As programmers, we can create tools to make programming easier. Often this means building something that can be easily invoked on the command line, whether it is a bash script or a CLI app with a full UI. This post focuses on optimizing the latter option.

## Performance Testing

To find bottlenecks in our programs, we can profile them. For this example, we will optimize the performance of the Python edition of Logria, which is powered by curses. According to `htop`, it consumes nearly 100% of the CPU when it runs, even when idle.

### Methodology

Python provides a tool called `cprofile`. To invoke it, we write:

```
python -m cProfile -s time logria/__main__.py
```

This invokes `cprofile` on the program's entry point, `__main__.py`. I allowed the app to run for about 10 seconds, then killed it with `^-c`.

This provides us with interesting information:

```
   10107910 function calls (10106894 primitive calls) in 11.319
seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     4468    4.357    0.001    4.357    0.001 {built-in method
time.sleep}
   114752    1.365    0.000    4.002    0.000
color_handler.py:99(_add_line)
   770183    0.762    0.000    0.762    0.000 {method 'addstr' of
'_curses.window' objects}
   655424    0.448    0.000    0.840    0.000
color_handler.py:80(_color_str_to_color_pair)
   770176    0.412    0.000    0.412    0.000 {method 'noutrefresh'
of '_curses.window' objects}
   116765    0.370    0.000    0.370    0.000 {method 'sub' of
're.Pattern' objects}
     4465    0.310    0.000    0.310    0.000 {method 'refresh' of
'_curses.window' objects}
   884959    0.251    0.000    0.251    0.000 {method 'split' of
'str' objects}
   770176    0.248    0.000    0.248    0.000
color_handler.py:59(_get_color)
     2167    0.242    0.000    5.670    0.003
shell_output.py:125(render_text_in_output)
    19194    0.230    0.000    0.230    0.000 {method 'poll' of
'select.poll' objects}
  1311264    0.183    0.000    0.183    0.000 {method 'get' of 'dict'
objects}
   114752    0.163    0.000    4.219    0.000
color_handler.py:129(_inner_addstr)
   770176    0.152    0.000    0.152    0.000 {built-in method
_curses.getsyx}
   114752    0.123    0.000    4.378    0.000
```

```
color_handler.py:145(addstr)
   770176    0.110    0.000    0.110    0.000 {built-in method
_curses.color_pair}
    19194    0.098    0.000    0.643    0.000 connection.py:917(wait)
   116765    0.089    0.000    0.667    0.000
regex_generator.py:26(get_real_length)
   116898    0.073    0.000    0.126    0.000 re.py:289(_compile)
   116765    0.072    0.000    0.563    0.000 re.py:203(sub)
        1    0.059    0.059   11.255   11.255
shell_output.py:279(main)
```

## Understanding

These lines are relevant to Logria. The rest are internal to Python or `curses`:

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.059    0.059   11.255   11.255
shell_output.py:279(main)
     2167    0.242    0.000    5.670    0.003
shell_output.py:125(render_text_in_output)
   114752    0.123    0.000    4.378    0.000
color_handler.py:145(addstr)
   114752    0.163    0.000    4.219    0.000
color_handler.py:129(_inner_addstr)
   114752    1.365    0.000    4.002    0.000
color_handler.py:99(_add_line)
   655424    0.448    0.000    0.840    0.000
color_handler.py:80(_color_str_to_color_pair)
    19194    0.098    0.000    0.643    0.000 connection.py:917(wait)
   116765    0.089    0.000    0.667    0.000
regex_generator.py:26(get_real_length)
   770176    0.412    0.000    0.412    0.000 {method 'noutrefresh'
of '_curses.window' objects}
   116765    0.370    0.000    0.370    0.000 {method 'sub' of
're.Pattern' objects}
     4465    0.310    0.000    0.310    0.000 {method 'refresh' of
'_curses.window' objects}
```

```
   770176    0.248    0.000    0.248    0.000
color_handler.py:59(_get_color)
```

## Issues Found

Let’s go line by line and see what the problem might be:

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.059    0.059   11.255   11.255
shell_output.py:279(main)
```

This is the main app loop, as evinced by a single call. This function cannot be optimized because all it does is run the app.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     2167    0.242    0.000    5.670    0.003
shell_output.py:125(render_text_in_output)
```

The method `render_text_in_output()` has the highest cumulative execution time. If this function gets called, we are guaranteed to check the message queues, parse the messages, apply user filters and rules, and refresh the screen.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
   114752    0.123    0.000    4.378    0.000
color_handler.py:145(addstr)
   114752    0.163    0.000    4.219    0.000
color_handler.py:129(_inner_addstr)
   114752    1.365    0.000    4.002    0.000
color_handler.py:99(_add_line)
   655424    0.448    0.000    0.840    0.000
```

```
color_handler.py:80(_color_str_to_color_pair)
```

The next few lines come from `color_handler`, a module that handles rendering output through curses with ASCII color codes. Since they are all direct calls to `curses`, these cannot be optimized. However, the functions in this module are only invoked when `render_text_in_output()` is called; if we can call `render_text_in_output()` less, we will invoke `color_handler` functions less.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
    19194    0.098    0.000    0.643    0.000 connection.py:917(wait)
```

`wait()` is an internal Python function that ensures we can read from the multiprocessing queues Logria uses to read messages. While this cannot be optimized, checking the queues less often will reduce the number of calls to this method.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
   116765    0.089    0.000    0.667    0.000
regex_generator.py:26(get_real_length)
```

`render_text_in_output()` calls `get_real_length()` to ensure we render the proper amount of lines, as calling `len()` won't give us the correct character count if there are ASCII sequences embedded in the string. It uses a regex to remove ASCII codes. While it could probably be optimized, calling it less is also a solution.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
   770176    0.412    0.000    0.412    0.000 {method 'noutrefresh'
of '_curses.window' objects}
```

This is an internal `curses` method. It is invoked inside `render_text_in_output()` whenever we want to redraw the screen.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
   116765    0.370    0.000    0.370    0.000 {method 'sub' of
're.Pattern' objects}
```

This is from the call to `get_real_length()` and cannot be optimized.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
     4465    0.310    0.000    0.310    0.000 {method 'refresh' of
'_curses.window' objects}
```

This call is inside of `render_text_in_output()` and is required to tell `curses` to render text.

```
   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
   770176    0.248    0.000    0.248    0.000
color_handler.py:59(_get_color)
```

Finally, this call checks a hashmap to ensure we choose the proper `curses` color_pair() when rendering. It cannot be optimized because it is a simple `O(1)` lookup that memoizes the `curses` method.

## Issues Found

From this analysis, we can tell that we are doing two things that significantly impact performance: drawing the UI via `render_text_in_output()` too much and reading from our message queues too frequently.

## Drawing Too Often

Every loop, the app calls `render_text_in_output()` to redraw the screen. If there is no limit to how fast the app runs, we will redraw as fast as the computer will let us, consuming all of the available CPU. To reduce this overhead, we do not need to reduce the number of calls to `render_text_in_output()`. We only need to not compute anything if there is nothing to render.

To accomplish this, we need to cache the last render to check if there is any new content to display. We want to use a data structure that takes up little space and is fast to compare to do this efficiently.

Logria uses lists for its message buffers, so we can store the start and end indexes where the render occurred. For example, if the `stderr` list is `100` items long and the display is `20` rows high, we render `17` rows of text. Thus, the start and end would be `(83, 100)`. A tuple containing two integer indexes works well for storing our previous render position. According to the documentation:

> *Sequences compare lexicographically using comparison of corresponding elements. The built-in containers typically assume identical objects are equal to themselves. That lets them bypass equality tests for identical objects to improve performance and to maintain their internal invariants.*

**Drawing Code**
To enforce this rule on the rendering logic, only a few lines of code are required. First, we store the render state on the Logria object by declaring it:

```
self.previous_render: Optional[Tuple[int, int]] = None
```

Second, we need to check this value every call. If it did not change, we do not want to execute the rest of the method. If it has changed, we need to update the state and continue with the function logic.

```
# Determine the start and end position of the render
start, end = determine_position(self, messages_pointer)
```

```
if not self.analytics_enabled and self.previous_render == (max(start,
0), end):
    return  # Early escape
self.previous_render = (max(start, 0), end)
```

Now, Logria will make heavy render calls only if there is new content to display.

## Polling Too Fast

Every loop, Logria checks multiprocessing queues for new messages to render. These checks can be slow as we wait for the interpreters to pickle the data back and forth, leading to a lot of wait time. We can reduce the number of calls by implementing a poll rate in the main app loop. This also helps call `render_text_in_output()` less often.

**Naive Solution**
Instead of running the app with a plain `while True` loop, we can add a delay at the start of each loop to prevent checking the queues too often:

```
while True:
    time.sleep(0.01)
```

While this reduces app CPU usage, it means we wait longer to render messages and makes user input feel sluggish as it has to wait to render any input.

**Smart Solution**
We only want to render as often as messages come into the queue or when the user wants to input something. We can add this feature by including some new code.

First, we need to set poll rate limits: a minimum and a maximum. We do not want the app to refresh too slowly, but we also do not want the app to consume too many resources.

```
FASTEST_POLL_RATE: float = 0.0001   # Fast enough for smooth typing,
1000 hz
SLOWEST_POLL_RATE: float = 0.1  # Poll ten times per second, 10 hz
```

Next, we need to calculate the rate at which we receive messages:

```
while True:
    # Update messages from the input stream's queues, track time
    t_0 = time.perf_counter()
    new_messages: int = 0
    for stream in self.streams:
        # repeat the below logic for stdout
        while not stream.stdout.empty():
            message = stream.stdout.get()
            self.stdout_messages.append(message)
            new_messages += 1
    t_1 = time.perf_counter() - t_0
    time.sleep(max(0, self.poll_rate - t_1))
    self.handle_smart_poll_rate(t_1, new_messages)
```

This checks the time it takes to read the messages from the queue and subtracts that from the poll rate so we do not delay extra time. It also calls to a new method called `handle_smart_poll_rate()`.

```
def handle_smart_poll_rate(self, t_1: float, new_messages: int) ->
None:
    """
    Determine a reasonable poll rate based on the speed of messages
received
    """
    if self.manually_controlled_line:
        self.poll_rate = constants.FASTEST_POLL_RATE
    elif self.smart_poll_rate:
        if not self.loop_time:
            self.loop_time = time.perf_counter()
        else:
            self.loop_time = t_1 - self.loop_time
            messages_per_second = new_messages / self.loop_time
            if messages_per_second > 0:
                # Update poll rate
                new_poll_rate = \
                    min(
                        max(
                            1 / messages_per_second,
                            constants.FASTEST_POLL_RATE
                        ),
                        constants.SLOWEST_POLL_RATE
                    )
                self.update_poll_rate(new_poll_rate)
```

This method ensures a few things. First, it checks if the user is inputting information. If that is the case, it will always use the maximum poll rate to ensure there is no latency[1].

If the user is not entering commands, the app then determines the optimum poll rate based on the time it took to read the messages and the number of messages found. It converts these data points to a message per second metric, then clamps that value to `SLOWEST_POLL_RATE..FASTEST_POLL_RATE`. This ensures that we always check the message queues as often as there are new messages, reducing resource usage as much as possible while also ensuring we are rendering when new messages appear.

As the number of messages read changes, so will the poll rate at which we check them.

## Conclusions

Using tools built into Python, developers can quickly and easily determine how their programs perform and where they are performing poorly. CLI apps that do more than print text once benefit from making fewer heavy drawing calls. Logria required additional optimization when polling for new data from its worker processes. `cprofile` provided visibility into what method calls were taking up time and CPU, allowing me to triage the issues and adjust each one.

While there are further optimizations to make, they will happen in the Rust edition of Logria.

---

View as: PDF, Markdown | Discussion: Hacker News

---

1. The caveat of this is that it introduces a “wake up time”, i.e. if the poll rate is close to `SLOWEST_POLL_RATE` then we might wait up to `SLOWEST_POLL_RATE` to adjust back to the maximum. This wakeup time is a tradeoff we make for general performance, as user input is rare, but streaming text is common.