

Software Transactional Memory in Pure Python

Dillon Niederhut^{‡*}

Abstract—There has been a growing interest in programming models for concurrency. Strategies for dealing with shared data amongst parallel threads of execution include immutable (as in Erlang) and locked (as in Python) data structures. A third option exists, called transactional memory (as in Haskell), which includes thread-local journaling for operations on objects which are both mutable and globally shared. Here, we present TraM, a pure Python implementation of the TL2 algorithm for software transactional memory.

Index Terms—concurrency, threading, transactional memory

Introduction

Methods for sharing resources between multiple processes have been of academic interest for quite some time [Lampert_1978]. Recently, the need for handling coincident events in client-server interactions and the increasing scale of easily available data, especially in combination with the reduced momentum in increasing the clock speed of CPUs, have promoted discussions of concurrent software architecture [Lampert_et_al_1997]. In an ideal world, a computationally intensive task could split its work across the cores of a CPU in a way that does not require changes to the structure of the task itself. However, sharing work or any other kind of data in a concurrent system removes the guarantee that events occur in a strict linear order, which in turn disrupts the atomicity and consistency inherent to single threads of control. To see concretely how this might become a problem, consider the example below, in which several Python threads are attempting to increment a global counter¹.

```
from threading import Thread
import time
import random

unsafe_number = 0

def unsafe_example():
    wait = random.random() / 10000
    global unsafe_number
    value = unsafe_number + 1
    time.sleep(wait)
    unsafe_number = value
```

In this particular example, we are forcing Python to behave asynchronously by inserting sleeping calls, which allow the interpreter to interrupt the execution of our "unsafe example", and give control in the interpreter to another thread. This creates the

opportunity for inconsistent data, as the value of the "unsafe number" might have changed between the time a thread reads it, and the time a thread overwrites it. Thus, in the code below, we would expect to see consistent output of the number 10, but in practice will see something smaller, depending on the length of the wait and the architecture of the system running the example².

```
if __name__ == '__main__':

    thread_list = []
    for _ in range(10):
        thread = Thread(target=unsafe_example)
        thread_list.append(thread)
    for thread in thread_list:
        thread.start()
    for thread in thread_list:
        thread.join()

    print(unsafe_number)
```

```
$ python run_test.py
$ 5
$ python run_test.py
$ 4
$ python run_test.py
$ 6
```

Models for handling shared data in memory, specifically, have included restricting data structures into being immutable, or restricting access into those data structures with locking mechanisms. The former solution is disadvantaged by the CPU and memory costs of redundant data copies, while the latter suffers from deadlocks and leaky abstractions [Peyton_2002].

A third approach involves the use of local operation journals that are validated before any data is modified in place [Le_et_al_2016]. The strategy is similar to that used in database transactions and self-correcting file systems, where atomicity, consistency, and durability are enforced in part by maintaining a history of changes that have been made to the copy of data in memory but not yet persisted to the copy of data on disk. Within a concurrent software application, each thread of control can keep a similar history of proposed changes, and only modify the data object shared across all threads once that journal of changes has been approved. This strategy, where incremental changes in each thread are applied all at once to shared, mutable structures, is called software transactional memory (STM).

Software Transactional Memory

A specific implementation of STM, called Transactional Locking Version II (TL2) was recently proposed which avoids most of the copy-based and lock-based errors, along with the temporary unsafety characteristic of earlier STM algorithms, by versioning its data [Dice_et_al_2006]. Briefly, the algorithm works by setting

* Corresponding author: dillon.niederhut@gmail.com

‡ Enthought

up a local journal for each thread, where proposed modifications to shared data are kept. If no other thread has modified the original data structures during the time needed to calculate the proposed changes, those changes are swapped in memory for the old version of the internal data.

Under work loads that are predominantly read operations, TL2 outperforms lock-based strategies because it employs non-blocking reads. Under workloads that are dominated by writes to shared data structures, TL2 outperforms immutable strategies in that it is possible to only copy pieces of a structure. The actual performance gain varies based on workload characteristics and number of CPUs, but a comparison against a coarse-grained POSIX mutex strategy shows gains of more than an order of magnitude; and, comparisons against previous implementations of STM are faster by constant factors roughly between 2 and 5 [Dice_et_al_2006].

The Python Implementation

The TraM package (available at <https://github.com/deniederhut/tram>) attempts to recreate the TL2 algorithm for transactional memory pythonically, and is not a one-for-one transliteration of the original Java implementation. The chief difference is that it does not use a global counter whose state is maintained by primitives in the language, but is instead using the system clock. This comes with the additional cost of making system calls, but prevents us from the necessity of building a concurrency strategy inside our concurrency strategy, since the clock state must be shared across all threads.

The algorithm starts by entering a retry loop, that will attempt to conduct the transaction a limited number of times before raising an exception. Ideally, this number is large enough that the retry limit would only be reached in the event of a system failure.

```
def transaction(self, *instance_list, write_action,
                read_action=None):
    """Conduct threadsafe operation"""
    if read_action is None:
        read_action = self.read
    retries = self.retries
    time.sleep(self.sleep) # for safety tests
    while retries:
        with self:
            read_list = read_action(instance_list)
            self.write(write_action(instance_list,
                                   read_list))
            self.sequence_lock(instance_list)
            time.sleep(self.sleep) #
            try:
                self.validate()
                time.sleep(self.sleep) #
                self.commit()
            except ValidationError:
                pass
            except SuccessError:
                break
            finally:
                self.sequence_unlock(instance_list)
                self.decrement_retries()
```

It then creates two thread local logs. In our Python implementation, this occurs inside of a context manager.

```
def __enter__(self):
    """initialize local logs"""
    self.read_log = []
    self.write_log = []
```

It then reads local copies of data into its read log, and writes proposed changes into its write log. The algorithm itself is agnostic to what the reading and writing operations actually do.

```
def write(self, pair_list):
    """Write instance-value pairs to write log"""
    for instance, value in pair_list:
        self.write_log.append(
            Record(instance, value, time.time())
        )
```

This makes it easy to extend TraM's threadsafe objects by writing decorated, transactional methods.

```
def __iadd__(self, other):
    @atomic
    def fun(data, *args, **kwargs):
        return data + other
    do = Action()
    do.transaction(self, write_action=fun)
    return self
```

The algorithm then compares the version numbers of the original objects against the local data to see if they have been updated.

```
def validate(self):
    """Raise exception if any instance reads are
    no longer valid
    """
    for record in self.read_log:
        if record.instance.version > record.version:
            raise ValidationError
```

If not, a lock is acquired only long enough to accomplish two instructions: pointing the global data structure to the locally modified data; and, updating the version number.

```
def commit(self):
    """Commit write log to memory"""
    for record in self.write_log:
        record.instance.data = record.value
        record.instance.version = record.version
    raise SuccessError
```

If the read log is not validated, the entire operation is aborted and restarted. This suggests that the worst case scenario for TL2 is when several threads are attempting to write to a single object, as the invalidated threads will waste resources cycling through the retry loop.

Using a similar safety test, we can see that the TraM Int object correctly handles separate threads attempting to update its internal data, even when the actions performed by each thread cannot be guaranteed to be atomic themselves.

```
from tram import Int

def safe_example():
    global safe_number
    safe_number += 1

if __name__ == '__main__':

    thread_list = []
    for _ in range(10):
        thread = Thread(target=safe_example)
        thread_list.append(thread)
    for thread in thread_list:
        thread.start()
    for thread in thread_list:
        thread.join()

    print(safe_number)
```

```
$ python run_test.py
$ 10
$ python run_test.py
$ 10
$ python run_test.py
$ 10
```

Future Directions

This implementation of TL2 is specifically limited by implementation details of CPython, namely the global interpreter lock (GIL), which ensures that all actions are executed in a linear order given a single Python interpreter. Python's libraries for concurrent operations, including threading and the more modern `async*`s, are still executed within a single interpreter and are therefore under control of the GIL. Python's library for multiple OS threads, multiprocessing, will perform operations in parallel, but has a small number of data structures that are capable of being shared.

In our motivating example, we have tricked the interpreter into behaving as if this is not the case. While it is probably not a good idea to encourage software developers to play fast and loose with concurrency, there is a lot to be said for compartmentalizing the complexity of shared data into the shared data structures themselves. Concurrent programs are notoriously difficult to debug, and part of that complexity has to do with objects leaking their safety abstraction into the procedures trying to use them.

However, the work on creating a transactional branch of PyPy shows that there is some interest in concurrent applications for Python. PyPy's implementation of STM is currently based on a global processing queue, modeled after the threading module, with the transactional algorithms written in C [Meier_et_al_2014]. We hope that presenting an additional abstraction for composing transactional objects will encourage the exploration of STM specifically and concurrency generally, in the python community. Even if this does not occur, seeing the algorithm written out in a read-friendly language may serve as an education tool, especially as a starting point for creating a more clever version of the implementation itself.

As an algorithm for threadsafe objects, TL2 itself has two major limitations. The first, mentioned above, is that the algorithm depends on a version clock which is used to create a post-hoc, partial synchronization of procedures. In the original implementation, this is a shared, global, mutable counter, which is incremented every time any object is updated. In this implementation, it is the system clock, which is shared but no longer mutable by structures inside the algorithm. Both strategies have drawbacks.

The second major limitation is that attaching versions to objects works fine for *updating* data, but not for *deleting* the object. In garbage collected languages like Java and Python, we can rely on the runtime to keep track of whether those objects are still needed, and can remove them only after their last reference. Any implementation in a language which without automated memory management will need its own solution to the deletion of versioned data to avoid memory leaks.

REFERENCES

- [Dice_et_al_2006] Dice, D., Shalev, O., & Shavit, N. (2006). Transactional locking II. In *International Symposium on Distributed Computing* (pp. 194-208). Springer Berlin Heidelberg. https://doi.org/10.1007/11864219_14.
- [Lamport_1978] Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. In *Communications of the ACM*, 21. (pp. 558-565).
- [Le_et_al_2016] Le, M., Yates, R., & Fluet, M. (2016). Revisiting software transactional memory in Haskell. <https://doi.org/10.1145/2976002.2976020>. In *Proceedings of the 9th International Symposium on Haskell* (pp. 105-113). ACM.
- [Meier_et_al_2014] Meier, R., & Rigo, A. (2014). A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE*. ACM. <https://doi.org/10.1145/2633301.2633305>.
- [Peyton_2002] Peyton Jones, S. (2002). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In *Engineering theories of software construction* (pp. 47-96).
- [Lamport_et_al_1997] Shavit, N. & Touitou, D. (1997). Software transactional memory. *Distributed Computing*, 10. (pp. 99-116). <http://doi.org/10.1007/s004460050028>.

1. Code has been modified from the original to avoid overfull hbox per the proceedings requirements

2. The order of magnitude for the wait time was chosen by experimentation to produce results between 3 and 7 on a 2.7GHz Intel Core i5.