Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

Rei Odaira
IBM Research – Tokyo

Jose G. Castanos
IBM Research – Watson Research Center

Hisanobu Tomari
University of Tokyo
Global Interpreter Locks in Scripting Languages

- Scripting languages (Ruby, Python, etc.) everywhere. → Increasing demands for speed.

- Multi-thread speed restricted by global interpreter locks.
  - Only one thread can execute interpreter at one time.
  - Not required by language specification.
  😊 Simplified implementation in interpreter and libraries.
  😞 No scalability on multi cores.
  - CRuby, CPython, and PyPy
Hardware Transactional Memory (HTM) Coming into the Market

- Improve performance by simply replacing locks with TM?
- Lower overhead than software TM via hardware.

IBM
- Blue Gene/Q 2012

Intel
- 4th Generation Core Processor 2013

IBM
- zEC12 2012

IBM
- POWER8
Our Goal

- What will realistic applications perform if we replace GIL with HTM?
  - Global Interpreter Lock (GIL)
- What modifications and new techniques are needed?
Our Goal

- What will realistic applications perform if we replace GIL with HTM?
  - Global Interpreter Lock (GIL)

- What modifications and new techniques are needed?

✓ Eliminate GIL in Ruby through HTM on zEC12 and Xeon E3-1275 v3 (aka Haswell)
✓ Evaluate Ruby NAS Parallel Benchmarks, WEBrick HTTP server, and Ruby on Rails

atomic { }
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
  - Remaining locks and incompatibility in class libraries

- Eliminate the GIL in Ruby ...
  - through less restricted HTM on real machines ...
  - and measure non-micro-benchmarks.
  - Concurrency and no incompatibility 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 operations.

- 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.
HTM

- Instruction set (zEC12)
  - TBEGIN: Begin a transaction
  - TEND: End a transaction
  - TABORT, etc.

- Micro-architecture
  - Read and write sets held in caches
  - Conflict detection using cache coherence protocol

- Abort in four cases:
  - Read set and write set conflict
  - Read set and write set overflow
  - Restricted instructions (e.g. system calls)
  - External interruptions, etc.
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.

No timer thread needed.
Tradeoff in Transaction Lengths

- No need to end and begin transactions at every yield point.
  = Variable transaction lengths at 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 transaction overflows
Dynamic Adjustment of Transaction Lengths

Transaction length = How many yield points to skip

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

1. Initialize with a long length (255).
2. Calculate abort ratio at each yield point
   - Number of aborts / Number of transactions
3. If the abort ratio exceeds a threshold (1% for zEC12 and 6% for Xeon), shorten the transaction length (x 0.75) and return to Step 2.
4. If a pre-defined number (300) of transactions started before the abort ratio exceeds the threshold, stop adjusting that yield point.
5 Sources of Conflicts and How to Remove Them

☑ Source code needs to be changed for higher performance, but each change is limited to only a few dozen lines of the interpreter’s source code.

- Refer to our paper for the details.
  - Global variables pointing to the current running thread
  - Manipulation of global free list when allocating objects
  - Conflicts and overflows in garbage collection
  - Updates to inline caches at the time of misses
  - False sharing in thread structures
5 Sources of Conflicts and How to Remove Them

- Source code needs to be changed for higher performance, but each change is limited to only a few dozen lines of the interpreter’s source code.

- Refer to our paper for the details.
  - Global variables pointing to the current running thread
  - Manipulation of global free list when allocating objects
  - Conflicts and overflows in garbage collection
  - Updates to inline caches at the time of misses
  - False sharing in thread structures
Thread-Local Free Lists to Avoid Conflicts

Object allocation

Single global free list

Thread 1

Thread 2

Thread-local free lists

Bulk movement
Outline

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

- Implemented in Ruby 1.9.3-p194.
- ~400 MB Ruby heap
- zEC12
  - 12-core 5.5-GHz zEC12 (1 hardware thread on 1 core)
  - 1-MB read set and 8-KB write set
  - Ported to z/OS 1.13 UNIX System Services.
- Xeon E3-1275 v3 (aka Haswell)
  - 4-core 2-SMT 3.5-GHz Xeon E3-1275 v3
  - 6-MB read set and 19-KB write set
  - Linux 3.10.5
Benchmark Programs

- Ruby NAS Parallel Benchmarks (NPB) [Nose et al., 2012]
  - Class S for BT, CG, FT, LU, SP, and Class W for IS and MG
- WEBrick: HTTP server included in Ruby distribution
  - Spawn 1 thread to handle 1 request.
- Ruby on Rails running on WEBrick
  - Executed a Web application to fetch a book list from SQLite3.
  - Measured only on Xeon E3-1275 v3.
GIL and HTM Configurations

- GIL: Original Ruby
- HTM-n (n = 1, 16, 256): Fixed transaction length (Skip n-1 yield points)
- HTM-dynamic: Dynamic adaptive transaction length
Throughput of FT in Ruby NPB

- HTM-dynamic was the best.
- HTM-1 suffered high overhead.
- HTM-256 incurred high abort ratios.
- zEC12 and Xeon showed similar speed-ups up to 4 threads.
- SMT did not help.
Throughput of WEBRick

- HTM was faster than GIL by 57% on Xeon E3-1275 v3.
- Scalability on zEC12 was limited by conflicts at malloc().

- HTM-dynamic was among the best.
- HTM-1 was better than HTM-16 and -256.
  - Frequent complex operations in WEBRick such as method invocations.
Ruby on Rails on Xeon E3-1275 v3 and Abort Ratios

- HTM improved the throughput by 24% over GIL.
- High abort ratios due to transaction overflows.
  - Regular expression libraries and method invocations.
  - Need to split into multiple transactions?

---

<table>
<thead>
<tr>
<th>Number of clients</th>
<th>Throughput (1 = 1 thread GIL)</th>
<th>Abort ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GIL</td>
<td>WEBrick / zEC12</td>
</tr>
<tr>
<td></td>
<td>HTM-1</td>
<td>WEBrick / Xeon</td>
</tr>
<tr>
<td></td>
<td>HTM-16</td>
<td>Rails / Xeon</td>
</tr>
<tr>
<td></td>
<td>HTM-256</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HTM-dynamic</td>
<td></td>
</tr>
</tbody>
</table>

---

- Abort ratios of HTM-dynamic
Single-thread Overhead

- Single-thread speed is important too.

- 18-35% single-thread overhead
  - Optimization: with 1 thread, use the GIL instead of HTM.  
    → 5-14% overhead in micro-benchmarks even with this optimization.

- Sources of the overhead:
  - Checks at the yield points
    → Adaptive insertion of yield points
  - Disabled method-invocation inline caches
    → Concurrent inline caches
Conclusion

- What was the performance of realistic applications when GIL was replaced with HTM?
  - Up to 4.4-fold speed-up with 12 threads in Ruby NPB.
  - 57% speed-up in WEBrick and 24% in Ruby on Rails.

- What was required for good scalability?
  - Proposed dynamic transaction-length adjustment.
  - Removed 5 sources of conflicts.

✔ Using HTM is an effective way to outperform the GIL without fine-grain locking.
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

\[
\begin{align*}
\text{if (GIL.acquired)} & \quad \text{Release GIL;} \\
\text{else} & \quad \text{TEND();}
\end{align*}
\]

Yielding a transaction (transaction boundary)

\[
\begin{align*}
\text{if (--yield_counter == 0) } & \quad \text{if (yield_counter == 0) } \\
& \quad \text{End a transaction;} \\
& \quad \text{Begin a transaction;}
\end{align*}
\]

Dynamic adjustment of a transaction length
Abort Ratios on zEC12

- Transaction lengths well adjusted by HTM-dynamic with 1% as a target abort ratio.
- No correlation to the scalabilities.
Cycle Breakdowns on zEC12

- 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 of Ruby NPB

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