A Way Forward in Parallelising Dynamic Languages

Position Paper

Armin Rigo

www.pypy.org

Remigius Meier

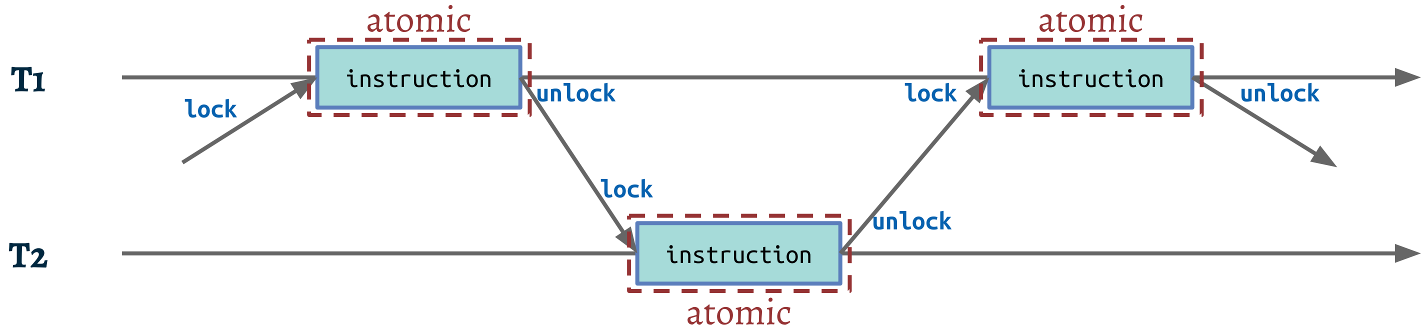
Department of Computer Science
ETH Zürich

Current situation

- Languages like Python & Ruby are very popular
- Concurrency using threads
  - e.g. background tasks
  - other models in other languages not covered here
- No parallelism in reference implementations
  - threads serialised using a single, global interpreter lock (**GIL**)
  - no speedup on multicore machines

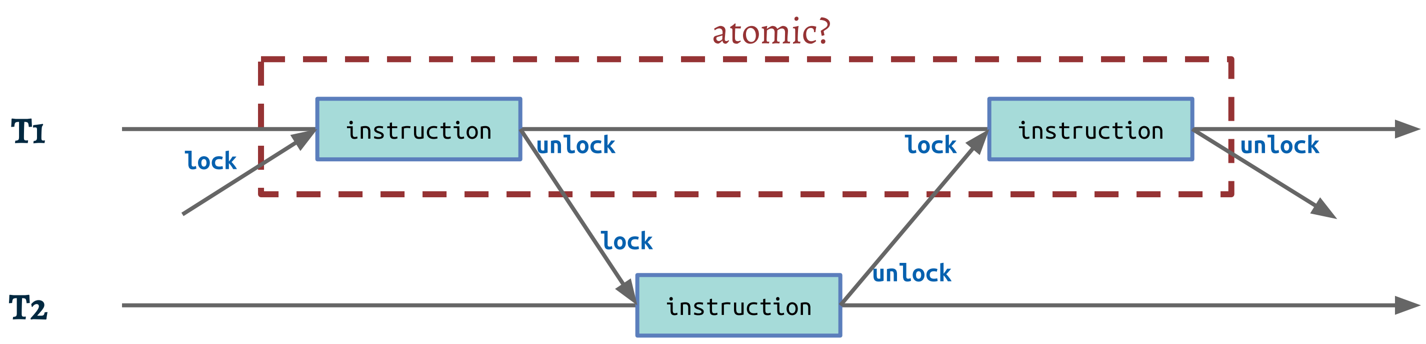
The GIL

- Initially: single-threaded interpreters
- For concurrency, provide threads
  - not for parallelism
- Challenges to adapt interpreter:
  - reference counting GC, concurrent access to lists / dicts / objects
- Easy solution:
  - ultra coarse-grained locking
  - acquire GIL around the execution of bytecode instructions
  - atomic & isolated execution



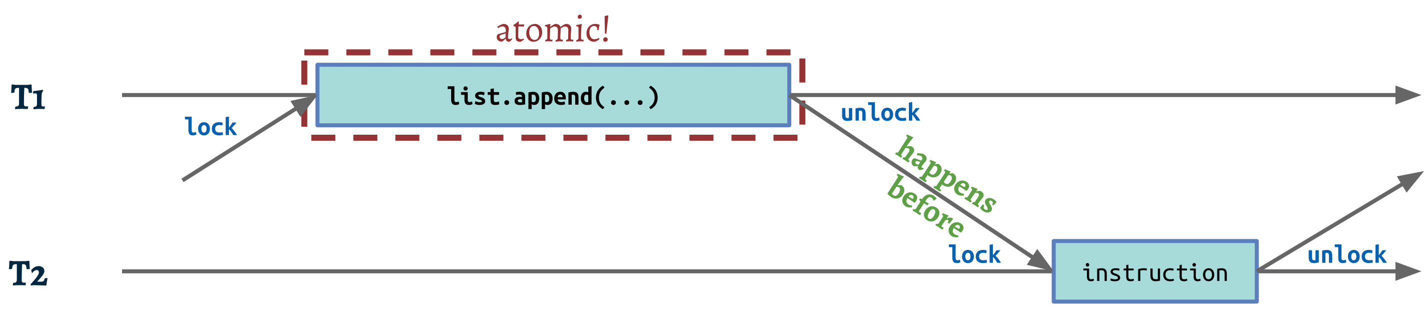
Consequences

- No parallelism
- GIL is interpreter-level sync, not available to applications
  - need application-level locking (semaphores, conditions, ...)
  - all challenges of concurrent programming still there



Consequences ✚

- Atomic & isolated instructions
  - things like `list.append()` are atomic
  - tons of websites mention this
  - latent races if language becomes really parallel —
- Sequential consistency
  - less surprises: “all variables volatile”
  - global sequential order of instructions



Sequential consistency

Example

<table>
<tr><th>Thread 1</th><th>Thread 2</th></tr>
<tr><td colspan="2"><code>A = B = 0</code></td></tr>
<tr><td><code>A = 1</code>
<code>if B == 0:</code></td><td><code>B = 1</code>
<code>if A == 0:</code></td></tr>
<tr><td colspan="2">At most one thread can enter here (seq. consistency)</td></tr>
</table>

No sequential consistency:

→ both threads could enter (e.g. x86 CPUs)

Where do we want to go?

- Remove / replace the GIL
  - allow threads to run in parallel
- Keep GIL semantics
- Keep current APIs
  - no changes to existing concurrent / threaded apps required

- Then: find new ways
  - better synchronisation mechanism than locking
  - new models for new apps: AME, tuple spaces, actors, ...

Avoiding the GIL

Approaches

1. Fine-grained locking
2. Shared-nothing
3. Transactional memory

Comparison

- Performance
  - single threaded
  - parallelisation
- Backwards compatibility:
  - support for existing threaded applications
  - sequential consistency
  - atomic instructions (`list.append()`)
- Implementation effort (interpreter)
  - large open-source communities
- Bonus: better application-level synchronisation mechanism to replace locking
  - e.g. exposing interpreter-level synchronisation to application
  - better = composable, no deadlocks, scalable / parallelisable

1. Fine-grained locking

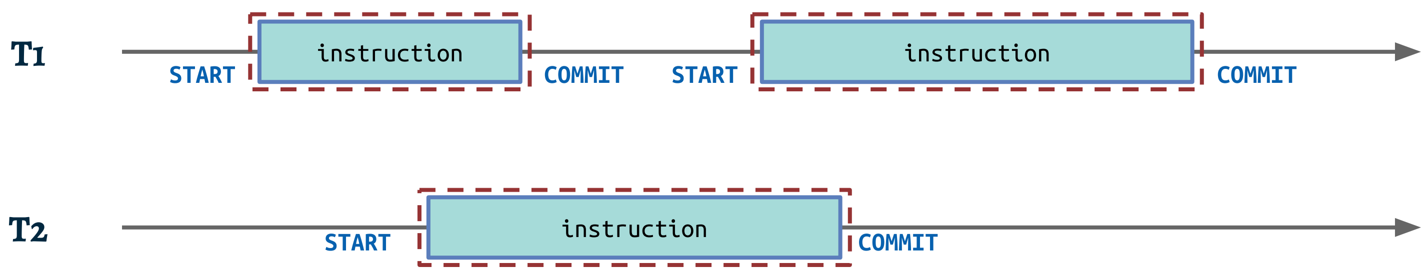
- Replace GIL with locks on objects / data structures
  - accessing different objects can run in parallel
- Possible to keep GIL semantics
- Harder to implement
  - many locks → deadlock risks
  - no centralised implementation
- Overhead of lock/unlock on objects
  - e.g. Jython depends on JVM for good lock removal
  - coarse GIL has less overhead
- Still needs application-level synchronisation

2. Shared-nothing

- Workaround instead of direct replacement
- Each independent part of the program gets its own interpreter (one GIL each)
- Simple implementation
- Not compatible with (most) existing threaded applications
  - extracting independent parts
- Explicit communication
  - good: clean model, no locks
  - bad: communication overhead

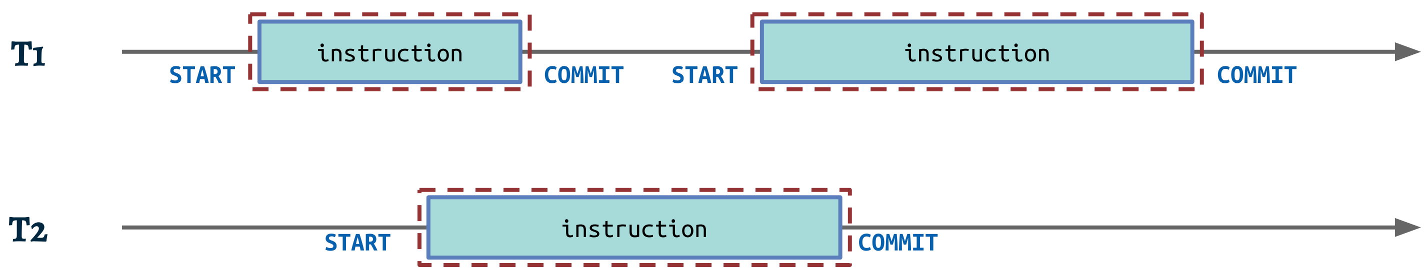
3. Transactional memory

- Transactions guarantee atomicity & isolation
- Direct replacement for the GIL
  - `lock` → transaction **START**
  - `unlock` → transaction **COMMIT**
- Keeps GIL semantics
  - sequential consistency (transaction schedule is serialisable)
  - instruction atomicity



3. Transactional memory

- Optimistically tries to execute in parallel
- Overhead:
  - validating memory accesses
  - start/commit/abort transactions
- Two categories:
  - HTM: hardware implementation
  - STM: software implementation
    also: some hybrids



3.1 Hardware TM

- Easy, direct replacement with low overhead (<40%)
- Implementations not widespread
- Limitations (Intel Haswell):
  - false sharing on cache line level
  - transaction size ~ cache size
- Some transactions never succeed → needs fallback
  - in current approaches: GIL
  - our experiments suggest the fallback is needed often (limited scaling)
- Backwards compatible with existing applications
- Centralised implementation
- Still needs application-level synchronisation

3.2 Software TM

- Same benefits as HTM, except performance
  - often overhead of 100-1'000%
  - commonly scale well, but no speedup on <8 cores
- No hardware limits
- Unlimited transaction size
  - transactions can be exposed to the application as atomic blocks
  - not possible with HTM
- Atomic blocks
  - guarantee atomicity & isolation
  - composable, deadlock-free
  - move global locking protocol to the language implementation

Rough summary

	GIL	Fine-grained locking	Shared-nothing	HTM	STM
single-threaded performance	++	+	++	++	--
multi-threaded performance	--	+	+	++	++
backwards compatibility	++	++	--	++	++
implementation effort	++	-	++	++	++
synchronisation mechanism	o	o	+	o	++

- Picture not entirely clear
- Fine-grained locking & shared-nothing mixed
- HTM looks good
    - may never be widespread enough
    - hardware limitations in the current generation

Rough summary

- STM
  - synchronisation mechanism: atomic blocks
- Push for an easier parallel programming environment
  - sequential consistency & atomic blocks
  - going further than e.g. Java, C#, etc.
- Ultimately needs better performance
  - maybe the long-term solution?
  - hybrid TMs?
- Our direction of research:

  Replace GIL with STM and move synchronisation from the application to the language implementation

Preview of PyPy-STM

- PyPy
  - Python interpreter
  - RPython: toolchain for producing dynamic language VMs
- Our own STM system: *STMGC-C7*
  - C library providing STM & garbage collection
  - optimised for use in dynamic language VMs (to replace GIL)
  - tight TM-GC cooperation: shared barriers, object lifetime optimisations
  - low overhead

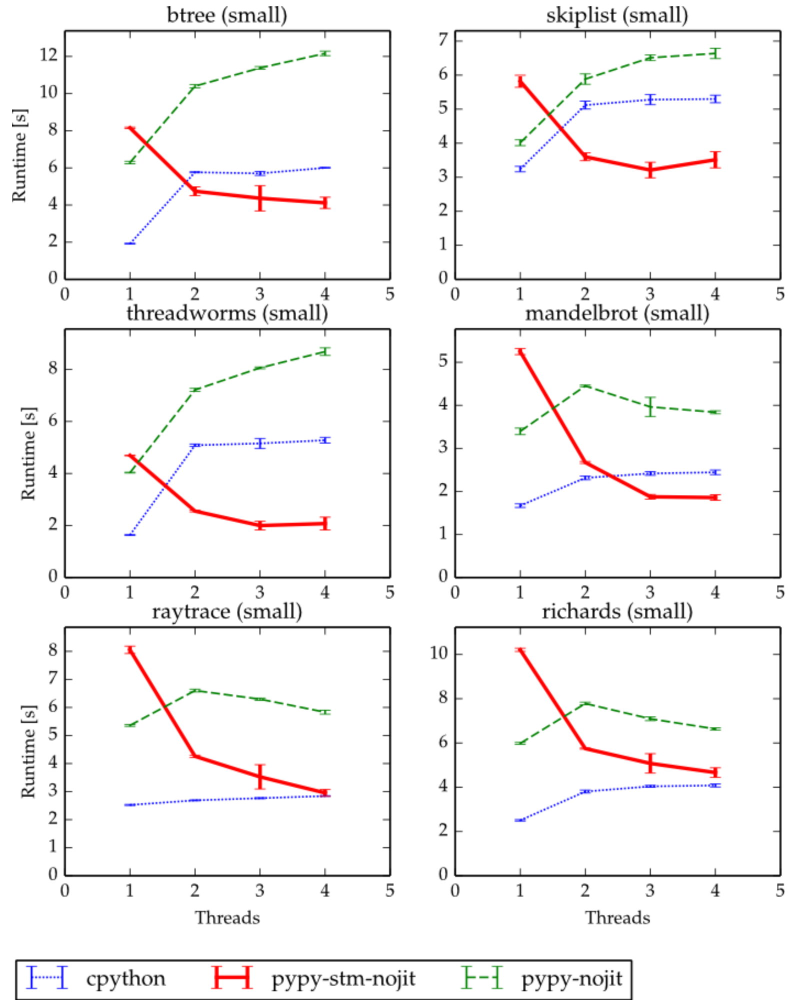
Preview of PyPy-STM

Single Threaded

- Max. 100% overhead
- Avg. 45% overhead

Multithreaded

- Speedup 1.1 - 1.9×



Questions & Discussion