Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8

Takuya Nakaike
IBM Research - Tokyo
nakaike@jp.ibm.com

Rei Odaira
IBM Research - Austin
rodaira@us.ibm.com

Matthew Gaudet
IBM Canada
mgaudet@ca.ibm.com

Maged M. Michael
IBM Watson Research Center
magedm@us.ibm.com

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

Abstract

Transactional Memory (TM) is a new programming paradigm for both simple concurrent programming and high concurrent performance. Hardware Transactional Memory (HTM) is hardware support for TM-based programming. It has lower overhead than software transactional memory (STM), which is a software-based implementation of TM. There are now four commercial systems, IBM Blue Gene/Q, IBM zEnterprise EC12, Intel Core, and IBM POWER8, offering HTM. Our work is the first to compare the performance of these four HTM systems. We measured the STAMP benchmarks, the most widely used TM benchmarks. We also evaluated the specific features of each HTM system. Our experimental results show that: (1) there is no single HTM system that is more scalable than the others in all of the benchmarks, (2) there are measurable performance differences among the HTM systems in some benchmarks, and (3) each HTM system has its own implementation characteristics that limit its scalability.

1. Introduction

Transactional memory (TM) [19] is a programming paradigm to enable both simple concurrent programming and high concurrent performance. In the TM programming environment, programmers simply define as transactions those program regions that access shared variables. TM runtime systems achieve high concurrent performance by optimistically executing the transactions in parallel. The runtime systems keep track of the accesses to the shared variables, buffer the stores to the shared variables during the transactions, and roll back some of the transactions when they detect conflicts among the accesses.

Hardware transactional memory (HTM) is becoming standard in modern processors because it provides lower overhead than software-based implementations of TM [25]. *IBM Blue Gene/Q* was the first to provide an accessible HTM implementation [1]. Although the *Azul* [4] and *Rock* [9] processors implemented HTM before Blue Gene/Q, their HTM systems were not usable because the Azul programming interface was not disclosed, so the HTM was hidden, and the Rock processor was canceled before reaching the market. After Blue Gene/Q, HTM was implemented on *IBM zEnterprise EC12 (zEC12)* [40], in the 4th generation of *Intel Core* [13], and in *IBM POWER8* [27] processors.

These HTM systems have been individually evaluated with various applications [1, 5, 7, 21, 23, 30, 31, 35, 36, 38, 39]. However, there is no single paper comparing the performance of all of the HTM systems using a common benchmark set. Because Blue Gene/Q, zEC12, Intel Core, and POWER8 are the first processors that implement HTM, clarifying their advantages and disadvantages is important to enhance the HTM implementations and improve the performance of the next generation of processors.

Our work is the first to quantitatively compare all of the existing HTM systems. For all four systems, we measured the STAMP benchmarks [3], the most widely used transactional memory benchmarks. To fairly compare the intrinsic performance of the HTM systems, we fixed the TM-unfriendly code that excessively increased nonessential transaction aborts in some of the STAMP benchmark programs. Also, we tuned the number of transaction retries so

ISCA '15, June 13 - 17, 2015, Portland, OR, USA
 ISBN 978-1-4503-3402-0/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750403

that the performance was maximized for each combination of an HTM system with the benchmarks.

Our major findings are: (1) there is no HTM system that is more scalable than the others for all of the benchmarks, (2) there are measurable performance differences among the HTM systems in some benchmarks, and (3) each HTM system has its own implementation limitations that degrade its scalability. These findings show that each HTM system still has room to improve its performance. A TM program may obtain performance benefits from future HTM systems even if it does not run well with the current systems.

Further, we evaluated the features specific to each HTM system: *constrained transactions* of zEC12, *hardware lock elision (HLE)* of Intel Core, and *suspend/resume instructions* and *rollback-only transactions* of POWER8. Our experimentations showed that each feature is beneficial to widen the applications of HTM.

Here are our contributions:

- We evaluated the HTM implementations of the Blue Gene/Q, zEC12, Intel Core, and POWER8 processors quantitatively on the STAMP benchmarks and clarified the advantages and disadvantages of each HTM implementation.
- We compared the maximum speed-up ratios of these HTM systems by repairing the TM-unfriendly code of the STAMP benchmarks and by exploring optimal transaction-retry counts for each HTM system.
- We analyzed in detail the causes of the transaction aborts and found the implementation-specific causes that degrade the scalability.
- We evaluated the features specific to each HTM system and showed the benefit of each feature.

Section 2 describes the differences in the HTM implementations of the Blue Gene/Q, zEC12, Intel Core, and POWER8 processors. Section 3 is about our transaction-retry mechanism. Section 4 describes our modifications for the STAMP benchmarks to reduce nonessential transaction aborts. Section 5 compares the performance of the four HTM systems and discusses the causes of the differences. Section 6 evaluates the specific features of each HTM system. Section 7 discusses the next generation of HTM systems. Section 8 reviews the related work. Section 9 concludes this paper.

2. Hardware Transactional Memory Implementations

In this section, we characterize the HTM implementations of the Blue Gene/Q, zEC12, Intel Core, and POWER8 processors and describe their differences.

These processors (except for Blue Gene/Q) provide machine instructions to begin, end, and abort transactions. A programmer uses the begin and end instructions to define transactions and uses the abort instruction to force the rollback of the transactions. When a transaction aborts, for example due to a memory access conflict, the execution is rolled back to immediately after the beginning of the transaction with a condition code set (zEC12 and POWER8) or jumps directly to an abort handler (Intel Core). These processors do not guarantee that a transaction will eventually succeed (*best-effort* HTM). Thus, a typical abort handler will determine whether the execution retries the transaction or reverts to a fallback mechanism such as locking.

In Blue Gene/Q, programmers use compiler-provided pragmas and C-language block constructs to specify the transactions. Blue Gene/Q does not disclose the underlying control mechanisms for the HTM. Programmers cannot write their own abort handling logic, but they can tune the system-provided code with environmental variables.

All four of the processors we experimented with implement the HTM facilities on top of their cache mechanisms. They keep track of the memory loads and stores during transactions in the caches or cache-like structures and detect conflicts using the cache coherence protocols. If two concurrent transactions access the same memory location and if at least one of the accesses is a store, then these transactions cause a conflict, and one of the transactions is aborted. The HTM implementations also buffer the stores during a transaction in the caches or buffers next to the caches, so that the stores are invisible from the other concurrent transactions. This buffering supports the rollbacks of transactions when a transaction aborts.

Although the four processors share implementation approaches, there are three major differences: (1) *conflict-detection granularity*, (2) *transaction capacity*, and (3) *abort-reason codes*. **Table 1** is a summary of the differences. There are multiple versions of the 4th generation Intel Core processor, but **Table** 1 shows only the Core i7-4770 that we tested.

Conflict-detection granularity. The zEC12, Intel Core, and POWER8 use their cache-line size as the conflict-detection granularity. Due to the cache-line granularity, a conflict is detected even when distinct bytes in a cache line are accessed concurrently by distinct transactions. We call this conflict a *false conflict*. The conflict-detection granularity of Blue Gene/Q is the L2 cache-line size (128 bytes) in the worst case. It can be reduced to 8 or 64 bytes based on certain conditions, such as the running mode described in Section 2.1.

As shown in **Table 1**, zEC12 has the largest conflict-detection granularity. A larger cache line helps exploit more spatial locality by allowing more variables to exist in the cache line, but from the viewpoint of HTM, it can increase the false conflicts.

Transaction capacity. The transaction capacity is the maximum amount of memory data that can be accessed in a

transaction. It is limited by the amount of the hardware resources needed to keep track of memory accesses for conflict detection and to buffer transactional stores. When a transaction tries to access a cache line that will exceed the capacity, it is aborted. We call this a *capacity-overflow abort*. In general, the load capacity is larger than the store capacity because the conflict detection has to record only the accessed memory addresses, while the store buffering needs to keep the stored data.

The second and third rows of Table 1 summarize the transaction capacities of the four processors. The load capacities are 1 MB or larger in these processors except for POWER8.

Note that there are two other factors that can cause the capacity-overflow aborts: cache-way conflicts and resource sharing among *simultaneous-multithreading (SMT)* threads. When a cache line accessed by a transaction is evicted from a cache because of a cache-way conflict, a capacity-overflow abort occurs even if the total amount of the transactional data does not exceed the capacity. Blue Gene/Q, Intel Core, and POWER8 support SMT, which allows multiple threads to run concurrently in a core. Since those SMT threads share the hardware resources for conflict detection and store buffering in each core, a transaction can encounter a capacity-overflow abort before it uses up the transaction capacity of the core.

Abort-reason code. The abort-reason code tells the users why a transaction aborted, such as a conflict or a capacity overflow. Each system has its own method to pass the code to the users. For example, Intel Core uses the EAX register. The abort-reason code is helpful not only for debugging but also as a hint about whether to retry the transaction.

As shown in **Table 1**, these four processors differ in the granularity of their abort reasons. For example, POWER8 reports a conflict due to a transactional access as different from one due to a non-transactional access, but the zEC12 and Intel Core do not distinguish between them. In addition to the codes for the specific abort reasons, zEC12, Intel Core, and POWER8 have a code that reports the processors' own decision about whether each transaction abort is persistent or transient.

In the next sections, we describe in detail the HTM implementation of each processor.

2.1 Blue Gene/Q

Blue Gene/Q uses the L2 cache for conflict detection and store buffering [1]. It assigns a unique speculation ID to each transaction and records the transactional accesses in the L2 directory with the speculation ID. The number of the speculation IDs is limited to 128. Although the IDs are periodically reclaimed for their reuse, the start of a new transaction is blocked if there is no available speculation ID.

Blue Gene/Q buffers the transactional stored data in its L2 cache. When a transactional store to a cache line occurs, the processor allocates a new way, which is different from the way that is storing the original cache-line data. Since six ways among the sixteen ways are reserved to store non-transactional data, the total transaction capacity combined for loads and stores is 20 MB (= 32 MB * 10/16).

Blue Gene/Q has two transactional execution modes: a short-running mode and a long-running mode. In the short-running mode, only the L2 cache buffers the transactional data, and thus every load of the transactional data requires access to the L2 cache. The long-running mode allows the L1 cache to buffer some of the transactional data though it invalidates all of the L1 cache lines at the start of each transaction.

2.2 zEC12

The zEC12 uses the L1 cache for conflict detection [6]. Each cache line has *tx-read* and *tx-dirty* bits, which are set by a transactional load and store, respectively. The zEC12 expands the transaction capacity for loads to 1 MB over the size of the L1 cache by recording the evicted cache lines in a special LRU-extension vector. The transactional stores are buffered in an 8-KB *gathering store cache*, which is private for each processor and is located between the L1 cache and the L2/L3 caches.

The zEC12 provides *constrained transactions* which are guaranteed to eventually commit. Our experiments on the STAMP benchmarks use normal transactions because the constrained transactions restrict the number of instructions to 32 and the transaction capacity to 256 bytes. We evaluated the performance of the constrained transactions by using the `ConcurrentLinkedQueue` data structure in the

Table 1. HTM implementations of Blue Gene/Q, zEC12, Intel Core i7-4770, and POWER8

Processor type	Blue Gene/Q	zEC12	Intel Core i7-4770	POWER8
Conflict-detection guranularity	8 - 128 bytes	256 bytes	64 bytes	128 bytes
Transactional-load capacity	20 MB (1.25 MB per core)	1 MB	4 MB	8 KB
Transactional-store capacity	20 MB (1.25 MB per core)	8 KB	22 KB	8 KB
L1 data cache	16 KB, 8-way	96 KB, 6-way	32 KB, 8-way	64 KB
L2 data cache	32 MB, 16-way, (shared by 16 cores)	1 MB, 8-way	256 KB	512 KB, 8-way
SMT level	4	None	2	8
Kinds of abort reasons	-	14	6	11

Java concurrent package because the enqueuing and dequeuing operations satisfy the restriction.

2.3 Intel Core

Intel Core uses the L1 cache for conflict detection and store buffering [28]. The details about the conflict detection and transaction capacities have not been disclosed. We measured the capacities by using a single-thread microbenchmark to execute transactions many times, gradually increasing the transactional loads or stores, and then measured the frequency changes in the capacity-overflow aborts. As a result of this experimentation, we concluded that the load and store capacities are 4 MB and 22 KB, respectively, on Core i7-4770. It has a larger transaction capacity for loads than the size of the L1 cache because it uses other resources to track the cache lines that were evicted from the L1 cache. The transaction capacity for the stores is within the size of the L1 cache.

Intel Core provides two programming interfaces to use HTM. *Hardware Lock Elision (HLE)* is an instruction-prefix-based interface to support the compatibility with processors that have no HTM. *Restricted Transactional Memory (RTM)* is a new instruction-set interface. Our experiments on the STAMP benchmarks used RTM, but we also evaluated HLE.

2.4 POWER8

POWER8 uses content addressable memory (CAM) linked with the L2 cache for conflict detection [10]. This CAM is called the *L2 TMCAM*. The L2 TMCAM records the cache-line addresses that are accessed in the transactions with bits to represent read and write. Although the transactional stored data is buffered in the L2 cache, the transaction capacity is bounded by the size of the L2 TMCAM. Since the number of the entries for the L2 TMCAM is 64, the total transaction capacity combined for loads and stores is 8 KB (=64*128 bytes).

POWER8 has *rollback-only transactions* which only support store buffering without the detection of data conflicts. The rollback-only transactions are useful for the single-thread speculative optimizations that do not need the detection of data conflicts [14, 24]. POWER8 also has instructions to suspend and resume transactions. A typical use case of these instructions in the user space is to output debug information. Another use case is Thread-Level Speculation (TLS) [18]. As described in [29], TLS can take advantage of HTM for conflict detection, but it also requires that transactions commit in the same order as the original sequential execution. Software implementation of order transactions involves accesses to a shared variable to control the commit order, but these accesses cause data conflicts among the transactions. The suspend/resume instructions can be used to escape from a transaction and to access the shared variable without data conflicts.

3. Transaction-Retry Mechanism

When a transaction aborts, the program can simply retry the transaction. However, since none of the HTM systems we evaluated guarantee that a transaction eventually commits (unless the transaction is a constrained transaction on the zEC12), these aborts can repeat indefinitely. Therefore a software fallback mechanism is required to guarantee forward progress. When TM is used to execute critical sections, as in the STAMP benchmarks, the standard fallback mechanism is to use a global lock to make transactions irrevocable.

Figure 1 shows the pseudocode for our transaction retry mechanism as used with zEC12, Intel Core, and POWER8. As described in Section 2, Blue Gene/Q can only use the system-provided retry mechanism, which we will describe later in this section. Each thread waits for the global lock to be released if it was acquired before the transaction began (Line 9) to avoid the lemming effect [8]. We implemented the global lock with a single memory word and spin waiting. Each transaction begins at Line 10, and when the transaction aborts, this pseudocode assumes that the program execution returns to immediately after the instruction that began the transaction (Line 11). After a transaction begins, the global lock is first checked, so that the HTM system can keep track of the lock word and abort the transaction if another thread acquires the global lock. If the global lock has

```
1:  lockRetryCount = MAX_LOCK_RETRY_COUNT;
2:  persistentRetryCount =
3:    MAX_PERSISTENT_RETRY_COUNT;
4:  transientRetryCount =
5:    MAX_TRANSIENT_RETRY_COUNT;
6:
7:  retry: // Label for retrying transactions
8:
9:  waitForLockToBeReleased();
10: tbegin(); // Begin a transaction
11: if (isTransactionAborted()) {
12:   // Return here on a transaction abort
13:   if (isLockAcquired()) {
14:     // Aborted due to a conflict on
15:     // the lock word
16:     if (--lockRetryCount > 0) goto retry;
17:   } else if (isAbortPersistent()) {
18:     // The abort code is persistent.
19:     if (--persistentRetryCount > 0)
20:       goto retry;
21:   } else if (--transientRetryCount > 0) {
22:     // The abort code is transient.
23:     goto retry;
24   }
25:   acquireLock();
26: } else if (isLockAcquired())
27:   tabort();
28: // Transaction body
29: if (isLockAcquired()) releaseLock();
30: else tend(); // End a transaction
```

Figure 1. Pseudocode for our transaction-retry mechanism

already been acquired, then this new transaction must abort (Line 27), because otherwise it could read inconsistent data.

When a transaction aborts, the thread determines whether or not it continues with transactional execution (Lines 11-25). There are three thread-local counters to control the number of retries before reverting to the global lock: (1) *lock-retry counter*, (2) *persistent-retry counter*, and (3) *transient-retry counter*.

The lock-retry counter controls the number of retries for those transaction aborts that are caused by conflicts on the global lock. We call these aborts *lock-conflict aborts*, which can be recognized by checking the global lock after the transactions aborted (Line 13). The persistent-retry counter controls the number of retries for the persistent transaction aborts (Line 17). On Intel Core and POWER8, we can recognize the persistent aborts based on the abort-reason code that reports the processors' own decisions about the persistence of the aborts. On zEC12, we treat capacity-overflow aborts as persistent. The transient-retry counter controls the number of retries for the other aborts. The maximum values for the three counters (Lines 1-5) are tuning parameters.

We separated the lock-retry counter from the transient-retry counter, because the conflicts for the global lock have different characteristics from the conflicts over normal shared data. When a thread acquires the global lock for the irrevocable execution of a transaction, all of the other concurrent transactions will be aborted. In contrast, a conflict over program data will abort only some of the transactions that happen to access the same data at the same time. Therefore, the best retry counts can be different for the lock-conflict aborts and the other transient aborts.

With the persistent-retry counter, we can allow transactions to retry even in the case of persistent aborts. Most of the persistent aborts are capacity-overflow aborts, but there are some cases where capacity-overflow aborts are not actually persistent. As described in Section 2, capacity-overflow aborts can be caused by cache-way conflicts or resource sharing among multiple SMT threads, so they may be transient.

An existing transaction retry mechanism [23] is the base of our retry mechanism. It uses a single retry counter and changes the degree of the decrement for the retry counter based on the abort reasons. For example, the retry count is halved in the case of a persistent transaction abort. Instead of changing the degree of the decrement, our mechanism uses different retry counters per abort reason.

In Blue Gene/Q, the system software offers a retry mechanism. It does not distinguish among lock-conflict, transient, or persistent aborts. It uses a single retry counter, and the users can specify the maximum number of retries with an environmental variable. The Blue Gene/Q also has an adaptation mechanism, with which transactions that too frequently fell back on the global lock will not be allowed to retry on the next abort. Finally, our retry mechanism checks the global lock at the beginning of a transaction (Line 13), while Blue Gene/Q's mechanism checks at the end when run in long-running mode, which is called lazy subscription [12].

4. STAMP Benchmarks

In this section, we describe the STAMP benchmarks and the modifications we made to prevent the nonessential transaction aborts. STAMP is the most widely used TM benchmark suite [3]. It consists of eight programs using both fine-grain and coarse-grain transactions. We used Version 0.9.10 of the STAMP benchmarks and their default runtime options for a non-simulator as specified in the README files.

In preliminary comparisons of the STAMP benchmarks on the four processors, we found that four benchmarks were not well programmed or tuned for best-effort HTM. This unfriendly code caused excessive nonessential transaction aborts, and thus we could not fairly compare the intrinsic performances of the HTM systems. The nonessential transaction aborts were caused by false conflicts and capacity overflows, which rarely occur in typical STM implementations. This explains why these problems have not been fixed since the STAMP benchmarks were released. We fixed these problems and used the modified versions for most of our evaluations, but we also show the performance comparison of the versions with and without our changes. Our fix is available in a Web site [32] as a code patch to Version 0.9.10 of the STAMP benchmarks. In the rest of this section, we describe each change in detail.

genome. In one transaction section of the genome benchmark, the genome segments are inserted into a hash table. There is a compile-time tuning parameter for how many genome segments to insert in each transaction. A larger number means a longer transaction and thus can alleviate the performance overhead to begin and end the transactions, but this also increases the probability of capacity-overflow aborts. We tuned this compile-time parameter (CHUNK_STEP_1) for each of the four platforms (9 for Blue Gene/Q and 2 for the other three processors) to achieve the best performance.

intruder. In the intruder benchmark, red-black trees are used for unordered sets, and linked lists for ordered sets. These data structures are not suitable for implementing their respective sets. On HTM, these unsuitable data structures result in excessive capacity-overflow aborts. We modified the code to use hash tables for the unordered sets and red-black trees for the ordered sets. Our hash table is similar to the concurrent hash table in the Java standard class library.

kmeans. In the kmeans benchmark, each transaction accesses and modifies one cluster, which consists of an inte-

ger number and floating-point numbers. The original code tries to avoid false conflicts by collocating a cluster in a contiguous memory region and by inserting padding between the clusters. However, because each cluster is not aligned to a cache line boundary, two clusters can coexist within a cache line, causing false conflicts. We modified the code to properly align the clusters to cache line boundaries.

vacation. In the vacation benchmark, red-black trees are used for unordered sets. As in intruder, we instead used hash tables for the unordered sets.

5. Performance Comparison of HTM Systems

We ran the STAMP benchmarks [3] on the following four platforms.

- 16-core 1.6-GHz A2 with 4 SMT threads (Blue Gene/Q), V1R2M2, 16 GB of main memory
- 16-core 5.5-GHz zEC12, z/OS V2.01, 64 GB of main memory
- 4-core 3.4-GHz Core i7-4770 with 2 SMT threads, Linux 3.14.5, 4 GB of main memory
- 6-core 4.1-GHz POWER8 with 8 SMT threads, AIX 7.1.3.16, 28.5 GB of main memory

The different numbers of cores indicate that the performance comparisons among the processors are fair only up to four concurrent application threads, because up to four threads can be assigned dedicated cores on each of the platforms. A processor cannot provide the maximum performance for each thread when the number of concurrent threads is larger than the number of its cores because multiple SMT threads share the hardware resources for the HTM, as described in Section 2. Our results for POWER8 are preliminary because we used a pre-release version of the processor.

Our main performance metric is the speed-up ratios of transactional execution over sequential execution and the transaction-abort ratios. Our baseline to calculate the speed-up ratios on each processor is the sequential non-HTM execution on that processor, because our purpose is to evaluate how each HTM system accelerates the multi-threaded performance. Comparing absolute performance is beyond the scope of our paper. A transaction-abort ratio is the percentage of the aborted transactions to all of the transactions without irrevocable transactions.

For the zEC12, Intel Core, and POWER8, we break down the cause of the transaction aborts into four categories: (1) capacity overflow, (2) data conflict, (3) other, and (4) lock conflict. The first three types of aborts are identified by checking the abort-reason code. Lock-conflict aborts are detected by the transaction-retry mechanism described in Section 3. Note that we can miss a lock-conflict abort if the global lock is released before the global lock is checked in the transaction-retry mechanism. In that case, the abort is categorized as a data conflict.

5.1 4-Thread Performance of Modified STAMP Benchmarks

In this section, we compare the performance of the HTM systems when we ran four threads for our modified version of the STAMP benchmarks. Since each system has at least four physical cores, each thread can have the exclusive use of a physical core. For the three retry counters explained in Section 3, we optimized the parameter values for each test case which consists of an HTM system and a benchmark. We also tuned the maximum retry count and the running mode for each benchmark on Blue Gene/Q. This approach allowed us to compare the best performance of each HTM system for each benchmark. We ran each benchmark four times and took the average of the runs.

Figure 2 shows the speed-up ratios of the transactional execution over the serial non-HTM execution and **Figure 3** shows the transaction-abort ratios. For the speed-up ratios, the error bars show the 95%-confidence intervals. Each abort ratio is divided into four categories, except for Blue Gene/Q. In each figure, the bayes benchmark was excluded from the calculation of the average numbers because of its non-deterministic behavior, which significantly affects the performance numbers [37]. We also dropped bayes from the later analyses presented here.

Only Blue Gene/Q showed scalable performance for yada because it has a larger transaction capacity than the other processors. In **Figure 3**, capacity-overflow aborts do not seem to be a major cause of aborts on zEC12, Intel Core, or POWER8, but that is because the maximum persistent-retry count was set to 1. Since capacity-overflow aborts occur persistently in the transaction section of yada, reducing the maximum persistent-retry count improves the performance. As a result, the number of capacity-overflow aborts seems small, but many of the transactions reverted to the global lock due to the capacity-overflow aborts. The serialization ratio, which is the percentage of the irrevocable transactions to all of the committed transactions, was about 10% on Blue Gene/Q, but about 20% on the other systems.

Blue Gene/Q has lower speed-up ratios than the other processors in these benchmarks, except for genome and yada, because of its relatively high single-thread overhead. For example, compared to perfectly linear speedup, Blue Gene/Q degraded the single-thread performance of kmeans-high by 40% while the performance degradation in the other processors was limited to 10%. Since Blue Gene/Q requires software-based register checkpointing, system calls to begin and end transactions, and L1-cache invalidation or bypass, it has higher single-thread overhead than the other processors. In addition, Blue Gene/Q frequently exhausted its

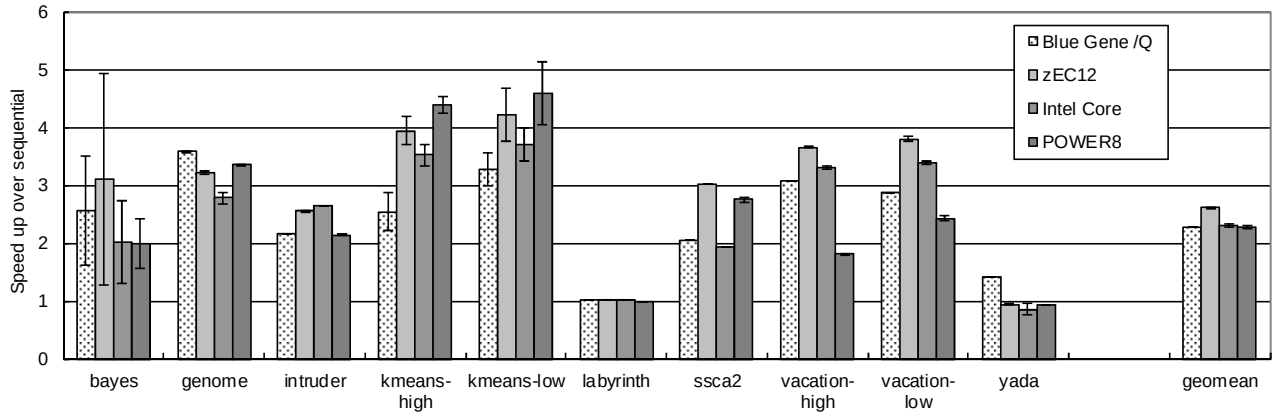

Figure 2. Speed-up ratios of transactional execution over serial execution in Blue Gene/Q, zEC12, Intel Core, and POWER8 with 4 threads. Our modified STAMP benchmarks were used.

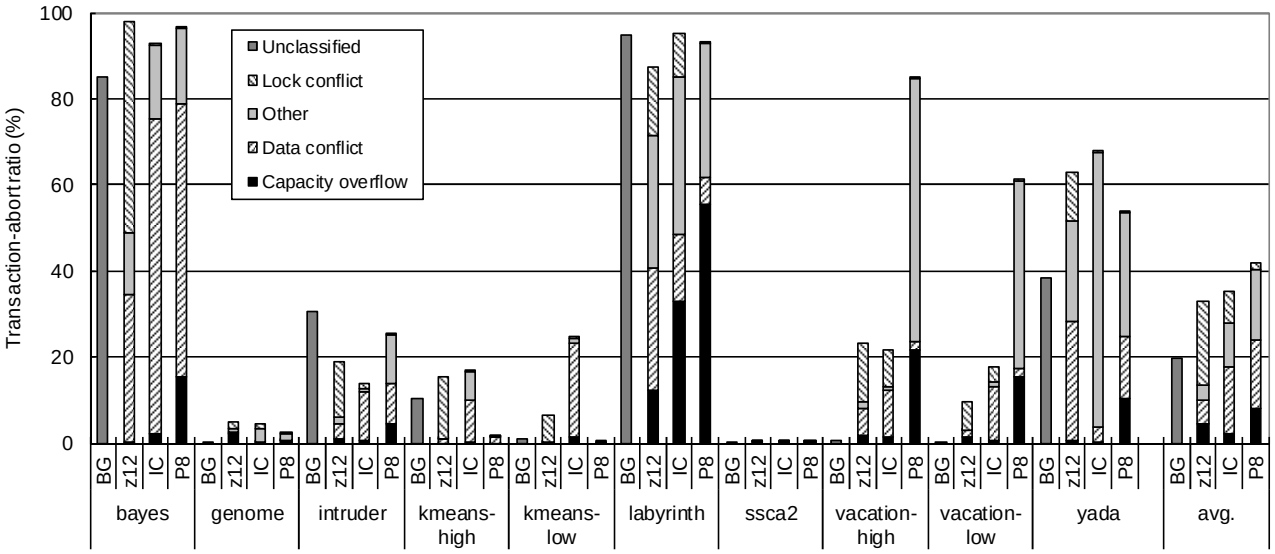

Figure 3. Transaction-abort ratios in Blue Gene/Q (BG), zEC12 (z12), Intel Core (IC), and POWER8 (P8) with 4 threads. Our modified STAMP benchmarks were used.

supply of speculation IDs in ssca2, because this benchmark executes many short transactions. Therefore, the start of a new transaction was often blocked until a speculation ID became available.

The zEC12 has the highest average speed-up ratio. Its distinctive characteristic is that most of the transaction aborts belong to the "other" category as shown in the grey bars of **Figure 3**. More specifically, the abort-reason code of the "other" aborts was *cache-fetch-related aborts*. Unfortunately, the meaning of this abort-reason code is not fully disclosed, even in the most detailed zEC12 documentation [40]. These aborts are transient and are caused by an implementation restriction when cache line fetches and stores happen in the transactions.

The zEC12 shows superlinear speedup in kmeans-low. This is caused by the non-deterministic threaded execution in kmeans as indicated by the larger error bars compared to the other benchmarks in **Figure 2**. POWER8 shows superlinear speedup in kmeans-high and -low for the same reason.

In kmeans-low, Intel Core has more data-conflict aborts than the other processors. We found that the hardware prefetching of Intel Core increased the data conflicts because the memory addresses of the clusters, which are the main shared data in kmeans, are successive. When a memory address is accessed, the hardware prefetching preloads the data at the adjacent address into the processor cache. In kmeans, each transaction updates a single cluster, which occupies cache lines because of the padding and alignment described in Section 4. During this update, some of the neighboring clusters can be prefetched into other cache lines. If the HTM system regards the prefetched cache line as transactional data, then it can detect a data conflict when another transaction concurrently updates one of the neighboring clusters. Note that this conflict is not necessary for the transaction because the transaction never accesses the neighboring cluster.

To assess the influence of hardware prefetching on data conflicts, we ran a second experiment by disabling the

hardware prefetching. Disabling the hardware prefetching reduced the abort ratios of kmeans-high and –low from 16% and 24 to 10% and 10%, respectively, and improves the speed-up ratios from 3.5 and 3.7 to 3.9 and 4.0, respectively. These results show that the hardware prefetching of Intel Core causes unnecessary data conflicts on the prefetched cache lines. Developers in Intel also validated our findings. Intel Core should be enhanced to avoid such false detection of data conflicts. Note that disabling the hardware prefetching is not a realistic solution because it can degrade the performance of other applications. Actually, we have seen performance degradation for the sequential execution of kmeans-high and –low.

Intel Core scaled worse than zEC12 and POWER8 in ssca2. This was not due to the HTM system, as indicated by the only 1% transaction-abort ratio on Intel Core. The inner-most loop of ssca2 causes many last-level cache misses, and the desktop Intel Core machine we used (Lenovo ThinkCentre M93p) had poorer performance of concurrent memory accesses than the other 3 systems.

POWER8 was worse than zEC12 and Intel Core in intruder and vacation because it has more capacity-overflow aborts than the other processors. Increasing the transaction capacity is an obvious approach to enhance the POWER8 HTM system.

5.2 4-Thread Performance of Original and Modified STAMP Benchmarks

In this section, we compare the performance of the original and modified versions of the STAMP benchmarks with four threads. We measured the maximum performance for each test case consisting of an HTM system and a benchmark by tuning the values for the maximum retry counts in the same manner as for the experiments in Section 5.1. **Figure 4** shows the speed-up ratios. This figure includes only the data for the modified benchmarks: genome, intruder, kmeans, and vacation. The geometric means are for all of the programs in the STAMP benchmarks.

Our modified version of genome shows 3.7 times performance gain for POWER8 over the original version. This large performance gain is due to the reduction in capacity-overflow aborts though we omitted the graph for the abort ratios because of the limited space for this paper. Our modifications also increased the performance more than 1.4 times for POWER8 in the intruder and vacation benchmarks due to the significant reduction in capacity-overflow aborts. However, POWER8 still suffered from more capacity-overflow aborts than the other processors.

In kmeans-high, our changes improved the performance of zEC12 and Intel Core by 20% and 28%, respectively. The major source of the performance improvement is the reduction in false conflicts that were caused by the misalignments of the shared data, as described in Section 4.

We saw some cases (intruder, kmeans-high, and vacation in Blue Gene/Q, and intruder in Intel Core) where our changes degraded the speed-up ratios even though they improved the absolute performance. This was caused by the significant improvement in the single-thread performance compared to the improvement in the multi-thread performance.

5.3 Scalability of Modified STAMP Benchmarks

Finally, we studied the scalabilities of the four HTM systems when we ran 1, 2, 4, 8, and 16 threads for our modified version. The values for the three maximum retry counts were optimized for each number of threads for each HTM system running the benchmarks. The mode and maximum retry count were also tuned on Blue Gene/Q in the same manner. Note that Intel Core and POWER8 cannot exclusively use a physical core for each thread with eight and sixteen threads. Therefore, a fair comparison of the eight- and sixteen-thread performance is possible only for Blue Gene/Q and zEC12. **Figure 5** shows the speed-up ratios. For Intel Core, we do not plot the sixteen-thread performance because sixteen is larger than the total number of possible SMT threads.

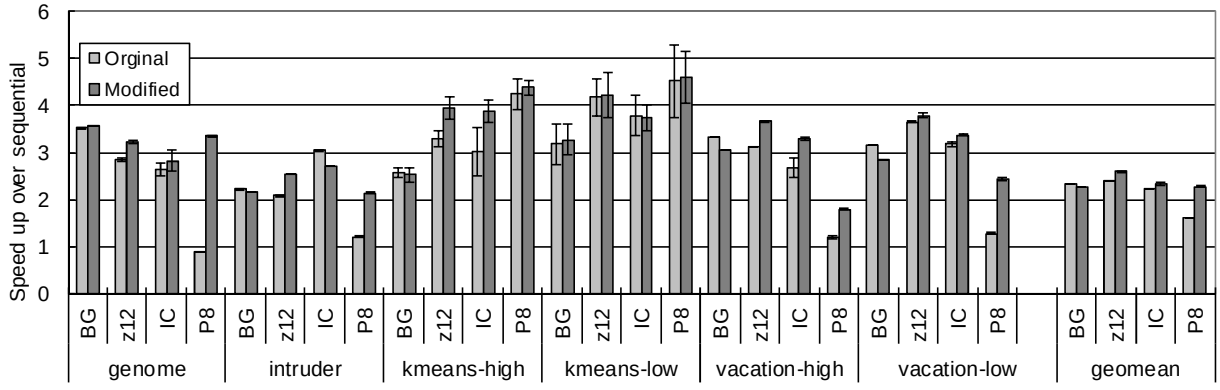

Figure 4. Speed-up ratios of transactional execution over sequential execution in Blue Gene/Q (BG), zEC12 (z12), Intel Core (IC), and POWER8 (P8) with 4 threads. The two bars are the speed-up ratios of the original and modified versions of the STAMP benchmarks, respectively.

In yada, Blue Gene/Q had a higher speed-up ratio than zEC12 because it has a larger transaction capacity than zEC12. However, with sixteen threads, data-conflict aborts limited the scalability. In intruder, ssca2, and vacation, the zEC12 had higher speed-up ratios than Blue Gene/Q. In ssca2, the speculation ID reclamation was the bottleneck for Blue Gene/Q. For intruder and vacation, there are two reasons. One is that zEC12 has less single-thread overhead than Blue Gene/Q. The second reason is that zEC12 had lower serialization ratios than Blue Gene/Q. For example, in intruder, Blue Gene/Q had a 56% serialization ratio with sixteen threads while zEC12 had only a 2% serialization ratio. This indicates that the adaptation mechanism in Blue Gene/Q acted too early in making the transactions fall back on the global lock.

In these benchmarks (except for kmeans), the zEC12 had higher speed-up ratios than Intel Core and POWER8 with eight and sixteen threads because it can assign an exclusive physical core to each thread. Meanwhile, the zEC12 had lower speed-up ratios than POWER8 for kmeans because it had excessive cache-fetch-related aborts. Basically, POWER8 showed higher speed-up ratios than Intel Core with eight threads because it has more physical cores than Intel Core. However, Intel Core had higher speed-up ratios than POWER8 in intruder and vacation because the transaction capacity of POWER8 is too small for those benchmarks.

6. Evaluation of Processor-Specific Features

In this section, we show the performance results of the three features specific to each HTM system: the constrained

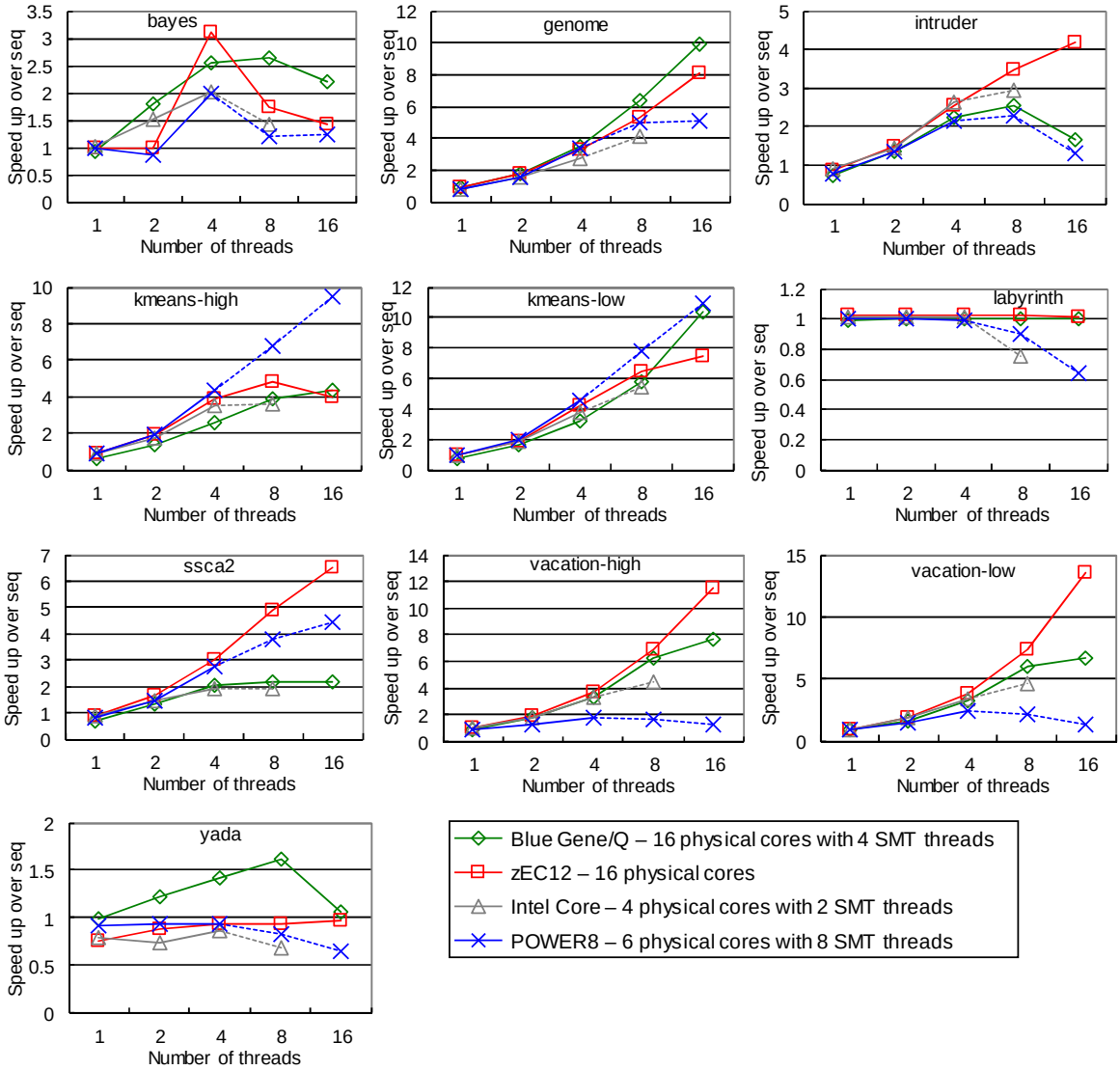

Figure 5. Speed-up ratios of transactional execution over serial execution in Blue Gene/Q, zEC12, Intel Core, and POWER8. Our modified version of the STAMP benchmarks was used with 1, 2, 4, 8, and 16 threads. Dotted lines are the speed-up ratios when the number of physical cores is smaller than the number of threads.

transactions in zEC12, the hardware lock elision in Intel Core, and the suspend/resume instructions in POWER8. Our purpose is to investigate whether these features are worth implementing in the future systems, from the performance perspective.

6.1 Performance of constrained transactions on zEC12

In this section, we compare the performance of the constrained and normal transactions on the zEC12 processor. As described in Section 2.2, constrained transactions are guaranteed to eventually commit, and thus they do not need abort handlers. We applied the constrained transactions to the enqueuing and dequeuing operations for the `ConcurrentLinkedQueue` data structure in the standard Java concurrent package because these operations satisfy the restriction for the constrained transactions. The enqueuing operation in a transaction adds a new element to the last element (tail) if the next pointer of the last element is null. Otherwise, it falls back to the original lock-free code which uses atomic operations. The dequeuing operation returns an object stored in the first element (head) if the object is not null. Otherwise, it falls back to the original code. We used a single retry counter for normal transactions because the data accessed in a transaction are within 256 bytes and thus capacity-overflow aborts will never occur. We do not need the lock-retry counter because we do not falls back to the lock. We tuned the retry count to obtain the maximum performance.

Figure 6 shows the relative execution times when each thread alternately enqueues to and dequeues from a single queue. The base line is the execution time of the original lock-free implementation of the `ConcurrentLinkedQueue` data structure with each number of threads. NoRetryTM used normal transactions without any retry, while OptRetryTM did with optimal retry counts. As shown in this figure, using transactions reduced the execution time when the number of threads was less than four. These speed-ups resulted from the reduction in the path length because the complicated lock-free operations were simplified by using transactions. When the number of threads was larger than two, NoRetryTM increased the execution time. Constrained transactions (ConstrainedTM) were comparable to OptRetryTM. We conclude that the constrained transactions do not provide performance benefits but they eliminate the work to implement the fallback paths and to tune the retry count.

Our conclusion is different from the conclusion of Jacobi et al. [6] who mention a performance benefit in constrained transactions compared to normal transactions for a highly contended microbenchmark. This is because we used a more realistic benchmark, which has lower contention, than their highly contended microbenchmark. Jacobi et al. themselves mentions that the high contention case is not common in real-world applications.

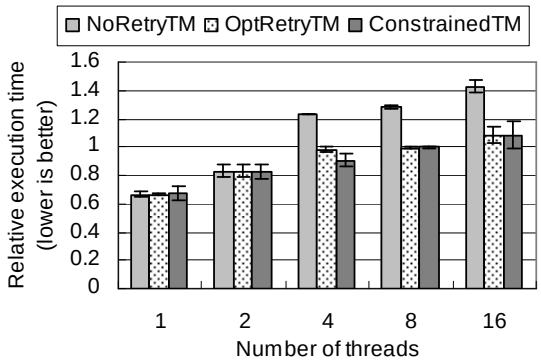

Figure 6. Relative execution times for normal and constrained transactions. The baseline is the execution time for the original lock-free implementation of the Java ConcurrentLinkedQueue data structure.

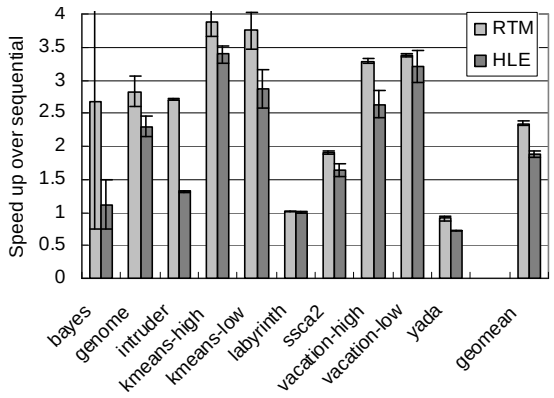

Figure 7. Speed-up ratios of RTM and HLE executions over serial execution in Intel Core. Our modified version of the STAMP benchmarks was used with 4 threads.

6.2 Performance of hardware lock elision (HLE) on Intel Core

In this section, we compare the performance of RTM and HLE on the Intel Core processor. **Figure 7** shows the speed-up ratios of the RTM and HLE execution over the serial execution with four threads in the STAMP benchmarks. The numbers of the RTM execution are same as the ones in **Figure 2** (i.e. the retry counts are tuned). On average, the speed-up ratio of the HLE execution reached 80% of that of the RTM execution. The performance gap remained, because the retry counts cannot be tuned for the HLE, which has no software-based retry mechanism. These results indicate that the HLE will provide modest speed-ups in many existing programs with little modification and no tuning effort.

6.3 Performance of thread-level speculation (TLS) on POWER8

As described in Section 2.4, the suspend and resume instructions in POWER8 can be exploited to implement efficient TLS. Following the experimental scheme in a previous study [29], we manually modified two benchmarks (433.milc and 482.sphinx3) in SPEC CPU2006 that can benefit from TLS. We applied TLS to frequently executed loops in each benchmark. **Figure 8**(a) shows an example loop and (b) is the loop after TLS is applied. Each thread executes this modified loop. Either the code in the dark gray or in the light gray is executed, depending on the availability of the suspend and resume instructions. Without these instructions, if a previous iteration has not yet finished, the current thread must abort (dark gray code). The suspend and resume instructions allow spin-waiting outside the transaction, without causing data conflicts (light gray code).

Figure 9 shows the throughput results normalized to the sequential execution of each benchmark. TLS provided speed-ups of up to 15% in 433.milc and 25% in 482.sphinx3 with 6 threads. In 482.sphinx3, TLS with the suspend and resume instructions was faster than TLS without them by 12%. By using these instructions, the abort ratio was reduced from 69% to 0.1%. In 433.milc, the improvement by using the suspend and resume instructions was only 2%. These instructions reduced the abort ratio from 83% to 10%, but false conflicts still remained. Overall, the suspend and resume instructions are effective extensions to HTM to implement efficient TLS.

(a)

```
for (i = 0; i < N; i++) {
  // Loop body
}
```

(b)

```
for (i=0; i < N; i += NumThreads)) {
retry:
  if (NextIterToCommit != i) {
    tbegin();
    if (isTransactionAborted()) goto retry;
  }
  // Loop body
  if (NextIterToCommit != i) tabort();
  suspend();
  while (NextIterToCommit != i) ;
  resume();
  if (isInTM()) tend();
  NextIterToCommit = i + 1;
}
```

Figure 8. Example loop for TLS with suspend/resume instructions of POWER8

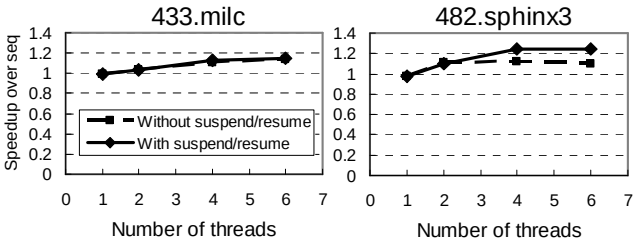

Figure 9. Speed-up ratios of TLS with and without the suspend/resume instructions on POWER8. Two SPEC CPU2006 benchmarks were measured.

7. Next Generation of HTM Systems

In this section, we discuss the next generation of HTM systems based on the experimental results shown in Section 5 and 6. The following are our recommendations for next generation HTM designs.

Precise Conflict Detection. The underlying implementation of the cache-coherency traffic should not compromise the conflict-detection mechanism of an HTM system. Intel Core has unnecessary transaction aborts because of the conflicts on the prefetched cache lines. The zEC12 detects false transaction-abort conditions (cache-fetch-related aborts) on the cache lines where in reality data conflicts do not occur. Although the precise conflict detection may complicate the hardware design, it is needed to maximize the performance.

Better Interaction with SMT. HTM should scale beyond the number of physical cores, using SMT. As shown in **Figure 5**, POWER8 and Intel Core cannot scale with SMT in the benchmarks of relatively large transactional sizes (the benchmarks other than kmeans and ssca2). One way to avoid capacity-overflow aborts due to the resource sharing among the SMT threads is to somehow restrict concurrent execution of transactions on the same core.

Non-transactional loads and stores. Non-transactional loads and stores enable conflict-detection-free memory accesses in transactions, not only for debugging but also for performance. For example, as shown in Section 6.3, the non-transactional load allows the efficient implementation of ordered transactions for TLS. Although such non-transactional loads and stores are not enough to achieve high TLS performance as mentioned in [29], they are basic and essential functions to build TLS on top of HTM.

Larger Transactional-Store Capacity. Next generation of HTM systems should have larger capacity, especially for transactional stores. **Figure 10** and **Figure 11** plot the relationship between the transaction sizes and transaction-abort ratios for loads and stores, respectively. Using a trace tool [2], we collected the data addresses accessed in transactions by running the STAMP benchmarks with the simulator runtime options on our Intel Core machine. We then calculated the transaction sizes by mapping the collected addresses to the cache lines of each processor. We show the 90-percentile transaction sizes. Even with the simulator runtime options, which specify smaller data sets than the non-simulator options, the transaction sizes of some benchmarks exceeds the supported transaction capacities,

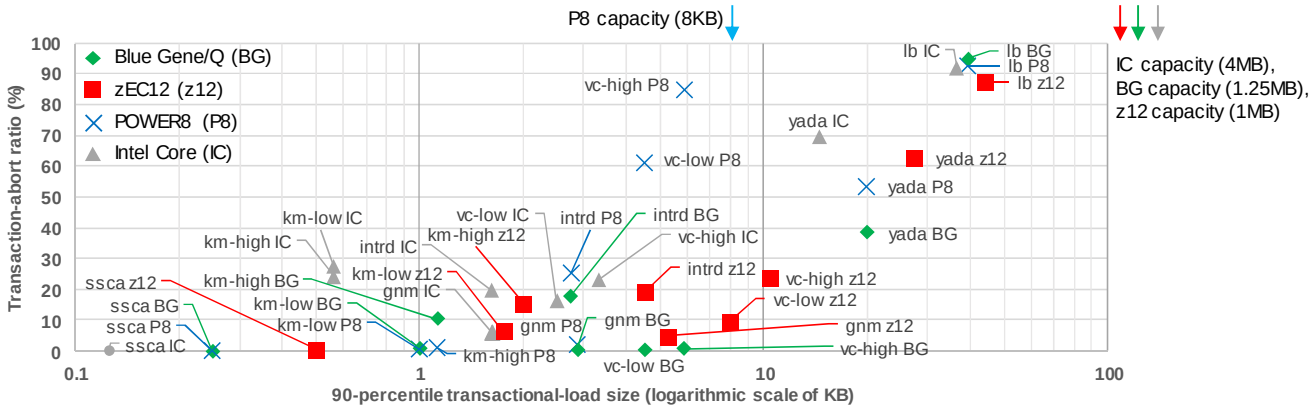

Figure 10. 90-percentile transactional-load sizes and transaction-abort ratios. Each plot corresponds to a pair of a benchmark and a processor. gnm: genome, intrd: intruder, km: kmeans, lb: labyrinth, ssca: ssca2, vc: vacation

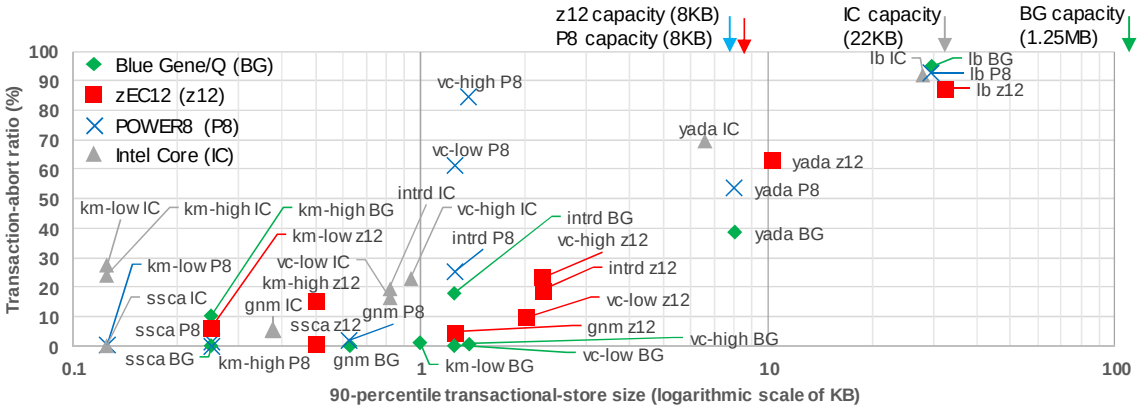

Figure 11. 90-percentile transactional-store sizes and transaction-abort ratios. Each plot corresponds to a pair of a benchmark and a processor. gnm: genome, intrd: intruder, km: kmeans, lb: labyrinth, ssca: ssca2, vc: vacation

especially for transactional stores. Discussing the tradeoff between the performance and the hardware resources is beyond the scope of this paper, but hardware should make its best efforts because it is not easy for software to avoid transaction capacity overflows.

8. Related Work

HTM has been studied for use in commercial processors [11, 16, 34] since the idea appeared in 1993 [19]. *Software transactional memory (STM)* [2, 15, 17, 25, 26] is an alternative to HTM. STM surpasses HTM in its portability and transaction capacity. It supports transactional execution in any processor and has practically no limits on its transaction capacity because it can use the all of memory for tracking the loads and stores. However, STM's high overhead is an obstacle to its use in real-world software.

Currently there are four processors with HTM: Blue Gene/Q [1], zEnterprise EC12 (zEC12) [6], the 4th generation of Intel Core [13], and POWER8 [27]. Although Azul [4] and Rock [9] processors implemented HTM before these four processors, their HTM implementations were not fully available because Azul's HTM programming interface was never published and the Rock processor was canceled before commercial production.

There is no prior research that directly compares the performance of actual HTM systems, though individual HTM systems have been evaluated with various applications [1, 5, 7, 21, 23, 30, 31, 35, 36, 38, 39]. Only Odaira et al. compared two HTM systems, zEC12 and Intel Core, using an HTM-enabled Ruby interpreter, but they did not evaluate the other platforms [30]. Our work is the first to compare all four of the HTM implementations, using a common benchmark set, the STAMP benchmarks [3]. Although the STAMP benchmarks have been independently measured on Blue Gene/Q, zEC12, and Intel Core by many researchers, it is important to compare the HTM systems using the same code base, because the transaction-retry mechanism described in Section 3 has a huge impact on the performance.

Wang el al. [1] provided a detailed performance analysis of a Blue Gene/Q HTM system with the STAMP benchmarks. Our speed-up ratio results mostly match theirs, although they tuned not only the modes (long-running or short-running) and maximum retry counts but also various other parameters. The yada benchmark showed a more than 2-fold speed-up with four threads in their results but only a 1.4-fold speed-up in ours. Their speed-up was because

function outlining by their compiler improved the single-thread performance of the HTM version of yada, but we did not observe this effect using our compiler.

Mitran et al. [20] evaluated three programs in the STAMP benchmarks on zEC12. Odaira et al. [31] analyzed all of the programs in the STAMP benchmarks on zEC12 to compare the performance of the C and Java versions of the STAMP benchmarks. In genome, intruder, and vacation, our speed-up ratios on zEC12 were better than the results by Odaira et al., even without our benchmark modifications, because we tuned the maximum transaction-retry counts.

Yoo et al. evaluated the benefits of an Intel Core HTM system on real HPC applications, but they also measured the STAMP benchmarks [28]. Diegues et al. also measured the STAMP benchmarks on Intel Core and found that the optimal transaction-retry count differs for each application [23]. Based on their findings, we used an optimal set of values for the transaction-retry counts for each HTM system with our benchmarks, allowing us to compare the maximum performance of each HTM system. The speed-up ratios obtained by Diegues et al. were similar to our results, although they used Xeon E3-1275, not Core i7-4770. Compared with their results, our unmodified version showed better speed-up ratios in kmeans-low (their 3 and our 3.8 with four threads) and worse ratios in genome (their 3.2 and our 2.7). Investigating these differences is our future work.

9. Conclusion

In this paper, we quantitatively compared the HTM systems that are implemented in four processors: Blue Gene/Q, zEC12, Intel Core, and POWER8. In order to find the maximum performance of each HTM system on the STAMP benchmarks, we modified the TM-unfriendly code of some benchmarks and tuned the transaction-retry counts to provide the most ideal comparisons possible. Our experimental results showed that there is no single HTM system that is more scalable than the others in all of the benchmarks. Each HTM system has its own implementation issues that limit the scalability. In Blue Gene/Q, the high single-thread overhead, which is caused by software-based register checkpointing and the system calls to control transactions, limits the performance. In addition, the cost of reclaiming the speculation IDs limits the scalability. The zEC12 suffers from mysterious transaction aborts that degrade its performance. Intel Core has extra transaction aborts due to the hardware prefetching. POWER8 has more capacity-overflow aborts than the other processors because of its overly small transaction capacity. Solving these implementation problems is important to improve the performance of concurrent applications to support the wider adaption of TM-based programming.

We also evaluated the features specific to each HTM system: the constrained transactions of zEC12, hardware lock elision (HLE) of Intel Core, and the suspend and resume instructions of POWER8. The constrained transactions, which do not need a software-based retry mechanism, had performance comparable to the normal transactions that require a software-based retry mechanism with tuned retry counts. The HLE showed modest speed-ups in most of the STAMP benchmarks without any tuning effort for the transaction-retry counts. The suspend and resume instructions were beneficial for avoiding data conflicts on a shared variable to implement ordered transactions, and thus improved the performance of TLS. Overall, these features specific to each HTM system will increase the adoption of and widen the application of HTM.

References

[1] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, 2012.

[2] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In Principles and Practice of Parallel Programming, Jan. 2006.

[3] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In International Symposium on Workload Characterization (IISWC), Sep. 2008.

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

[5] C. G. Ritson, T. Ugawa, and R. E. Jones. Exploring garbage collection with Haswell hardware transactional memory. Proceedings of the 2014 International Symposium on Memory Management, 2014.

[6] C. Jacobi, T. Slegel, and D. Greiner. Transactional Memory Architecture and Implementation for IBM System Z. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012.

[7] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit. StackTrack: an automated transactional approach to concurrent memory reclamation. Proceedings of the Ninth European Conference on Computer Systems, 2014.

[8] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaun. Applications of the Adaptive Transactional Memory Test Platform. In 3rd ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2008.

[9] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, 2009.

[10] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. Transactional memory support in the IBM POWER8 processor. IBM Journal of Research and Development, Vol. 59, 2015.

[11] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the power architecture. Proceedings of the 40th Annual International Symposium on Computer Architecture, 2013.

[12] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy. Improved Single Global Lock Fallback for Best-effort Hardware Transactional Memory. In 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2014.

[13] Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1. Chapter 15: Programming with Intel® Transactional Synchronization Extensions. June 2014.

[14] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges. Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, 2003.

[15] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In Principles and Practice of Parallel Programming, Jan. 2010.

[16] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004.

[17] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In ACM Symp. on Parallelism in Algorithms and Architectures (SPAA), June 2008.

[18] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. In Proceedings of the 19th Annual International Symposium on Computer Architecture, 1992.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Proceedings of the 1993 Annual International Symposium on Computer Architectures.

[20] M. Mitran and V. Vokhshori. Evaluating the zEC12 Transactional Execution Facility. IBM Systems Magazine, 2012.

[21] M. Schindewolf, B. Bihari, J. Gyllenhaal, M. Schulz, A. Wang, and W. Karl. What scientific applications can benefit from hardware transactional memory? Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012.

[22] macsim. Simulator for heterogeneous architecture. https://code.google.com/p/macsim/

[23] N. Diegues and P. Romano. Self-Tuning Intel Transactional Synchronization Extensions. Proceedings of the 1st International Conference on Autonomic Computing (ICAC), 2014.

[24] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware atomicity for reliable software speculation. Proceedings of the 34th Annual International Symposium on Computer Architecture, 2007.

[25] N. Shavit and D. Touitou. Software Transactional Memory. In Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, 1995.

[26] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. IEEE Transactions on Parallel and Distributed Systems, Dec. 2010.

[27] Power ISA™ Version 2.07, https://www.power.org/documentation/power-isa-version-2-07/, 2013.

[28] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013.

[29] R. Odaira and T. Nakaike. Thread-Level Speculation on Off-the-Shelf Hardware Transactional Memory. In Proceedings of the 2014 IEEE International Symposium on Workload Characterization, 2014.

[30] R. Odaira, J. G. Castanos, and H. Tomari. Eliminating global interpreter locks in Ruby through hardware transactional memory. Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2014.

[31] R. Odaira, J. G. Castanos, and T. Nakaike. Do C and Java programs scale differently on Hardware Transactional Memory? Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC), 2013.

[32] R. Odaira. Patching and Building HTM-enabled STAMP. http://researcher.watson.ibm.com/researcher/view_person_subpage.php?id=6045.

[33] S. Gong and S. Heisig. Experiences with Disjoint Data Structures in a New Hardware Transactional Memory System. Proceedings of the 25th International Symposium on Computer Architecture and High Performance Computing, 2013.

[34] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, 2009.

[35] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner. Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions. Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), 2014.

[36] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. Proceedings of the 2014 IEEE International Conference on Data Engineering (ICDE), 2014.

[37] W. Ruan, Y. Liu, and M. Spear. STAMP Need Not Be Considered Harmful. In 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2014.

[38] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent Cuckoo hashing. Proceedings of the Ninth European Conference on Computer Systems, 2014.

[39] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. Proceedings of the Ninth European Conference on Computer Systems, 2014.

[40] z/Architecture Principle of Operations. SA22-7832-09.