CoSched: Controlled Concurrency Testing for Python

Project Report for CS6215 by: *Aditya Ranjan Jha*

Introduction

In concurrent programming [2], multiple threads or processes run simultaneously, potentially sharing resources. Most modern systems, including operating systems, are multi-threaded. However, the interleavings among threads can be non-deterministic, making it very hard to formally verify [4,7.8] the correctness of concurrent programs. An alternative approach is to use testing to identify potential bugs [1,5,6]. Systematically testing a concurrent program to explore all possible interleavings is not feasible because it would require testing at the granularity of each instruction. For a program with N threads, each with M lines of code, this could mean $N! \times M!$ different orders—a computationally infeasible task. Instead, we can exercise control over possible schedules by making choices at interesting points in the program, such as synchronization operations. This is called controlled concurrency testing. In this project, we propose **CoSched**, a tool that controls interleavings by generating schedules based on the synchronization primitives used between different threads.

Background

Python is a high-level programming language that manages its own runtime. The Python interpreter historically uses a Global Interpreter Lock (GIL) to ensure that only one thread executes at a time. However, in recent versions of Python, the GIL has been removed to unlock the true benefits of multi-threaded applications, especially on multi-core systems. This change increases the risk of concurrency bugs, making controlled concurrency testing essential. Many servers, website backends, and deployed applications are written in Python and require a testing framework to identify potential issues. Since Python abstracts low-level execution, controlling thread execution—such as preemption—becomes difficult without foreign function interfaces (FFI) with C.

One possible solution is to have threads cooperatively yield control to a main thread or tool. This could be achieved using generators, but it would require inserting manual yields into the program itself, which is impractical for existing codebases. A better approach is to wrap the threading library primitives. When a thread yields control to the interpreter to operate on synchronization primitives, we can intercept the call and yield control to a scheduler, enabling the creation of interesting interleavings. Despite Python's high-level nature, logical concurrency bugs are common and can be classified into three distinct categories:

Deadlocks

A deadlock occurs when multiple threads are waiting for each other to release resources, resulting in a standstill. Consider the following code from a benchmark:

```
def t1():
    A = 0
    m.acquire()
    A += 1
    if A == 1:
```

```
            l.acquire()
        m.release()
        m.acquire()
        A -= 1
        if A == 0:
            l.release()
        m.release()
```

Here, `m` and `l` are locks. If multiple threads execute this function, one thread might acquire `m`, then `l`, while another thread acquires `m` after the first releases it. The second thread then waits for `l` (held by the first thread), while the first thread waits for `m` (now held by the second thread). This circular wait leads to a deadlock in certain interleavings.

Resource Starvation

Resource starvation occurs when a thread waiting for a resource—such as a lock, semaphore, or another thread—cannot acquire it because the resource is never released or remains unavailable. Consider this example:

```
def thread():
    rlock_1.acquire()
    rlock_1.acquire()
    rlock_1.release()
```

If multiple threads run this code, the first thread to execute acquires the reentrant lock `rlock_1` twice, increasing its reference count to two. When it releases the lock once, the count drops to one, but the lock remains held. Since the thread terminates, the lock is never fully released, causing all other threads to starve as they wait indefinitely to acquire it.

Data Races

Data races occur when multiple threads access shared data without proper synchronization, leading to non-deterministic outcomes. For instance, consider this code:

```
def thread_1():
    global x
    x = 1
    lock.acquire()
    lock.release()
    x = 2

def thread_2():
    global x
    lock.acquire()
    lock.release()
    x = 3
```

Here, two threads write to the global variable `x`. Depending on the interleaving, the final value of `x` could be either 2 or 3, illustrating a data race where the outcome depends on which thread executes last.

Approach

The goal of this project is to design **CoSched**, a tool that tests existing Python programs using the `threading` library by controlling execution to generate interleavings and detect potential errors. The tool runs threads one at a time, yielding control to a scheduler that decides the next thread to execute based on a scheduling policy. CoSched is designed to work with existing Python programs without requiring code changes, achieved by wrapping threading library primitives.

Key Challenge

Simulating thread preemption is difficult in Python without modifying the program's code, as the language abstracts low-level thread control. To address this, CoSched intercepts calls to synchronization primitives (e.g., `m.acquire()`), using these points to yield control to the scheduler.

Implementation

CoSched uses **greenlets**, lightweight cooperative coroutines, to manage thread execution. The workflow of CoSched is as follows:

- **Library Integration**: The `cosched` library is imported in place of the standard `threading` library. All calls to threading primitives invoke `cosched` wrappers.
- **Thread Registration**: When a thread is created, it is registered with the scheduler, and a new greenlet is created for it. Invoking `start` or `run` does not immediately execute the thread; the scheduler controls execution.
- **Scheduler and Thread States**: The scheduler tracks each thread's state:
    - `RUNNING`: The thread is currently executing.
    - `BLOCKED`: The thread is waiting on a synchronization primitive.
    - `TERMINATED`: The thread has completed.
    - `IDLE`: The thread is ready to run but awaiting selection.
- **Synchronization Wrappers**: Wrappers for primitives (e.g., `Lock`) manage their state. For a lock, this includes `lock_holder` (the current owner) and `lock_queue` (threads waiting to acquire it). When a thread calls `acquire()`, the wrapper updates the state and yields to the scheduler via greenlet switching. Similarly, `release()` updates the state for the calling thread and other threads that are waiting on it.
- **Scheduling Policies**: CoSched supports three policies to explore interleavings:
    - **Random**: Randomly selects the next thread to run, useful for broad exploration with high entropy.
    - **Priority**: Assigns each thread a priority that decreases each time it runs, increasing the granularity of new thread execution thereby helpful for detecting deadlocks.
    - **Interactive**: Allows the user to manually select the next thread at each synchronization point, ideal for debugging specific scenarios.

Example Workflow

Consider a thread attempting to acquire a lock:

```
def t1():
    m.acquire()  # Intercepted by wrapper, yields to scheduler
    m.release()  # Intercepted, updates state, yields back
```

1. The scheduler starts by switching to `t1`'s greenlet.
2. At `m.acquire()`, the wrapper checks the lock's state, updates it, and switches back to the scheduler.
3. The scheduler selects the next thread (e.g., `t2`) based on the policy.
4. When `t2` releases a resource or blocks, control returns to the scheduler, which may resume `t1`.

This process continues, systematically exploring interleavings until the ready queue (threads ready to be executed) is empty, at the end we systematically explore the concurrency bugs based on thread state. Concretely the following conditions are checked:

- **Deadlock Detection**: For all the synchronization primitives used in the progeram (and therefore registered with the scheduler), we check if the `queue` of the lock is not empty and the `holder` is not `terminated`. We add them to a deadlock dependency graph and keep tracking all the resources, till we find one of the `holders` that is in the deadlock dependency graph (thereby a cycle).
- **Resource Starvation Detection**: For all the primitives that are registered with the scheduler, we check if the `queue` is not empty and the `holder` has terminated thereby never releasing the resource.
- **Data Race Detection**: For data race we have to add assertions in the code for the value of shared variables to check if the value of the shared variable is as expected or not and if the value is not we can detect a logical data race. However, this is not a complete solution, as it requires manual assertions in the code. But we found that benchmarks also used similar primitives to detect data race. Another approach in future could be tracking global state with the scheduler and checking the state of the global variable over different runs of the program (differential testing).

Evaluation

We tested **CoSched** on python translation of various benchmark programs from the C concurency testing tool *Period*. The benchmarks include the above mentioned concurrency bugs.

The benchmark programs are:

Benchmark	Type	Description
`benchmark_carter_c01.py`	Deadlock	Demonstrates potential deadlock through circular wait
`deadlock01.py`	Deadlock	Shows deadlock through multiple lock acquisitions

Benchmark	Type	Description
wait_join.py	Deadlock	Illustrates deadlock through thread join dependencies
rlock_test.py	Resource Starvation	Tests reentrant lock starvation scenarios
semaphore_test.py	Resource Starvation	Demonstrates semaphore-based resource starvation
account_bad.py	Data Race	Shows race condition in bank account transactions
circular_buffer.py	Data Race	Demonstrates producer-consumer race conditions

The success rate of detecting the concurrency bugs in these benchmarks is as follows:

Test Name	Priority Policy (10 runs)	Random Policy (10 Runs)	Time (in µs) (CoSched)	Time (in µs) (Python Interpreter)
benchmark_carter_c01.py	6/10	7/10	60.1206	18.5208
deadlock01.py	10/10	5/10	52.4875	124.288
wait_join.py	10/10	10/10	44.9918	16.0832
rlock_test.py	10/10	10/10	1.00542e+06	14.0667
semaphore_test.py	10/10	10/10	52.8666	15.2083
account_bad.py	3/10	4/10	64.4792	201.229
circular_buffer.py	10/10	9/10	84.2124	126.962

These rates demonstrate that the priority policy as the intuition suggests is better in exploring problematic interleavings, since each thread is given a chance to run before one of the previously ran thread is picked again, thereby creating more dense interleavings. But we also observed in some runs random policy also performed well but the regeneration of these schedules is not deterministic (depends on the random seed), but priority will pick the thread in similar permutations thereby maximizing the detection of concurrency issues deterministically in cases where the interleavings are very fine grained (after being serialized).

The performance of Cosched has comparable overheads to the Python interpreter, with the added benefit of detecting concurrency bugs. The large overhead in case of rlock is due to the nature of the benchmark itself where there are only sync primitives and each time a lock is acquired, the scheduler has to update the state of the lock and the thread and check if some threads can be released or not.

Conclusion

CoSched offers a practical approach to testing concurrent Python programs by controlling thread execution through synchronization primitives and a flexible scheduler. Its use of greenlets ensures compatibility with

existing code, making it a valuable tool as Python moves beyond the GIL. Future enhancements could include more sophisticated scheduling policies and performance optimizations.

Acknowledgments

This project design sketch was suggested by Project Mentors Dylan Wolff and Zhao Huan. The project was developed as part of the CS6215 course project "Advanced Topics in Program Analysis" where the ideas were proposed by teaching assistant Yuntong Zhang. The benchmarks were adapted from the C concurrency testing tool Period [1].

References

1. Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled concurrency testing via periodical scheduling. In Proceedings of the 44th International Conference on Software Engineering (ICSE '22). Association for Computing Machinery, New York, NY, USA, 474–486. https://doi.org/10.1145/3510003.3510178

2. Fu, H., Wang, Z., Chen, X., & Fan, X. (2017). A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. Software Quality Journal, 26(3), 855–889. https://doi.org/10.1007/s11219-017-9385-3

3. Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. 2007. RELAY: static race detection on millions of lines of code. In Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC-FSE '07). Association for Computing Machinery, New York, NY, USA, 205–214. https://doi.org/10.1145/1287624.1287654

4. Agarwal, Pratyush, et al. "Stateless model checking under a reads-value-from equivalence." International Conference on Computer Aided Verification. Cham: Springer International Publishing, 2021.

5. Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. 2021. Detecting concurrency vulnerabilities based on partial orders of memory and thread events. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 280–291. https://doi.org/10.1145/3468264.3468572

6. Dylan Wolff, Zheng Shi, Gregory J. Duck, Umang Mathur, and Abhik Roychoudhury. 2024. Greybox Fuzzing for Concurrency Testing. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), Vol. 2. Association for Computing Machinery, New York, NY, USA, 482–498. https://doi.org/10.1145/3620665.3640389

7. Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound sequentialization for concurrent program verification. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022). Association for Computing Machinery, New York, NY, USA, 506–521. https://doi.org/10.1145/3519939.3523727

8. Rasin, Dan, Orna Grumberg, and Sharon Shoham. "Modular verification of concurrent programs via sequential model checking." Automated Technology for Verification and Analysis: 16th International

Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings 16. Springer International Publishing, 2018.