`, but on JRuby and TruffleRuby it's defined as a hash table with a mutex. Officially Rails doesn't support TruffleRuby or JRuby, but in practice, we tend to accommodate them with this sort of small changes.

Just Remove It Already

That's why there's "removing the GVL" and "removing the GVL".

The *simple* way would be to do what TruffleRuby and JRuby do: nothing. Or close to nothing.

Since these alternative implementations are based on the Java Virtual Machine, which is memory-safe, they delegate to the JVM runtime the hard job of failing but not hard crashing in such cases. Given MRI is implemented in C, which is famously not memory-safe, just removing the GVL would cause the virtual machine to run into a segmentation fault (or worse) when your code triggers this sort of race condition, so it wouldn't be as simple.

Ruby would need to do something similar to what the JVM does, having some sort of atomic counter on every object that could be subject to race conditions. Whenever you access an object you increment it and check it is set to `1` to ensure nobody else is currently using it.

This in itself is quite a challenging task, as it means going over all the methods implemented in C (in Ruby itself but also popular C extensions), to insert all these atomic increments and decrements.

It would also require some extra space in most Ruby objects for that new counter, likely 4 or 8 bytes, because atomic operations aren't easily done on smaller integer types. Unless of course there's some smart trick I'm not privy of.

It would also result in a slow-down of the virtual machine, as all these atomic increments and decrements likely would have a noticeable overhead, because atomic operations mean that the CPU has to ensure all cores see the operation at the same time, so it essentially locks that part of the CPU cache. I won't try to guess how much that overhead would be in practice, but it certainly isn't free.

And then the result would be that a lot of existing pure Ruby code, that used to be effectively thread safe, would no longer be. So beyond the work ruby-core would have to do, Ruby users would also likely need to debug a bunch of thread safety issues in their code, gems, etc.

That's why despite the impressive efforts of JRuby and TruffleRuby teams to be as compatible as possible with MRI, the absence of a GVL, which is a feature, makes it so that most non-trivial codebases likely need at least some debugging before they can run properly on either of them. It's not necessarily a ton of effort, it depends, but it's more work than your average yearly Ruby upgrade.

Replace It By Something

But that's not the only way to remove the GVL, another way that is often envisioned is to replace the one global lock, by a myriad of small locks, one per every mutable object.

In terms of work needed, it's fairly similar to the previous approach, you'd need to go over all the C code and insert explicitly lock and unlock statements whenever you touch a mutable object. It

would also require some space on every object, likely a bit more than just a counter though.

With such approach, C extensions would still likely need some work, but pure Ruby code would remain fully compatible.

If you've heard about the semi-recent effort to remove Python's GIL (that's what they call their GVL), that's the approach they're using. So let's look at the sort of changes they made, starting with their base object layout that is defined in `object.h`

It has lots of ceremonial code, so here's a stripped-down and simplified version:

```
/* Nothing is actually declared to be a PyObject, but every pointer to
 * a Python object can be cast to a PyObject*.  This is inheritance built by
 */
#ifndef Py_GIL_DISABLED
struct _object {
    Py_ssize_t ob_refcnt
    PyTypeObject *ob_type;
};
#else
// Objects that are not owned by any thread use a thread id (tid) of zero.
// This includes both immortal objects and objects whose reference count
// fields have been merged.
#define _Py_UNOWNED_TID            0

struct _object {
    // ob_tid stores the thread id (or zero). It is also used by the GC and
    // trashcan mechanism as a linked list pointer and by the GC to store th
    // computed "gc_refs" refcount.
    uintptr_t ob_tid;
    uint16_t ob_flags;
    PyMutex ob_mutex;           // per-object lock
    uint8_t ob_gc_bits;         // gc-related state
    uint32_t ob_ref_local;      // local reference count
    Py_ssize_t ob_ref_shared;   // shared (atomic) reference count
    PyTypeObject *ob_type;
};
#endif
```

There's quite a lot in there, so let me describe it all. My entire explanation will assume a 64-bit architecture, to make things simpler.

Also note that while I used to be a Pythonista, that was 15 years ago, and nowadays I'm just spectating Python's development from afar. All this to say, I'll do my best to correctly describe what they are doing, but it's entirely possible I get some of it wrong.

Anyway, when the GIL isn't disabled as part of compilation, every single Python object starts with a header of `16B`, the first `8B` called `ob_refcnt` is used for reference counting as the name implies, but actually only `4B` is used as a counter, the other `4B` is used as a bitmap to set flags on the object, just like in Ruby. Then the remaining `8B` is simply a pointer to the object's class.

For comparison, Ruby's object header, called `struct RBasic` is also `16B`. Similarly, it has one pointer to the class, and the other `8B` is used as a big bitmap that stores many different things.

However, when the GIL is disabled during compilation, the object header is now `32B`, double the size. It starts with an `8B` `ob_tid`, for thread ID, which stores which thread owns that particular object. Then `ob_flags` is explicitly laid out, but has been reduced to `2B` instead of `4B`, to make space for a `1B` `ob_mutex`, and another `1B` for some GC state I don't know much about.

The `4B` `ob_refcnt` field is still there, but this time named `ob_ref_local`, and there is another `8B` `ob_ref_shared`, and finally, the pointer to the object class.

Just with the change in the object layout, you can already have a sense of the extra complexity, as well as the memory overhead. Sixteen extra bytes per object isn't negligible.

Now, as you may have guessed from the `refcnt` field, Python's memory is mainly managed via reference counting. They also have a mark and sweep collector, but it's only there to deal with circular references. In that way, it's quite different from Ruby, but looking at what they had to do to make this thread safe is interesting regardless.

Let's look at `Py_INCREF`, defined in `refcount.h`. Here again, it's full of `ifdef` for various architecture and such, so here's a stripped-down version, with only the code executed when the GIL is active, and some debug code removed:

```
#define _Py_IMMORTAL_MINIMUM_REFCNT ((Py_ssize_t)(1L << 30))

static inline Py_ALWAYS_INLINE int _Py_IsImmortal(PyObject *op)
{
    return op->ob_refcnt >= _Py_IMMORTAL_MINIMUM_REFCNT;
}

static inline Py_ALWAYS_INLINE void Py_INCREF(PyObject *op)
{
    if (_Py_IsImmortal(op)) {
        return;
    }
    op->ob_refcnt++;
}
```

It's extremely simple, even if you are unfamiliar with C you should be able to read it. But basically, it checks if the refcount is set to a magical value that marks immortal objects, and if it isn't immortal, it simply does a regular, non-atomic, hence very cheap, increment of the counter.

A sidenote on immortal objects, it's a very cool concept introduced by Instagram engineers which I've been meaning to introduce in Ruby too. It's well worth a read if you are interested in things like Copy-on-Write and memory savings.

Now let's look at that same `Py_INCREF` function, with the GIL removed:

```
#define _Py_IMMORTAL_REFCNT_LOCAL UINT32_MAX
# define _Py_REF_SHARED_SHIFT        2

static inline Py_ALWAYS_INLINE int _Py_IsImmortal(PyObject *op)
{
    return (_Py_atomic_load_uint32_relaxed(&op->ob_ref_local) ==
            _Py_IMMORTAL_REFCNT_LOCAL);
}

static inline Py_ALWAYS_INLINE int
_Py_IsOwnedByCurrentThread(PyObject *ob)
{
    return ob->ob_tid == _Py_ThreadId();
}

static inline Py_ALWAYS_INLINE void Py_INCREF(PyObject *op)
{
    uint32_t local = _Py_atomic_load_uint32_relaxed(&op->ob_ref_local);
    uint32_t new_local = local + 1;
    if (new_local == 0) {
        // local is equal to _Py_IMMORTAL_REFCNT_LOCAL: do nothing
        return;
    }
    if (_Py_IsOwnedByCurrentThread(op)) {
        _Py_atomic_store_uint32_relaxed(&op->ob_ref_local, new_local);
    }
    else {
        _Py_atomic_add_ssize(&op->ob_ref_shared, (1 << _Py_REF_SHARED_SHIFT)
    }
}
```

This is now way more involved. First the `ob_ref_local` needs to be loaded atomically, which as mentioned previously is more costly than loading it normally as it requires CPU cache synchronization. Then we still have the check for immortal objects, nothing new.

The interesting part is the final `if`, as there are two different cases, the case where the object is owned by the current thread and the case where it isn't. Hence the first step is to compare the `ob_tid` with `_Py_ThreadId()`. That function is way too big to include here, but you can check its implementation in `object.h`, on most platform it's essentially free because the thread ID is always stored in a CPU register.

When the object is owned by the current thread, Python can get away with a non-atomic increment followed by an atomic store. Whereas in the opposite case, the entire increment has to be atomic, which is way more expensive as it involves compare and swap operations. Meaning that in case of a race condition, the CPU will retry the incrementation until it happens without a race condition.

In pseudo-Ruby it could look like this:

```ruby
def atomic_compare_and_swap(was, now)
  # assume this method is a single atomic CPU operation
  if @memory == was
    @memory = now
    return true
  else
    return false
  end
end

def atomic_increment(add)
  loop do
    value = atomic_load(@memory)
    break if atomic_compare_and_swap(value + add, value)
  end
end
```

So you can see how what used to be a very mundane operation, that is a major Python hotspot, became something noticeably more complex. Ruby doesn't use reference counting, so this particular case wouldn't immediately translate to Ruby if there was an attempt to remove the GVL, but Ruby still has a bunch of similar routines that are very frequently called and would be similarly impacted.

For instance, because Ruby's GC is generational and incremental, whenever a new reference is created between two objects, say `A` towards `B`, Ruby might need to mark `A` as needing to be rescanned, and it is done by flipping one bit in a bitmap. That's one example of something that would need to be changed to use atomic operations.

But we still haven't got to talk about the actual locking. When I first heard about Python's renewed attempt to remove their GIL, I expected they'd leverage the existing reference counting API to shove the locking in it, but clearly, they didn't. I'm not certain why, but I suppose the

semantics don't fully match.

Instead, they had to do what I mentioned earlier, go over all the methods implemented in C to add explicit lock and unlock calls. To illustrate, we can look at the `list.clear()` method, which is the Python equivalent to `Array#clear`.

Prior to the GIL removal effort, it looked like this:

```
int
PyList_Clear(PyObject *self)
{
    if (!PyList_Check(self)) {
        PyErr_BadInternalCall();
        return -1;
    }
    list_clear((PyListObject*)self);
    return 0;
}
```

It looks simpler than it actually is because most of the complexity is in the `list_clear` routine, but regardless, it's fairly straightforward.

Quite a while after the project started, Python developers noticed they forgot to add some locks to `list.clear` and a few other methods, so they changed it for:

```
int
PyList_Clear(PyObject *self)
{
    if (!PyList_Check(self)) {
        PyErr_BadInternalCall();
        return -1;
    }
    Py_BEGIN_CRITICAL_SECTION(self);
    list_clear((PyListObject*)self);
    Py_END_CRITICAL_SECTION();
    return 0;
}
```

Not that much worse, they managed to encapsulate it all in two macros that are just noops when Python is built with the GIL enabled.

I'm not going to explain everything happening in `Py_BEGIN_CRITICAL_SECTION`, some of it flies over my head anyway, but long story short it ends up in `_PyCriticalSection_BeginMutex`, which has a fast path and a slow path:

```
static inline void
_PyCriticalSection_BeginMutex(PyCriticalSection *c, PyMutex *m)
{
    if (PyMutex_LockFast(m)) {
        PyThreadState *tstate = _PyThreadState_GET();
        c->_cs_mutex = m;
        c->_cs_prev = tstate->critical_section;
        tstate->critical_section = (uintptr_t)c;
    }
    else {
        _PyCriticalSection_BeginSlow(c, m);
    }
}
```

What the fast path does, is that it assumes the object's `ob_mutex` field is set to `0`, and tries to set it to `1` with an atomic compare and swap:

```
//_Py_UNLOCKED is defined as 0 and _Py_LOCKED as 1 in Include/cpython/lock.h
static inline int
PyMutex_LockFast(PyMutex *m)
{
    uint8_t expected = _Py_UNLOCKED;
    uint8_t *lock_bits = &m->_bits;
    return _Py_atomic_compare_exchange_uint8(lock_bits, &expected, _Py_LOCKE
}
```

If that works, it knows the object was unlocked so it can just to a little bit of book keeping.

If that doesn't work, however, it enters the slow path, and there it starts to become quite complicated but to describe it quickly, it first uses a spin-lock with 40 iterations. So in a way, it does the same compare and swap logic 40 times in a raw with the hope that it might work eventually. And if that still doesn't work, it then "parks" the thread and will wait for a signal to resume. If you are interested in knowing more you can look at `_PyMutex_LockTimed` in `Python/lock.c` and follow the code from there. Ultimately the mutex code isn't that interesting for our current topic, because the assumption is that most objects are only ever accessed by a single thread, so the fast path is what matters the most.

But beyond the cost of that fast path, what is also important is how to integrate the lock and unlock statements in an existing codebase. If you forget one `lock()`, you might cause a VM crash, and if you forget one `unlock()`, you might cause a VM dead-lock, which is arguably even worse.

So let's go back to that `list.clear()` example:

```
int
PyList_Clear(PyObject *self)
{
    if (!PyList_Check(self)) {
        PyErr_BadInternalCall();
        return -1;
    }
    Py_BEGIN_CRITICAL_SECTION(self);
    list_clear((PyListObject*)self);
    Py_END_CRITICAL_SECTION();
    return 0;
}
```

You may have noticed how Python does error checking. When a bad precondition is found, it generates an exception with a `PyErr_*` function and returns `-1`. That's because `list.clear()` always returns `None` (Python's `nil`), so the return type of its C implementation is just an `int`. For a method that returns a Ruby object, on an error condition it would return a `NULL` pointer.

For instance `list.__getitem__`, which is Python's equivalent to `Array#fetch` is defined as:

```
PyObject *
PyList_GetItem(PyObject *op, Py_ssize_t i)
{
    if (!PyList_Check(op)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    if (!valid_index(i, Py_SIZE(op))) {
        _Py_DECLARE_STR(list_err, "list index out of range");
        PyErr_SetObject(PyExc_IndexError, &_Py_STR(list_err));
        return NULL;
    }
    return ((PyListObject *)op) -> ob_item[i];
}
```

You can see that error if you try accessing a Python list with an out-of-bound index:

```
>>> a = []
>>> a[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You can recognize the same `IndexError` and the same `list index out of range` message.

So in both cases, when the Python methods implemented in C need to raise an exception, they build the exception object, store it in some thread local state, and then return a specific value to let the interpreter know that an exception happened. When the interpreter notices the return value of the function is one of these special values, it starts unwinding the stack. In a way, Python exceptions are syntactic sugar for the classic `if (error) { return error }` pattern.

Now let's look at Ruby's `Array#fetch`, and see if you notice any difference in how the out-of-bound case is handled:

```
static VALUE
rb_ary_fetch(int argc, VALUE *argv, VALUE ary)
{
    // snip...
    long idx = NUM2LONG(pos);
    if (idx < 0 || RARRAY_LEN(ary) <= idx) {
        if (block_given) return rb_yield(pos);
        if (argc == 1) {
            rb_raise(rb_eIndexError, "index %ld outside of...", /* snip... *
        }
        return ifnone;
    }
    return RARRAY_AREF(ary, idx);
}
```

Did you notice how there is no explicit `return` after `rb_raise`?

That's because Ruby exceptions are very different from Python exceptions, as they rely on `setjmp(3)` and `longjmp(3)`.

Without going into too much detail, these two functions essentially allow you to make some sort of "savepoint" of the stack and jump back to it. When they are used, it's a bit like a non-local `goto`, you directly jump back to a parent function and all the intermediate functions never return.

As a consequence, a Ruby equivalent of `Py_BEGIN_CRITICAL_SECTION` would need to call `setjmp`, and push the associated checkpoint on the execution context (essentially the current fiber) using the `EC_PUSH_TAG` macro, so essentially every core method would now need a `rescue` clause, and that's not free. It's doable, but likely more costly than `Py_BEGIN_CRITICAL_SECTION`.

Shall We?

But we were so preoccupied with whether or not we could remove the GVL, we didn't stop to think if we should.

In the case of Python, from my understanding, the driving force behind the effort to remove the GIL is mostly the machine learning community, in big part, because feeding graphic cards efficiently requires a fairly high level of parallelism, and `fork(2)` isn't very suitable for it.

But, again from my understanding, the Python Web community, such as Django users, seem to be content with `fork(2)`, even though Python is at a major disadvantage over Ruby in terms of Copy-on-Write effectiveness, because as we saw previously, its reference counting implementation means most objects are constantly written to, so CoW pages are very quickly invalidated.

On the other hand, Ruby's mark-and-sweep GC is much more Copy-On-Write friendly, as almost all the GC tracking data isn't stored in the objects themselves but inside external bitmaps. Hence, one of the main arguments for GVL free threading, which is to reduce memory usage, is much less important in the case of Ruby.

Given that Ruby (for better or for worse) is predominantly used for the Web use case, it can at least partially explain why the pressure to remove the GVL isn't as strong as it has been with Python. Similarly, Node.js and PHP don't have free threading either, but as far as I know their respective communities aren't complaining much about it, unless I missed it.

Also if Ruby were to adopt some form of free threading, it would probably need to add some form of lock in all objects, and would frequently mutate it, likely severely reducing Copy-on-Write efficiency. So it wouldn't be purely an additive feature.

Similarly, one of the main blocker for removing Python's GIL has always been the negative impact on single-thread performance. When you are dealing with easily parallelizable algorithms, even if single-thread performance is degraded, you can probably come out on top by using more parallelism. But if the sort of thing you use Python for isn't easily parallelizable, free-threading may not be particularly appealing to you.

Historically, Guido van Rossum's stance on removing the GIL was that he'd welcome it as long as it had no impact on single-thread performance, hence why it never happened. Now that Guido is no longer Python's benevolent dictator, it seems that the Python steering council is willing to accept some regression on single-thread performance, but it isn't yet clear how much it will actually be. There are some numbers flying around, but mostly from synthetic benchmarks and such. Personally, I'd be interested to see the impact on Web applications before I'd be enthusiastic about such change happening to Ruby. It is also important to note that the removal has been accepted but with some proviso, so it isn't yet done and it's not impossible that they might decide to backtrack at one point.

Another thing to consider is that the performance impact on Ruby might be worse than for Python, because the objects that need the extra overhead are the mutable ones, and contrary to Python, in Ruby that includes strings. Think of how many string operations the average web application is doing.

On the other side, one argument I can think of in favor of removing the GVL though, would be YJIT. Given the native code YJIT generates, and the associated metadata it keeps are scoped to the process, no longer relying on `fork(2)` for parallelism would save quite a lot of memory, just by sharing all this memory, that being said, removing the GVL would also make YJIT's life much harder, so it may just as much hinder its progress.

Another argument in favor of free threading is that forked processes can't easily share connections. So when you start scaling Rails application to a large number of CPU cores, you end up with a lot more connections to your datastore than with stacks that have free threading, and this can be a big bottleneck, particularly with some databases with costly connections like PostgreSQL. Currently, this is largely solved by using external connection poolers, like PgBouncer or ProxySQL, which I understand aren't perfect. It's one more moving piece that can go wrong, but I think it's much less trouble than free threading.

And finally, I'd like to note that the GVL isn't the whole picture. If the goal is to replace `fork(2)` by free-threading, even once the GVL is removed, we might still not quite be there because Ruby's GC is "stop the world", so with much more code execution happening in a single process, hence much more allocations, we may find out that it would become the new contention point. So personally, I'd rather aim for a fully concurrent GC before wishing the GVL removed.

So It Is Urgent To Do Nothing?

At this point, some of you may feel like I'm trying to gaslight people into thinking that the GVL is never a problem, but that's not exactly my opinion.

I do absolutely think the GVL is currently causing some very real problems in real world applications, namely contention. But this is very different from wanting the GVL removed and I beleive the situation could be noticeably improved in other ways.

If you've read my short article on how to properly measure IO time in Ruby, you may be familiar with the GVL contention problem, but let me include the same test script here:

```ruby
require "bundler/inline"

gemfile do
  gem "bigdecimal" # for trilogy
  gem "trilogy"
  gem "gvltools"
end
```

```ruby
GVLTools::LocalTimer.enable

def measure_time
  realtime_start = Process.clock_gettime(Process::CLOCK_MONOTONIC, :float_mi
  gvl_time_start = GVLTools::LocalTimer.monotonic_time
  yield

  realtime = Process.clock_gettime(Process::CLOCK_MONOTONIC, :float_millisec
  gvl_time = GVLTools::LocalTimer.monotonic_time - gvl_time_start
  gvl_time_ms = gvl_time / 1_000_000.0
  io_time = realtime - gvl_time_ms
  puts "io: #{io_time.round(1)}ms, gvl_wait: #{gvl_time_ms.round(2)}ms"
end

trilogy = Trilogy.new

# Measure a first time with just the main thread
measure_time do
  trilogy.query("SELECT 1")
end

def fibonacci( n )
  return  n  if ( 0..1 ).include? n
  ( fibonacci( n - 1 ) + fibonacci( n - 2 ) )
end

# Spawn 5 CPU-heavy threads
threads = 5.times.map do
  Thread.new do
    loop do
      fibonacci(25)
    end
  end
end

# Measure again with the background threads
measure_time do
  trilogy.query("SELECT 1")
end
```

If you run it, you should get something like:

```
realtime: 0.22ms, gvl_wait: 0.0ms, io: 0.2ms
realtime: 549.29ms, gvl_wait: 549.22ms, io: 0.1ms
```

This script demonstrates how GVL contention can cause havoc on your application latency. And even if you use a single-threaded server like Unicorn or Pitchfork, it doesn't mean the applications only use a single thread. It's incredibly common to have various background threads to perform some service tasks, such as monitoring. One example of that is the `statsd-instrument` gem. When you emit a metric, it's collected in memory, and then a background thread takes care of serializing and sending these metrics in batch. It's supposed to be largely IO work, hence shouldn't have too much impact on the main threads, but in practice, it can happen that these sorts of background threads hold the GVL for much longer than you'd like.

So while my demo script is extreme, you can absolutely experience some level of GVL contention in production, regardless of the server you use.

But I don't think trying to remove the GVL is necessarily the best way to tame that problem, as it would take years of tears and sweat before you'd reap any benefits.

Prior to something like 2006, multi-core CPUs were basically non-existent, and yet, you were perfectly able to multi-task on your computer in a relatively smooth way, crunching numbers in Excel while playing some music in Winamp, and this without any parallelism.

That's because even Windows 95 had a somewhat decent thread scheduler, but Ruby still doesn't. What Ruby does when a thread is ready to execute and has to wait for the GVL, is that it puts it in a FIFO queue, and whenever the running thread releases the GVL, either because it did some IO or because it ran for its allocated 100ms, Ruby's thread scheduler pops the next one.

There is no notion of priority or anything. A semi-decent scheduler should be able to notice that a thread is mostly IO and that interrupting the current thread to schedule the IO-heavy thread faster is likely worth it.

So before trying to remove the GVL, it would be worth trying to implement a proper thread scheduler. Credit goes to John Hawthorn for that idea.

In the meantime, Aaron Patterson shipped a change in Ruby 3.4 to allow reducing the 100ms quantum via an environment variable. It doesn't solve everything, but it can probably already help in some cases, so it's a start.

Another idea John shared in one of our conversations[2], would be to allow more CPU operations with the GVL released. Currently, most database clients only really release the GVL around the IO, think of it like it:

```
def query(sql)
  response = nil
  request = build_network_packet(sql)

  release_gvl do
    socket.write(request)
    response = socket.read
```

```
  end

  parse_db_response(response)
end
```

For simple queries that return a non-trivial amount of data, it is likely that you are actually spending much more time building the Ruby objects with the GVL acquired, than waiting on the DB response with the GVL released.

This is because very very few of the Ruby C API can be used with the GVL released, notably, anything that allocates and object, or could potentially raise an exception MUST acquire the GVL.

If this constraint was removed, such that you could create basic Ruby objects such as String, Array, and Hashes with the GVL released, it would likely allow the GVL to be released much longer and significantly reduce contention.

Conclusion

I'm personally not really in favor of removing the GVL, I don't think the tradeoff is quite worth it, at least not yet, nor do I think it would be as much of a game-changer as some may imagine.

If it didn't have any impact on the classic (mostly) single-threaded performance, I wouldn't mind it, but it is almost guaranteed to degrade single-threaded performance significantly, hence this feels a bit like "a bird in the hand is worth two in the bush" kind of proposition.

Instead, I believe there are some much easier and smaller changes we could make to Ruby that would improve the situation on a much shorter timeline and with much less effort both for Ruby-core and for Ruby users.

But of course that is just the perspective of a single Ruby user with mostly my own use case in mind, and ultimately this is for Matz to decide, based on what he thinks the community wants and needs.

For now, Matz doesn't want to remove the GVL and He instead accepted the Ractor proposal[3]. Perhaps his opinion may change one day, we'll see.

1. MRI: Matz's Ruby Interpreter, the reference implementation of Ruby, sometimes referred to as CRuby. ↩️
2. If you didn't notice, John is incredibly clever. ↩️
3. Ractors which I also wanted to discuss in this post, but it's already too long, so maybe another time. ↩️