Do C and Java Programs Scale Differently on Hardware Transactional Memory?

Rei Odaira*, Jose G. Castanos+, Takuya Nakaike*

* IBM Research – Tokyo
Toyosu, Tokyo, Japan
{odaira, nakaike}@jp.ibm.com

+ IBM Research – T. J. Watson Research Center
Yorktown Heights, NY, USA
castanos@us.ibm.com

***Abstract*— People program in many different programming languages in the multi-core era, but how does each programming language affect application scalability with transactional memory? As commercial implementations of Hardware Transactional Memory (HTM) enter the market, the HTM support in two major programming languages, C and Java, is of critical importance to the industry. We studied the scalability of the same transactional memory applications written in C and Java, using the STAMP benchmarks. We performed our HTM experiments on an IBM mainframe zEnterprise EC12. We found that in 4 of the 10 STAMP benchmarks Java was more scalable than C. The biggest factor in this higher scalability was the efficient thread-local memory allocator in our Java VM. In two of the STAMP benchmarks C was more scalable because in C padding can be inserted efficiently among frequently updated fields to avoid false sharing. We also found Java VM services could cause severe aborts. By fixing or avoiding these problems, we confirmed that C and Java had similar HTM scalability for the STAMP benchmarks.**

Keywords—transactional memory; programming language; C; Java

I. Introduction

Transactional memory is a promising technology in the multi-core era for better scalability with less programming effort. A programmer can simply enclose the critical sections with begin and end directives to define transactions. Each 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, which allows transactional memory to outperform global locking. Although transactional memory is attractive for its potential concurrency, pure software implementations, called Software Transactional Memory (STM), incur high overhead.

Recently, chipmakers began designing and producing special hardware for transactional memory, called Hardware Transactional Memory (HTM). Programmers want to write applications that take advantage of transactional memory hardware. Sun announced the Rock processor with an HTM facility [2], though the processor was cancelled before reaching the market. Intel documented an instruction set architecture called Transactional Synchronization Extensions [11] and implemented it in their Haswell processors. IBM has released Blue Gene/Q and the mainframe processor zEnterprise EC12 (zEC12) with HTM support [7,18]. IBM also created HTM extensions for the Power ISA [9].

As transactional memory becomes widely available, more and more programming languages support transactional memory. One technical report proposes transactional language constructs for C++ [1]. The GNU C/C++ compiler already supports transactional memory intrinsics for x86 and the IBM XL C/C++ compiler offers them for a mainframe architecture [15]. *Intrinsics* are special API calls that are replaced with the corresponding hardware instructions in the compiled code. Many Software Transactional Memory (STM) systems have been developed for various programming languages, such as DSTM2 [8] for Java, TinySTM [21] for C/C++, the STM package for Haskell [19], and Clojure's STM support [4].

In spite of the broad support for transactional memory in these programming languages, it is still unclear how each programming language and its implementations affect the performance of the applications that use transactional memory. Of particular importance are the HTM scalability characteristics of C and Java, two major imperative programming languages. With the recent releases of commercial HTM implementations, more and more applications will be written in these leading programming languages [22] using HTM.

In this paper, we compare the HTM scalabilities of the same transactional memory applications implemented in both C and Java. We measured the STAMP benchmarks [14], which were originally written in C, and later ported to Java [13]. To run the Java programs using transactional memory on HTM, we developed platform-independent Java APIs for the HTM intrinsics so that the transaction begin, end, and other operations supported by the underlying hardware can be invoked from Java. The calls to the HTM intrinsics are recognized by our just-in-time (JIT) compiler and are converted to HTM instructions embedded in the JIT-compiled code. Using the HTM intrinsics for Java and inline assembly for C, we implemented the same transaction retry logic in both C and Java for fair comparison.

Our experimental platform was IBM's HTM implementation in zEC12 and this paper is also the first to report the results of the full set of STAMP benchmarks on zEC12.

Here are our contributions:

- We compared the scalability characteristics of two major programming languages, C and Java, on a commercial HTM implementation using the de facto standard STAMP benchmarks.

- We developed platform-independent HTM intrinsics for Java and corresponding JIT-compiler optimizations in the IBM J9/TR Java VM [6].
- We performed the first full measurements of the STAMP benchmarks on the HTM implementation of the IBM mainframe zEnterprise EC12 (zEC12).

Section II describes the HTM implementation used in our studies. Section III presents our HTM intrinsics for Java and Section IV explains the original STAMP benchmarks and how they were ported to Java. Section V shows the experimental results and Section VI covers related work. Section VII concludes this paper.

II. HTM Implementation

We used HTM in an IBM mainframe zEC12 [18] for our studies. This section briefly describes the instruction set architecture that supports the HTM and its micro-architecture implementation. A complete overview of the zEC12 HTM implementation was described by Jacobi et al. [12] and the full instruction set architecture is defined in the manual [10].

A. *Instruction Set Architecture*

Each transaction begins with a TBEGIN instruction and is ended by a TEND instruction. The TBEGIN instruction saves the general purpose registers but not the floating point registers. Therefore, the programmer is responsible for saving and restoring the floating point registers as needed.

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 SMP system. Each core has a 96-KB L1 data cache and a 1-MB L2 data cache. 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 special transaction-backup register file. The maximum supported 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 the tx-dirty bit set. 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 cache. Thus the maximum read-set size is roughly the size of the L2 cache. 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. An overflow abort happens if the read-set or write-set size exceeds their respective limitations.

III. HTM Intrinsics For Java

This section describes our HTM intrinsics for Java. *Intrinsics* are special API calls that are replaced with the corresponding hardware instructions in the compiled code. Because the Java standard does not support HTM programming, we designed special methods to call the HTM operations. These methods can be implemented using the Java Native Interface (JNI). However, since JNI is heavyweight, we modified our JIT compiler to recognize these methods and to generate the HTM instructions directly in the JIT-compiled code. First, we present the Application Programming Interface (API) with examples for lock elision. Second, we explain how the intrinsics are handled by our JIT compiler.

A. *API*

Table I shows the Java API for the HTM intrinsics. The HTM class includes the basic operations, while the HTM.DiagnosticInfo class abstracts the diagnostic methods needed to investigate why a transaction aborted. The actual diagnostic methods are implemented in a platform-dependent subclass, HTM.DiagnosticInfoZ for our mainframe. The sub-class also contains platform-dependent methods to directly access the diagnostic information. For instance, the getTDB() method is provided to read the contents of the Transaction Diagnostic Block (TDB).

TABLE I. APIs OF HTM INTRINSICS FOR JAVA.

Method	Description
public class HTM	
public static boolean begin()	Begin a transaction and return true. If the transaction aborts, the execution returns back to this method and the return value is false.
public static void end()	End a transaction.
public static void abort()	Abort a transaction.
public static abstract class HTM.DiagnosticInfo	
public static DiagnosticInfo create()	Create and return a platform-dependent object to diagnose aborted transactions (on our mainframe, return an instance of the HTM.DiagnosticInfoZ class).
public abstract void read()	Read diagnostic information into instance fields.
public abstract boolean isValid()	Valid diagnostic infromation?
public abstract boolean abortByConflict()	Abort due to transaction conflict?
public abstract boolean abortByRestrictedInstruction()	Abort due to a restricted instruction?
public abstract boolean abortByFootprintOverflow()	Abort due to a transaction footprint overflow?
public abstract boolean abortByNestingOverflow()	Abort due to too many nested transactions?
public abstract boolean abortByTABORT()	Abort due to a TABORT instruction?
public static class HTM.DiagnosticInfoZ extends HTM.DiagnosticInfo	
Implementations of the abstract methods in HTM.DiagnosticInfo	
public long[] getTDB()	Return the contents of TDB.

Fig. 1 shows a code example using the HTM intrinsics for lock elision. When executing the transactions in the STAMP benchmarks in our experiments, we used almost the same algorithm except for the optimizations described in the next section. A utility class AtomicRegion is defined in Lines 7-54. This is used to enclose the critical section in Lines 1-6. Because most of the HTM implementations use best-effort algorithms, a fallback mechanism is needed. In this example, a simple global spin lock is used (Line 9). Other than the difference in the concurrency control mechanism, the transactional path and the fallback path execute the same code in the critical section (Line 3). The simple spin lock does not support nesting, but it is easy to extend it to a reentrant lock.

AtomicRegion.begin() first tries to execute a critical section as a transaction. After beginning the transaction, it reads the lock into the transaction read-set in Line 18, so that the transaction can be aborted later if the lock is acquired by another thread. The transaction must abort immediately if the lock is already acquired, because otherwise the transaction could read data that was modified.

If a transaction aborts, the abort's cause is checked to determine if it is persistent or transient. If it is persistent, the execution reverts to the lock (Lines 30 and 42-47). If the abort is transient, the transaction is retried some dozen times before acquiring the lock (Lines 33-36). In zEC12, the TBEGIN instruction can determine whether an abort's cause is persistent or transient from the condition code, but we instead use the abort code reported by the TDB (Lines 26-28) so that we can fine tune the retry logic based on the abort code. Our implementation retries the transaction up to 16 times for transient aborts. In our preliminary experiments, we confirmed that it was unlikely that a transaction would ever succeed after 16-or-more consecutive transient aborts.

AtomicRegion.end() in Lines 48-53 releases the lock or ends the transaction, depending on whether or not this critical section has been executed with an acquired lock.

B. Implementation of the HTM Intrinsics

The default implementation of the HTM.begin() method in the class library simply returns false, and the HTM.end() and HTM.abort() methods do nothing. However, we modified the JIT compiler in IBM J9/TR to recognize these methods and to embed the corresponding HTM instructions directly into the JIT-compiled code. Therefore the critical section is executed correctly but never as a transaction during interpreted execution. Enabling transactional execution for the interpreter is future work.

The front-end of the JIT compiler detects calls to these methods and converts them into special intermediate-language operators for TBEGIN, TEND, and TABORT. The back-end of the compiler generates the corresponding hardware instructions. The floating-point registers that must live across a TBEGIN instruction are saved to the stack before the TBEGIN instruction and are restored if the transaction aborts. The address of thread-local memory is passed to the TBEGIN instruction as a TDB area because the HTM.DiagnosticInfoZ class relies on the information recorded in the TDB.

To reduce overhead, our JIT compiler also recognizes calls to HTM.DiagnosticInfo.read() because its default implementation calls the heavyweight JNI. On our mainframe, the JIT compiler generates code to copy the contents of the TDB into an instance field of the HTM.DiagnosticInfoZ object.

IV. C AND JAVA STAMP BENCHMARKS

This section describes how we measured the C and Java versions of the STAMP benchmarks. STAMP [14] is the most widely used transactional-memory benchmark suite. It was originally written in C and consists of 8 programs using both fine-grain and coarse-grain transactions. Table II shows the benchmarks and their default runtime options.

A. C STAMP Benchmarks

We used Version 0.9.10 of the C STAMP benchmarks. The C version encloses the critical sections with TM_BEGIN() and TM_END() macros. Users of the benchmarks must implement their own TM_BEGIN() and TM_END() for their specific measurement environments.

We implemented three versions of TM_BEGIN() and TM_END(). The first one uses the HTM of zEC12, the second one uses global locking for reference, and the third one is a baseline sequential version, which emits no code for TM_BEGIN() or TM_END(). The algorithm of the HTM version is similar to AtomicRegion.begin() and end() shown in

```
1.  AtomicRegion.begin();
2.  try {
3.    // Critical section
4.  } finally {
5.    AtomicRegion.end();
6.  }

7.  import java.util.concurrent.atomic.AtomicBoolean;
8.  class AtomicRegion {
9.    private static final AtomicBoolean lock
        = new AtomicBoolean(false);
10.   private static final HTM.DiagnosticInfo diag
        = HTM.DiagnosticInfo.create();
11.   private static final int RETRY_COUNT_MAX = 16;

12.   public static void begin() {
13.     int retryCount = RETRY_COUNT_MAX;
14.     while (true) {
15.       // (A) See Section IV.
16.       if (HTM.begin()) {
17.         // Transaction
18.         if (lock.get())
19.           HTM.abort();
20.         return;
21.       } else {
22.         // Abort
23.         diag.read();
24.         // (B) See Sectin IV.
25.         if (! diag.isValid() ||
26.             diag.abortByFootprintOverflow() ||
27.             diag.abortByRestrictedInstruction() ||
28.             diag.abortByNestingOverflow()) {
29.           // Persistent abort
30.           fallBackGlobalLock();
31.           return;
32.         } else {
33.           // Transient abort
34.           if (--retryCount > 0)
35.             continue;
36.           fallBackGlobalLock();
37.           return true;
38.         }
39.       }
40.     }
41.   }

42.   private static void fallBackGlobalLock() {
43.     do {
44.       while (lock.get())
45.         ;
46.     } while (! lock.compareAndSet(false, true));
47.   }

48.   public static void end() {
49.     if (lock.get())
50.       lock.set(false);
51.     else
52.       HTM.end();
53.   }
54. }
```

Fig. 1. Code example of lock elision using the HTM intrinsics for Java.

Fig. 1. It uses inline assembly to invoke the zEC12 instructions for TBEGIN, TEND, and TABORT. As an optimization, in Lines 15 and 24 of Fig. 1, we check whether the lock was acquired and if yes, spin-wait until it is released. After the spin wait, the transaction is retried up to 16 times. The global-locking version uses a single global spin lock, which is the same as the fallback spin lock in Fig. 1.

The barrier synchronization function included in the C STAMP benchmarks uses Pthread's mutex locks, which are heavyweight calls on our mainframe platform. For this reason, we implemented the sense-reversing barrier synchronization using spin locks.

B. *Java STAMP Benchmarks*

The Java version of the STAMP benchmarks [13] was ported from the C version by the Programming Languages Research Group at the University of California, Irvine. It was written specifically for their own Java-to-C translator that accepts a dialect of the Java language. This Java version does not compile and run as-is on a standard Java environment.

We rewrote the Java STAMP benchmarks using the HTM intrinsics, allowing us to build and run them using our J9/TR Java VM. Other than the HTM intrinsics, the rewritten version relies only on the standard Java features. The critical sections are enclosed with AtomicRegion.begin() and end() as shown in Fig. 1. AtomicRegion.begin() uses the optimizations described in Section IV.A at Lines 15 and 24. We implemented the same barrier synchronization algorithm as used in the C version. We also found and fixed several bugs in the Java version. In addition, we modified the random number generator and the hash code calculator so that they generated the same values as the corresponding C routines. For reference, we developed a global-locking version using a spin lock and a baseline sequential version. We are planning to publicly release the rewritten Java STAMP benchmarks.

Fig. 2 compares code excerpts from the C and Java versions. Fig. 2 part (a) shows the C and Java code to insert an element into a hash table. The C function takes a pointer to a table as the first argument, whereas in Java the method is an instance method of the Table class. The Java STAMP benchmarks rarely use the collection classes in the standard Java class libraries. Instead, they include their own basic collection classes that are translated from the corresponding data structures in the C STAMP benchmarks. As shown in Fig. 2 part (b), each access to a field of a structure in C is directly translated to an access to a corresponding instance field of an object in Java. In the C STAMP benchmarks, the reads and writes to shared data are annotated with the TM_SHARED_READ() and TM_SHARED_WRITE() macros,

TABLE II. STAMP BENCHMARK SUITE

Benchmark	Description	Default Runtime Options
bayes	Learns structure of a Bayesian network	-v32 -r4096 -n10 -p40 -i2 -e8 -s1
genome	Performs gene sequencing	-g16384 -s64 -n16777216
intruder	Detects network intrusion	-a10 -l128 -n262144 -s1
kmeans-high	Implement K-means clustering	-m15 -n15 -t0.05 -i inputs/random2048-d16-c16.txt
kmeans-low		-m40 -n40 -t0.05 -i inputs/random2048-d16-c16.txt
labyrinth	Routes paths in maze	-i inputs/random-x512-y512-z7-n512.txt
ssca2	Creates efficient graph representation	-s20 -i1.0 -u1.0 -l3 -p3
vacation-high	Emulates travel reservation system	-n4 -q60 -u90 -r1048576 -t4194304
vacation-low		-n2 -q90 -u98 -r1048576 -t4194304
yada	Refines a Delaunay mesh	-a15 -i inputs/ttimeu1000000.2

(a-1) C / sequencer_run() in genome

```
1.  TM_BEGIN();
2.  status = TMTABLE_INSERT(startHashToConstructEntryTables[j],
                            (ulong_t)startHash,
                            (void*)constructEntryPtr );
3.  TM_END();
```

(a-2) Java / Sequencer.run() in genome

```
1.  AtomicRegion.begin();
2.  try {
3.    check = startHashToConstructEntryTables[newj]
               .table_insert(startHash, constructEntryPtr);
4.  } finally {
5.    AtomicRegion.end();
6.  }
```

(b-1) C / checkTaskList() in bayes

```
1.  TM_BEGIN();
2.  float globalBaseLogLikelihood =
      TM_SHARED_READ_F(learnerPtr->baseLogLikelihood);
3.  TM_SHARED_WRITE_F(learnerPtr->baseLogLikelihood,
              (baseLogLikelihood + globalBaseLogLikelihood));
4.  TM_END();
```

(b-2) Java / Learner.checkTastList() in bayes

```
1.  AtomicRegion.begin();
2.  try {
3.    float globalBaseLogLikelihood =
        learnerPtr.baseLogLikelihood;
4.    learnerPtr.baseLogLikelihood =
        (baseLogLikelihood + globalBaseLogLikelihood);
5.  } finally {
6.    AtomicRegion.end();
7.  }
```

Fig. 2. Comparison of code excerpts from the C and Java versions of the STAMP benchmarks.

so that the STM can instrument these accesses. With the HTM, these macros do not perform any instrumentation and simply read and write the data.

In many Java VMs, frequently executed methods are compiled by a JIT compiler. However, because the STAMP benchmarks run for only 3 to 30 seconds with the default parameters, they finish before a sufficient number of methods have been JIT-compiled. Therefore, we added a harness class to the Java STAMP benchmarks to invoke the main() method of a specified benchmark multiple times during a single run. We tried different parameters to run the benchmarks longer and found the scalability tendencies did not change, but we decided to keep using the default parameters to insure our results are comparable with other work using the STAMP benchmarks.

V. EXPERIMENTAL RESULTS

This section describes our experimental environment and then compares the results of the C and Java versions of the STAMP benchmarks.

A. Experimental Environment and Settings

We evaluated the STAMP benchmarks on zEC12 running the mainframe z/OS 1.13 with UNIX System Services (USS). The experimental system was divided into multiple Logical PARtitions (LPARs), and each LPAR corresponded to a virtual machine. Our LPAR was assigned 16 cores, each running at 5.5 GHz, and with 6 GB of main memory. Although the system was not totally dedicated to our experiments, no other processes were running when we collected our data.

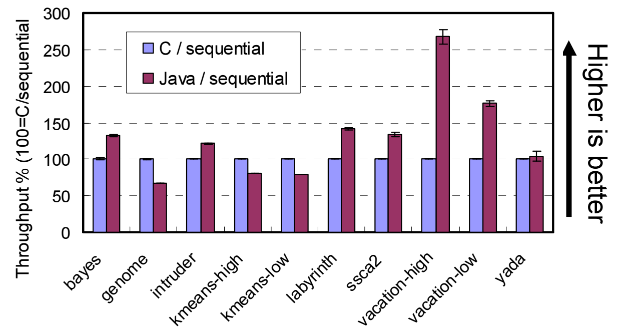

Fig. 3. Comparison of the throughput of sequential execution in C and Java. The 95% confidence intervals are also shown.

The C versions were compiled as 64-bit programs with the IBM XL C/C++ compiler for z/OS. The O3 optimization option was specified. The Java version was run on 64-bit IBM J9/TR 1.7.0 SR1 with 4 GB of Java heap and mark-and-sweep GC. The GC time accounted for less than 5% of the execution time for all of the benchmarks.

We ran each benchmark 4 times and took the average of the runs. For Java, we made each run iterate a benchmark for at least 2 minutes and averaged the execution times of the second half of the iterations as the result for that run. This measurement method masked the effect of the JIT compiler, since most of the JIT compilation was done during the first half of the iterations.

B. Comparison between C and Java

Fig. 3 compares the throughput of each sequential version of C and Java using the default parameters shown in Table II. The results are normalized against the C results. The purpose of this paper is not to analyze the absolute performance of C and Java but to compare their scalabilities. However, if their absolute performance differed greatly, for example by an order of magnitude, then it would be meaningless to do any detailed scalability comparison. In Fig. 3, the performance of Java was within 67% to 267% of that of C. This means that the absolute performance results of C and Java were close enough for meaningful comparisons.

Fig. 4 shows the throughput of the C and Java STAMP benchmarks using the lock and HTM. We changed the number of threads from 1 to 2, 4, 8, and then 16. We also show the 95% confidence intervals. The C and Java results are normalized to their respective sequential version results, so that we can compare the scalability of C and Java. We compare these results with the previous report on Blue Gene/Q in Section V.D.

The HTM versions of C and Java scaled similarly for labyrinth, ssca2, and yada. In bayes, the C and Java versions showed different average characteristics, but the fluctuations were so large that we could not draw any conclusions. This was because the running time of the multi-threaded bayes tends

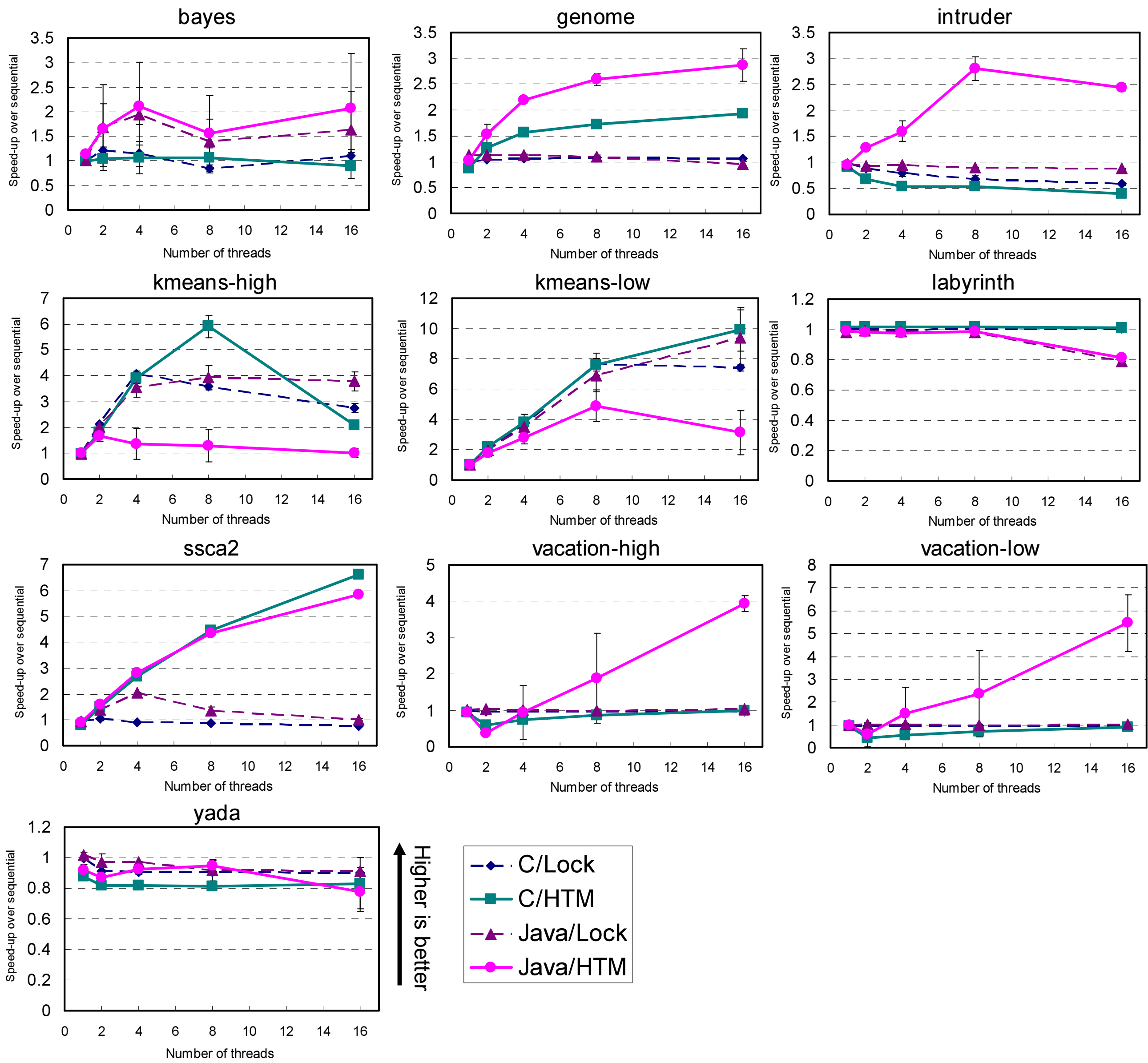

Fig. 4. Throughput of the C and Java STAMP benchmarks when running with the default benchmark parameters, using a single global lock and HTM. The C and Java results are normalized to their respective sequential versions' results. The 95% confidence intervals are also shown.

to vary depending on the insertion order of the edges into the Bayesian network. In genome, intruder, vacation-high, and vacation-low, Java was more scalable than C, while in kmeans-high and kmeans-low C outperformed Java. In the remainder of this section, we explain the causes of these scalability differences and suggest ways to address them.

1) Thread-local memory allocator

The lower scalability of C in genome, intruder, vacation-high, and vacation-low was caused by excessive transaction conflicts in the malloc() memory allocator. These benchmarks allocate many objects within the transactions. A naïve implementation of a memory allocator allocates an object from global free lists. This mechanism obviously causes conflicts in multi-threaded execution. In contrast, the J9/TR Java VM and most of the other high-performance Java VMs allocate objects on a thread-local basis to avoid conflicts. The HotSpot Java VM uses Thread-Local Allocation Buffers (TLABs) [16]. Each application thread allocates objects from its own TLAB. If the TLAB becomes empty, another TLAB is allocated from the global Java heap and assigned to the thread. The J9/TR Java VM also uses a similar mechanism.

To avoid the conflicts at the memory allocator, the C version of the STAMP benchmarks includes a simple but incomplete thread-local memory allocator. This resembles the TLAB in the HotSpot Java VM, but it does not support the release of objects. Since this incomplete allocator can run the C

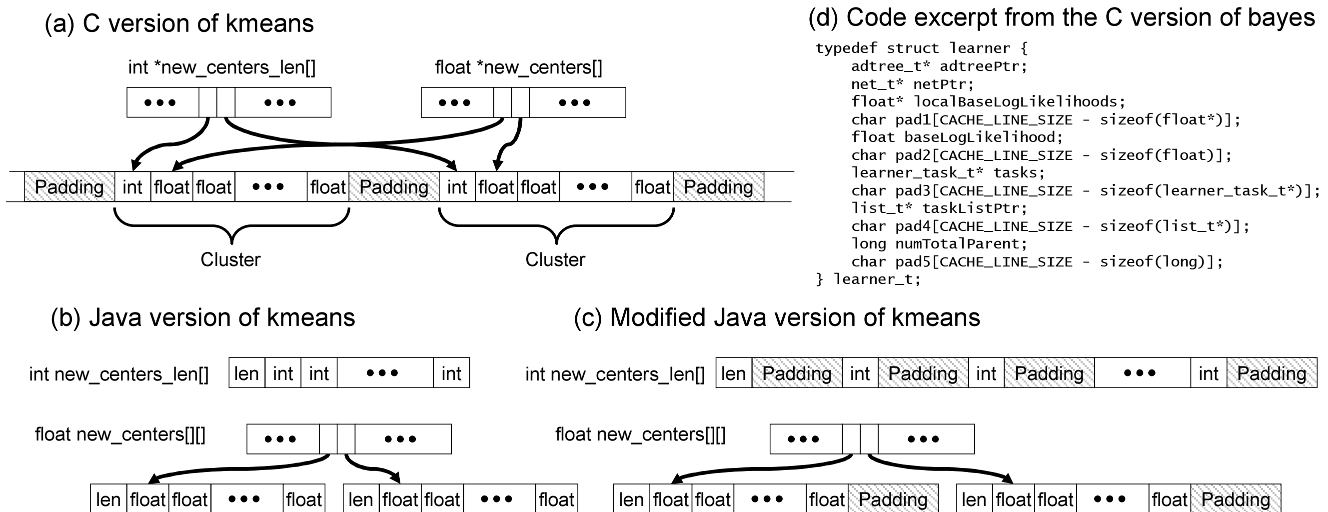

Fig. 5. Data structures used in the C and Java versions of kmeans and bayes. (a) The C version of kmeans inserts padding within the sets of an integer and floating-point numbers representing clusters. (b) The original Java version of kmeans does not insert any padding. The "len" fields contain the lengths of the arrays for array bound checking. (c) The modified Java version separates the data belonging to different clusters with padding, but this uses more memory than in the C version. (d) The C version of bayes also inserts padding.

STAMP benchmarks without any errors, we replaced the invocations of malloc() with calls to the thread-local memory allocator. These measurement results appear in the next section. We could instead use an efficient multi-threaded memory allocator, such as TCMalloc [20], but its performance on the HTM is still unknown and studying it is future work.

2) Padding to avoid false sharing

C was more scalable than Java in kmeans-high and kmeans-low, because the C version inserts padding between frequently updated fields to avoid false sharing. If each of the fields that is frequently updated by different threads were placed into the same cache line, they would cause excessive conflicts due to false sharing. Inserting padding among such fields to place them in different cache lines is a widely used programming technique. In kmeans, the data structures of a cluster consist of an integer and an array of floating-point numbers, as shown in Fig. 5(a). Because each transaction updates one cluster at a time, the data structures of a cluster are all located in a contiguous memory region. In contrast, the data structures of different clusters are separated by padding, so that different transactions updating different clusters do not conflict with one another.

In the Java version, since it is not possible to allocate the integer and the array of floating-point numbers in a contiguous area, they are represented as different arrays, as shown in Fig. 5(b). This representation obviously causes conflicts due to false sharing because the data belonging to different clusters can be placed into the same cache line. We modified the original Java version by inserting padding around each integer and each array of floating-point numbers, as shown in Fig. 5(c). We show the results later. This implementation is less memory-efficient than the C version. In particular, on the mainframe platform, each integer occupies a 256-byte cache line. Note that each array in Java has a hidden field at the head to hold the length of the array, and padding is necessary between the length field and the first element. Because the length field is read in every transaction for array bound checking, without the padding, conflicts happen when transactions modify the first element. Depending on the Java VM, the length field is not necessarily at the head of its array, so this padding method is Java VM-dependent.

The difference in padding between C and Java is not specific to kmeans. For example, the C version of bayes uses the data structure shown in Fig. 5(d), which has padding among the frequently-updated fields. This type of padding cannot be implemented in the Java language because arrays cannot be embedded into an object and the field order within an object is not necessarily the same as the order written in the source code. Therefore the corresponding data structure in the Java version of bayes does not include any padding.

In general, the Java language has difficulties in handling data structures with padding, as shown in these examples. More sophisticated VM support is desired in Java, such as a feedback-directed mechanism to automatically co-locate the data structures accessed within the same transaction and to separate frequently updated fields.

3) Java VM services

In vacation-high and vacation-low, although Java was more scalable than C, we found that Java VM services invoked during the transactions reduced the scalability. Specifically, JIT-compiled code with profiling instrumentation was executed during the transactions, and the profiling code caused severe conflicts. Our J9/TR Java VM performs multi-level JIT compilation, where methods are first executed by the interpreter, and then frequently executed methods are JIT-compiled at a lower optimization level. If a JIT-compiled method is more frequently executed, then it is JIT-compiled again with profiling instrumentation. The instrumented JIT-compiled code is executed for a short period and finally the

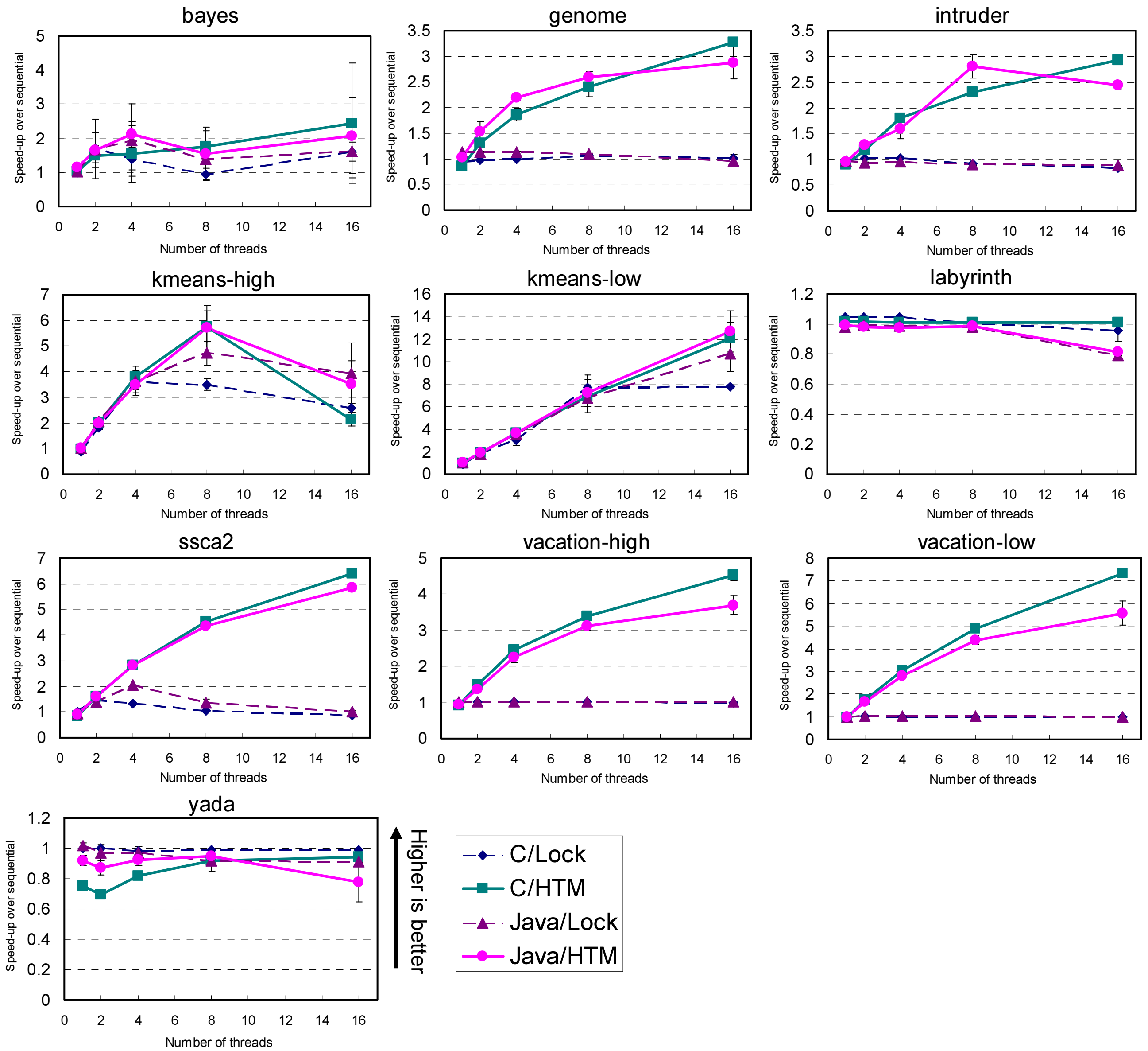

Fig. 6. Throughput of the C and Java STAMP benchmarks after modifing the three points described in Section V.B. Overall, C/HTM and Java/HTM scaled similarly. When compared with Fig. 4, the throughput of the C version was improved by the thread-local memory allocator, especially in intruder, vacation-high, and vacation-low. The Java version scaled better in kmeans-high and kmeans-low because of the inserted padding and in vacation-high and vacation-low by avoiding profiling code exection during the transactions.

method is JIT-compiled again at a higher optimization level with the collected profile. A similar mechanism is also implemented in other Java VMs. The profiling is controlled by global data structures, which are where the contentions occurred. In most of the benchmarks, the profiling was done before the measurement period (the last minute of the 2-minute execution period), but in vacation-high and vacation-low, the profiling code for a method was executed during the measurement period. We avoided these contentions by disabling the profiling for the specific method that caused the contentions in vacation-high and vacation-low. A more fundamental solution would be to implement a profiling mechanism that accesses fewer global data structures.

The profiling mechanism for the JIT compilation is not the only Java VM service that can cause many aborts. For example, a JIT compiler can cause transaction overflows if it is invoked during a transaction. Another example is code patching, which is not allowed in most of the HTM implementations. Therefore, to get better scalability in Java, Java VMs must be aware of HTM and make their services HTM-friendly.

C. *Comparison between C and Java after Modification*

Fig 6 shows the scalabilities of the C and Java versions of the STAMP benchmarks with the modifications described in the previous section. Compared with Fig. 4, the thread-local

TABLE III. PERCENTAGE OF SERIALIZED TRANSACTIONS AND ABORTED TRANSACTIONS, AND AVERAGE NUMBER OF RETRIES

Benchmark		Serialization ratio (# threads)					Abort ratio (# threads)					Average # of retries (# threads)				
		1	2	4	8	16	1	2	4	8	16	1	2	4	8	16
bayes	C	32%	48%	52%	59%	68%	39%	94%	96%	97%	98%	0.1	7.6	10.2	13.7	18.7
	Java	30%	38%	45%	56%	58%	35%	89%	94%	97%	97%	0.1	4.4	8.1	12.6	15.0
genome	C	13%	13%	12%	11%	10%	16%	41%	52%	59%	68%	0.0	0.5	0.9	1.2	1.8
	Java	11%	9%	9%	10%	10%	14%	30%	41%	54%	69%	0.0	0.3	0.5	1.0	1.9
intruder	C	1%	2%	2%	4%	4%	2%	26%	42%	59%	69%	0.0	0.3	0.7	1.3	2.1
	Java	1%	2%	6%	10%	12%	1%	31%	69%	80%	81%	0.0	0.4	2.1	3.6	3.6
kmeans-high	C	0%	0%	1%	7%	0%	0%	15%	39%	76%	27%	0.0	0.2	0.6	2.9	0.4
	Java	0%	0%	2%	0%	1%	0%	9%	52%	31%	40%	0.0	0.1	1.0	0.4	0.7
kmeans-low	C	0%	0%	1%	0%	0%	0%	5%	14%	6%	19%	0.0	0.1	0.2	0.1	0.2
	Java	0%	0%	0%	0%	0%	0%	3%	4%	20%	18%	0.0	0.0	0.0	0.2	0.2
labyrinth	C	50%	50%	50%	55%	50%	51%	74%	86%	95%	94%	0.0	0.9	2.6	8.8	7.0
	Java	50%	51%	51%	55%	84%	53%	75%	86%	96%	99%	0.1	0.9	2.5	9.9	19.2
ssca2	C	0%	0%	0%	0%	0%	0%	11%	29%	32%	29%	0.0	0.1	0.4	0.5	0.4
	Java	0%	0%	0%	0%	0%	1%	1%	1%	2%	4%	0.0	0.0	0.0	0.0	0.0
vacation-high	C	15%	13%	13%	12%	12%	20%	46%	56%	69%	79%	0.1	0.6	1.0	1.8	3.3
	Java	15%	15%	15%	17%	13%	19%	56%	67%	80%	80%	0.0	1.0	1.5	3.0	3.4
vacation-low	C	7%	5%	5%	6%	5%	12%	24%	36%	49%	60%	0.1	0.2	0.5	0.8	1.4
	Java	6%	6%	6%	5%	6%	9%	39%	44%	55%	68%	0.0	0.5	0.7	1.1	2.0
yada	C	17%	18%	18%	18%	19%	23%	76%	79%	83%	87%	0.1	2.4	3.1	4.0	5.5
	Java	19%	19%	19%	25%	25%	21%	68%	74%	89%	90%	0.0	1.5	2.1	4.8	6.5

memory allocator in the C/HTM improved the scalability of intruder, vacation-high, and vacation-low. The C version of bayes was improved too, but the fluctuations were still large. The Java versions scaled better in kmeans-high and kmeans-low when the appropriate padding was inserted. The Java versions of vacation-high and vacation-low were improved by avoiding the execution of profiling code during the transactions.

Table III shows the serialization ratios and abort ratios. A serialization ratio is the percentage of committed transactions that were executed with the global lock being acquired. An abort ratio is the percentage of executed transactions that were aborted. In ssca2, the C version suffered from more aborts than the Java version, but they did not affect the scalability because the transaction coverage is small. Although not shown in the table, persistent aborts accounted for less than 20% of all of the aborts with 2 to 16 threads. Genome, vacation-high, and labyrinth suffered from relatively high persistent abort ratios, due to read-set overflows (for genome and vacation-high) and write-set overflows (for labyrinth). Most of the transient aborts were caused by conflicts. Table III also presents the average number of retries. The numbers can be more than 16, which is the maximum number of retries specified in Line 11 of Fig. 1, because the transactions can retry not only for transient aborts but also for the acquired global lock (Line 24 of Fig. 1).

Overall, after the modifications described in Section V.B were applied, the scalability differences between C and Java became smaller than before. There are still gaps in vacation-high and vacation-low with 16 threads. This investigation is also for future research.

D. Comparison against Blue Gene/Q

We compared the results of the C STAMP benchmarks on zEC12 (Fig. 6) with those on another HTM implementation in Blue Gene/Q (BG/Q) [23]. In bayes, although the fluctuations were large, BG/Q scaled slightly better than zEC12. BG/Q achieved a 3-fold speed-up with 16 threads. Our zEC12 suffered from many conflicts. In genome, the scalability gap between BG/Q and zEC12 was even larger. The abort ratio on BG/Q was 13% with 16 threads, while that on zEC12 was 68%, as shown in Table III. Intruder and labyrinth showed similar scalability characteristics on BG/Q and zEC12. The zEC12 was more scalable than BG/Q for kmeans-high, kmeans-low, and ssca2, while vacation-high, vacation-low, and yada were less scalable on the zEC12. These benchmarks suffered from frequent conflicts on the zEC12.

In summary, the zEC12 was more scalable than BG/Q in 3 and less scalable in 5 of the 10 C STAMP benchmarks. Except for kmeans-high and kmeans-low, the abort ratios of zEC12 in Table III were higher than the corresponding results in Table 3 of the BG/Q paper [23]. Most of the aborts on the zEC12 were caused by transaction conflicts. We believe that the zEC12's 256-byte cache line size, which is longer than BG/Q's 64-byte L1 and 128-byte L2 cache line sizes, adversely affected the scalability, due to false sharing.

VI. RELATED WORK

Rolett evaluated the C++ and Java versions of micro-benchmarks on the Rochester Software Transactional Memory system [17]. Experimental results showed that the C++ version outperformed the Java version. The lower performance of the Java version was caused by indirect accesses to objects. Since the Java version wrapped all of the shared objects so that the objects were accessed only through getter and setter methods, it always went through the wrapper objects to access the shared objects. The indirect object accesses were also necessary for atomic operations such as compare-and-swap, because the implementation of atomic operations was boxed in the java.util.concurrent package classes. These types of overhead do not occur in an HTM because they are caused by the operations to implement the STM.

Wu et al. measured C/C++ and Java applications on an STM runtime system which was implemented in C [24]. The

compilers for C/C++ and Java generated the instrumentation code that called the same STM runtime library functions to manage transactions. The C/C++ applications had higher instrumentation overhead than the Java applications. This was because stack variables can be shared among multiple threads through pointers, and thus they needed to be instrumented in C/C++. The instrumentation for stack variables was not needed in Java because local variables were guaranteed to be isolated. This difference is not observed in an HTM system because all of the accesses to the variables in memory during the transactions are tracked in the same way by the hardware for both C/C++ and Java.

There are few reports that evaluate Java applications on a real HTM implementation. Only some micro-benchmarks were evaluated on the HTM of a pre-release Rock processor [5]. Since the Rock processor implemented much simpler HTM, the results might not apply to more recent HTM implementations. Although a large Java application was evaluated on the Azul HTM system, no detailed analysis was presented [3]. Therefore our paper is the first to evaluate the Java version of the STAMP benchmarks on a real HTM system and to provide a detailed performance analysis.

Wang et al. [23] evaluated the C version of the STAMP benchmarks on the HTM implementation of Blue Gene/Q and clarified the advantages and disadvantages of the HTM by comparing it with TinySTM [21]. We evaluated the same benchmarks on the HTM implementation of zEC12 and provided additional data that may help in designing future HTM systems.

VII. Conclusion

This paper investigates how programming language choice affects application performance on Hardware Transactional Memory (HTM). We compared the C and Java versions of the STAMP benchmarks. We developed new HTM intrinsics for Java, so that Java programs can call the HTM operations and the JIT compiler can embed the corresponding HTM instructions in the JIT-compiled code. We ran our test on a commercial HTM implementation on an IBM mainframe zEnterprise EC12. We found that in 4 of the 10 STAMP benchmarks Java was more scalable than C. The biggest factor in this higher scalability was the efficient thread-local memory allocator in a Java VM. In two of the STAMP benchmarks C was more scalable because in C padding can be inserted efficiently among frequently updated fields to avoid false sharing. We also found that Java VM services could cause many aborts. By using a thread-local memory allocator, inserting padding, and avoiding invoking the Java VM services during the transactions, we confirmed that C and Java had similar scalability on HTM in the STAMP benchmarks.

Acknowledgment

We would like to thank the members of the Commercial Systems group in IBM Research – Tokyo for helpful discussions. We are also grateful to the authors of the C and Java versions of the STAMP benchmarks.

References

[1] Boehm, H., Gottschlich, J., Luchangco, V., Michael, M., Moir, M., Nelson, C., Riegel, T., Shpeisman, and T., Wong, M., “Transactional language constructs for C++,” N3341=12-0031, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3341.pdf, 2012.

[2] 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.

[3] Click, C., “Azul’s Experiences with Hardware Transactional Memory,” 2009 Transactional Memory Workshop.

[4] Clojure, http://clojure.org/.

[5] Dice, D., Lev, Y., Moir, M., and Nussbaum, D., “Early experience with a commercial hardware transactional memory implementation,” In Architectural Support for Programming Languages and Operating Systems, Mar. 2009.

[6] Grcevski, N., Kilstra, A., Stoodley, K., Stoodley, M., and Sundaresan, V, “Java just-in-time compiler and virtual machine improvements for server and middleware applications,” in Proceedings of the 3rd Virtual Machine Research and Technology Symposium, pp. 151-162, 2004.

[7] 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.

[8] Herlihy, M., Luchangco, V., and Moir, M., “A flexible framework for implementing software transactional memory,” in OOPSLA, pp. 253-262, 2006.

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

[13] Java STAMP Benchmark Suite Version 0.5, http://demsky.eecs.uci.edu/software.php .

[14] Minh, C. C., Chung, J., Kozyrakis, C., and Olukotun, K, “STAMP: Stanford Transactional Applications for Multi-processing,” in IISWC, pp. 35-46, 2008.

[15] Mitran, M., and Vokhshoori, V., “IBM XL C/C++ compiler maximizes zEC12’s transactional execution capabilities,” IBM Systems Magazine, 2012

[16] Oracle, “Tuning the Java runtime system,” http://docs.oracle.com/cd/E19644-01/817-5051/pt_tuningjava.html

[17] Rolett, A., “A Java Implementation of the Rochester Software Transactional Memory Library,” Thesis --University of Rochester. Dept. of Computer Science, 2008.

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

[19] STM package for Haskell, http://hackage.haskell.org/package/stm .

[20] TCMalloc, http://goog-perftools.sourceforge.net/doc/tcmalloc.html .

[21] TinySTM, http://www.tmware.org/tinystm.

[22] TIOBE Programming Community Index, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html .

[23] Wang, A., Gaudet, M., Wu, P., Amaral, J. N., Ohmacht, M., Barton, C., Silvera, R., and Michael, M., “Evaluation of Blue Gene/Q Hardware Support for Transactional Memories,” Proceedings of the 21st international conference on Parallel architectures and compilation techniques, 2012.

[24] Wu, P., Michael, M. M., Praun, C., Nakaike, T., Bordawekar, R., Cain, H. W., Cascaval, C., Chatterjee, S., Chiras, S., Hou, R., Mergen, M., Shen, X., Spear, M. F., Wang, H. Y., and Wang, K., “Compiler and runtime techniques for software transactional memory optimization,” Concurrency and Computation: Practice & Experience - Compilers for Parallel Computers 2007 Workshop. pp. 7-23, Vol. 21, 2009.