Multi-core programming without threads: the transactional memory approach

Armin Rigo and Remi Meier

A common story

- Write a program or website
- The program gets popular
- Datasets grow bigger and bigger

➔ The program needs to scale

Problem

“We have all these cores, we really need to use more than one now”

- Multiple processes? Not if the program is too “irregular”
- Multiple threads?

Threads are hard

- Concurrent, independent execution streams
- Shared memory
  - Manual synchronization and coordination
  - Locks: Locks are hard

Our approach

- Not automatic parallelization, but automatic synchronization
- Programmer...
  - identifies code where parallel execution is ***possible***
  - splits code into atomic units that may run in any order
- Runtime...
  - ensures atomic and isolated execution
  - ensures serializable schedule
  - handles conflicts transparently

Example

```python
...
for block in pending:
    blockcount += 1
    transform(block)
    already_seen.add(block)
```

```python
...
tq = transaction.TransactionQueue()
for block in pending:
    tq.add(transform, block)
tq.run()
blockcount += len(pending)
already_seen.update(pending)
```

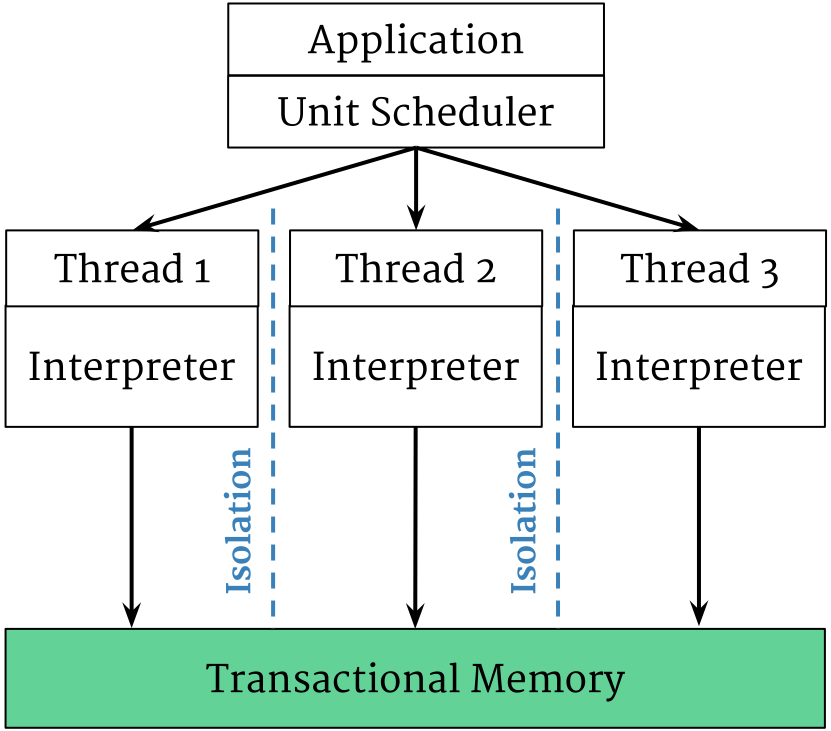
- The calls to `transform()` should be mostly independent
- But they do not need to be *fully* independent
- Benefits from parallelism **if possible**

Our solution

- Transactional memory (TM)
  - coarse-grained atomicity and serializability
- Modified Python VM (PyPy)
- All Python-level code executes in TM
  - no statically known code boundaries for “parallel units”
- Completely transparent / native:
  - TM in the virtual machine
  - no explicit, exposed TM
  - only “TransactionQueue”, an API for running parallel units

Speculative parallel execution

- Application enqueues units
- Units get scheduled
- Unit runs fully in one transaction
  - atomically, isolated
  - in parallel if possible
- Units schedule is serializable



Consequences

- No (exposed) threads, no manual synchronization
- Gradually introduce parallelism
  - programmer marks promising spots
  - if parallel execution is not possible, the code still executes correctly…
  - …but not faster: programmer may have to tweak the code to minimize conflicts
- Requires execution of **all** code in TM
  - performance challenge
  - so far, 25%-40% typical performance hit, up to 100% in some cases (but fully working :-)
  - Just-In-Time compiler may help more

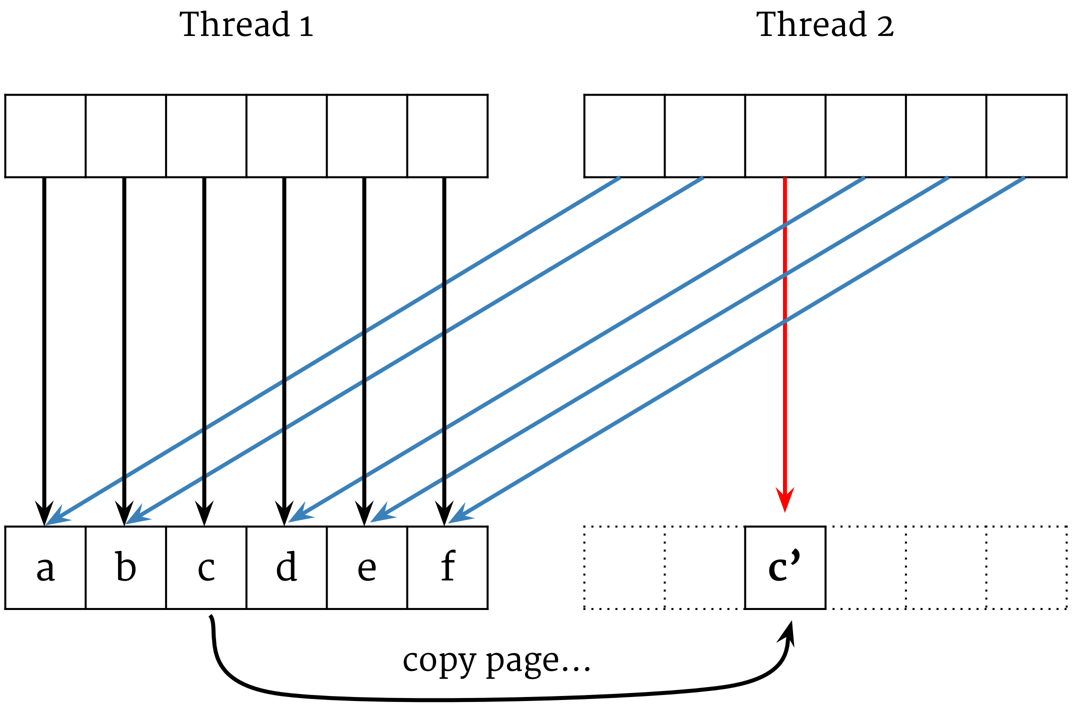
STMGC

- C library that handles both GC and STM
- Used by PyPy
  - benefits from PyPy being written in “RPython”, a flexible language we control
  - but can be used more generally

STMGC API

- Classical part: the GC
  - obj = allocate(size)
  - pushing/popping on the “root stack” of objects
  - finding pointers inside objects
- New part: STM
  - stm_read(obj)   /* no return value, extremely cheap */
  - stm_write(obj)   /* no return value */
  - stm_start_transaction(), stm_commit_transaction()
- Concurrent transactions see separate memory
- Unmodified memory pages are shared (with mmap)

Memory views



Conclusion

- Not automatic parallelization, but close
- Works on existing, irregular programs
  - No need to learn a specific model
- Drawback: the programmer may have to add some tweaks to reduce conflicts
  - But the program is always “correct”
- STMGC+PyPy shows a reasonable(?) performance hit so far