Is it me, or the GIL?

Christoph Heer
EuroPython 2019 – 10.07.2019
Background

- Quality assurance for SAP HANA
- Automated testing of ~800 commits per day
- Mainly testing with physical hardware: ~1600 machines, 610 TB RAM (256 GB – 8TB)
- Development of optimized tools and services
Inspect offer handling thread

select offer for task 724.09ms
prepare task
API call
Service #1: process request
API call
Service #1: process request
...
select offer for task
...
select offer for task
Inspect offer handling thread

Observations

- Different function runtimes

select offer for task 724.09ms
prepare task
API call

Service #1: process request
API call

Service #1: process request
...

select offer for task 551.02ms

Service #1: process request
30ms
186ms
30ms

select offer for task 200ms

select offer for task 32.21ms

select offer for task 292.08ms
Inspect offer handling thread

select offer for task
prepare task
API call
Service #1: process request
API call
Service #1: process request
...
select offer for task
...
select offer for task

Observations
- Different function runtimes
- Increased latency
Inspect offer handling thread

select offer for task 724.09ms
prepare task
API call

Service #1: process request

API call

Service #1: process request

... 

select offer for task

... 

select offer for task

Observations

- Different function runtimes
- Increased latency
- Gaps between operations
Inspect offer handling thread

Observations
- Different function runtimes
- Increased latency
- Gaps between operations

Assumption: Thread released GIL multiple times and had to wait for re-acquire

Service #1: process request
- API call
- 30ms
- 30ms
- 186ms
- 30ms

select offer for task
- 724.09ms

prepare task

select offer for task
- 551.02ms

Service #1: process request
- API call
- 200ms

select offer for task
- 32.21ms

select offer for task
- 292.08ms
Mitigate GIL contention

Python’s ecosystem offers various ways:

- multithreading => asyncio
- multithreading => multiprocessing (+ asyncio)
- CPU-intensive functions => Cython without the GIL
- Python => $faster language
- ...

But, Rewriting and major refactoring are expensive

- Verify the assumption and measure GIL contention
- Decide about solution based on collected metrics instead of intuition
Look at the GIL

$ python3.7
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.get_gil_stats()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: module 'sys' has no attribute 'get_gil_stats'
Wish list

- Provides metrics about GIL contention
  - wait time
  - hold time
- Additional context
  - Thread identifier/name
  - Python/C-Function
  - Trace/Request Id
- Usable for productive environments
  - Low overhead
  - Dynamically attachable to running Python processes
- Integration into existing observability stack
Related work and instrumentation approaches

A **Zoomable Interactive Python Thread Visualization** by Dave Beazley

- Understanding the Python GIL (PyCon 2010)
- Adjustments in CPython 2.6 to store events about the GIL
Related work and instrumentation approaches

**Thread Concurrency Visualization by PyCharm**

- Visualizes and reveals locking issues but omits GIL
Related work and instrumentation approaches

**gil_load** by Chris Billington

- Statistical profiler based on sampling states of all python threads every 50ms
- Installable python package (not Python 3.7 compatible yet)

```
$ python example_two.py
[2019-06-21 23:23:20] GIL load: 1.00 (1.00, 1.00, 1.00)
[2019-06-21 23:23:21] GIL load: 1.00 (1.00, 1.00, 1.00)
[2019-06-21 23:23:23] GIL load: 1.00 (1.00, 1.00, 1.00)
[2019-06-21 23:23:23] GIL load: 1.00 (1.00, 1.00, 1.00)
[2019-06-21 23:23:25] GIL load: 1.00 (1.00, 1.00, 1.00)
[2019-06-21 23:23:26] GIL load: 1.00 (1.00, 1.00, 1.00)
[2019-06-21 23:23:27] GIL load: 1.00 (1.00, 1.00, 1.00)
```
Related work and instrumentation approaches

**py-spy** by Ben Frederickson

- Promising sampling profiler in Rust for Python applications
- Includes GIL utilization metric but no breakdown into usage per thread
“There is no magic GIL performance analysis tool”

a sad Python developer
Creating a tool that reveals the GIL

- Base: SystemTap
- Analyze Linux applications by attaching handlers to events
- CPython 3.6 introduced DTrace/SystemTap support & markers
  - function__entry
  - function__return
  - Great documentation: Instrumenting CPython with DTrace and SystemTap
- Pre-build Linux packages often compiled without: --with-dtrace
- No GIL related markers
Adding SystemTap markers about the GIL

```diff
diff --git a/Python/ceval_gil.h b/Python/ceval_gil.h
index ef5189068e..aecd3c99aa 100644
@@ -static void drop_gil(PyThreadState *tstate) {
-    MUTEX_LOCK(_PyRuntime.ceval.gil.mutex);
-    _Py_atomic_store_relaxed(&_PyRuntime.ceval.gil.locked, 0);
-    if (PyDTrace_GIL_DROP_ENABLED())
-        PyDTrace_GIL_DROP(PyThread_get_thread_ident());
-    if (PyDTrace_GIL_CLAIM_ENABLED())
-        PyDTrace_GIL_CLAIM(PyThread_get_thread_ident());
-    COND_SIGNAL(_PyRuntime.ceval.gil.cond);
-    MUTEX_UNLOCK(_PyRuntime.ceval.gil.mutex);
+
+    MUTEX_LOCK(_PyRuntime.ceval.gil.mutex);
+    _Py_atomic_store_relaxed(&_PyRuntime.ceval.gil.locked, 0);
+    if (PyDTrace_GIL_DROP_ENABLED())
+        PyDTrace_GIL_DROP(PyThread_get_thread_ident());
+    if (PyDTrace_GIL_CLAIM_ENABLED())
+        PyDTrace_GIL_CLAIM(PyThread_get_thread_ident());
+    COND_SIGNAL(_PyRuntime.ceval.gil.cond);
+    MUTEX_UNLOCK(_PyRuntime.ceval.gil.mutex);

@@ -static void take_gil(PyThreadState *tstate) {
+    if (tstate == NULL)
+        Py_FatalError("take_gil: NULL tstate");
+    if (PyDTrace_GIL_CLAIM_ENABLED())
+        PyDTrace_GIL_CLAIM(PyThread_get_thread_ident());
+    err = errno;
+    MUTEX_LOCK(_PyRuntime.ceval.gil.mutex);
```
Measure time between GIL markers

probe process("libpython3.7m.so.1.0").mark("gil_claim") {
    last_timestamp[$arg1] = gettimeofday_ns();
}

probe process("libpython3.7m.so.1.0").mark("gil_acquired") {
    wait_time_ns = gettimeofday_ns() - last_timestamp[$arg1];
    gil_wait_aggregate[$arg1] <<< wait_time_ns;
    last_timestamp[$arg1] = gettimeofday_ns();
}

probe process("libpython3.7m.so.1.0").mark("gil_drop") {
    hold_time_ns = gettimeofday_ns() - last_timestamp[$arg1];
    gil_hold_aggregate[$arg1] <<< hold_time_ns;
}

- Attach probes (event handlers) to GIL markers
- Calculate timing of transitions
  - claim - acquired - drop
- Store measurements in SystemTap statistical aggregate type per thread
Produce summary of collected data

```plaintext
probe begin {println("Start tracing of Python GIL probes")}

probe end {
    println("Terminate tracing")
    println("Summary (all time measurements in ns)")
    foreach (thread in gil_wait_aggregate) {
        printf("Python Thread \%d\n", thread)
        printf(
            "Aggregated GIL wait time: \%d\n",
            @sum(gil_wait_aggregate[thread])
        )
        printf(
            "Aggregated GIL hold time: \%d\n",
            @sum(gil_hold_aggregate[thread])
        )
        printf("GIL wait latency\n")
        println(@hist_log(gil_wait_aggregate[thread]))
        printf("GIL hold time\n")
        println(@hist_log(gil_hold_aggregate[thread]))
    }
}
```

- Handlers for start and end of tracing session
- Report GIL wait and hold time as sum and histogram
Experiment 1: Process with 2 IO-bound threads

def io_work(n=150):
    while n > 0:
        n -= 1
        time.sleep(0.1)

def main():
    io1 = Thread(name='io1', target=io_work)
    io2 = Thread(name='io2', target=io_work)
    io1.start()
    io2.start()
    io1.join()
    io2.join()

Summary (all time measurements in ns)
Python Thread 140604373862144    # MainThread
Aggregated GIL wait time: 1102539   # 1.10ms
Aggregated GIL hold time: 27794967  # 27.79ms

Python Thread 140604351805184    # io1
Aggregated GIL wait time: 452269   # 0.45ms
Aggregated GIL hold time: 1490119  # 1.49ms

Python Thread 140604343412480    # io2
Aggregated GIL wait time: 462690   # 0.46ms
Aggregated GIL hold time: 1382457  # 1.38ms

App. runtime:  15064.3ms
GIL Hold time:  29.7ms 0.20% of runtime
GIL Wait time:  1.9ms 0.01% of runtime
Experiment 2: CPU-bound thread

def io_work(n=150):
    while n > 0:
        n -= 1
        time.sleep(0.1)

def cpu_spin():
    while True:
        pass

def main():
    io1 = Thread(name='io1', target=io_work)
    io2 = Thread(name='io2', target=io_work)
    cpu1 = Thread(
        name='cpu1', target=cpu_spin, daemon=True
    )

    ...
## Experiment 2: GIL wait latency for IO threads

<table>
<thead>
<tr>
<th>Python Thread 140347192317696  # io1</th>
<th>GIL wait latency</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1048576</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2097152</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4194304</td>
<td>150</td>
<td>149</td>
</tr>
<tr>
<td>8388608</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16777216</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GIL hold time</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>value</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32768</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>65536</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>131072</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>262144</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python Thread 140347116091136  # io2</th>
<th>GIL wait latency</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1048576</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2097152</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4194304</td>
<td>150</td>
<td>149</td>
</tr>
<tr>
<td>8388608</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>16777216</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>GIL hold time</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>value</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>1024</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2048</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4096</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8192</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16384</td>
<td>2</td>
<td>115</td>
</tr>
<tr>
<td>32768</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>65536</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>131072</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>262144</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Stable wait latency between 4 – 8ms
Experiment 2: Evaluation

- GIL contention affects overall application performance
  - Additional 5ms latency on each GIL acquire attempt after one blocking IO operation

>>> import sys
>>> sys.getswitchinterval()
0.005

- switchinterval defines the grace period before a waiting thread requests GIL drop
- GIL-holding thread may keep the GIL longer due to
  - long-running bytecode operation
  - external C function
Analyze production application with SystemTap

**Plan**
- Deploy container with customized CPython and SystemTap
- Attach to process
- Get clear and precise insides about GIL contention

**Relativity**
- Deploy container with customized CPython
- Install SystemTap on host with kernel sources, compiler toolchain etc.
- Copy libpython3.7m.so.1.0 from container into host filesystem
- Attach to process = Load custom kernel extension with your systemtap handlers
- Get huge text file with report
Result: Analysis of productive task scheduler

Observation period: 120091.89ms
GIL hold time: 105680.91ms 88.00% of timeframe
GIL wait time: 352997.64ms 293.94% of timeframe

- Proved, our application suffers from GIL contention
- More questions:
  - Are there threads that hold the GIL longer than 5ms?
  - If yes, which functions are so expensive?
  - Is it possible to identify time-based patterns with higher contention?

- Problem: With 31 threads the text report isn’t well understandable
Timelines are easier to understand

select offer for task 724.09ms

prepare task 551.02ms

API call 200ms

Service #1: process request 30ms

API call 186ms

Service #1: process request 30ms

... 32.21ms

select offer for task 292.08ms

...
Replace text report with some colorful charts

1. Collect and store data with SystemTap
2. Load data into Jupyter notebook
3. Transform data
4. Visualize data with Bokeh
Distribution of GIL hold time

metrics-collector
80075.77ms (75.77%)
Fixing our GIL contention

• Started replacing of an expensive C-Extension that doesn’t release the GIL

• Increased sleep time of metrics-collector thread from 10sec to 120sec

Before:
Observation period: 120091.89ms
GIL hold time: 105680.91ms 88.00% of timeframe
GIL wait time: 352997.64ms 293.94% of timeframe

After:
Observation period: 300112.84ms
GIL hold time: 130806.91ms 43.59% of timeframe
GIL wait time: 240568.60ms 80.16% of timeframe

• Without major refactoring the system is now able to utilize all available resources

• Collected data will helps us to decide about coming architectural changes
Many additional ideas

- Bring toolset (systemtap script & visualization) in a public usable state
- Enhance CPython for data collection with SystemTap
  - Integrate GIL markers into CPython
  - C API for thread names
- Maybe collect and provide GIL metrics directly with CPython
  - `sys.get_gil_stats()`
  - Easier integration into existing observability tooling like distributed tracing
  - No need to compile custom kernel extensions in your productive environment

- If you are also interested in that area, let’s talk!
Thank you.

Christoph Heer

christoph.heer@sap.com
@ChristophHeer