# Software Transactional Memory with PyPy

Armin Rigo

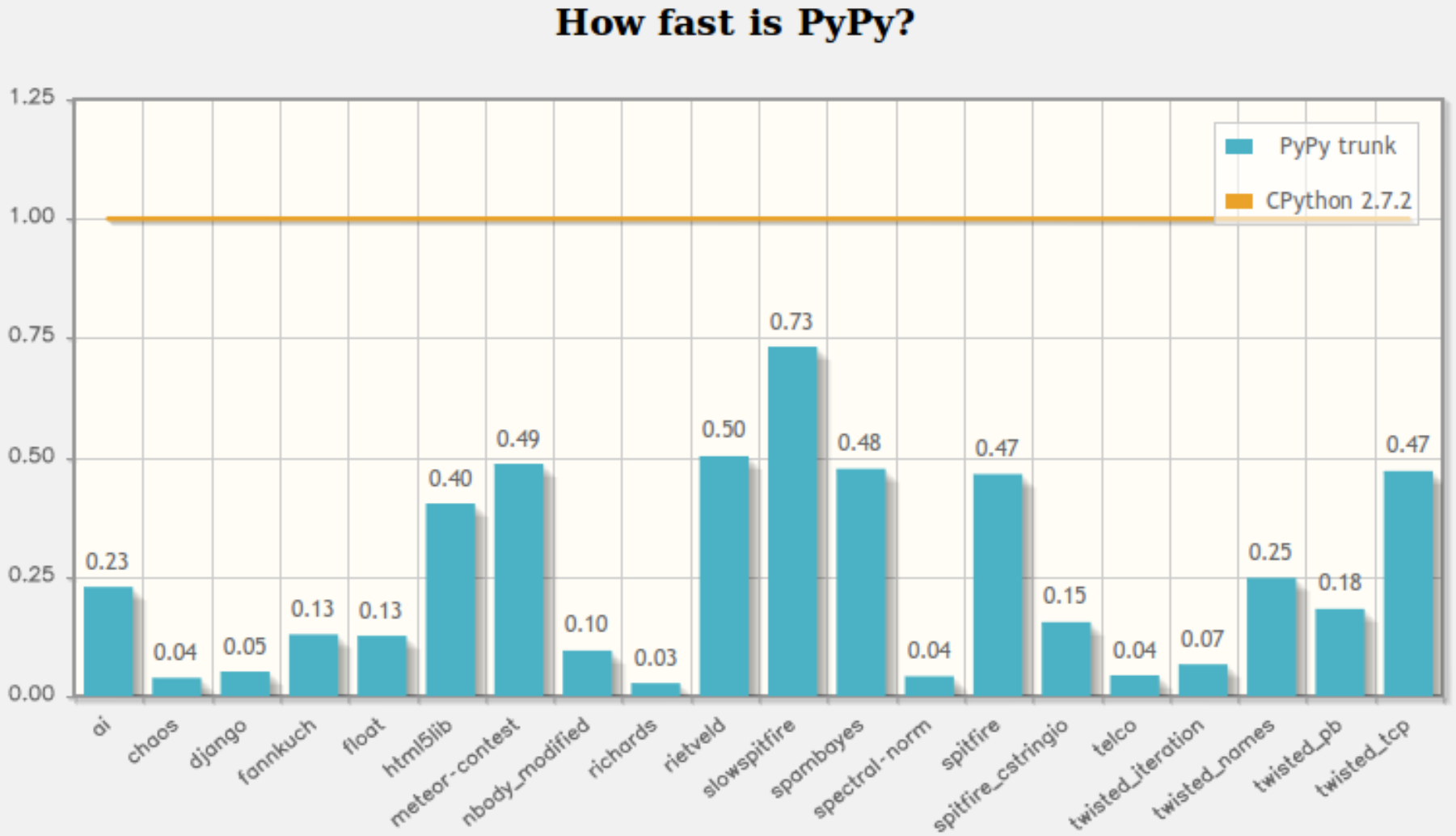
PyCon ZA 2013

4th October 2013

# Introduction

- me: Armin Rigo
- what is PyPy: an alternative implementation of Python
- very compatible
- main focus is on speed

# Introduction




Plot 1: The above plot represents PyPy trunk (with JIT) benchmark times normalized to CPython. Smaller is better.

It depends greatly on the type of task being performed. The geometric average of all benchmarks is 0.16 or **6.3** times *faster* than CPython

# SQL by example

# SQL by example

```
BEGIN TRANSACTION;
SELECT * FROM ...;
UPDATE ...;
COMMIT;
```

# Python by example

```
...
x = obj.value
obj.value = x + 1
...
```

# Python by example

```
...
obj.value += 1
...
```

# Python by example

```
...
x = obj.value
obj.value = x + 1
...
```

# Python by example

```
begin_transaction()
x = obj.value
obj.value = x + 1
commit_transaction()
```

# Python by example

```
the_lock.acquire()
x = obj.value
obj.value = x + 1
the_lock.release()
```

# Python by example

```
with the_lock:
    x = obj.value
    obj.value = x + 1
```

# Python by example

```
with atomic:
    x = obj.value
    obj.value = x + 1
```

# Locks != Transactions

```
BEGIN TRANSACTION;
SELECT * FROM ...;
UPDATE ...;
COMMIT;
```

```
BEGIN TRANSACTIO
SELECT * FROM ..
UPDATE ...;
COMMIT;
```

# Locks != Transactions

```
with the_lock:
  x = obj.val
  obj.val = x + 1
```

```
with the_lock:
  x = obj.val
  obj.val = x + 1
```

# Locks != Transactions

```
with atomic:
    x = obj.val
    obj.val = x + 1
```

```
with atomic:
    x = obj.val
    obj.val = x + 1
```

# STM

- Transactional Memory
- advanced but not magic (same as databases)

# By the way

- STM replaces the GIL (Global Interpreter Lock)
- any existing multithreaded program runs on multiple cores

# By the way

- the GIL is necessary and very hard to avoid, but if you look at it like a lock around every single subexpression, then it can be replaced with *with atomic* too

# So...

- yes, any existing multithreaded program runs on multiple cores
- yes, we solved the GIL
- great

# So...

- no, it would be quite hard to implement it in standard CPython
- too bad for now, only in PyPy
- but it would not be completely impossible

# But...

- but only half of the story in my opinion *:-)*

# Example 1

```
def apply_interest(self):
    self.balance *= 1.05


for account in all_accounts:
    account.apply_interest()
```

# Example 1

```
def apply_interest(self):
    self.balance *= 1.05


for account in all_accounts:
    account.apply_interest()
    ^^^ run this loop multithreaded
```

# Example 1

```
def apply_interest(self):
    #with atomic: --- automatic
        self.balance *= 1.05

for account in all_accounts:
    add_task(account.apply_interest)
run_all_tasks()
```

# Internally

- *run_all_tasks()* manages a pool of threads
- each thread runs tasks in a *with atomic*
- uses threads, but internally only
- very simple, pure Python

# Example 2

```
def next_iteration(all_trains):
    for train in all_trains:
        start_time = ...
        for othertrain in train.deps:
            if ...:
                start_time = ...
        train.start_time = start_time
```

# Example 2

```
def compute_time(train):
    ...
    train.start_time = ...

def next_iteration(all_trains):
    for train in all_trains:
        add_task(compute_time, train)
    run_all_tasks()
```

# Conflicts

- like database transactions
- but with *objects* instead of *records*
- the transaction aborts and retries automatically

# Inevitable

- "inevitable" (means "unavoidable")
- handles I/O in a *with atomic*
- cannot abort the transaction any more

# Current status

- basics work, JIT compiler integration almost done
- different executable (*pypy-stm* instead of *pypy*)
- slow-down: around 3x (in bad cases up to 10x)
- real time speed-ups measured with 4 or 8 cores
- Linux 64-bit only

# User feedback

- implemented:

```
Detected conflict:
  File "foo.py", line 58, in wtree
    walk(root)
  File "foo.py", line 17, in walk
    if node.left not in seen:
Transaction aborted, 0.047 sec lost
```

# User feedback

- not implemented yet:

```
Forced inevitable:
  File "foo.py", line 19, in walk
    print >> log, logentry
Transaction blocked others for XX s
```

# Asynchronous libraries

- future work
- tweak a Twisted reactor: run multithreaded, but use *with atomic*
- existing Twisted apps still work, but we need to look at conflicts/inevitables
- similar with Tornado, eventlib, and so on

# Asynchronous libraries

```
while True:
    events = epoll.poll()
    for event in events:
        queue.put(event)
```

And in several threads:

```
while True:
    event = queue.get()
    with atomic:
        handle(event)
```

# More future work

- look at many more examples
- tweak data structures to avoid conflicts
- reduce slow-down, port to other OS'es

# STM versus HTM

- Software versus Hardware
- CPU hardware specially to avoid the high overhead (Intel Haswell processor)
- too limited for now

# Under the cover

- 10’000-feet overview
- every object can have multiple versions
- the shared versions are immutable
- the most recent version can belong to one thread
- synchronization only at the point where one thread "steals" another thread’s most recent version, to make it shared
- integrated with a generational garbage collector, with one nursery per thread

# Summary

- transactions in Python
- a big change under the cover
- a small change for Python users
- (and the GIL is gone)
- this work is sponsored by crownfunding (thanks!)
- *Q & A*