# Eliminating Global Interpreter Locks in Ruby through Hardware Transactional Memory

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

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

**Hisanobu Tomari**
University of Tokyo
tomari@is.s.u-tokyo.ac.jp

## Abstract

Many scripting languages use a Global Interpreter Lock (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 presents our first results eliminating the GIL in Ruby using Hardware Transactional Memory (HTM) in the IBM zEnterprise EC12 and Intel 4th Generation Core processors. Though prior prototypes replaced a GIL with HTM, we tested realistic programs, the Ruby NAS Parallel Benchmarks (NPB), the WEBrick HTTP server, and Ruby on Rails. We devised a new technique to dynamically adjust the transaction lengths on a per-bytecode basis, so that we can optimize the likelihood of transaction aborts against the relative overhead of the instructions to begin and end the transactions. Our results show that HTM achieved 1.9- to 4.4-fold speedups in the NPB programs over the GIL with 12 threads, and 1.6- and 1.2-fold speedups in WEBrick and Ruby on Rails, respectively. The dynamic transaction-length adjustment chose the best transaction lengths for any number of threads and applications with sufficiently long running times.

***Categories and Subject Descriptors*** D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel programming

***Keywords*** Global interpreter lock; hardware transactional memory; scripting language; lock elision

## 1. Introduction

Scripting languages such as Ruby [26] and Python [21] offer high productivity, but at the cost of slow performance. The single-thread performance is limited because of interpreted execution, dynamic typing, and the support for metaprogramming. Many projects [10,24,30] have attempted to overcome these limitations through Just-in-Time (JIT) compilation with type specialization.

Meanwhile, the multi-thread performance of the scripting languages is constrained by 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 one thread that acquires 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' 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 [9,10,12,13] 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.

Transactional memory has been proposed as an alternative 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). Intel published documentation for an instruction set called Transactional Synchronization Exten-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*PPoPP '14*, February 15 - 19 2014, Orlando, FL, USA
Copyright © 2014 ACM 978-1-4503-2656-8/14/02…$15.00.
http://dx.doi.org/10.1145/2555243.2555247

sions [8] and implemented it in the 4th Generation Core processor. IBM has released Blue Gene/Q and the mainframe zEnterprise EC12 (zEC12) with HTM support [5,27]. IBM also defined HTM extensions for POWER ISA [6] to be implemented in POWER8 [28]. These HTM implementations will offer effective concurrency with low overhead.

There have been previous studies [1,23,29] of replacing the GIL of a scripting language with HTM, but they measured only micro-benchmarks on simulators or on the limited HTM of Sun's Rock processor [3]. Therefore, these studies did not propose solutions to or even reveal many transaction aborts encountered when realistic programs are executed on less-restrictive HTM hardware.

This paper shows the empirical results of removing the GIL from Ruby and proposes dynamic transaction length adjustment to reduce transaction aborts and overhead. We used IBM's HTM implementation in zEC12 and Intel's implementation in the 4th Generation Core processor (Xeon E3-1275 v3). In addition to micro-benchmarks, we measured the NAS Parallel Benchmarks ported to Ruby [17], the WEBrick HTTP server attached in the Ruby distribution, and Ruby on Rails [25]. Our results are also applicable to other HTM implementations. Unlike the results [29] on Sun's Rock processor, the HTM implementations on which we experimented are similar to the generic ones proposed in the literature, e.g. [22].

Running these programs on the real HTM hardware revealed a tradeoff between transaction aborts and overhead. On the one hand, we desire long transactions as we amortize the overhead to begin and end each transaction. On the other hand, no HTM implementation allows for transactions of unlimited lengths, and longer transactions increase the amount of discarded work when a transaction aborts.

To balance the transaction aborts against the overhead, we propose a new technique to dynamically adjust the transaction lengths on a per-yield-point basis. 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.

Here are our contributions:

- We propose an algorithm for the GIL elimination that adjusts the transaction lengths at each yield point.
- We implemented and evaluated the GIL elimination on real machines that support 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 2 describes the HTM implementations used in our studies and Section 3 explains Ruby's GIL implementation. Section 4 shows how we replaced the GIL with the HTM and then balanced the aborts against the overhead, leading to the experimental results in Section 5. Section 6 covers related work. Section 7 concludes the paper.

# 2. HTM Implementations

We used the HTM implementations in the IBM mainframe zEC12 and the Intel 4th Generation Core processor (Xeon E3-1275 v3) for our studies. This section briefly describes the instruction set architectures supporting the HTM and their micro-architectures. A complete description of the HTM in zEC12 appeared in a paper [11]. The instruction set architectures of zEC12 and the 4th Generation Core processor are also available [7,8].

## 2.1 Instruction set architectures

In zEC12, each transaction begins with a TBEGIN instruction and is ended by a TEND instruction. In the 4th Generation Core processor, XBEGIN and XEND correspond to TBEGIN and TEND, respectively. If a transaction aborts, then the execution returns back to the instruction immediately after the TBEGIN in zEC12, while the argument of the XBEGIN specifies a relative offset to a fallback path.

A transaction can abort for various reasons. The most frequent causes include footprint overflows and conflicts. 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 transaction footprint overflow, the program should cancel the execution of the transaction. The condition code in zEC12 and the EAX register in the 4th Generation Core processor report whether the abort is transient or persistent. A transaction can also be aborted by software with a TABORT (in zEC12) or an XABORT (in the 4th Generation Core processor) instruction.

## 2.2 Micro-architectures

The zEC12 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. 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. Each core supports a single hardware thread. The cache line size is 256 bytes.

The Xeon E3-1275 v3 processor contains 4 cores and each core supports 2 simultaneous multi-threading (SMT) threads. Each core has 32-KB L1 data and 256-KB L2 unified caches. The 4 cores share an 8-MB L3 cache. The cache line size is 64 bytes.

The zEC12 HTM facilities are built on top of its cache structure. Each L1 data cache line is augmented with its own tx-read and tx-dirty bits. An abort is triggered if a cache-coherency request from another CPU conflicts with a transactionally read or written line. 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 this cache size, which is 8 KB.

The detailed design of the Xeon E3-1275 v3 processor's HTM has not been revealed, but it takes advantage of the cache structure, like zEC12. Our preliminary experiments showed that the maximum read-set size is 6 MB, and the maximum write-set size is about 19 KB.

# 3. 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.

## 3.1 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 [26] 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 and is executed by an interpreter. CRuby does not have a JIT compiler.

## 3.2 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 environments, 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 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. Note that the yield points have nothing to do with the "yield" expression in the Ruby language.

As an optimization, each application thread does need not yield the GIL at every yield point, because the yield operation is heavy. 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 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.

# 4. GIL Elimination through HTM

This section presents our algorithm for eliminating the GIL by using an HTM. Our algorithm is based on Transactional Lock Elision (TLE) [3,22]. 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 aborts after several retries, the thread acquires 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 were too coarse-grained for the HTM, which caused footprint overflows. Thus we added new yield points as explained in Section 4.2.

## 4.1 Beginning and ending a transaction

Figure 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.

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 4.3.

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.

The TBEGIN() function (Line 13) is a wrapper for the TBEGIN or XBEGIN instruction described in Section 2.1. The TBEGIN() function initially returns 0. If the transac-

tion aborts, the execution returns back to within the TBEGIN() function and then it returns an abort reason code.

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 4.3. 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, retrying the transaction will not succeed, so the execution immediately reverts to the GIL (Lines 28-29). 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 (Figure 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 or XEND instruction is issued.

### 4.2 Yielding a transaction

As described in Section 3.2, 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 footprint overflows.

In our preliminary experiments, we found the original yield points were too coarse-grained for the HTM. As described in Section 3.2, 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: getlocal, getinstancevariable, getclassvariable, send, opt_plus, opt_minus, opt_mult, and opt_aref. The first three bytecode types are to read variables. The send bytecode invokes a method or a block. The opt_ variants perform their corresponding operations (plus, minus, multiplication, and array reference) but are optimized for certain types such as integer numbers. We chose these bytecodes because they appear frequently in bytecode sequences or they access much data in memory. 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 interpreters, the bytecode boundaries are natural

```
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. }
```

**Figure 1.** Algorithm to begin a transaction.

yield points. Because the bytecode instructions can be generated in any order, 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. Applications properly synchronized by using Mutex objects are not affected by our changes.

At each yield point, we call the transaction_yield() function in Figure 2, which simply calls the functions to end and begin transactions (Lines 12-13), but with two optimizations. First, as described in Section 3.2, 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 Figure 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 4.3. Unlike the original

```
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. }
```

**Figure 2.** Algorithm to end and yield a transaction.

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.

### 4.3 Dynamic transaction-length adjustment

As shown in Figure 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.

***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 footprint 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 or XBEGIN but also the surrounding code in Figure 1.

The best transaction length depends on each yield point. If the intervals (i.e. the number of 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 an instruction that is not allowed in transactions, e.g. a system call, then the length of any transaction starting at A should be one. If the length was two or more, the transactions would definitely abort.

***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

```
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. }
```

**Figure 3.** Algorithm to set and adjust a transaction.

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 Figure 3 is invoked from Line 5 in Figure 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 Figure 1). If the transaction length has not yet reached the minimum value or 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.

### 4.4 Conflict removal

To obtain better scalability with the HTM, any transaction conflicts must be removed. We fixed five major sources of conflicts in CRuby.

The most severe conflicts occurred 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 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.

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 this overhead, we reduced the frequency of GC by increasing the initial Ruby heap. We changed the initial number of free objects from 10,000 to 10,000,000, which corresponded to about 400 MB.

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 conflict. For method invocations, we changed the caching logic so that each cache is filled only at the first miss. For instance-variable accesses, we reduced the cache misses by changing the inline cache guard. Originally, the cached content is used if the class of the object is the same as the class recorded in the cache when it is filled. However, the cached content is valid even when the classes are different, as long as the instance-variable table of the class of the object is the same as that of the recorded class. Therefore, we reduced the cache misses by using the instance-variable-table equality check instead of the class-equality check.

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

Each of these conflict removals was limited to a few dozen modified lines in the source code.

## 5. Experimental Results

This section describes our implementation for zEC12 and the 4th Generation Core processor. Then our experimental results are presented for the Ruby NAS Parallel Benchmarks (NPB) [17], the WEBrick HTTP server, and Ruby on Rails [25].

### 5.1 Implementation

We implemented the algorithms and optimizations explained in Section 4 in CRuby 1.9.3-p194. CRuby ran on Linux 3.10.5 and the 4th Generation Core processor without any modifications. To run CRuby on zEC12, we ported it into the UNIX System Services (USS) of z/OS 1.13.

For the conflict resolutions in Sections 4.4, we also implemented the thread-local free lists in the original CRuby. We tested a back-port to the original CRuby of the global variable removal, and the changes in the inline caches, but found they degraded the performance. The new yield points (Section 4.2) 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 Figure 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 of Figure 3 was set to 255, and the PROFILING_PERIOD to 300. Unless set to extremely large values like 10,000, these constants did not affect the performance. The target abort ratio of the dynamic transaction-length adjustment was set to 1% on zEC12 and 6% on the 4th Generation Core processor, based on our preliminary experiments. The best target abort ratios are independent of the applications but depend on the HTM implementations, especially the abort costs. Accordingly, ADJUSTMENT_THRESHOLD (Line 16 in Figure 3) was set to 3 on zEC12 and 18 on the 4th Generation Core processor, which meant that the ADJUSTMENT_THRESHOLD / PROFILING_PERIOD = 3 / 300 = 1% on zEC12 and 18 / 300 = 6% on the 4th Generation Core processor. The ATTENUATION_RATE (Line 19 in Figure 3) was set to 0.75.

### 5.2 Experimental environments

The experimental zEC12 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. Our system for the 4th Generation Core processor has one Xeon E3-1275 v3 chip, running at 3.5 GHz. Its microcode version was 0x8. Although the systems were not totally dedicated to our experiments, no other process was running during our measurements, and the performance fluctuations were negligible.

The malloc() function in z/OS can cause many conflicts because it is not a thread-local allocator by default. When running with the HTM, we specified the HEAPPOOLS runtime option to enable thread-local allocation in malloc().

### 5.3 Benchmarks

We measured two micro-benchmarks, seven programs in the Ruby NPB, the WEBrick HTTP server, and Ruby on Rails. We ran them four times and took the averages.

As preliminary experiments, we created the two micro-benchmarks to assess how the HTM works for embarrassingly parallel programs. Figure 4 shows the workloads for each thread. The results showed good scalability for the HTM, while the GIL did not scale at all. The best HTM configurations for each benchmark achieved an 11- to 10-fold speedup over the GIL using 12 threads on zEC12 in the While and Iterator benchmarks, respectively.

The Ruby NPB [17] was translated from the Java version of the NPB version 3.0 [16]. 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 ran in 10 to 300 seconds.

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 [12] 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 6. JRuby does not run on zEC12 because it does not support the EBCDIC character encoding. To measure the scalability up to 12 threads, we ran JRuby on a 12-core 2.93-GHz Intel Xeon X5670 machine (with hyper-threading disabled) running Linux 2.6.32 and HotSpot Server VM 1.7.0_06.

The Java 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 X5670 machine, using the interpreter of IBM J9 VM 1.7.0 SR3. We disabled the JIT compiler because the class sizes of S

While benchmark

```
1. def workload(numIter)
2.   x = 0
3.   i = 1
4.   while i <= numIter
5.     x += i
6.     i += 1
7.   end
8. end
```

Iterator benchmark

```
1. def workload(numIter)
2.   x = 0
3.   (1..numIter).each do |i|
4.     x += i
5.   end
6. end
```

**Figure 4.** Each thread's workloads in the two embarrassingly parallel micro-benchmarks.

and W were small. With the JIT compiler, the execution time was too short to outweigh the parallelization overhead.

WEBrick is the default HTTP server for Ruby on Rails. It is implemented in Ruby and is included in the CRuby distribution. It creates one Ruby thread for each incoming request and discards the thread after returning a response. We ran WEBrick version 1.3.1. To measure its scalability, we changed the number of HTTP clients simultaneously accessing WEBrick. Each run sent 30,000 requests for a 46-byte page from the same machine as the one running WEBrick. We took the peak throughput (requests per second) as the result of each run. The HTTP clients consumed less than 5% of the CPU cycles.

Ruby on Rails is a popular Web application framework in Ruby. Using Ruby on Rails version 4.0.0, we created an application to fetch a list of books from a database. We used SQLite3 as the database manager and WEBrick as the HTTP server. Ruby on Rails is thread-safe, but for backward compatibility it has a global lock to serialize the request processing. In our experiments, we disabled the global lock. The measurement method was the same as in WEBrick. We ran Ruby on Rails only on Xeon E3-1275 v3 because we encountered problems installing it in z/OS.

### 5.4 Results of the NAS Parallel Benchmarks

Figure 5 shows the throughput of the Ruby NAS Parallel Benchmarks on zEC12 and Xeon E3-1275 v3, normalized to GIL with 1 thread. The number of threads was set from 1 to 2, 4, 6, and 8 on Xeon E3-1275 v3, and to 12 on zEC12. HTM-1, -16, and -256 denote the fixed transaction lengths of 1, 16, and 256. These configurations correspond to Lines 2-3 in Figure 3. HTM-dynamic uses the dynamic transaction-length adjustment described in Section 4.3.

In zEC12, HTM-dynamic showed up to a 4.4-fold speedup in FT with 12 threads and at the minimum 1.9-fold speedups 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 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

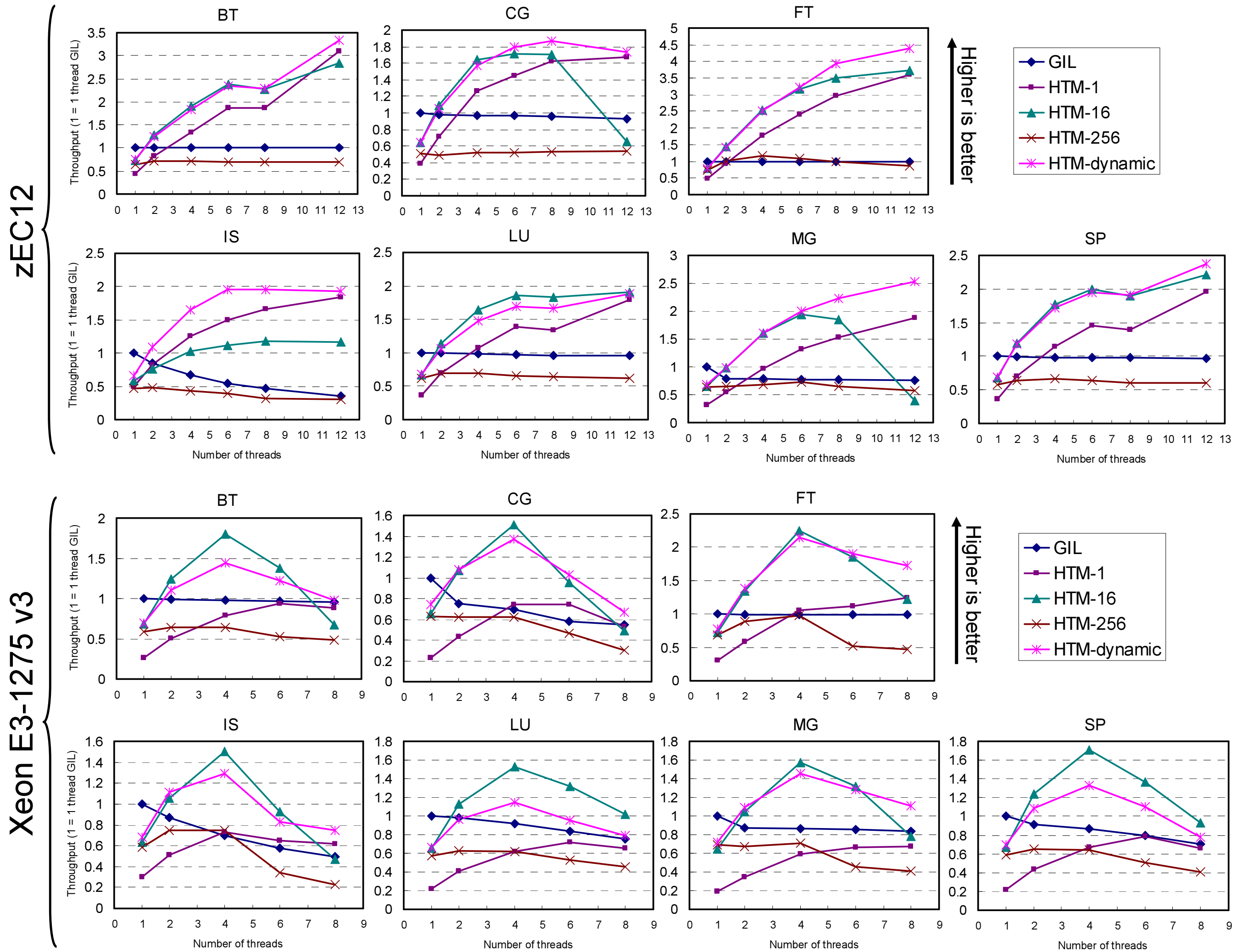


**Figure 5.** Throughput of the Ruby NAS Parallel Benchmarks on 12-core zEC12 (top) and 4-core 2-SMT Xeon E3-1275 v3 (bottom), normalized to the 1-thread GIL. HTM-1, -16, and -256 ran transactions of fixed lengths 1, 16, and 256. HTM-dynamic uses our proposed dynamic transaction-length adjustment.

configurations, but it incurred more conflict aborts as the number of threads increased.

In the 4-core Xeon E3-1275 v3 machine, HTM-16 showed the best throughput. Its speedups for up to 4 threads were almost the same as the corresponding speedups of HTM-dynamic on zEC12. Beyond 4 threads, HTM-16 did not scale using SMT, while HTM-1 did scale in some benchmarks. With SMT, HTM-16 suffered from many transaction footprint overflows because a pair of threads on the same core share the same caches, thus halving the maximum read- and write-set sizes.

HTM-dynamic did not scale better than HTM-16 in Xeon E3-1275 v3. This was due to the learning algorithm of Xeon E3-1275 v3's HTM. We found that the HTM in Xeon E3-1275 v3 changed its abort criteria for each transaction, based on the abort statistics. We created a test program to simulate our dynamic transaction-length adjustment. In each iteration, it sequentially wrote a specified amount of data to memory during a transaction. In one process, it first wrote 24 KB 10,000 times, and then 20 KB 10,000 times, and so on. We measured the transaction success ratios for each 100 iterations. Figure 6(a) shows the results. When the write-set size was shrunk from 20 KB to 16 KB and then to 12 KB, the success ratios did not jump sharply but instead increased gradually. It took about 5,000 iterations to reach a steady state. This implies that the HTM in Xeon E3-1275 v3 eagerly aborts a transaction that has suffered from many footprint overflows and thus cannot quickly adapt to change in the data set size.

This learning algorithm of Xeon E3-1275 v3's HTM can conflict with our dynamic transaction-length adjustment. The running times of the Ruby NPB were too short for both

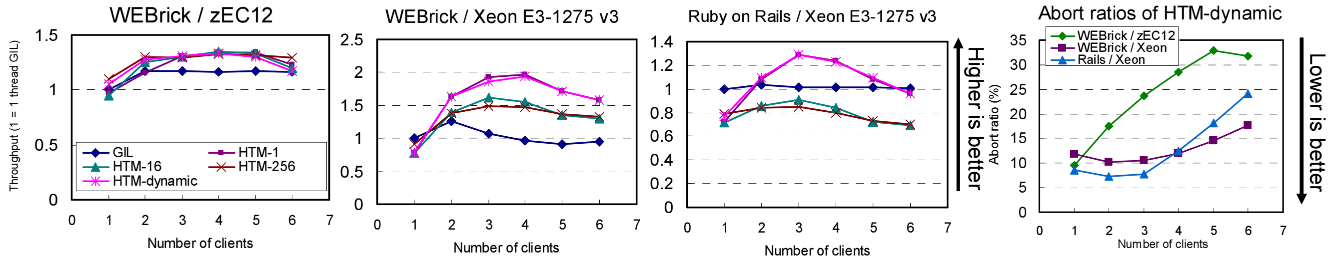

**Figure 7.** Throughput and abort ratios of the WEBrick HTTP server on 12-core zEC12, on 4-core 2-SMT Xeon E3-1275 v3, and Ruby on Rails on Xeon E3-1275 v3. The throughput results are normalized to the 1-thread GIL. HTM-1, -16, and -256 ran transactions of fixed lengths, and HTM-dynamic uses our proposed dynamic transaction-length adjustment.

the underlying HTM and our algorithm to reach a steady state. We ran the benchmarks longer by increasing the class sizes and confirmed HTM-dynamic was equal to or better than HTM-16. Figure 6(b) presents the results of BT with the class size W.

Without the new yield points described in Section 4.2, all of the benchmarks except for CG in the Ruby NPB suffered from more than 20% slowdowns compared with the GIL. Without the conflict removals in Section 4.4, the HTM provided no acceleration in any of the benchmarks.

### 5.5 Results of WEBrick and Ruby on Rails

Figure 7 shows the throughput and abort ratios of WEBrick and Ruby on Rails on zEC12 and Xeon E3-1275 v3. As described in Section 5.3, we ran Ruby on Rails only on Xeon E3-1275 v3. The throughput was normalized to GIL with 1 thread. We changed the number of concurrent clients from 1 to 6. In WEBrick, HTM-1 and HTM-dynamic achieved a 33% speedup on zEC12 and a 97% speedup on Xeon E3-1275 v3 (over 1-thread GIL). GIL also showed speedups of 17% and 26% on zEC12 and Xeon E3-1275 v3, respectively, because the GIL is released during I/O. As a result, HTM-1 and HTM-dynamic were faster than GIL by 14% and 57% on zEC12 and Xeon E3-1275 v3, respectively. Similarly, in Ruby on Rails, HTM-1 and HTM-dynamic improved the throughput by 24% over GIL. The throughput degraded when the number of clients was increased from 4 to 6, due to the rise in the abort ratio, as we explain in Section 5.6.

The HTM scaled worse on zEC12 than on Xeon E3-1275 v3, because many conflicts occurred in malloc(). As described in Section 5.2, we used a thread-local allocator, but there still remained conflict points. Due to these excessive conflicts, HTM-1 and HTM-dynamic did not show better throughput than HTM-16 and HTM-256 on zEC12.

Unlike the Ruby NPB, HTM-1 was the best or one of the best among the fixed transaction-length configurations in WEBrick and Ruby on Rails. HTM-dynamic also chose the best transaction lengths in these programs. In summary, with HTM-dynamic, the users do not need to specify different transaction lengths for different programs and numbers

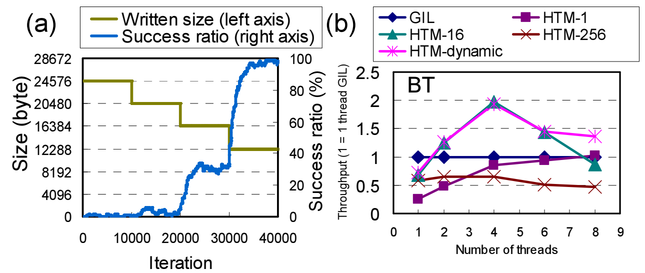

**Figure 6. (a)** Results of a test program shrinking the write-set size on Xeon E3-1275 v3. The transaction success ratio increased gradually when the size shrank. **(b)** Throughput of BT with a bigger class size (W) on Xeon E3-1275 v3. HTM-dynamic showed the best throughput.

of threads to obtain near optimal performance, as long as the programs run long enough on the Xeon E3-1275 v3. With 12 threads on zEC12, 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.

### 5.6 Further optimization opportunities

We present the abort ratios of HTM-dynamic in the rightmost part of Figure 7 and in Figure 8. In the Ruby NPB, the abort ratios were mostly below 2% on zEC12 and 7% on Xeon E3-1275 v3, indicating that HTM-dynamic adjusted the transaction lengths properly with the respective 1% and 6% target abort ratios (of Section 5.1). In WEBrick and Ruby on Rails, HTM-dynamic could not control the abort ratios because most of the transaction lengths reached 1 and could not be shortened further.

The cycle breakdowns of 12-thread HTM-dynamic on zEC12 in Figure 8 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 the actual execution, because 79% of the time was spent in data initialization, which was outside of the measurement period.

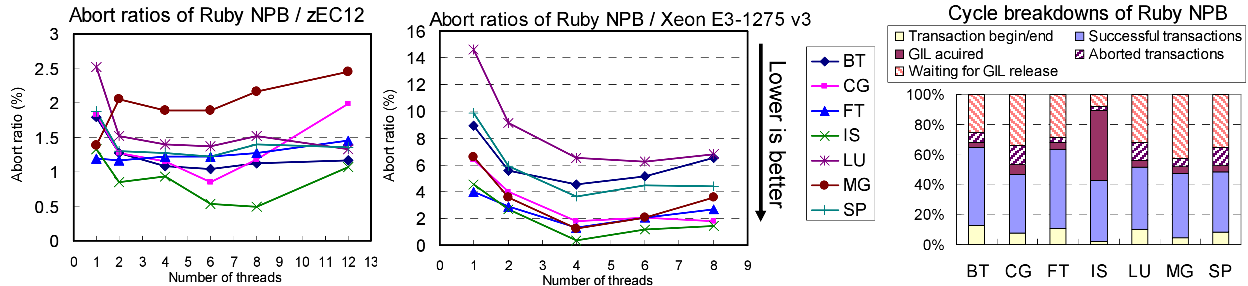


**Figure 8.** Abort ratios and cycle breakdowns (when running with 12 threads on zEC12) of HTM-dynamic in the Ruby NPB.

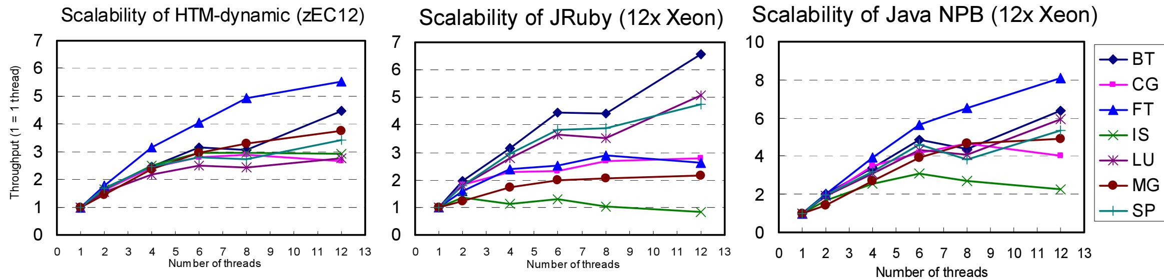


**Figure 9.** Scalability comparison of the Ruby NAS Parallel Benchmarks on HTM-dynamic/CRuby on 12-core zEC12, fine-grained locking/JRuby, and the Java NAS Parallel Benchmarks. JRuby and the Java version ran on 12-core Intel Xeon X5670 (no hyper-threading).

Investigation of the abort reasons that caused the GIL to be acquired revealed that read-set conflicts accounted for more than 80% for all of the Ruby NPB with 12 threads. Except for IS, more than 50% of those read-set conflicts occurred at the time of object allocation. Even with the thread-local free lists described in Section 4.4, the global free list still needed occasional manipulation. Also, when the global free list became empty, lazy sweeping of the heap was triggered and thus caused more conflicts. To overcome the conflicts in the object allocations, 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.

For WEBrick on zEC12, malloc() caused many conflicts, as described in Section 5.5. In WEBrick on Xeon E3-1275 v3 with 4-clients HTM-dynamic, footprint overflows and conflicts accounted for 34% and 29%, respectively, of the aborts that resulted in the GIL acquisition, but the CPU did not report the abort reasons for the others. In Ruby on Rails on Xeon E3-1275 v3 with 4-client HTM-dynamic, 87% of the aborts were due to transaction footprint overflows. Most of these aborts in WEBrick and Ruby on Rails occurred in the regular-expression library and method invocations. The regular expression library is written in C, so there is no yield point in it. A method invocation is a complex operation in CRuby, and since it is a single byecode, it does not have a yield point in it either. To reduce the aborts in the library and method invocations, these operations should be split into multiple transactions.

The single-thread overhead of HTM-dynamic against the GIL was 18% to 35% in Figures 5 and 7. Aborts due to overflows and external interrupts occurred even with one thread, but there were three more overhead sources. First, the checking operation in Line 9 of Figure 2 and the new yield points described in Section 4.2 caused 5%-14% overhead. Second, as described in Section 4.4, we changed the logic of the method-invocation inline caches to reduce conflicts. This change degraded the single-thread performance by up to 5%. To avoid this degradation, HTM-friendly inline caches, such as thread-local caches, are required. Third, on zEC12, access to Pthread's thread-local storage accounted for 9% of the execution cycles on average. As explained in Section 4.4, we moved several global variables to the thread-local storage. Unfortunately, the access function, pthread_getspecific(), is not optimized in z/OS, but it is highly tuned in some environments, including Linux.

### 5.7 Scalability characterization

The abort ratios and cycle breakdowns of the Ruby NPB in Figure 8 had little correlation with the speedups in Figure 5. These facts suggest that although the speedups 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 Figure 9, we compare the scalability of HTM-dynamic on zEC12, JRuby, and the Java NPB, from which

the Ruby NPB was translated. Figure 9 shows that even the Java NPB hit scalability bottlenecks and HTM-dynamic resembled the Java NPB in terms of the scalability. These results confirmed that the differences in the speedups by HTM-dynamic among the benchmarks originated from each program's own scalability characteristics. 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 Figure 9). We believe the characteristics of each benchmark were different between HTM-dynamic and JRuby because of JRuby's internal scalability bottlenecks.

# 6. Related Work

Riley et al. [23] 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 a real HTM. Tabba [29] 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 function returns or tolerate TLB misses. These prototype results cannot be extended to real-world applications. The GIL in Ruby was eliminated through HTM in our prior report [18]. This paper is the first to evaluate larger benchmarks including WEBrick and Ruby on Rails on two implementations of less-restrictive HTM, zEC12 and Xeon E3-1275 v3.

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. 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 other types of aborts, which calls for the dynamic transaction-length adjustment.

Dice et al. [3] evaluated a variety of programs using HTM on the Sun Rock processor. Wang et al. [31] measured the STAMP benchmarks [15] 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 [9,10,12,13,24] use or are going to use fine-grained locking instead of the GIL. JRuby [12] 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 2.2.1 of Rubinius [24] uses fine-grained locking. However, the Rubinius support for the CRuby-compatible extension libraries conflicts with further removing the locks. In contrast, replacing the GIL with HTM creates no compatibility problems in the libraries and can yet increase scalability. PyPy is planning to eliminate the GIL through software transactional memory (STM) [20], but it is unclear whether the scalability improvement can offset the overhead of the STM.

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 [19]. The cloning makes a GIL unnecessary, but it is as heavy as fork() and restricts shared-memory programming. Lua [14] 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) [4] does not support multi-threading, so the programs must be written in an asynchronous event-handling style.

# 7. Conclusion and Future Work

This paper shows the first empirical results of eliminating the Global Interpreter Lock (GIL) in a scripting language through Hardware Transactional Memory (HTM) to improve the multi-thread performance of realistic programs. We proposed a new automatic mechanism to dynamically adjust the transaction lengths on a per-yield-point basis. Our mechanism chooses 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 footprint overflows. We experimented on the HTM facilities in the mainframe processor IBM zEC12 and the Intel 4th Generation Core processor (Xeon E3-1275 v3). We evaluated the Ruby NAS Parallel Benchmarks (NPB), the WEBrick HTPP server, and Ruby on Rails. Our results show that HTM achieved up to a 4.4-fold speedup in the Ruby NPB, and 1.6-fold and 1.2-fold speedups in WEBrick and Ruby on Rails, respectively. The dynamic transaction-length adjustment chose the best transaction lengths. On Xeon E3-1275 v3, programs need to run long enough to benefit from the dynamic transaction-length adjustment.

From all of these results, we concluded that HTM is an effective approach to achieve higher multi-thread performance compared to the GIL.

Our techniques will be effective also in Python, because our GIL elimination and dynamic transaction-length adjustment do not depend on Ruby. Conflict removal can be specific to each implementation. For example, the original Python implementation (CPython) uses reference counting GC, which will cause many conflicts, while PyPy uses copying GC and thus is more suitable for the GIL elimination through HTM.

## Acknowledgments

We would like to thank our colleagues in IBM Research for helpful discussions. We are also grateful to the anonymous reviewers for valuable comments.

## 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] Dice, D., Lev, Y., Moir, M., and Nussbaum, D. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS*, pp. 157-168, 2009.

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

[5] 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.

[6] IBM. Power ISA Transactional Memory. Power.org, 2012.

[7] IBM. z/Architecture Principles of Operation Tenth Edition (September, 2012). http://publibfi.boulder.ibm.com/epubs/pdf/dz9zr009.pdf .

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

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

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

[11] Jacobi, C., Slegel, T., and Greinder, D. Transactional memory architecture and implementation for IBM System z. In *MICRO 45*, 2012.

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

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

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

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

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

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

[18] Odaira, R. and Castanos, J. G. Eliminating global interpreter locks in Ruby through hardware transactional memory. Research Report RT0950, IBM Research – Tokyo, 2013.

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

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

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

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

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

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

[25] Ruby on Rails. http://rubyonrails.org/ .

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

[27] Shum, C.-L. IBM zNext: the 3rd generation high frequency micro-processor chip. In *HotChips* 24, 2012.

[28] Stuecheli, J. Next Generation POWER microprocessor. In *HotChips* 25, 2013.

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

[30] 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.

[31] 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.