Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

Rei Odaira
IBM Research – Tokyo,

Jose G. Castanos
IBM Research – Watson Research Center
Global Interpreter Locks in Scripting Languages

- Scripting languages (Ruby, Python, etc.) everywhere. → Increasing demand on speed.

- Many existing projects for single-thread speed.
  - JIT compiler (Rubinius, ytljit, PyPy, Fiorano, etc.)
  - HPC Ruby

- Multi-thread speed restricted by global interpreter locks.
  - Only one thread can execute interpreter at one time.
  - No parallel programming needed in interpreter.
  - No scalability on multi cores.
Hardware Transactional Memory (HTM) Coming into the Market

- Improve performance by simply replacing locks with TM?
- Realize lower overhead than software TM by hardware.

IBM
Blue Gene/Q
2012

IBM
zEC12
2012

Sun Microsystems
Rock Processor
Cancelled

Intel
Transactional Synchronization eXtensions, 2013
Our Goal

- What will the performance of real applications be if we replace GIL with HTM?
  - Global Interpreter Lock (GIL)

- What modifications and new techniques are needed?
Our Goal

- What will the performance of real applications be if we replace GIL with HTM?
  - Global Interpreter Lock (GIL)

- What modifications and new techniques are needed?

- Eliminate GIL in Ruby through zEC12’s HTM
- Evaluate Ruby NAS Parallel Benchmarks

atomic { }

zEC12
Related Work

- Eliminate Python’s GIL with HTM
  - Micro-benchmarks on non-cycle-accurate simulator [Riley et al., 2006]
  - Micro-benchmarks on cycle-accurate simulator [Blundell et al., 2010]
  - Micro-benchmarks on Rock’s restricted HTM [Tabba, 2010]

- Eliminate Ruby and Python’s GILs with fine-grain locks
  - JRuby, IronRuby, Jython, IronPython, etc.
  - Huge implementation effort
  - Incompatibility in multithreaded execution of class libraries

- Eliminate the GIL in Ruby ...
  - through HTM on a real machine ...
  - and evaluate it using non-micro-benchmarks.
  - Small implementation effort
  - No incompatibility problem in the class libraries
Outline

- Motivation
- Transactional Memory
- GIL in Ruby
- Eliminating GIL through HTM
- Experimental Results
- Conclusion
Transactional Memory

- At programming time
  - Enclose critical sections with transaction begin/end instructions.

- At execution time
  - Memory operations within a transaction observed as one step by other threads.
  - Multiple transactions executed in parallel as long as their memory operations do not conflict.
  -> Higher parallelism than locks.

```
lock();
a->count++;
unlock();

Thread X

tbegin();
a->count++;
tend();

Thread Y

tbegin();
a->count++;
tend();

tbegin();
a->count++;
tend();

tbegin();
b->count++;
tend();
```
HTM in zEC12

- Instruction set
  - TBEGIN: Begin a transaction
  - TEND: End a transaction
  - TABORT, NTSTG, etc.

- Micro-architecture
  - Hold read set in L1 and L2 caches (~2MB)
  - Hold write set in L1 cache and store buffer (8KB)
  - Conflict detection using cache coherence protocol

- Roll back to immediately after TBEGIN in the following cases:
  - Read set and write set conflict
  - Read set and write set overflow
  - Restricted instructions (e.g. system calls)
  - External interruptions, etc.

```
TBEGIN
if (cc!=0) goto abort handler
...
...
TEND
```
Ruby Multi-thread Programming and GIL based on 1.9.3-p194

- Ruby language
  - Program with Thread, Mutex, and ConditionVariable classes

- Ruby virtual machine
  - Ruby threads assigned to native threads (Pthread)
  - Only one thread can execute the interpreter at any given time due to the GIL.
  
- GIL acquired/released when a thread starts/finishes.
- GIL yielded during a blocking operation, such as I/O.
- GIL yielded also at pre-defined yield-point bytecode ops.
  - Conditional/unconditional jumps, method/block exits, etc.
How GIL is Yielded in Ruby

- It is too costly to yield GIL at every yield point.
  → Yield GIL every 250 msec using a timer thread.

```
if (flag is true){
  gil_release();
  sched_yield();
  gil_acquire();
}
```

Actual implementation is more complex to ensure fairness.
Outline

- Motivation
- Transactional Memory
- GIL in Ruby
  - Eliminating GIL through HTM
    - Basic Algorithm
    - Dynamic Adjustment of Transaction Lengths
    - Conflict Removal
- Experimental Results
- Conclusion
Eliminating GIL through HTM

- Execute as a transaction first.
  - Begins/ends at the same points as GIL’s yield points.

- Acquire GIL after consecutive aborts in a transaction.

Diagram:
- Begin transaction
- End transaction
- Conflict
- Retry if abort
- Wait for GIL release
- Acquire GIL in case of consecutive aborts
- Release GIL
Beginning a Transaction

- **Persistent aborts**
  - Overflow
  - Restricted instructions

- **Otherwise, transient aborts**
  - Read-set and write-set conflicts, etc.

- **Abort reason reported by CPU**
  - Using a specified memory area

```ruby
if (TBEGIN()) {
  /* Transaction */
  if (GIL.acquired)
    TABORT();
  else {
    /* Abort */
    if (GIL.acquired) {
      if (Retried 16 times)
        Acquire GIL;
      else {
        Retry after GIL release;
      } else if (Persistent abort) {
        Acquire GIL;
      } else {
        /* Transient abort */
        if (Retried 3 times)
          Acquire GIL;
        else
          Retry;
      }
``` Execute Ruby code;
Where to Begin and End Transactions?

- Should be the same as GIL’s acquisition/release/yield points.
  - Guaranteed as critical section boundaries.

😊 However, the original yield points are too coarse-grained.
  - Cause many transaction overflows.

- Bytecode boundaries are supposed to be safe critical section boundaries.
  - Bytecode can be generated in arbitrary orders.
  - Therefore, an interpreter is not supposed to have a critical section that straddles a bytecode boundary.

😢 Ruby programs that are not correctly synchronized can change behavior.

→ Added the following bytecode instructions as transaction yield points.
  - getinstancevariable, getclassvariable, getlocal, send, opt_plus, opt_minus, opt_mult, opt_aref
  - Criteria: they appear frequently or are complex.
Ending and Yielding a Transaction

Ending a transaction

if (GIL.acquired)
   Release GIL;
else
   TEND();

Yielding a transaction (transaction boundary)

if (--yield_counter == 0) {
   End a transaction;
   Begin a transaction;
}

Dynamic adjustment of a transaction length

✓ Source code changes limited to only part of files
   ←→ Changes for fine-grain locking scattered throughout files
Tradeoff in Transaction Lengths

- No need to end and begin transactions at every yield point.
  - Variable transaction lengths with the granularity of yield points.

- Longer transaction = Smaller relative overhead to begin/end transaction.
- Shorter transaction = Smaller abort overhead
  - Smaller amount of wasteful work rolled-back at aborts
  - Smaller probability of size overflow
Dynamic Adjustment of Transaction Lengths

Adjust transaction lengths on a per-yield-point basis.

1. Initialize with a long length (255).

2. Calculate abort ratio at each yield point base on the following two numbers:
   - Number of transactions started from the yield point
   - Number of aborted transactions started from the yield point

3. If the abort ratio exceeds a threshold (1%), shorten the transaction length \((x \times 0.75)\) and return to Step 2.

4. If a pre-defined number (300) of transactions started before the abort ratio exceeds the threshold, finish adjustment for the yield point.
5 Sources of Conflicts and How to Remove Them (1/2)

- Global variables pointing to the current running thread
  - Cause conflicts because they are written every time transactions yield
    - Moved to Pthread’s thread-local storage

- Updates to inline caches at the time of misses
  - Cache the result of hash table access for method invocation and instance-field access
    - Cause conflicts because caches are shared among threads.
    - Implemented miss reduction techniques.
5 Sources of Conflicts and How to Remove Them (2/2)

- Manipulation of global free list when allocating objects
  - Introduced thread-local free lists
    - When a thread-local free list becomes empty, take 256 objects in bulk from the global free list.

- Garbage collection
  - Cause conflicts when multiple threads try to do GC.
  - Cause overflows even when a single thread performs GC.
  - Reduced GC frequency by increasing the Ruby heap.

- False sharing in Ruby’s thread structures (rb_thread_t)
  - Added many fields to store thread-local information.
    - Conflicted with other structures on the same cache line.
  - Assign each thread structure to a dedicated cache line.
Platform-Specific Optimizations

- Performed spin-wait for a while at GIL contention.
  - Original GIL is acquired and released every ~250 msec.
  - Fallback GIL for HTM is acquired and released far more frequently.
    → Parallelism severely damaged if sleep in OS every time GIL is contended.
  - Some environments already include this optimization in Pthread implementation.

- Implemented our own setjmp() without a restricted instruction.
  - setjmp() in z/OS contains an address-space manipulation instruction.
  - It suffices to save general-purpose registers for Ruby’s purpose.
Outline

- Motivation
- Transactional Memory
- GIL in Ruby
- Eliminating GIL through HTM
- Experimental Results
- Conclusion
Experimental Environment

- Implemented in Ruby 1.9.3-p194.
- Ported to z/OS 1.13 UNIX System Services.
  - Not yet succeeded in building full-featured Ruby due to EBCDIC.
  - Instead, used miniruby, which supports core class libraries.
- Experimented on 12-core 5.5-GHz zEC12
  - 1 hardware thread on 1 core

Configurations
- GIL: Original Ruby
- HTM-n \( (n = 1, 16, 256) \):
  Fixed transaction length (Skip \( n-1 \) yield points)
- HTM-dynamic:
  Dynamic adaptive transaction length
Benchmark Programs

- Two micro-benchmarks
  - Embarrassingly parallel programs running while and iterator loops.
  - 11-fold speed-ups over 1-thread GIL by HTM with 12 threads.
  - At least 5-14% single-thread overhead

- Ruby NAS Parallel Benchmarks (NPB)
  [Nose et al., 2012]
  - 7 programs translated from the Java version
  - Consist of single-threaded and multi-threaded sections
  → Inherent scalability limitation exists.

- Trying to measure Web-based workloads
  - Difficulties in EBCDIC support on z/OS, etc.
Throughput of Ruby NPB (1/2)

- 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.
Throughput of Ruby NPB (2/2)

- **IS**
  - Throughput (1 = 1 thread GIL) vs. Number of threads

- **LU**
  - Throughput (1 = 1 thread GIL) vs. Number of threads

- **MG**
  - Throughput (1 = 1 thread GIL) vs. Number of threads

- **SP**
  - Throughput (1 = 1 thread GIL) vs. Number of threads
Abort Ratios

- Transaction lengths well adjusted by HTM-dynamic with 1% as a target abort ratio.
- No correlation to the scalabilities.

Abort ratios of HTM-dynamic

Abort ratio (%) vs Number of threads

- BT
- CG
- FT
- IS
- LU
- MG
- SP
Cycle Breakdowns

- No correlation to the scalabilities.
  - Result of IS is not reliable because 79% of its execution was spent in initialization, outside of the measurement period.

Cycle breakdowns

- Transaction begin/end
- GIL acquired
- Waiting for GIL release
- Successful transactions
- Aborted transactions

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>CG</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Categorization by Abort Reasons

- Conflicts at read set accounted for most of the aborts.
  - Cache fetch-related + Fetch conflict

![Abort categorization by reasons (HTM-dynamic / 12 threads)](image)
Categorization by Functions Aborted by Fetch Conflicts

- Half of the aborts occurred in manipulating the global free list (rb_newobj) and lazy sweep in GC (gc_lazy_sweep).
  - A lot of Float objects allocated.
  - To be fixed in Ruby 2.0 with Flonum?
Comparing Scalabilities of CRuby, JRuby, and Java

- CRuby (HTM) was similar to Java (almost no VM-internal bottleneck).
- Scalability saturation in CRuby (HTM) is inherent in the applications.
- JRuby (fine-grain locking) behaved differently from CRuby (HTM).
- On average, HTM achieved the same scalability as fine-grain locking.
Single-thread Overhead

- Single-thread speed is important too.

- With 1 thread, use the GIL instead of HTM.
  - 5-14% overhead in the micro-benchmarks even with this optimization.

- Sources of the overhead:
  - Checks at the yield points
  - Newly added yield points
  - Slow access to Pthread’s thread-local storage (z/OS specific)
Conclusion

- What was the performance of real applications when GIL was replaced with HTM?
  - Up to 4.4-fold speed-up with 12 threads.

- What was required for that purpose?
  - Modified only a couple of source code files to replace the GIL with HTM.
  - Changed up to a few dozens of lines to remove each conflict.
  - Proposed dynamic transaction-length adjustment.

✔ Using HTM is an effective way to outperform the GIL with only small source code changes