Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

Rei Odaira, Jose G. Castanos and Hisanobu Tomari

IBM Research and University of Tokyo

April 8, 2014
The Problem:

Script languages interpreters often use a Global Interpreter Lock (GIL)

- Allow easy interpreter design
- But serialize all execution
Observation:

Work for single-thread performance

- JIT compilers (Rubinius, PyPy, etc.)
- HPC Ruby

No scalability on multicores
Observation:

Work for multicore performance

- Fine-grain synchronization in interpreters (e.g. JRuby)

Replacing GIL with fine-grain locking leads to incompatible extension libraries support
Approach:

- Replace GIL with HTM to get scalability
- How/Where to put transactions?
- Dynamic-length transactions
- Evaluation on real applications
Outline

1. Background
2. Eliminating GIL with HTM
3. Dynamic-length transactions
4. Evaluation
5. Conclusions
## Hardware Transactionnal Memory

### Goals:
- Best-effort mechanism to improve fast-paths
- Simple interface to define critical sections
- Intel RTM API (?):
  - XBEGIE
  - XEND
  - XABORT
  - XTEST
- Programmer needs to provide fallback handler

### Lifecycle:
- Critical sections are executed optimistically
- Check for conflicts eagerly using cache coherency protocol
- Aborted and Rollbacked if conflict or capacity overflow
- Committed to global memory only upon validation

---

Rei Odaira, Jose G. Castanos and Hisano
Background on Ruby

**Dynamic language**
- Dynamic typing
- meta-programming
- closures
- Just like Python or Perl

**CRuby reference implementation**
- Bytecode interpretation
- Use native threads
- No JIT
Background on GIL

**Not a normal lock:**
- Always held when executing
- Acquired when thread starts, released when it stops
- Released for blocking operations (I/O)
- To enable ”concurrency”: Yield points

**Yield points:**
- Release GIL; sched_yield(); Acquire GIL;
- Set at loop-back edges and when exiting a method/block
- Heavy ⇒ does not happen at every yield point
- Timer thread set a yield flag every 250ms
- No yield when only one thread
Outline

1. Background
2. Eliminating GIL with HTM
3. Dynamic-length transactions
4. Evaluation
5. Conclusions
Approach

Based on Transactional Lock Elision

- Replace acquiring/release GIL with transaction begin/end
- Use GIL as a fallback
- Retry on abort (depends on the cause)
- Transactions span 2 yield points
- Remove timer thread
Aborts

Yield points are too coarse-grain

- Causes a lot of capacity aborts

⇒ Add more yield points:
  - getlocal, getinstancevariable, getclassvariable (read variable)
  - send (call method or block)
  - opt_plus, opt_aref, opt_minus, opt_mult (classic operations)

New yield points

- Located at bytecode boundaries
- Are safe, unless application is not synchronized correctly
- Applications should not rely on GIL for synchronization
- in NAS Parallel Benchmarks, half of the instructions are yield points
Outline

1. Background
2. Eliminating GIL with HTM
3. Dynamic-length transactions
4. Evaluation
5. Conclusions
Optimization - dynamic-length transactions

**Transaction length:**
- Tradeoff between XBEGIN/XEND cost, and abort rate
- Best choice depends on the yield point
  - imagine a syscall just after a yield point

**Transactions can now span more than 2 yield point**
- The size of a transaction (number of yield points it crosses):
  - Count best length for transaction at each yield point
  - Count number of transactions start at each yield point (for abort rate)
  - Start high (long transaction)
  - Shorten if abort rate above a threshold
  - Stop profiling when steady state reached
## Sources of conflicts

1. Global variables pointing to the running thread
2. Single global linked list for free objects
3. Mark&Sweep GC
   - Not parallel (causes conflicts)
   - Too long (overflows capacity)
   ⇒ has to be from GIL
4. Shared inline caches (cache hashtable lookup to access methods/variables)
5. False sharing in thread structures
Optimization - conflict removal

Solutions

1. Put those global variables in thread-local storage
2. Thread-private free objects lists
3. Increase heap size 1000x (400MB) to GC less often
4. Change caching logic (less cache updates, less caches misses)
5. Dedicate cache line to thread structures
Outline

1. Background
2. Eliminating GIL with HTM
3. Dynamic-length transactions
4. Evaluation
5. Conclusions
Evaluation

Machines

- Intel Haswell with RTM
- IBM zEC12 with HTM support

- 2 microbenchmarks
- Ruby NPB (NAS)
- WEBrick HTTP server
- Ruby on Rails
Achieved up to 4.4-fold speed-up in FT.

HTM-dynamic was the best in 6 of 7 benchmarks.

HTM-1 suffered high overhead.

HTM-256 incurred high abort ratios.
NPB on Haswell machine

Throughput (1 = 1 thread GIL)

Number of threads

Throughput (1 = 1 thread GIL)

Number of threads

Throughput (1 = 1 thread GIL)

Number of threads

Throughput (1 = 1 thread GIL)

Number of threads

Throughput (1 = 1 thread GIL)

Number of threads

GIL
HTM-1
HTM-16
HTM-256
HTM-dynamic

Rei Odaira, Jose G. Castanos and Hisanobu Tomari
HTM in CRuby
20 / 26
Differences with Intel’s RTM and IBM’s HTM

RTM has a learning algorithm

- Micro-benchmark write different sizes
- Success ratio increase *gradually*, not sharply
- This conflicts with HTM-dynamic, making adjustment phase longer
- A longer benchmark run achieves a steady-state, with HTM-dynamic performing best
Optimizations impact

New yield points
- Without new yield points, all NPB benchmarks have 20% slowdown over GIL

Remove conflicts
- Without conflict removal optimts, HTM provides no acceleration in any benchmark
Further Measurements

Abort rate

- HTM-dynamic targets abort ratio of 1%
- Observed abort ratio is between 0.5% and 2.5%
- No correlation between observed abort ratio and scalability

Single thread overhead

- When only 1 thread, use GIL instead of HTM
  ⇒ 5-14% overhead in microbenchmarks, due to:
    ▶ checks at yield points
    ▶ newly added yield points
    ▶ slow access to Pthread’s thread local storage (z/OS specific)
Outline

1. Background
2. Eliminating GIL with HTM
3. Dynamic-length transactions
4. Evaluation
5. Conclusions
Conclusions

Summary

- Use HTM to replace GIL in Ruby’s interpreter
- Keep compatibility with extension libraries
- Get up to 4.4 speedup with 12 threads
  - Required small code modifications
  - Small changes to remove conflicts
  - Introduce dynamically-sized transactions

Future work: Remove GIL in Python

Work on PyPy as GC in CPython is refcount
Thanks for your attention!