June 14, 2013

RT0950
Computer Science 12 pages

Research Report

Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

Rei Odaira and Jose G. Castanos

IBM Research - Tokyo
IBM Japan, Ltd.
NBF Toyosu Canal Front Building
6-52, Toyosu 5-chome, Koto-ku
Tokyo 135-8511, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Limited Distribution Notice
This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

Rei Odaira
IBM Research – Tokyo
odaira@jp.ibm.com

Jose G. Castanos
IBM Research – Thomas J. Watson Research Center
castanos@us.ibm.com

***Abstract*—Many scripting languages use Global Interpreter Locks (GIL) to simplify the internal designs of their interpreters, but this kind of lock severely lowers the multi-thread performance on multi-core machines. This paper shows the first results eliminating the GIL in Ruby using the Hardware Transactional Memory (HTM) in the new mainframe zEnterprise EC12 processor. Though prior prototypes replaced the GIL with HTM, we tested realistic programs, the Ruby NAS Parallel Benchmarks (NPB), as well as micro-benchmarks. We devised a new technique to dynamically adjust the transaction lengths on a per-bytecode basis, so that we can automatically optimize the likelihood of transaction aborts against the relative overhead of the instructions to begin and end the transactions. Our current results show that HTM achieved an 11-fold speed-up over the GIL on 12 cores in the micro-benchmarks and 1.9- to 4.4-fold speed-ups in the NPB programs. The dynamic transaction-length adjustment improved the throughput by up to 18%. Our investigation on the scalability and overhead revealed further optimization opportunities.**

Keywords*— *global interpreter lock; hardware transactional memory; Ruby; lock elision

I. INTRODUCTION

Scripting languages such as Ruby [29] and Python [25] offer high productivity [7, 23], but at the cost of slow performance. The single-thread performance is limited because of interpreted execution, dynamic typing, and the support for meta-programming. Many projects [13,28,32] attempt to overcome these limitations through Just-in-Time (JIT) compilation with type specialization.

Meanwhile, the multi-thread performance of the scripting languages is restricted due to the *Global Interpreter Lock (GIL)*, or the *Giant VM Lock (GVL)* in Ruby's terminology. Although each application thread is mapped to a native thread, only the single thread that acquired the GIL can actually run. At pre-defined yield points, each thread releases the GIL, yields the CPU to another runnable thread if any exists, and then reacquires the GIL. The GIL eases the programming of the interpreters' internal logic and the extension libraries because they do not need to worry about concurrency. In return, the GIL significantly limits the performance on multi-core systems. Some new implementations of the Ruby and Python languages [12,13,16,17] use complex fine-grained locking to remove the GIL. However, their class libraries still need to be globally protected to remain compatible with the original implementations of the languages. As a result, the typical use of the scripting languages is a multi-process approach. For example, in Web frameworks, a Web server instantiates many instances of the interpreter to handle the requests rather than relying on the multi-thread features of the languages.

Transactional memory has been proposed as an alternative way to eliminate global locking without requiring the complex semantics of fine-grained locking. In transactional memory, a programmer encloses critical sections with transaction begin and end directives. A transaction is executed atomically so that its memory operations appear to be performed in a single step. Transactions can be executed concurrently as long as their memory operations do not conflict, so that transactional memory can outperform global locking. Although transactional memory is attractive for its potential concurrency, pure software implementations are slow [2].

Chipmakers in the industry regard transactional memory as a promising technology for parallel programming in the multi-core era and are designing or producing hardware for transactional memory, called Hardware Transactional Memory (HTM). The Azul Vega processer uses HTM to replace contended Java locks [4]. Sun announced the Rock processor with a limited HTM facility [3], though the processor was cancelled before release into the market. Intel published documentation for an instruction set called Transactional Synchronization Extensions [11] and implemented it in the Haswell processor. IBM has released Blue Gene/Q and mainframe processor zEnterprise EC12 (zEC12) with HTM support [8,30]. IBM also presented HTM extensions for Power ISA [9]. These HTM implementations will offer effective concurrency with low overhead. There were a couple of previous studies [1,27,31] on replacing the GIL of a scripting language with HTM, but none have ever measured realistic benchmarks on real HTM hardware.

This paper shows the empirical results of removing the GIL from Ruby using IBM's HTM implementation in zEC12 [30] and measuring the NAS Parallel Benchmarks ported to Ruby [21], in addition to micro-benchmarks. Our obtained results are also applicable to other proposed HTM implementations. Unlike the results [31] on the more restrictive HTM of Sun's Rock processor, the HTM implementation on which we experimented is similar to the generic ones proposed in the literature, e.g. [26]. Transactional data is marked as such in a CPU cache line, the conflict detection is incorporated into a cache-coherence protocol, and the transactional write data is buffered in a CPU cache.

To reduce transaction aborts and overhead, we propose a new technique to dynamically adjust the transaction lengths on a per-yield-point basis. On one hand, we desire long transactions as we amortize the overhead to begin and end a transaction. On the other hand, no HTM implementation allows for transactions of unlimited lengths, and longer transactions increase the amount of discarded work in case of transaction aborts. As in the GIL, a transaction ends and begins at pre-defined transaction yield points, but it need not do so at every transaction yield point. We dynamically adjust the transaction lengths, i.e. how many yield points to skip, based on the abort statistics of the transactions started at each yield point.

Our contributions are as follows:

- We proposed an algorithm for the GIL elimination to adjust transaction lengths at each yield point.
- We implemented and evaluated the GIL elimination on a real machine that supports HTM, using real-world applications in addition to micro-benchmarks.
- We eliminated the GIL from Ruby, going beyond prior work focusing on Python. We also removed transaction conflict points in the Ruby interpreter.

Section II describes the HTM implementation used in our studies and Section III explains Ruby's GIL implementation. Section IV shows how we replaced the GIL with the HTM and then reduced the overhead, leading to the experimental results in Section V. Section VI covers related work. Section VII concludes this paper.

II. HTM IMPLEMENTATION

We used the HTM in a new IBM mainframe zEC12 [29] for our studies. This section briefly describes the instruction set architectures to support the HTM as well as its micro-architecture implementation. A complete description appeared in [15]. The instruction set architecture is defined in [10].

A. *Instruction Set Architecture*

Each transaction begins with a TBEGIN instruction and is ended by a TEND instruction. Transactions can be nested. In that case, all of the transactions commit when the outermost transaction commits (also called flattened nesting). The TBEGIN instruction saves the general purpose registers.

The TBEGIN instruction initially sets the condition code to 0. If a transaction aborts, then the execution returns back to the instruction immediately after the outermost TBEGIN. All of the transactionally written data is discarded and the saved general purpose registers are restored. The hardware transaction facilities also set the condition code to 2 or 3, depending on whether the cause of the abort is transient or persistent, respectively. Therefore, a program typically checks the condition code immediately after TBEGIN and jumps to a fallback path if it is not 0.

A transaction can abort for various reasons. The most frequent causes include external interrupts, overflows, conflicts, and restricted instructions. Aborts are classified as either transient or persistent by the CPU and the condition code is set accordingly. When the abort is transient, e.g. because of a conflict, simply retrying the transaction is likely to succeed. On persistent aborts, e.g. due to attempted execution of a restricted instruction, the program should cancel the execution of the transaction. Restricted instructions include system calls and access-register manipulation, but most of the non-privileged instructions are allowed. A transaction can also be aborted by software with a TABORT instruction.

The programmer can specify the address of a 256-byte memory in the operand of the TBEGIN instruction. This memory area is called a Transaction Diagnostic Block (TDB) and is used for storing debug information when a transaction aborts. A TDB contains the abort reason code and the instruction virtual address where the abort was detected.

B. *Micro-Architecture*

The Central Processor (CP) chip has 6 cores, and 6 CP chips are packaged in a multi-chip module (MCM). Up to 4 MCMs can be connected in a single cache-coherent system. Each core has 96-KB L1 and 1-MB L2 data caches. Both the L1 and L2 caches are store-through with 256-byte cache lines. The 6 cores on a CP chip share a 64-MB L3 cache and the 6 CP chips share an off-chip 384-MB L4 cache included in the same MCM. All four levels of the caches are inclusive. Each core supports a single hardware thread. The TBEGIN instruction saves the general purpose registers to a transaction-backup register file. The maximum nesting depth is 16.

The HTM facilities of zEC12 are built on top of its cache structure. Each L1 data cache line is augmented with its own tx-read and tx-dirty bits. A load instruction during a transaction sets a tx-read bit. Transactionally written data is stored into the L1 with an active tx-dirty bit. An abort is triggered if a cache-coherency request from another CPU conflicts with a transactionally read or written line. This means zEC12 uses an eager abort scheme and provides strong atomicity. On an abort, all of the lines whose tx-dirty bits are set are invalidated. The general purpose registers are restored from the transaction backup register file.

A special LRU-extension vector records the lines that are transactionally read but evicted from the L1. Thus the maximum read-set size is roughly the size of the L2. The transactionally written data is buffered in the Gathering Store Cache between the L1 and the L2/L3. The maximum write-set size is limited to the cache size, which is 8 KB.

III. RUBY IMPLEMENTATION

This section introduces the Ruby language and its original implementation, often called CRuby, and describes how the GIL works in CRuby. Our description is based on CRuby 1.9.3-p194.

A. *The Ruby language and CRuby*

Ruby is a modern object-oriented scripting language that is widely known as part of the Ruby on Rails Web application framework. Nevertheless, Ruby is a general-purpose language that has many features in common with other scripting languages such as JavaScript, Python, or Perl: a flexible,

dynamic type system; meta-programming; closures; and an extensive standard runtime library.

The reference Ruby implementation [29] is called CRuby, which is written in the C language. The recent releases of CRuby use a stack-based virtual machine. Ruby code is compiled into an internal bytecode representation at runtime. On the mainframe platform CRuby uses a large switch-case statement to interpret the bytecode.

B. *The GIL in CRuby*

The Ruby language supports multi-threaded programming through objects of the standard Thread class that synchronize through standard concurrency constructs like Mutex or ConditionVariable objects. Since CRuby 1.9, the interpreter supports native threads, where each Ruby application thread maps to a kernel thread. In our studies, we ported CRuby to the mainframe z/OS UNIX System Services (USS), which supports the POSIX interfaces. Thus each application thread corresponds to a Pthread.

Unfortunately, the concurrency is limited. The interpreter has a GIL, guaranteeing that only one application thread is executing in the interpreter at any given time. The GIL eliminates the need for complex concurrency programming in the interpreter and libraries. Unlike normal locking, which holds a lock only during part of the execution, the GIL is almost always held by one of the application threads during execution and is released only when necessary. An application thread acquires and releases the GIL when it starts and ends, respectively. It also releases the GIL when it is about to perform a blocking operation, such as I/O, and it acquires the GIL again after the operation is finished.

However, if the GIL were released only at the blocking operations, compute-intensive application threads could not be switched with one another at all. Therefore, at certain pre-defined points, the application thread yields the GIL by releasing the GIL, calling the sched_yield() system call to yield the CPU, and then acquiring the GIL again. To insure reaching a yield point within a finite time, CRuby sets yield points at loop back-edges and each exit of a method and block.

As an optimization, each application thread does not necessarily yield the GIL at every yield point, because the yield operation is heavyweight. To allow for occasional yields, CRuby runs a timer thread in background. It wakes up every 250 msec and sets a flag in the per-thread data structure of the running application thread. Each application thread checks the flag at each yield point and yields the GIL only when it is set. In addition, if there is only one application thread, then no yield operations will be performed at all.

In Python, the GIL is implemented in a similar wary. The yield points are set in some of the bytecode operations. Instead of using a timer thread, each application thread runs through a pre-defined number (100, by default) of yield points before yielding the GIL.

IV. GIL ELIMINATION THROUGH HTM

This section presents a new algorithm for eliminating the GIL by using an HTM. Our algorithm is based on Transactional Lock Elision (TLE) [5,26]. Like TLE, our algorithm retains the GIL as a fallback mechanism. A thread first executes Ruby code as a transaction. If the transaction aborts, the thread can retry the transaction, depending on the abort reason. If the transaction does not succeed after several retries, the thread uses the GIL to proceed.

Transactions should begin and end at the same points as where the GIL is acquired, released, and yielded, because those points are guaranteed as safe critical-section boundaries. However, our preliminary experiments showed the original yield points in CRuby were too coarse-grained for the HTM, which caused overflows. Thus we added new yield points as explained in Section IV.B.

We first show the algorithms to begin and end a transaction that replace the GIL acquisition and release, respectively. The simple replacement causes many transaction aborts due to conflicts and overflows. Therefore, in the later parts of this section we introduce dynamic transaction-length adjustment.

A. *Beginning and Ending a Transaction*

Fig. 1 shows the algorithm to begin a transaction. The GIL status is tracked by the global variable GIL.acquired, which is set to true when the GIL is acquired. The original CRuby also uses this global variable for the same purpose.

If there is no other live application thread in the interpreter, then the algorithm reverts to the GIL (Lines 2-3), because concurrency is unnecessary in this case. Otherwise, the algorithm first sets the length of the transaction to be executed (Line 5). This will be explained in Section IV.C.

Before beginning a transaction, the algorithm checks the GIL and if it has been acquired by some other thread, waits until it is released (Lines 6-8 and 40-48). This is not mandatory but an optimization. Rather than uselessly rushing into a transaction that will never succeed because the GIL was already acquired, the current thread should wait for its release.

Lines 9 and 10 initialize local retry counters for transient aborts and conflicts at the GIL, respectively. The first_retry flag (Line 11) is used to adjust the transaction length only at the first retry (Lines 17-20).

The TBEGIN() function (Line 13) is a wrapper for the TBEGIN instruction described in Section II.A. The TBEGIN() function initially returns 0. If the transaction aborts, the execution returns back to within the TBEGIN() function and then it returns an abort reason code, referring to a TDB.

Lines 14-15 are within the transaction. As in the original TLE, the transaction first reads the GIL (Line 15) into its transaction read set, so that later the transaction can be aborted if the GIL is acquired by another thread. The transaction must abort immediately if the GIL is already acquired, because otherwise the transaction could read data being modified.

Lines 16-37 are for abort handling. We describe Lines 17-20 in Section IV.C. If the GIL is acquired (Line 21), there is a conflict at the GIL. In the same way as in Lines 6-8, Lines 22-27 waits until the GIL is released. The algorithm first tries to use spin locking, but after GIL_RETRY_MAX-time aborts, it forcibly acquires the GIL (Line 27). If the abort is persistent,

```
1.  transaction_begin(current_thread, pc) {
2.    if (there is no other live thread) {
3.      gil_acquire();
4.    } else {
5.      set_transaction_length(current_thread, pc);
6.      if (GIL.acquired) {
7.        if (spin_and_gil_acquire()) return;
8.      }
9.      transient_retry_counter = TRANSIENT_RETRY_MAX;
10.     gil_retry_counter = GIL_RETRY_MAX;
11.     first_retry = 1;
12.   transaction_retry:
13.     if ((tbegin_result = TBEGIN()) == 0) {
14.       /* transactional path */
15.       if (GIL.acquired) TABORT();
16.     } else { /* abort path */
17.       if (first_retry) {
18.         first_retry = 0;
19.         adjust_transaction_length(pc);
20.       }
21.       if (GIL.acquired) {
22.         gil_retry_counter--;
23.         if (gil_retry_counter > 0) {
24.           if (spin_and_gil_acquire()) return;
25.           else goto transaction_retry;
26.         }
27.         gil_acquire();
28.       } else if (is_persistent(tbegin_result)) {
29.         gil_acquire();
30.       } else {
31.         /* transient abort */
32.         transient_retry_counter--;
33.         if (transient_retry_counter > 0)
34.           goto transaction_retry;
35.         gil_acquire();
36.       }
37.     }
38.   }
39. }

40. spin_and_gil_acquire() {
41.   Spin for a while until the GIL is released;
42.   if (! GIL.acquired) return false;
43.   gil_acquire();
44.   return true;
45. }

46. gil_acquire() {
47.   /* Omitted. Original GIL-acquisition logic. */
48. }
```

Fig. 1. Algorithm to begin a transaction.

retrying the transaction will not succeed, so the execution immediately reverts to the GIL (Lines 28-29). The abort reason code in the TDB is used to determine whether the abort is persistent. We regard overflows and restricted instructions as persistent. The other abort reasons, such as conflicts are considered transient. For the transient aborts, we retry the transaction TRANSIENT_RETRY_MAX times before falling back on the GIL (Lines 31-35).

Ending a transaction is much simpler than beginning a transaction, as shown in Fig. 2. The acquired GIL (Line 2) means this transaction has been executed not as a transaction but with the GIL being held. Thus the GIL must be released. Otherwise, the TEND instruction is issued.

B. Yielding a Transaction

As described in Section III.B, the original CRuby implementation has a timer thread to force yielding among the application threads. We no longer need the timer thread because the application threads are running in parallel using the HTM, but we still need the yield points. Without them, some transactions would last so long that there would be many conflicts and overflows.

```
1.  transaction_end() {
2.    if (GIL.acquired) gil_release();
3.    else TEND();
4.  }

5.  gil_release() {
6.    /* Omitted.  Original GIL-release logic */
7.  }

8.  transaction_yield(current_thread, pc) {
9.    if (there is other live thread) {
10.     current_thread->yield_point_counter--;
11.     if (current_thread->yield_point_counter == 0) {
12.       transaction_end();
13.       transaction_begin(current_thread, pc);
14.     }
15.   }
16. }
```

Fig. 2. Algorithm to end and yield a transaction.

In our preliminary experiments, we found the original yield points in CRuby were too coarse-grained for the HTM. As described in Section III.B, the original CRuby sets yield points at branches and method and block exits. With only these yield points, most of the transactions abort due to store overflows. Therefore, we defined the following bytecode types as additional yield points: get_local, getinstancevariable, getclassvariable, send, opt_plus, opt_minus, opt_mult, and opt_aref. We chose these bytecodes because they appear frequently in bytecode sequences or they consume many CPU cycles. This means that in the NAS Parallel Benchmarks, more than half of the bytecode instructions are now yield points.

We also need to guarantee that the new yield points are safe. In language interpreters, the bytecode boundaries are natural yield points. Because the bytecode instructions can be generated in any orders, it is unlikely that the interpreters internally have a critical section straddling a bytecode boundary. However, for applications that are incorrectly synchronized, such as those assuming the GIL can be yielded only at branches or method exits, the new yield points can change their behavior.

At each yield point, we call the transaction_yield() function in Fig. 2, which simply calls the functions to end and begin transactions (Lines 12-13), but with two optimizations. First, as described in Section III.B, no yield operation is performed if there is only one application thread (Line 9). Note that the GIL is used in this case (Line 3 of Fig. 1). Second, a transaction does not yield at every yield point but only after a set number of yield points (using yield_point_counter) have been passed (Lines 10-11). This optimization is described in Section IV.C. Unlike the original GIL-yield operation, we do not need to call the sched_yield() system call, because the multiple threads are already running in parallel and the OS is scheduling them.

C. Dynamic Transaction-Length Adjustment

As shown in Fig. 2, each transaction will skip a predetermined number of yield points before it ends. This means that the transaction lengths vary with the granularity of the yield points. The length of a transaction means the number of yield points the transaction passes through plus one.

```
1.  set_transaction_length(current_thread, pc) {
2.    if (transaction length is constant) {
3.       current_thread->yield_point_counter =
             TRANSACTION_LENGTH;
4.    } else {
5.       if (transaction_length[pc] == 0)
6.         transaction_length[pc] =
               INITIAL_TRANSACTION_LENGTH;
7.       current_thread->yield_point_counter =
             transaction_length[pc];
8.       if (transaction_counter[pc] < PROFILING_PERIOD)
9.         transaction_counter[pc]++;
10.   }

11. adjust_transaction_length(pc) {
12.   if (transaction length is NOT constant &&
13.        transaction_length[pc] > 1 &&
14.        transaction_counter[pc] <= PROFILING_PERIOD) {
15.     num_aborts = abort_counter[pc];
16.     if (num_aborts <= ADJUSTMENT_THRESHOLD) {
17.       abort_counter[pc] = num_aborts + 1;
18.     } else {
19.       transaction_length[pc] =
              transaction_length[pc] * ATTENUATION_RATE;
20.       transaction_counter[pc] = 0;
21.       abort_counter[pc] = 0;
22.     }
23.   }
24. }
```

Fig.3. Algorithm to set and adjust a transaction length.

1) Tradeoff in transaction length

In general, there are three reasons the total abort overhead decreases as the transaction lengths shorten. First, the amount of work that becomes useless and has to be rolled-back at the time of an abort is smaller. Second, the probabilities of overflows are smaller, because they depend on the amount of data accessed in each transaction. Third, if the execution reverts to the GIL, the serialized sections are shorter.

In contrast, the shorter the transactions are, the larger the relative overhead to begin and end the transactions. In particular, beginning a transaction suffers from the overhead of not only TBEGIN but also the surrounding code in Fig. 1.

The best transaction length depends on each yield point. If the intervals (i.e. the number of bytecode instructions) between the subsequent yield points are small, then the lengths of the transactions starting at the current yield point should be long. As another example, suppose there are three consecutive yield points, A, B, and C. If the code between B and C contains a restricted instruction, then the length of any transaction starting at A should be one. If the length was two or more, then the transactions would definitely abort.

2) Adjustment algorithm

We propose a mechanism to adjust the transaction lengths on a per-yield-point basis. The transaction length is initialized to a certain large number at each yield point. The abort ratios of the transactions starting at each yield point are monitored. If the abort ratio is above a threshold at a particular yield point, then the transaction length is shortened. This process continues during a profiling period until the abort ratio falls below the threshold.

The set_transaction_length() function in Fig. 3 is invoked from Line 5 in Fig. 1 before each transaction begins. The parameter pc is the program counter of the yield-point bytecode from which this transaction is about to start. If the Ruby interpreter is configured to use a constant transaction length, that constant value is assigned to the transaction length (yield_point_counter) at Line 3. Otherwise, the yield-point-specific length is assigned at Line 7. If it has not yet been initialized, then a pre-defined long length is assigned (Lines 5-6). To calculate the abort ratio, this function also counts the number of the transactions started at each yield point (Line 9). To avoid the overhead of monitoring the abort ratio after the program reaches a steady state, there is an upper bound for the counter (Line 8).

The adjust_transaction_length() function is called when a transaction aborts for the first time (Line 19 in Fig. 1). If the transaction length has not yet reached the minimum value 1 (Line 13), and if this is during a profiling period (Line 14), then the abort ratio is checked and updated (Lines 16-17). If the number of aborts in the transactions started from the current yield point exceeds a threshold (Line 16) before the PROFILING_PERIOD number of transactions began, then the transaction length is shortened (Line 19). The two counters to monitor the abort ratio are reset (Lines 20-21), to extend the profiling period.

Note that even when the execution reverts to the GIL, the length of the transaction is unchanged. If the current length is 3, for example, the current thread passes through 2 yield points and releases the GIL at the third one.

3) Implementation

We allocate additional arrays to associate the three variables used in Fig. 3 with each yield point, that is, transaction_length, transaction_counter, and abort_counter. The additional arrays are the same size as the bytecode-sequence arrays for each method or block. From each yield-point bytecode, the associated variables can be accessed easily at the same offset in the additional array. This implementation works for the Ruby NAS Parallel Benchmarks, but a more memory-efficient structure would be required for larger applications. These variables are referenced and updated outside of the transactions, but they could still cause conflicts on the memory bus. Therefore, we limit the accesses only to the profiling periods (Line 8 in Fig. 3).

We needed to modify only 6 source-code files among the 125 files of the CRuby interpreter to implement the algorithms in Sections IV.A, B, and C.

D. Conflict Removal

To obtain better scalability with the HTM, any transaction conflicts must be removed. We fixed five major sources of conflicts in CRuby, which appeared one by one. Removing the first conflict source exposed the second one, and so on. Each of the five conflict removals was limited to a few dozen modified lines in the source code.

The most severe conflicts happened at global variables pointing to the Ruby-thread structure of the running thread. Immediately after the GIL is acquired, the global variables point to the running thread. If multiple threads write to these variables every time any transaction begins, they will cause many store conflicts. Therefore we moved these variables from the global scope to the Pthread thread-local storage.

The second source of severe conflicts is the head of the single global linked list of free objects. CRuby allocates each

new object from the head of the list. This mechanism obviously causes conflicts in multi-threaded execution. We modified CRuby's memory allocator, so that each thread maintains a short thread-local free list. A specified number (256, in our implementation) of objects are moved in bulk from the global free list to the thread-local free list, and each new object is allocated on a thread-local basis, without conflicts.

Garbage collection (GC) is the third conflict point. The mark-and-sweep GC in CRuby is not parallelized. GC will cause conflicts if invoked from multiple transactions. Even if it is triggered from one transaction, the transaction size will overflow. This implies that GC is always executed with the GIL acquired. To mitigate the serialization by GC, we reduced the frequency of GC by increasing the initial Ruby heap size. We changed the initial number of free objects from 10,000 to 10,000,000, which corresponded to about 400 MB on z/OS.

Fourth, inline caches cause aborts when they miss. CRuby searches a hash table to invoke a method or to access an instance variable. To cache the search result, a one-entry inline cache is collocated with each method-invocation and instance-variable-access bytecode. Since the inline caches are shared among threads, an update to an inline cache at the time of a cache miss can result in a transaction conflict. We reduced the cache misses at instance-variable accesses by changing the inline cache guard from a class-equality check to an instance-variable-table equality check, because some classes share the same instance-variable table. Because we could not use the same technique at method invocations, we changed the caching logic so that each cache is filled only at the first miss.

Finally, as we added frequently updated fields, such as yield_point_counter (Line 10 in Fig. 2), to CRuby's thread structures, they began to cause false sharing. We avoided this by aligning the thread structures to the cache line boundaries.

E. Platform-Specific Optimizations

The fallback GIL is far more frequently acquired and released than the original GIL. The original is acquired and released roughly every 250 msec, while the fallback GIL is whenever a persistent abort or excessive transient aborts happen. If each thread went to sleep in the OS each time it failed to acquire the GIL, then it would significantly degrade the parallelism. This is because even when the GIL is released and a waiting thread is notified, the thread cannot immediately wake up and return to the user-space. Therefore, we added the spin-waiting before the original GIL logic in the gil_acquire() function (Lines 46-48 in Fig. 1) to briefly keep the thread waiting in the user space. In some platforms, the mutex lock in the native thread library already implements this mechanism. In those platforms, this optimization is unnecessary. Also note that this optimization does not speed up the original GIL, which is infrequently acquired and released.

The setjmp() function on the z/OS USS includes the Set Address Space Control (SAC) instruction, which is not allowed in a transaction. CRuby uses the setjmp() function to implement exception jumps and to save the register contents. The saved registers are later scanned by the garbage collector as part of the root set. For both of these usages, we found it suffices to save the contents of the general-purpose registers in a buffer. Therefore, we replaced the calls to the setjmp() function with calls to our own implementation, effectively avoiding aborts due to the restricted instruction.

V. Experimental Results

This section describes our implementation for the z/OS UNIX System Services (USS) on zEC12. Then our experimental results are presented for our micro-benchmarks and the Ruby NAS Parallel Benchmarks (NPB) [21].

A. Implementation

We ported CRuby 1.9.3-p194 into the USS of z/OS 1.13. The building process for CRuby first builds a core subset of the Ruby interpreter, called miniruby, which implements all of Ruby's language features but supports only the core class library. This miniruby is then used to generate Makefiles and some C source code for the extension libraries. All of our experiments used miniruby, because we encountered problems in building the extension libraries, mainly due to the EBCDIC character encoding used in the mainframe. Miniruby is capable of running the NPB.

We implemented our algorithms and optimizations explained in Section IV in the ported CRuby. For the conflict removals in Section IV.D and the platform-specific optimizations in Section IV.E, we implemented the thread-local free lists in the original CRuby too, although they did not affect the performance. For fair comparison, we replaced the setjmp() calls in the original CRuby with our own implementation because it improved the performance. We tested a back-port to the original CRuby of the global variable removal, the changes in the inline caches, and the spin-lock before the GIL but found they degraded the performance. The new yield points (Section IV.B) were not added in the original CRuby because they would increase the overhead without any benefit. In all of the experiments, the initial Ruby heap size was set to 10,000,000, using the RUBY_HEAP_MIN_SLOTS environmental variable.

The values of TRANSIENT_RETRY_MAX and GIL_RETRY_MAX in Fig. 1 were set to 3 and 16, respectively. In our preliminary experiments, it was unlikely that a transaction would ever succeed after 3-or-more consecutive transient aborts. In contrast, a thread should wait more patiently for the GIL release, because the GIL will eventually be released and the fallback to GIL is very slow. The INITIAL_TRANSACTION_LENGTH from Fig. 3 was set to 255, and the PROFILING_PERIOD to 300. Unless set to extremely large values, these constants did not affect the performance. Our target abort ratio was 1%, so ADJUSTMENT_THRESHOLD (Line 16 in Fig. 3) was set to 3, which meant that the ADJUSTMENT_THRESHOLD / PROFILING_PERIOD = 3 / 300 = 1%. The ATTENUATION_RATE (Line 19 in Fig. 3) was set to 0.75.

B. Experimental Environment

We evaluated our GIL elimination using the HTM in zEC12 [30]. The experimental system was divided into multiple Logical PARtitions (LPARs), and each LPAR corresponds to a virtual machine. Our LPAR was assigned 12

cores, all running at 5.5 GHz. Although the system was not totally dedicated to our experiments, no other process was running at the time of the experiments, and the performance fluctuations were negligible.

C. Benchmarks

We measured two micro-benchmarks and 7 programs in the Ruby NPB. We ran them four times and took the averages.

The two micro-benchmarks are similar to the Python benchmarks used in [31]. Both of the micro-benchmarks are embarrassingly parallel. The top of Fig. 4 shows the workloads of each thread. The While benchmark uses a while statement. The Iterator benchmark uses an iterator that takes a block construct as an argument. Block constructs are almost the same as methods in CRuby, which means the Iterator benchmark contains a method invocation in its innermost loop. These benchmarks are useful to assess how the HTM works for these embarrassingly parallel programs.

The Ruby NPB [21] was semi-automatically translated from the Java version of the NPB version 3.0 [20]. It contains 7 programs, BT, CG, FT, IS, LU, MG, and SP. We chose the class size W for IS and MG and S for the other programs. With these sizes, the programs took 10 to 300 seconds to finish.

The NPB programs are composed of serialized sections and multi-threaded sections. To investigate their scalability characteristics, we ran the Ruby NPB on JRuby 1.7.3 [16] as well as the original Java NPB. JRuby is an alternative implementation of the Ruby language written in Java. JRuby is suitable as a comparison target for HTM because it minimizes its internal scalability bottlenecks by using fine-grained locking instead of the GIL. Note that this means JRuby sacrifices its compatibility with CRuby, as discussed in Section VI. Because JRuby does not support the EBCDIC character encoding on z/OS, we measured it on a 12-core 2.93 GHz Intel Xeon machine (with hyper-threading disabled) running Linux and HotSpot Server VM 1.7.0_06.

The Java version of the NPB is useful for estimating the scalability of the application programs themselves, because the Java VM has even fewer VM-internal scalability bottlenecks than JRuby. We ran the Java NPB on the same Xeon machine, using IBM J9 VM 1.7.0 SR3. Since the class sizes of S and W are small and Java is much faster than Ruby, each run of the Java NPB took only several seconds. To give the just-in-time compiler time to compile the methods at high optimization levels, we invoked each NPB program multiple times for 2 minutes in a single run on the Java VM and calculated the average of the execution times of the invocations.

D. Results of Micro-Benchmarks

We first show the scalability results of the micro-benchmarks and then their single-thread performance.

1) Scalability

The middle of Fig. 4 shows the throughput, with the number of threads set to 1 to 2, 4, 6, 8, and 12. HTM-1, -16, and -256 denote the fixed transaction lengths of 1, 16, and 256, respectively. These configurations correspond to Lines 2-3 in Fig. 3. HTM-dynamic uses the dynamic transaction-length

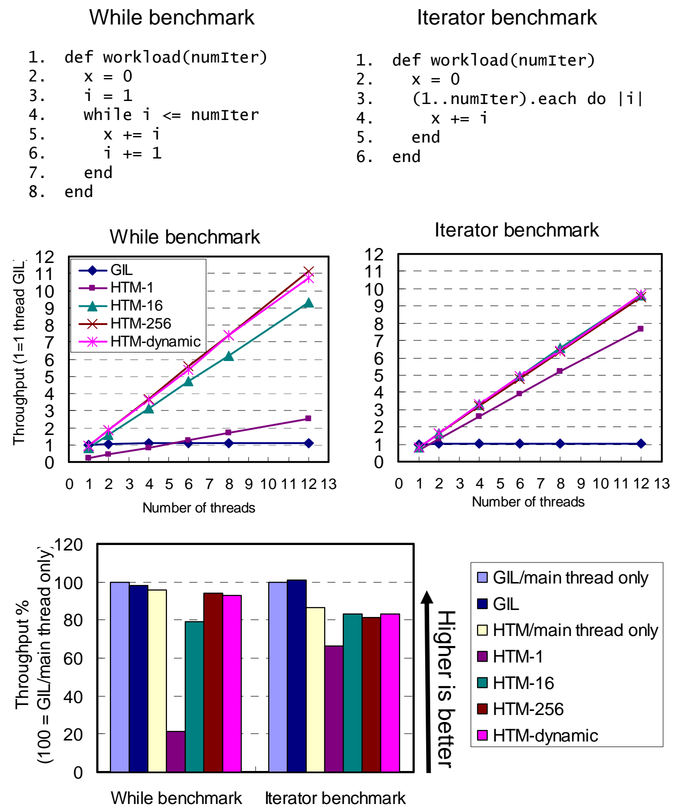

Fig.4. **(Top)** The two embarrassingly parallel micro-benchmarks. **(Middle)** Throughput of the two micro-benchmarks, normalized to the GIL with 1 thread. The HTM achieved 11- and 10-fold speed-ups on 12 cores. **(Bottom)** Single-thread performance of the micro-benchmarks. The HTM suffered from at least 5-14% overhead because of the additional yield points and the checking operation at each yield point.

adjustment described in Section IV.C. The throughput results are normalized to the 1-thread GIL.

The best HTM configurations for each benchmark achieved an 11- and 10-fold speed-ups over the GIL using 12 threads in the While and Iterator benchmarks, respectively. These results are better than the previous study [31] of Python on Sun's Rock processor. In that study, the While and Iterator benchmarks showed only 4.5- and 7-fold speed-ups, respectively, over the GIL with 16 threads. This was due to their implementation's large single-thread overhead.

Among the four HTM configurations, HTM-dynamic delivered performance close to the best of the other three configurations. In the While benchmark, HTM-1 incurred the overhead to begin and end the transactions because the number of instructions (both in bytecode and in native CPU) between the yield points was small in this benchmark. In contrast, the innermost loop of the Iterator benchmark contained a method invocation, which is a complex operation in CRuby. Therefore, even with HTM-1, the relative overhead to begin and end the transactions was not as large as in the While benchmark.

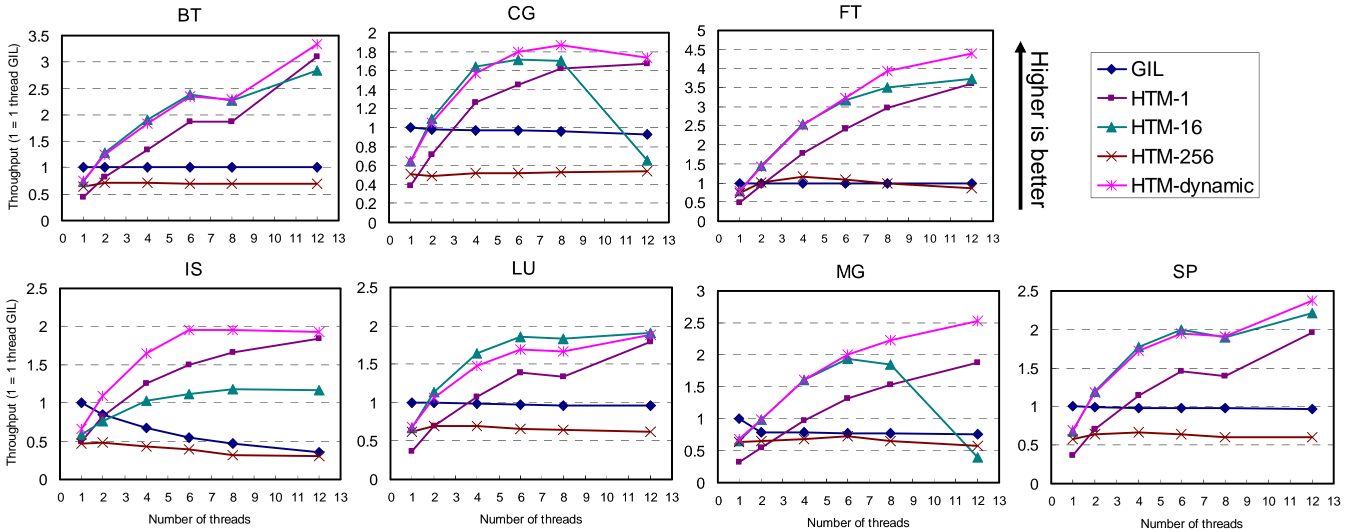

Fig.5. Throughput of the Ruby NAS Parallel Benchmarks, normalized to the GIL with 1 thread. HTM-1, -16, and -256 ran transactions of fixed lengths 1, 16, and 256, respectively. HTM-dynamic uses our proposed dynamic transaction-length adjustment.

2) Single-thread performance

The bottom of Fig. 4 presents the single-thread performance. Since aborts were rare swith one thread, these results expose the overhead of the yield-point operation in Fig. 3 as well as the overhead to begin and end the transactions. The term "a single thread" has two meanings in this experiment. On one hand, if the main application thread executes the workload functions in Fig. 4, then there is truly only one live application thread in the interpreter. In this case, the original CRuby does not perform any yield operations, and neither does our interpreter, but always uses the GIL (Lines 2-3 in Fig. 1 and Line 9 in Fig. 2). "GIL/main thread only" and "HTM/main thread only" in Fig. 4 correspond to this case. On the other hand, if the main application thread spawns another thread to execute the workload functions and goes to sleep, then there are two live application threads, although only one of them is running. In this case, the optimization is not enabled, and our interpreter uses the HTM. The original CRuby has a further optimization so that if there is only one thread running and the others are asleep, then it does not perform any yield operations. GIL, HTM-1, -16, -256, and HTM-dynamic in Fig. 4 and throughout this paper correspond to this case.

The results at the bottom of Fig. 4 show that even without using the HTM, "HTM/main thread only" incurred 5% to 14% overhead. This was due to the checking operation in Line 9 of Fig. 2 and the new yield points described in Section IV.B. If the transactions end and begin at each yield point, as in HTM-1, the additional overhead is 74% and 20% in the While and Iterator benchmarks, respectively. Most of the overhead was due not to the TBEGIN and TEND instructions but to the code surrounding them. GIL and "GIL/main thread only" had the same performance, as expected.

E. *Results of the NAS Parallel Benchmarks*

Fig. 5 shows the throughput of the Ruby NAS Parallel Benchmarks, normalized to GIL with 1 thread. HTM-dynamic showed up to a 4.4-fold speed-up in FT with 12 threads and at the minimum 1.9-fold speed-ups in CG, IS, and LU. From the four HTM configurations, HTM-dynamic was almost always the best or close to the best. HTM-dynamic was 18% faster than HTM-16 in FT with 12 threads. HTM-1 was worse than HTM-dynamic because of its larger overhead, although its abort ratios were lower. HTM-256 showed almost no scalability. Due to its excessively long transaction lengths, its abort ratios were above 90%, and the execution almost always fell back on the GIL. HTM-16 was the best among the fixed-transaction-length configurations, but it incurred more conflict aborts as the number of threads increased.

In summary, using HTM-dynamic, users do not need to specify different transaction lengths for different programs and numbers of threads to obtain near optimal performance. With 12 threads, 40% of the frequently executed yield points had the transaction length of 1 in the Ruby NPB. That means HTM-dynamic effectively chose better lengths for the other points.

If the new yield points were not added as described in Section IV.B, all of the benchmarks except for CG suffered from more than 20% slow-downs compared with the GIL. Without the conflict removals in Section IV.D, the HTM provided no acceleration in any of the benchmarks.

F. *Further Improvement Opportunities*

We present the abort ratios and cycle breakdowns of HTM-dynamic in Fig. 6. The abort ratios were mostly below 2% regardless of the number of threads, indicating that HTM-dynamic successfully adjusted the transaction lengths with 1% as a target abort ratio (Section V.A).

The cycle breakdowns of 12-thread HTM-dynamic in Fig. 6 show that the time spent waiting for the GIL release was longer than the time for cycles wasted on aborted transactions. The cycle breakdown of IS does not represent its actual

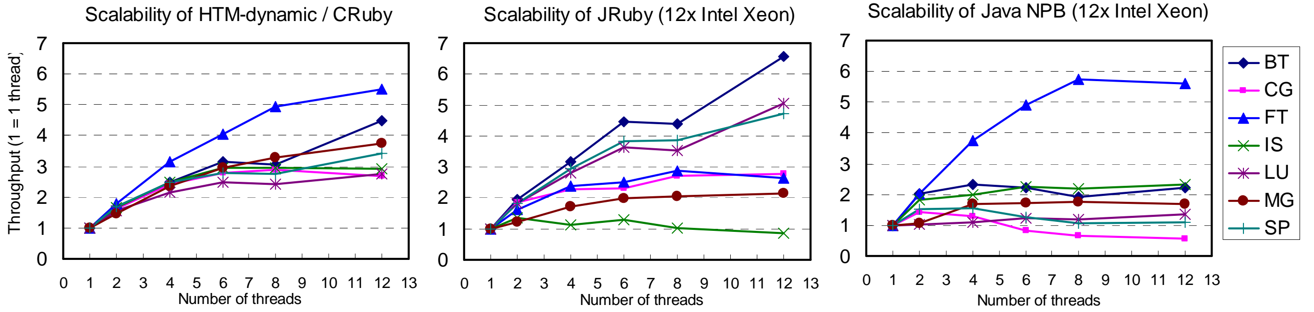

Fig.7. Scalability comparison of the Ruby NAS Parallel Benchmarks on HTM-dynamic/CRuby, fine-grained locking/JRuby, and the Java NAS Parallel Benchmarks. JRuby and the Java version ran on 12-core Intel Xeon X5670 (Westmere-EP 2.93GHz, no hyper-threading).

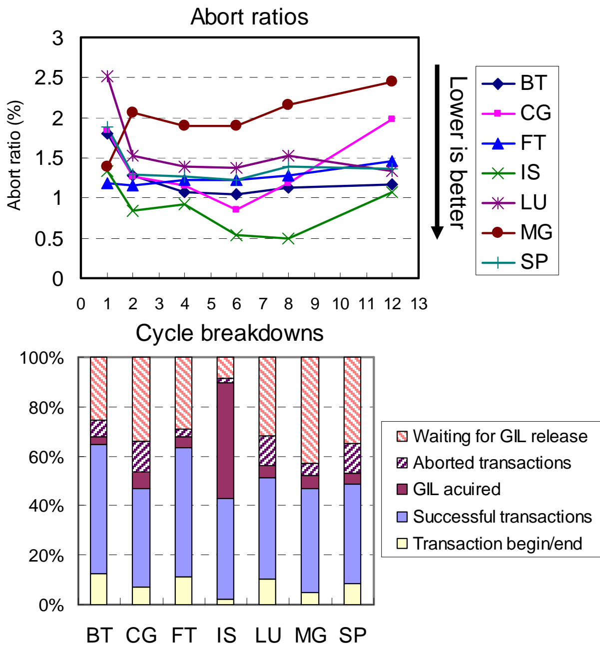

Fig. 6. Abort ratios and cycle breakdowns (when running with 12 threads) of HTM-dynamic.

execution, because 79% of its time was spent in data initialization, which was outside of the measurement period.

Investigation on the abort reasons that caused the GIL to be acquired revealed that fetch (read-set) conflicts accounted for more than 80% in all of the benchmarks with 12 threads. Except for IS, more than 50% of those fetch conflicts occurred at the time of object allocation. Even with the thread-local free lists described in Section IV.D, the global free list still needed occasional manipulation. Also, when the global free list became empty, lazy sweeping of the heap was triggered and caused more conflicts. All of the benchmarks other than IS heavily use floating-point numbers, which are implemented as objects in CRuby 1.9.3. Because the latest CRuby 2.0 represents floating-point numbers as unboxed values, we believe most of these conflicts have been eliminated.

To overcome the conflicts at the general object allocation, the global free list must be eliminated. When a thread-local free list becomes empty, the lazy sweeping should be done on a thread-local basis. GC should also be parallelized or thread-localized. The HTM can be utilized as a synchronization mechanism during GC.

The single-thread overhead of the HTM against the GIL was 25% to 35% in Fig. 5. These numbers were even worse than the results of the micro-benchmarks in Fig. 4. This was because the Ruby NPB exposed two more overhead sources. First, Fig. 6 shows the Ruby NPB incurred 1.2% to 2.5% abort ratios even with a single thread. About 40% of the aborts were due to store overflows and 30% due to external interrupts. Both of these kinds of aborts can be reduced by shortening the transaction lengths, but at the cost of increased relative overhead in beginning and ending the transactions. Second, access to Pthread's thread-local storage accounted for 9% of the total execution cycles on average. As explained in Section IV.D, we moved several global variables to the thread-local storage. Unfortunately, the access function, pthread_getspecific(), is not optimized in z/OS USS, but it is highly tuned in some environments, including Linux.

G. Scalability Characterization

In spite of the large differences in the speed-ups among the 7 programs in Fig. 5, their abort ratios and cycle breakdowns did not much differ in Fig. 6 and had little correlation with the speed-ups. These facts suggest that although the overall speed-ups achieved by HTM-dynamic were limited by the conflicts at the time of object allocation, the differences among the programs were due to their inherent scalability characteristics.

In Fig. 7, we compare the scalability of HTM-dynamic, JRuby, and the Java NPB, from which the Ruby NPB was translated. The Java version did not scale well because we used the smallest or the second smallest problem size in our experiments. Since the Java version was 10 to 100 times faster than the Ruby version, the parallelization overhead negated the speed-ups in such small problems.

Fig. 7 shows that HTM-dynamic resembled the Java NPB rather than JRuby in terms of the scalability. These results confirmed that the differences in the speed-ups by HTM-dynamic among the benchmarks originated from each program's own scalability characteristics. We tried to run

larger problems in the Ruby NPB, but the execution times were prohibitively long. When compared with JRuby, HTM-dynamic achieved the same scalability on average: 3.6-fold with HTM-dynamic and 3.5-fold with JRuby, running 12 threads (not shown in the figure). We guess the characteristics of each benchmark were different between HTM-dynamic and JRuby because of JRuby's internal scalability bottlenecks.

Overall, the GIL elimination through HTM is an effective technique to deliver higher multi-thread performance than the GIL with a small implementation cost. As discussed in Section IV.C and IV.D, we needed to modify only a limited number of source-code files to replace the GIL with the HTM, and each conflict removal modified at most a few dozen lines of code.

VI. Related Work

Riley et al. [27] used HTM to eliminate the GIL in PyPy, one of the implementations of the Python language. However, because they experimented with only two micro-benchmarks on a non-cycle-accurate simulator, it is hard to assess how their implementation would behave on real HTM. Tabba [31] used the HTM of an early-access version of Sun's Rock processor to remove the GIL in the original Python interpreter. Although their measurements were on real hardware, they ran only three synthetic micro-benchmarks. Also, the HTM on Rock had a severe limitation in that transactions could not contain any function returns or tolerate TLB misses. These prototype results cannot be extended to real-world applications. This paper is the first to evaluate larger benchmarks on less-restrictive HTM hardware.

RETCON [1] applied speculative lock elision to the GIL in Python. The focus of the work was on reducing conflicts due to reference-counting GC by symbolic re-execution. By avoiding the conflicts, RETCON achieved a 25-fold speed-up with 32 processors for a micro-benchmark. However, because it was evaluated on a simulator supporting an unlimited transaction size, the aborts in the experiment were mostly due to conflicts. In our experience with a real HTM implementation, the effectiveness of GIL elimination is limited by overflows and various other types of aborts. Thus the dynamic transaction-length adjustment is necessary.

Dice et al. [5] evaluated a variety of programs using HTM on an early-access version of the Sun Rock processor. Wang et al. [33] measured the STAMP benchmarks [19] on the HTM in Blue Gene/Q. Neither of these evaluations covered GIL elimination for scripting languages.

Some alternative implementations of the Ruby and Python languages [12,13,16,17,28] use or are going to use fine-grained locking instead of the GIL. JRuby [16] maps Ruby threads to Java threads and then uses concurrent libraries and synchronized blocks and methods in Java to protect the internal data structures. However, JRuby has two types of incompatibility with CRuby. First, while some of the standard-library classes in CRuby are written in C and are implicitly protected by the GIL, JRuby rewrites them in Java and leaves them unsynchronized for performance reasons. Thus any multi-threaded programs that depend on the implicitly protected standard-library classes in CRuby may behave differently in JRuby. Second, because JRuby does not support CRuby-compatible extension libraries, it does not need the GIL to protect the thread-unsafe extension libraries. The current version 1.2.4 of Rubinius [28] has the GIL, but there are plans to remove it in a future version. However, the Rubinius support for the CRuby-compatible extension libraries conflicts with removing the GIL completely. In contrast, replacing the GIL with HTM creates no compatibility problems in the libraries. PyPy is planning to eliminate the GIL by using software transactional memory [24], but it is unclear whether the scalability improvement can offset the overhead of the software transactional memory.

Because of the GIL, programmers who need to exploit multiple cores in Ruby or Python have been using multi-processing. Whether or not multi-threading is better than multi-processing is beyond the scope of this paper, but the implementations of general-purpose languages should not inhibit a particular programming model because of their implementation-specific reasons.

Scripting languages other than Ruby and Python mostly do not have a GIL, but that is because they do not support shared-memory multi-thread programming, and thus their programming capabilities are limited on multi-core systems. Perl's ithreads clone the entire interpreter and its data when a thread is created, and any data sharing among threads must be explicitly declared as such [22]. The cloning makes a GIL unnecessary, but it is as heavy as fork() and restricts shared-memory programming. Lua [18] does not support multi-threading but uses coroutines. The coroutines switch among themselves by explicitly calling a yield function. This means they never run simultaneously and do not require a GIL. JavaScript (AKA ECMAScript) [6] does not support multi-threading, so the programs must be written in an asynchronous event-handling style.

VII. Conclusion and Future Work

This paper shows the first empirical results of eliminating the Global Interpreter Lock (GIL) in a scripting language through a real Hardware Transactional Memory (HTM) to improve the multi-thread performance of realistic programs. We eliminated the GIL in Ruby using the HTM facilities in the mainframe processor zEC12 and evaluated the Ruby NAS Parallel Benchmarks (NPB) and some micro-benchmarks. We proposed a new automatic mechanism to dynamically adjust the transaction lengths on a per-yield-point basis. Our mechanism chose a near optimal tradeoff point between the relative overhead of the instructions to begin and end the transactions and the likelihood of transaction conflicts and overflows. Our results show that HTM achieved a 11-fold speed-up over the GIL with 12 threads in the micro-benchmarks and up to a 4.4-fold speed-up in the Ruby NPB programs. The dynamic transaction-length adjustment improved the throughput by up to 18%. From these results, we concluded that HTM is an effective approach to achieve higher multi-thread performance than the GIL at a small implementation cost. We also discussed further improvement opportunities for HTM.

In addition to the performance improvement, the interactions between just-in-time-compiled code and the HTM-

based GIL elimination must be studied further. Finally, we plan to measure Web-related workloads such as WEBrick and Ruby on Rails once we succeed in building CRuby’s extension libraries on z/OS USS.

REFERENCES

[1] Blundell, C., Raghavan, A., and Martin, M. M. K., "RETCON: transactional repair without replay," in ISCA, pp. 258-269, 2010.

[2] Cascaval, C., Blundell, C., Michael, M., Cain, H. W., Wu, P., Chiras, S., and Chatterjee, S., "Software transactional memory: why is it only a research toy?" ACM Queue, 6(5), pp. 46-58, 2008.

[3] Chaudhry, S., Cypher, R., Ekman, M., Karlsson, M., Landin, A., Yip, S., Zeffer, H., and Tremblay, M., "Rock: A high-performance SPARC CMT processor," IEEE Micro, 29(2), pp. 6-16, 2009.

[4] Click, C., "Azul's experiences with hardware transactional memory," In HP Labs - Bay Area Workshop on Transactional Memory, 2009.

[5] Dice, D., Lev, Y., Moir, M., and Nussbaum, D., "Early experience with a commercial hardware transactional memory implementation," in ASPLOS, pp. 157-168, 2009.

[6] ECMAScript. http://www.ecmascript.org/

[7] Fowler, Martin., "Ruby at ThoughtWorks," http://martinfowler.com/articles/rubyAtThoughtWorks.html , 2009

[8] Haring, R. A., Ohmacht, M., Fox, T. W., Gschwind, M. K., Satterfield, D. L., Sugavanam, K., Coteus, P. W., Heidelberger, P., Blumrich, M. A., Wisniewski, R.W., Gara, A., Chiu, G. L.-T., Boyle, P.A., Chist, N.H., and Kim, C., "The IBM Blue Gene/Q compute chip," IEEE Micro, 32(2), pp. 48-60, 2012.

[9] IBM, “Power ISA Transactional Memory,” Power.org, 2012.

[10] IBM, "z/Architecture Principles of Operation Tenth Edition (Septem-ber, 2012)," http://publibfi.boulder.ibm.com/epubs/pdf/dz9zr009.pdf

[11] Intel Corporation, "Intel Architecture Instruction Set Extensions Programming Reference," 319433-012a edition, 2012.

[12] IronPython, http://ironpython.codeplex.com/

[13] IronRuby, http://www.ironruby.net/

[14] Ishizaki, K., Ogasawara, T., Castanos, J., Nagpurkar, P., Edelsohn, D., and Nakatani, T., "Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs," in VEE, pp. 169-180, 2012.

[15] Jacobi, C., Slegel, T., and Greinder, D., "Transactional memory architecture and implementation for IBM System z," in MICRO45, 2012.

[16] JRuby, http://jruby.org/

[17] Jython, http://www.jython.org/

[18] Lua, http://www.lua.org/

[19] Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K., "STAMP: Stanford transactional applications for multi-processing," in IISWC, pp. 35-46, 2008.

[20] NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html

[21] Nose, T., "Ruby version of NAS Parallel Benchmarks 3.0," http://www-hiraki.is.s.u-tokyo.ac.jp/members/tknose/

[22] Perl threads, http://perldoc.perl.org/perlthrtut.html

[23] Prechelt, L., "An empirical comparison of seven programming languages," IEEE Computer, 33(10), pp. 23-29, 2000.

[24] PyPy Status Blog, "We need Software Transactional Memory," http://morepypy.blogspot.jp/2011/08/we-need-software-transactional-memory.html

[25] Python programming language, http://www.python.org/

[26] Rajwar, R. and Goodman, J. R., "Speculative lock elision: enabling highly concurrent multithreaded execution," in MICRO, pp. 294-305, 2001.

[27] Riley, N. and Zilles, C., "Hardware transactional memory support for lightweight dynamic language evolution," in Dynamic Language Symposium (OOPSLA Companion), pp. 998-1008, 2006.

[28] Rubinius, http://rubini.us/

[29] Ruby programming language, http://www.ruby-lang.org/

[30] Shum, C.-L., "IBM zNext: the 3rd generation high frequency micro-processor chip," in HotChips 24, 2012.

[31] Tabba, F., "Adding concurrency in python using a commercial processor's hardware transactional memory support," ACM SIGARCH Com-puter Architecture News, 38(5), pp. 12-19, 2010.

[32] Tatsubori, M., Tozawa, A., Suzumura, T., Trent, S., Onodera, T., "Evaluation of a just-in-time compiler retrofitted for PHP," in VEE, pp. 121-132, 2010.

[33] Wang, A., Gaudet, M., Wu, P., Ohmacht, M., Amaral, J. N., Barton, C., Silvera, R., Michael, M. M., "Evaluation of Blue Gene/Q hardware support for transactional memories," in PACT, pp. 127-136, 2012.