Thread-Level Speculation on Off-the-Shelf Hardware Transactional Memory

Rei Odaira and Takuya Nakaike
IBM Research - Tokyo
Toyosu, Tokyo, Japan
{odaira, nakaike}@jp.ibm.com

***Abstract*—Thread-level speculation can speed up a single-thread application by splitting its execution into multiple tasks and speculatively executing those tasks in multiple threads. Efficient thread-level speculation requires hardware support for memory conflict detection, store buffering, and execution rollback, and in addition, previous research has also proposed advanced optimization facilities, such as ordered transactions and data forwarding. Recently, implementations of hardware transactional memory (HTM) are coming into the market with minimal hardware support for thread-level speculation. However, few implementations offer advanced optimization facilities. Thus, it is important to determine how well thread-level speculation can be realized on the current HTM implementations, and what optimization facilities should be implemented in the future.**

In our research, we studied thread-level speculation on the off-the-shelf HTM implementation in Intel TSX. We manually modified potentially parallel benchmarks in SPEC CPU2006 for thread-level speculation. Our experimental results showed that thread-level speculation resulted in up to an 11% speed-up even without the advanced optimization facilities, but actually degraded the performance in most cases. In contrast to our expectations, the main reason for the performance loss was not the lack of hardware support for ordered transactions but the transaction aborts due to memory conflicts. Our investigation suggests that future hardware should support not only ordered transactions but also memory data forwarding, data synchronization, multi-version cache, and word-level conflict detection for thread-level speculation.

Keywords—thread-level speculation; transactional memory

I. INTRODUCTION

As CPU frequency scaling slowed down in the 2000s, chip makers began implementing more and more cores and hardware threads on each chip to continue increasing the total processing capability while retaining power efficiency. Multi-threaded applications can exploit these increases in processing capability. To facilitate such exploitation, the chip makers are providing new hardware support such as Hardware Transactional Memory (HTM).

Unfortunately, single-threaded applications cannot benefit from the increasing number of cores and hardware threads. Although parallelizing compilers have been studied for many years, they have shown benefits only in simple computation kernels. To find parallelism in a single-threaded program, the compilers must find provably data-independent tasks in the code, which is often impossible in complex applications.

Thread-Level Speculation (TLS), or Speculative Multithreading (SpMT), has been proposed [2][4][12][20][21][22] to overcome this limitation of the parallelizing compilers. Most of the proposed systems for the thread-level speculation incorporate at least three kinds of hardware support: (1) detection of address conflicts among loads and stores from multiple threads, (2) store buffering during the execution of specified code regions, and (3) execution rollback to the beginning of a region when a conflict is detected. With such hardware support, compilers no longer need to prove the data independence among the tasks. The compilers only need to find probably data-independent tasks in the code, and such tasks can be speculatively executed in parallel. It can be left to the hardware and runtime to preserve the correct sequential semantics of the program by detecting data-dependence violations and by re-executing the rolled-back tasks until the data dependence is resolved.

While thread-level speculation has only been proposed in research papers, recent releases of publicly available CPUs implement HTM [5][6][7][8], which supports the basic hardware features required for thread-level speculation. Specifically, HTM offers hardware support for memory conflict detection, store buffering, and execution rollback. Although researchers have noticed that thread-level speculation can be implemented on top of HTM [18], no one has ever evaluated thread-level speculation on real HTM hardware. Therefore, it is important to determine how well thread-level speculation can improve single-thread performance on current HTM implementations.

Beyond the raw performance data, what is more interesting is why thread-level speculation often fails to show good performance on the current HTM implementations. This is because typical proposals [2][4][12][20][21][22] for thread-level speculation include advanced hardware facilities such as ordered transactions, data forwarding, data synchronization, or word-level conflict detection for optimization, but most of the current HTM implementations do not provide such hardware. For example, each task in thread-level speculation must commit in the same order as the sequential execution order, and the hardware support for ordered transactions allows a task to wait for the previous tasks to finish. Without this hardware support, if a task is about to finish before its previous tasks, then that task will have to be rolled back. Only Blue Gene/Q [5] supports even one of those advanced facilities, ordered transactions. Thus, we want to know if the thread-level speculation can show speed-ups even without the advanced hardware facilities, and if not, what are the obstacles to achieving performance improvement.

In this paper, we studied the thread-level speculation on an off-the-shelf implementation of HTM, the Intel TSX [8], part of the 4th generation of the Intel Core processor. As target benchmarks, we selected the 6 programs in SPEC CPU2006 that can potentially be sped up by thread-level speculation, based on previous studies [16][17]. Instead of implementing a compiler for thread-level speculation, we manually modified the source code of the selected benchmarks, focusing on their frequently executed loops. We assigned each iteration (or consecutive multiple iterations) of the loops as a task to a different thread. We executed each task as a transaction on the HTM. Because the Intel TSX does not support ordered transactions as hardware, we needed to roll back a task when its previous task had not yet finished.

Here are our contributions:

- We show the first experimental results of thread-level speculation on a real HTM hardware that does not support any advanced hardware facilities such as ordered transactions, data forwarding, or word-level conflict detection.
- For each of the executed benchmarks in SPEC CPU2006, we present detailed reasons for the performance improvement or degradation and suggest what kinds of hardware extensions are necessary in the future.

Section II describes the instruction set and microarchitecture of the Intel TSX in the Intel 4th Generation Core processor, as an example of an off-the-shelf HTM implementation. Section III explains how to support thread-level speculation on the off-the-shelf HTM. Section IV shows our experimental results and Section V covers related work. Section VI concludes this paper.

II. HTM Implementation

We experimented on the Intel TSX [8] implemented in the Intel 4th Generation Core processor (Core i7-4770). This section briefly describes its instruction set and micro architectures. Intel recently announced that the Intel TSX implemented in the 4th Generation Core processor had a functional problem [10]. Intel will disable the Intel TSX in the processors to be shipped until a fix is implemented. Because the problem occurs only "under a complex set of internal timing conditions," [10] we believe our results are valid, but we will revalidate them once the fix becomes available.

The Core i7-4770 processor contains 4 cores and each core supports 2 simultaneous multi-threading (SMT) threads. Each core has 32-KB L1 data and 256-KB L2 unified caches. The 4 cores share an 8-MB L3 cache. The cache line size is 64 bytes.

In the Intel TSX, each transaction begins with an XBEGIN instruction and is ended by an XEND instruction. The detailed design of the Core i7-4770 processor's HTM has not been revealed, but it takes advantage of the CPU cache structure. The hardware keeps track of the read and write sets of each transaction, using the caches. Transactionally written data is not visible to the other threads until the transaction is committed by an XEND instruction.

A transaction can abort for various reasons, and the execution jumps to a program point specified by the argument of the XBEGIN instruction. All of the transactionally written data is discarded, and the registers are rolled back to the image immediately before the XBEGIN instruction. The most frequent causes for aborts include conflicts and footprint overflows. The EAX register reports the abort reason. Transactions conflict with each other when the read or write set of a transaction overlaps with the write set of the other transaction. When a conflict occurs, one of the transactions is aborted by the hardware, but the software cannot control which one to abort. Conflicts are detected at the granularity of a 64-byte cache line. Also, because the read and write sets are kept in caches, there are upper limits on their sizes. Our preliminary experiments showed that the maximum read-set size is 4 MB, and the maximum write-set sized is about 22 KB in the Core i7-4770. A transaction can also be aborted by software with an XABORT instruction.

These hardware features, i.e. conflict detection, store buffering, and execution rollback, are minimal requirements for efficient thread-level speculation. The other available HTM implementations such as the ones in the IBM zEnterprise EC12 [7] and POWER8 [6] offer the same set of features.

III. Thread-Level Speculation on HTM

This section describes the basics of thread-level speculation and how to implement it on the currently available HTM implementations.

We call a unit of work for speculation a *task*. Since we use HTM for speculation, each task is executed as a *transaction*, except for the case described in Section III.B.

A. Thread-Level Speculation for Loops

Our research focuses on thread-level speculation for frequently executed loops. Another typical target of thread-level speculation is function calls. Section III.D explains why we did not consider function calls as our initial target.

Fig. 1(a) is a code excerpt from the 429.mcf benchmark in SPEC CPU2006. Lines 8-18 are a frequently executed loop. Parallelizing compilers cannot prove that the iterations of this loop are data-independent, because Lines 12-15 modify shared data, the basket_size variable and the objects pointed to by the perm array. However, because the conditional checks in Lines 9 and 11 are only occasionally true, this loop can be executed in parallel for many of its iterations. This loop is a good example in which thread-level speculation can improve the single-thread performance but parallelizing compilers cannot.

(a) Frequently executed loop in 429.mcf: pbeampp.c

```
1. static long basket_size;
2. static BASKET *perm[B+K+1];
3. static long nr_group;

4. arc_t *primal_bea_mpp(..., arc_t *stop_arcs, ...) {
5.   ...
6.   cost_t red_cost;
7.   arc_t *arc = ...;
8.   for( ; arc < stop_arcs; arc += nr_group ) {
9.     if( arc->ident > BASIC ) {
10.      red_cost = arc->cost -
           arc->tail->potential +
           arc->head->potential;
11.      if( bea_is_dual_infeasible( arc, red_cost ) ) {
12.        basket_size++;
13.        perm[basket_size]->a = arc;
14.        perm[basket_size]->cost = red_cost;
15.        perm[basket_size]->abs_cost = ABS(red_cost);
16.      }
17.    }
18.  }
19.  ...
20.}
```

(b) Example of TLS execution

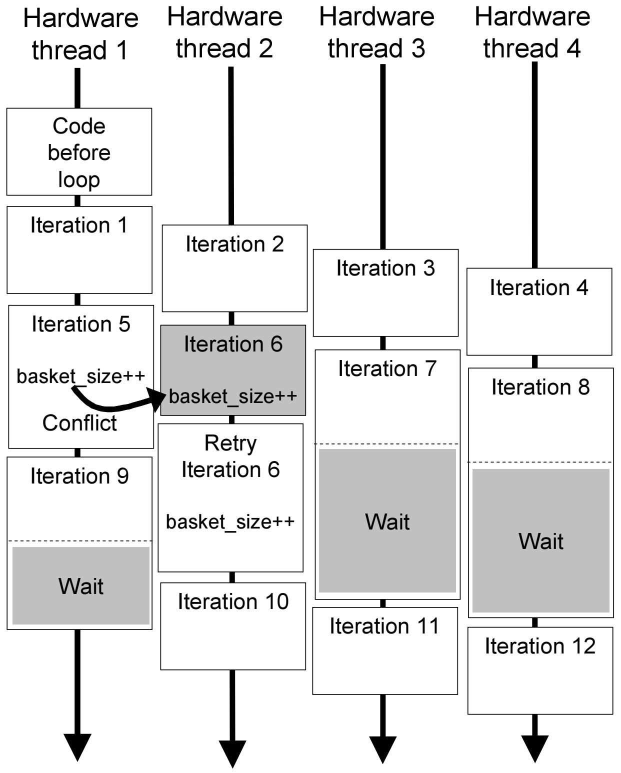

Fig. 1. (a) Frequently executed loop in 429.mcf of SPEC CPU 2006. (b) Example of execution by TLS on 4 threads, assuming that the system supports ordered transactions. Wasted CPU cycles are in grey. Iterations 5 and 6 increment a variable basket_size in global memory, resulting in a conflict. Iterations 7, 8, and 9 wait for the previous iterations to finish.

Fig. 1(b) is an example execution sequence of the loop in Fig. 1(a) using thread-level speculation. In this example, each iteration of the loop is executed as a single task. We assume that this system has 4 hardware threads and supports ordered transactions in hardware. In thread-level speculation, all of the tasks must commit in the same order as the original sequential order, because the system can insure that there is no data-

```
1. static volatile long basket_size;
2. static BASKET *perm[B+K+1];
3. static long nr_group;

4. arc_t *primal_bea_mpp(..., arc_t *stop_arcs, ...) {
5.   ...
6.   arc_t *arc = ...;
7.   arc_t *Next_Iter_To_Commit = arc;
8.   TLS_arguments Args[Num_TLS_Threads];
9.   for (Thr = 0; Thr < Num_TLS_Threads; Thr++) {
10.    Args[Thr].arc = arc;
11.    arc += nr_group * Loop_Distance;
12.    Args[Thr].stop_arcs = stop_arcs;
13.  }
14.  Invoke_TLS(TLS_primal_bea_mpp, Args,
               &Next_Itr_To_Commit);
15.  ...
16.}

17.void TLS_primal_bea_mpp(TLS_Thread *Thread) {
18.  cost_t red_cost;
19.  arc_t *arc = Thread->Args->arc;
20.  arc_t *stop_arcs = Thread->Args->stop_arcs;
21.  arc_t * volatile *Ptr_Next_Itr_To_Commit =
       Thread->Ptr_Next_Itr_To_Commit;

22.  long Inc = nr_group * (Num_TLS_Threads - 1) *
                Loop_Distance;
23.  long Distance = nr_group * Loop_Distance;
24.  for( ; arc < stop_arcs; arc += inc ) {
25.    arc_t *Original_arc = arc;
26.    bool Execute_in_HTM;
27.  Retry:
28.    if (*Ptr_Next_Itr_To_Commit != arc) {
29.      Execute_in_HTM = true;
30.      if (XBEGIN())
31.        goto Retry:
32.    } else {
33.      Execute_in_HTM = false;
34.    }
35.    for ( ; arc - Original_arc < Distance &&
               arc < stop_arcs;
               arc += nr_group ) {
36.      /* Omit.
            The same as Lines 9-17 in Fig.1(a). */
37.    }
38.    if (Execute_in_HTM) {
39.      if (*Ptr_Next_Itr_To_Commit != Original_arc)
40.        XABORT();
41.      XEND();
42.    }
43.    *Ptr_Next_Itr_To_Commit =
          Original_arc + Distance;
44.  }
45. }
```

Fig. 2. TLS version of the loop in Fig. 1, using HTM.

dependency violation in a task only after all of its preceding tasks have committed. The hardware support for ordered transactions allows a task to wait immediately before its commit point if its preceding tasks have not yet finished. In most of the proposed systems for thread-level speculation, the hardware support for ordered transactions also means that the hardware rolls back a more speculative task when a conflict is detected.

In Fig. 1(b), the hardware threads 1, 2, 3, and 4 execute iterations 1, 2, 3, and 4, respectively. Suppose the conditional checks in Lines 9 and 11 are both true in the iterations 5 and 6. These tasks cause a conflict on the updates to the basket_size variable, and the hardware rolls back the more speculative task,

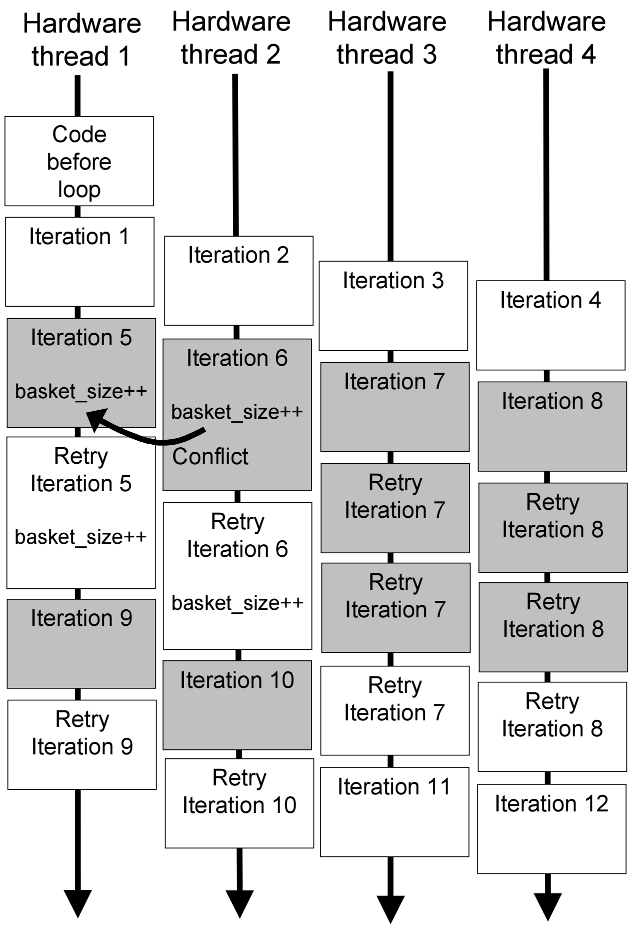

Fig. 3. Example of execution by TLS using HTM, in contrast to the execution by TLS with hardware-supported ordered transactions in Fig. 1(b). Wasted CPU cycles are in grey. The conflict between the iterations 5 and 6 aborted the less speculative iteration 5. The iterations 7-10 have to be retried because they cannot wait for the preceding iterations to finish

the iteration 6. In some proposed systems for thread-level speculation, the hardware supports data forwarding from a less speculative task to a more speculative one, so the iteration 6 would not need to be rolled back, but we do not assume such hardware support exists in this example.

The iterations 7, 8, and 9 do not encounter any conflicts, but their preceding tasks are delayed due to the conflict between the iterations 5 and 6. The hardware support for ordered transactions allows them to wait until their respective preceding tasks commit.

B. Thread-Level Speculation using HTM

Fig. 2 shows the same loop as the one in Fig. 1(a), but with thread-level speculation using HTM. Instead of implementing a compiler to perform this transformation automatically, we manually modified the source code in our experiments. In this example, memory fence instructions are omitted for clarity. The XBEGIN, XABORT, and XEND functions in Lines 30, 40, and 41 are wrapper functions for their corresponding instructions. The XBEGIN function returns false when a transaction begins, but when the transaction aborts, the execution jumps back to within the XBEGIN function, which then returns true.

The loop in Lines 8-18 in Fig. 1(a) is moved to a new function in Lines 17-45. In this example, we assume that there are 4 software threads 1-4, each bound to a corresponding hardware thread. Each software thread executes multiple tasks that are pre-assigned to that thread. For example, the software thread 1 executes iterations 1, 5, 9, and so on. The original function (Lines 4-16) sets up the arguments for each software thread and starts thread-level speculation by calling a utility function (Line 14). The utility function (not shown) passes the arguments to the software threads and lets them execute the loop body in the new function (Lines 17-45). Alternatively, instead of pre-assigning multiple tasks to a software thread, we can assign only one task to each software thread. In this style, each task spawns a new thread, which executes the next task, and when a task finishes, the corresponding thread is discarded. This dynamic thread spawning allows more flexibility in scheduling, but requires a low-overhead threading mechanism.

In the new function (Lines 17-45), the loop is transformed into doubly-nested loops, and the inner loop (Lines 35-37) is executed as a single task. This means the inner loop is executed as a single transaction, enclosed with the XBEGIN and XEND instructions (Lines 30 and 41). In this way, multiple iterations can be executed in a transaction to mitigate the execution overhead of the XBEGIN and XEND instructions. The trip count of the inner loop can be adjusted by the Loop_Distance variable. For example, when Loop_Distance is 2, the software thread 1 executes iterations 1, 2, 9, 10, and so on, assuming 4 software threads. Since there are upper limits in the read and write sets of a transaction, Loop_Distance cannot be freely increased.

Another important reason for executing multiple iterations in a transaction is to avoid false sharing, because the conflict detection is at the granularity of a cache line, which is 64 bytes long in Core i7-4770. For example, if each iteration of a loop consecutively writes to a 4-byte element of an array, then Loop_Distance should be 64 bytes / 4 bytes = 16, so that multiple transactions do not write to different parts of the same cache line. Therefore, this transformation into the doubly-nested loops is specific not to the thread-level speculation using HTM but to any thread-level speculation with cache-line-level conflict detection.

Ordered transactions are implemented by software, using a shared variable Next_Itr_To_Commit defined in Line 7. This variable holds the value of a loop induction variable for the iteration that can commit next. Each thread accesses this shared variable through a pointer (Line 21). If the induction variable of the first iteration of a task does not match Next_Itr_To_Commit (Line 39), then that task has to be rolled back by the XABORT instruction. Note that it is not a good idea to spin-wait on Next_Itr_To_Commit in the transaction, because once this variable is read in Line 39, it is kept in the read set of the transaction. When a previous task updates Next_Itr_To_Commit in Line 43, the spin-waiting transaction would be aborted due to the conflict and be rolled back anyway.

TABLE I. EVALUATED BENCHMARKS IN SPEC CPU2006

Benchmark	Description	Loop location in source code	Loop coverage	Loop_Distance
429.mcf	Combinatorial optimization	pbeampp.c, 165	41%	20
433.milc	Quantum Chromodynamics (QCD)	quark_stuff.c, 1523	23%	4
456.hmmer	Gene sequence database search	fast_algorithm.c, 133	95%	16
464.h264ref	Video compression	mv-search.c, 394	29%	16
470.lbm	Computational fluid dynamics	lbm.c, 186	98%	3
482.sphinx3	Speech recognition	vector.c, 513	35%	8

At the beginning of each task in Line 28, if it turns out that the current task can commit next, then this task is non-speculative, and therefore it is executed without using HTM. This is necessary because if all of the tasks were executed as transactions, even non-speculative tasks could be rolled back indefinitely, and forward progress could not be guaranteed.

Finally, note that we privatize the red_cost variable, which was originally in Line 6 of Fig. 1(a) but moved to the new function in Line 18 of Fig. 2, assuming that it is not difficult for a compiler to prove this variable is iteration-local.

C. *Example of Execution by TLS using HTM*

Fig. 3 shows an example of the execution of the loop in Fig. 2, in contrast to the execution on the system supporting ordered transactions in hardware in Fig. 1(b). The iterations 5 and 6 caused a conflict in the same way as in Fig. 1(b), but this time the hardware rolls back the iteration 5, because the hardware is not aware of which transaction is more speculative. As a result, not only the iteration 5 but also the iteration 6 has to be rolled back, because when the iteration 6 is about to commit, the iteration 5 has not yet finished.

For the same reason, the iterations 7-10 have to be retried. The retrial is less efficient than the wait in Fig 1(b), because a thread cannot start the next task as soon as the preceding task commits.

D. *Thread-Level Speculation for Function Calls*

We focus on thread-level speculation for loops, but another typical target of thread-level speculation is function calls. That is, the callee of a function call and its continuation are speculatively executed in parallel. However, it is difficult to efficiently implement this type of thread-level speculation on the current HTM. When a thread finishes the execution of the callee, it must notify the other thread executing the continuation, so that the other thread can commit the transaction. Unfortunately, the current HTM implementations do not provide such notification mechanisms without aborting the transactions.

IV. EXPERIMENTS

This section describes our benchmarks and implementation of the thread-level speculation using HTM and then presents the experimental results of a micro-benchmark and SPEC CPU2006.

A. *Benchmarks*

We used a micro-benchmark and SPEC CPU2006 to evaluate the thread-level speculation on HTM. The micro-benchmark is an embarrassingly parallel program that contains one loop. Each iteration increments an iteration-local variable 1,000 times, and the loop iterates 10,000,000 times. We created this micro-benchmark to study the best speed-up we can obtain from the thread-level speculation on HTM.

For SPEC CPU2006, we evaluated the benchmarks that can potentially be sped up by thread-level speculation, according to previous studies [16][17]. Specifically, we selected 6 of the 7 programs that showed more than 1.5-fold speed-ups over the sequential execution using 4 cores in the previous studies. These studies evaluated 13 benchmarks in SPEC CPU2006, using a simulator based on STAMPede [21] with hardware-supported ordered transactions, data forwarding, data synchronization, and word-level conflict detection. Therefore, we do not need to consider those benchmarks that showed no or marginal speed-ups even with these advanced hardware facilities. We excluded 444.namd because it generates multiple methods from a single C++ macro function, so it was difficult to manually modify a particular frequently executed loop.

Table I summarizes the selected 6 benchmarks. For each of these benchmarks, we applied the thread-level speculation to one frequently executed loop reported in Table 4 of the previous study [17]. The third column of Table I shows the locations of our target loops in the source code. The forth column of Table I is the fractions of the total execution time covered by our target loops, measured in our experimental environment described in Section IV.C. We used the reference data sets for our evaluations.

B. *Implementation*

We implemented the thread-level speculation on the Intel TSX using Pthreads. Because creating a Pthread is heavyweight on our experimental platform (Linux), we spawned the same number of Pthreads as the number of hardware threads at the start-up time of a program. Each Pthread spin-waits on a flag for some task to be assigned to it. This method obviously wastes CPU cycles. The system should provide a lightweight mechanism to spawn or to wake up threads, but that is beyond the scope of our current work.

We manually modified the source code of the frequently executed loops for the thread-level speculation. The loop bodies (for example, Line 36 in Fig. 2) are exactly the same as the original loops, but due to the transformation into the doubly-nested loops, the entire source code looks different from the original code. Therefore, compiler optimizations can result in generated code that is different from the originally generated code. To assess this undesired effect, we measured not only the original sequential version but also the 1-thead execution of the thread-level speculation version. In the 1-thread execution, our system spawned no Pthreads, and the parallelization overhead such as the argument setup is minimal. Therefore, we can distinguish the speed-ups or slow-downs by the thread-level speculation from the effects of the compiler optimizations.

C. *Experimental Environment and Settings*

We evaluated our implementation on one Intel Core i7-4770 processor, running at 3.4 GHz. Our machine had 4 GB of memory and ran Linux 2.6.32-431. As described in Section II, the Core i7-4770 has 4 cores with 2-way SMT. We compiled our benchmarks with GCC 4.9.0. We did not change the compiler flags specified in the configuration file provided by SPEC CPU2006.

In the following sections, we show the throughput results. Throughput is the reciprocal of the entire execution time of an application, not the execution time of the loop to which thread-level speculation was applied. We ran each benchmark 4 times and took the averages. The performance fluctuations were negligible.

We conducted preliminary experiments to find the best Loop_Distance as explained in Section III.B for each of the benchmarks. The results are shown in the 5th column of Table I. For the micro-benchmark, Loop_Distance was set to 100. In addition, when we executed more than 1 software thread on a core by enabling SMT, we halved the Loop_Distance for 429.mcf, 433.milc, and 470.lbm, because the effective read and write set sizes decreased by half.

D. *Results of Micro-Benchmark*

Fig. 4(a) shows the throughput of the micro-benchmark, normalized to the throughput of the sequential execution. The number of software threads was set from 1 to 2, 4, 6, and 8. Note that the execution with 6 and 8 threads used the 2-way SMT. The thread-level speculation on HTM scaled up to 4 threads, but the speed-up was only 1.6-fold. With 6 and 8 threads, it did not show any speed-up.

Fig. 4(b) presents the total abort ratio of the micro-benchmark and the abort ratios for each abort reason reported by the CPU. An abort ratio is the number of aborted transactions divided by the total number of transactions attempted. Note that with 1 thread, all of the tasks were executed without using HTM, so there were no aborts. The Intel TSX has 5 abort reasons, but we did not include "Hit Break Point" or "During Nested Transaction" because they never occurred in our experiments. The "Other" reason in Fig. 4 (b) means the CPU set no flag in the EAX register. The "Order inversion" was caused by the XABORT instruction in Line 40 of Fig. 2. These results indicate that the scalability of the micro-benchmark was limited by the lack of hardware-supported ordered transactions. Although each task only increments its local variable the same amount of times in this

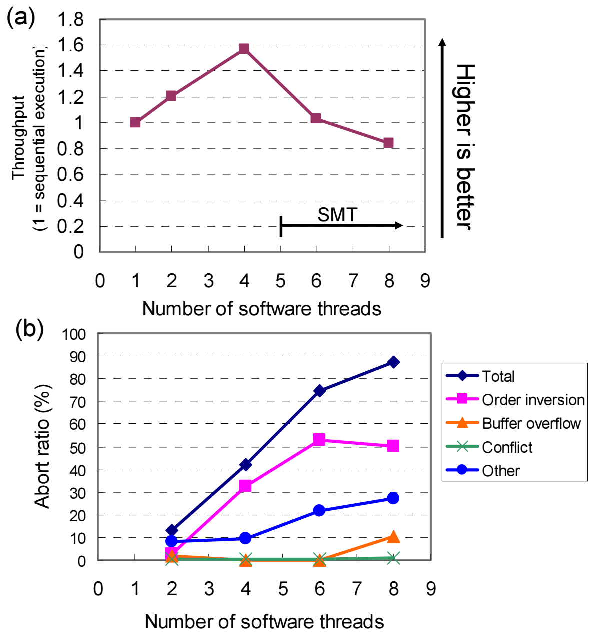

Fig. 4. (a) Throughput of the micro-benchmark on Intel Core i7-4770, normalized to the throughput of the sequential execution. (b) Abort ratios of each abort reason of the micro-benchmark.

micro-benchmark, the execution time of each task fluctuates, which can result in inversions of the committing order. The fluctuation can be exacerbated when multiple tasks are executed on the SMT.

This micro-benchmark results show the upper limit of the speed-up we can achieve with the thread-level speculation on HTM without any additional hardware facilities.

E. *Results of SPEC CPU2006*

Fig. 5(a) shows the throughput of the 6 selected benchmarks in SPEC CPU2006, normalized to the throughput of the sequential execution. The thread-level speculation on HTM improved the performance by 11% in 482.sphinx3, 8% in 429.mcf and by 5% in 433.milc using 2 or 4 threads, but degraded the performance in most of the cases. The results of the 1-thread execution were within 3% of those of the sequential execution in 433.milc, 464.h264ref, 470.lbm, and 482.sphinx3. In 429.mcf and 456.hmmer, the 1-thread executions were slower by 7% and 15%, respectively. As described in Section IV.B, these differences were caused by the effects of the compiler optimizations. If the thread-level speculation versions were as optimized by the compiler as the sequential versions, these benchmarks could show better performance.

Fig. 5(b) presents the abort ratios of the SPEC CPU2006 benchmarks. The transactions in 456.hammer, 464.h264ref, and 470.lbm almost always aborted, which resulted in the performance losses in Fig. 5(a). 456.hammer and 470.lbm scaled worse than 464.h264ref simply because their loop coverage was more than 95%, as shown in Table I. The other 3 benchmarks, 429.mcf, 433.milc, and 482.sphinx3 showed

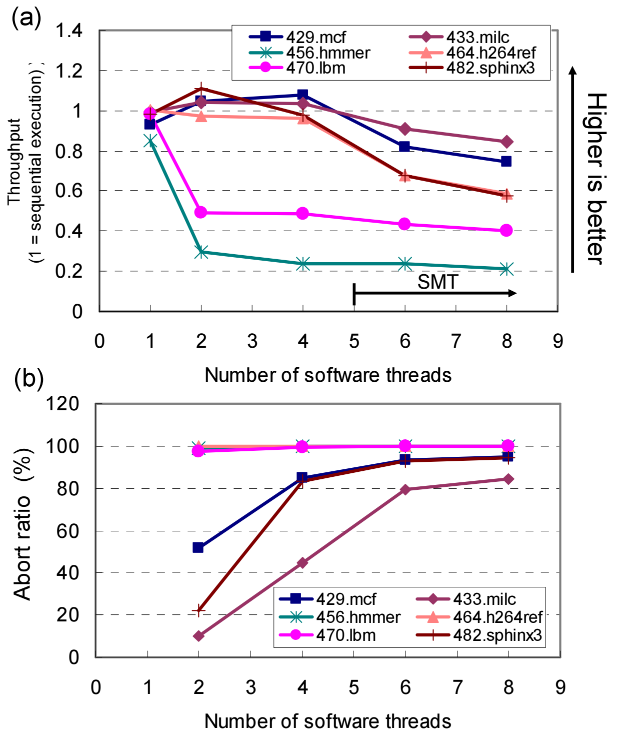

Fig. 5. (a) Throughput of SPEC CPU2006 benchmarks by TLS execution on Intel Core i7-4770, normalized to the throughput of the sequential execution of each benchmark. (b) Abort ratios of the SPEC CPU2006 benchmarks

speed-ups to some extent with 2 or 4 threads, because of their relatively lower abort ratios.

F. *Detailed Analysis of SPEC CPU2006*

Fig. 6 shows the abort ratios for each abort reason in SPEC CPU2006. In this section, we analyze the results of each benchmark by referring to the abort statistics and source code.

1) 429.mcf

We have already presented the source code of the frequently executed loop of 429.mcf in Fig. 1(a). This loop causes occasional conflicts due to the update to the basket_size variable. According to Fig. 6, the main reason for the aborts was these conflicts, not committing order inversion, even with 2 threads. We can lower the probability of the conflicts by simply making Loop_Distance smaller. However, our preliminary experiments show performance degradation with a smaller Loop_Distance, not only because of the higher relative overhead of the XBEGIN and XEND instructions, but also because of more aborts by committing order inversion. We believe that too small tasks are subject to execution time fluctuations. Hardware-supported ordered transactions could help handle the fluctuations, but the commit-time wait is inefficient in any case. A better solution would be to support data forwarding from a less speculative task to a more speculative task.

(a) Frequently executed loop in 456.hmmer

```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1]   + tpmm[k-1];
  if ((sc = ip[k-1]  + tpim[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])  mc[k] = sc;
  if ((sc = xmb  + bp[k])         > mc[k])  mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;
  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;
  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

(b) Frequently executed loop in 464.h264ref

```
int**   block_sad = ...;
...
for (pos = 0; pos < max_pos; pos++)
{
  ...
  block_sad[...][pos] = LineSadBlk0;
  ...
}
```

(c) Frequently executed loop in 482.sphinx3

```
int32 *score;
...
for (r = offset; r < end-1; r += 2) {
  m1 = gautbl->mean[r];
  m2 = gautbl->mean[r+1];
  v1 = gautbl->var[r];
  v2 = gautbl->var[r+1];
  dval1 = gautbl->lrd[r];
  dval2 = gautbl->lrd[r+1];

  for (i = 0; i < veclen; i++) {
    diff1 = x[i] - m1[i];
    dval1 -= diff1 * diff1 * v1[i];
    diff2 = x[i] - m2[i];
    dval2 -= diff2 * diff2 * v2[i];
  }

  if (dval1 < gautbl->distfloor)
      dval1 = gautbl->distfloor;
  if (dval2 < gautbl->distfloor)
    dval2 = gautbl->distfloor;

  score[r] = (int32)(f * dval1);
  score[r+1] = (int32)(f * dval2);
}
```

Fig. 7. (a) Frequently executed loop in 456.hmmer. (b) Frequently executed loop in 464.h264ref. (c) Frequently executed loop in 482.sphinx3.

2) 433.milc

433.milc is the most parallel among the 6 benchmarks, from the viewpoint of thread-level speculation. The speed-up was worse than that of 482.sphinx3, because of the lower loop coverage of 23%. The main abort reason was committing order inversion, which means hardware support for ordered transactions could provide better speed-ups. Because we halved Loop_Distance when using the SMT with 6 or 8 software threads, buffer overflows were not a severe problem.

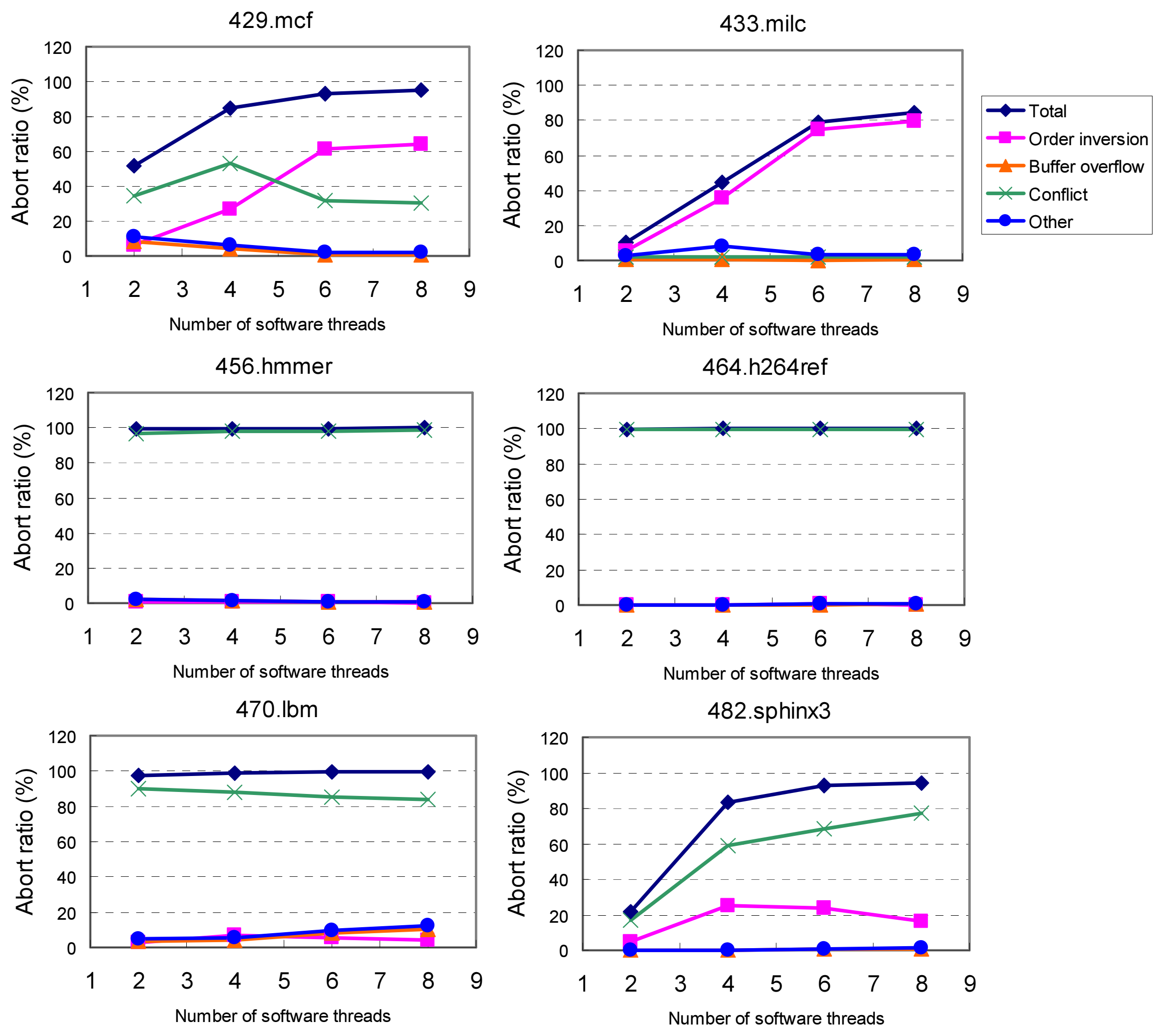

Fig. 6. Abort ratios of each abort reason in SPEC CPU2006.

3) 456.hmmer

Fig. 7(a) shows the frequently executed loop to which we applied the thread-level speculation with HTM. This loop has a great deal of loop-carried dependence, such as at the dc array in Line 8. This is why we encountered almost 100% conflict aborts, as shown in Fig. 6. The previous studies [16][17] were able to achieve good speed-ups in 456.hmmer, by taking advantage of advanced hardware facilities, such as data forwarding and synchronization.

4) 464.h264ref

The benchmark 464.h264ref is another one that suffered from severe conflicts. There are two sources of the conflicts in this benchmark. One is a write-after-read (WAR) dependence, and the other is false sharing due to multiple writers.

In 456.hmmer, the data dependence is in the loop body of the frequently executed loop, but in 464.h264ref, the data dependence is in another function called from its frequently executed loop. This function (UMVLine16Y_11) uses a static variable, so it is not reentrant. In fact, because the reads and writes to this static variable are private to each iteration, it does not have a read-after-write (RAW) dependence, but a WAR dependence. It is difficult for a compiler to analyze this data dependence, because it is in a different function of a different source file from the frequently executed loop. We need a hardware solution for this problem such as the multiple-version caches with ordered transactions proposed in some previous studies (e.g. [18]).

The other source of the conflicts is false sharing due to multiple writers. Fig. 7(b) shows a simplified version of the frequently executed loop. Each iteration of this loop consecutively writes a 4-byte integer to an array. If one iteration is assigned to one transaction, multiple transactions will write to the same cache line, causing conflicts. As described in Section III.B, we avoided this false sharing by setting Loop_Distance to 16 and by aligning the array to a cache line boundary.

To estimate a possible speed-up, we eliminated the first WAR dependence by manually modifying the UMVLine16Y_11 function. Interestingly, the abort ratio of the

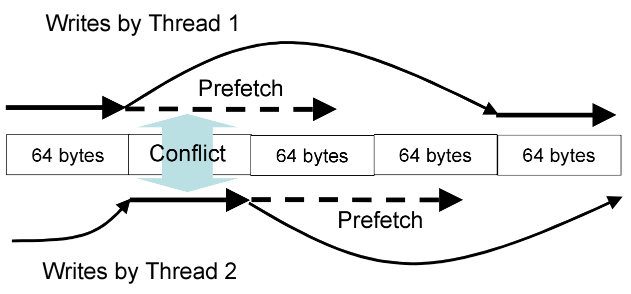

Fig. 8. Writes to a cache line triggers prefetching of the next cache line, but the prefetched cache line causes a conflict with writes by another thread.

conflicts was still as high as 93%. We found this was due to the same cause described in the section for 482.sphinx3.

5) 470.lbm

The benchmark 470.lbm also suffers from false sharing by multiple writers. Unfortunately, increasing Loop_Distance does not help, because the consecutive iterations of the frequently executed loop write to non-consecutive elements of an array. To solve this problem, word-level conflict detection is necessary.

6) 482.sphinx3

The frequently executed loop of sphinx3 shown in Fig. 7(c) contains the same pattern as in Fig. 7(b). The writes to the score array in Lines 21 and 22 cause false sharing by multiple writers. Therefore, we set Loop_Distance to 8 and inserted ramp-up code before the loop to align the accesses to a cache line boundary.

Although this benchmark showed a speed-up of 11% with 2 threads, an interesting point was that the main reason for the aborts was not committing order inversion but conflicts, as shown in Fig. 6. Since the only writes to shared memory in this loop are the ones at Lines 21 and 22, why does this benchmark still cause conflicts? We believe it was caused by the adjacent cache line prefetcher in the Core i7-4770 [9]. Fig. 8 illustrates this situation. Suppose there are 4 threads, each writing to a 64-byte cache line in a task. While a transaction running on Thread 1 is writing to a cache line, the CPU triggers the prefetching of the adjacent memory location. Because the prefetched cache line is part of the read or write set of the transaction, it will cause a conflict with another transaction running on Thread 2, which is writing to the same memory location. Because the BIOS of the machine we used (Lenovo ThinkCentre M93p) does not provide a menu to disable the adjacent cache line prefetcher, we have not yet confirmed this hypothesis.

7) Summary

Overall, 5 of the 6 benchmarks in SPEC CPU2006 suffered from various kinds of conflicts. Therefore, to extend the HTM implementations for more efficient thread-level speculation, hardware support for ordered transactions does not appear to be the best general approach. We discovered that data forwarding, data synchronization, multiple-version caches, and word-level conflict detection are necessary to speed up these benchmarks.

This observation contradicts the execution examples in Fig. 1(b) and Fig. 2, which show the benefits of hardware support for ordered transactions. This is because the actual execution of the realistic programs incurred many memory conflicts, and the execution of most of the transactions did not reach the committing point, where the transaction order mattered.

V. Related Work

Porter et al. [18] studied possible ways to extend a baseline HTM architecture for the efficient thread-level speculation. They added ordered line invalidation, data forwarding, word-level conflict detection, and a write-update protocol to the baseline architecture, and evaluated the benefits of each change. The biggest difference between their work and our research is that their baseline HTM architecture supported ordered transactions. In contrast, we based our research on the currently available HTM implementations, which do not support ordered transactions. Our results show that even without hardware support for ordered transactions, the thread-level speculation can achieve speed-ups of up to 11%, and more importantly, aborts due to not having the hardware-supported ordered transactions are not the main reason for the performance loss in many benchmarks.

Blue Gene/Q [5] supports ordered transactions in hardware. Also, the HTM of the upcoming POWER8 processor [6] will support transaction suspend and resume, which can be used to efficiently implement ordered transactions. However, our research revealed that to obtain good speed-ups in SPEC CPU2006, data forwarding, data synchronization, and other advanced facilities will be required, too.

Packirisamy et al. [16][17] applied thread-level speculation to SPEC CPU2006. They first analyzed inter-thread register- and memory-oriented data dependences in the frequently executed loops in SPEC CPU2006. They then evaluated thread-level speculation in SPEC CPU2006 on a trace-driven simulator modeling STAMPede [21]. The 6 SPEC CPU2006 benchmarks we chose were the 6 most scalable programs in their simulation results. They only briefly described why each benchmark was or was not scalable, while we provide detailed analyses for each of the 6 programs. In addition, their results assumed thread-level speculation hardware with advanced optimization facilities. In contrast, we show whether or not those 6 potentially scalable programs are actually scalable on off-the-shelf HTM.

There have been many research papers published using off-the-shelf HTM systems. They either evaluated the performance of the HTM systems by measuring transactional memory benchmarks [3][14][23][25] or used HTM to speed up applications such as database systems [11][13][24], memory managers [1][19], and programming language interpreters [15]. However, none of them evaluated thread-level speculation on an off-the-shelf HTM system.

VI. Conclusion

Thread-level speculation has been studied for many years, but only recently can we investigate its performance on real hardware by taking advantage of the hardware transactional memory available in the latest processors. In our research, we studied thread-level speculation on the off-the-shelf HTM implementation in Intel TSX. We manually modified potentially parallel benchmarks in SPEC CPU2006 for thread-level speculation. Our experimental results showed that thread-level speculation resulted in up to an 11% speed-up even without advanced optimization facilities, but actually degraded the performance in most cases. These are the first experimental results of thread-level speculation on real HTM hardware that does not support any advanced hardware facilities such as ordered transactions, data forwarding, or word-level conflict detection. Conflicting with our expectations, the main reason for the performance loss was not the lack of hardware support for ordered transactions, but rather the transaction aborts due to memory conflicts. Our investigation suggests that future hardware should support not only ordered transactions but also data forwarding, data synchronization, multi-version cache, and word-level conflict detection for thread-level speculation.

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 anonymous reviewers for providing us with helpful comments.

REFERENCES

[1] Alistarh, D., Eugster, P., Herlihy, M., Matveev, A., and Shavit, N., "StackTrack: an automated transactional approach to concurrent memory reclamation," in Proceedings of the Ninth European Conference on Computer Systems, pp. 25:1-25-14, 2014.

[2] Franklin, M. and Sohi, G. S., "The expandable split window paradigm for exploiting fine-grain parallelism," in Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 58-67, 1992.

[3] Goel, B., Titos-Gil, R., Negi, A., McKee, S. A., and Stenstrom, P., "Performance and energy analysis of the restricted transactional memory implementation on Haswell," in Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pp. 615-624, 2014.

[4] Gopal, S., Vijaykumar, T. N., Smith, J. E., and Sohi, G. S., "Speculative versioning cache," in Proceedings of the 4th International Symposium on High-Performance Computer Architecture, pp. 195-205, 1998.

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

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

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

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

[9] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 248966-028 edition, 2013.

[10] Intel Corporation, "Intel Xeon Processor E3-1200 v3 Product Family Specification Update August 2014 Revision 007," 328908-007 edition, 2014.

[11] Karnagel, T., Dementiev, R., Rajwar, R., Lai, K., Legler, T., Schlegel, B., and Lehner, W., "Improving in-memory database index performance with Intel Transactional Synchronization Extensions," in Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture, pp. 476-487, 2014.

[12] Krishnan, V., and Torrellas, J., "A chip-multiprocessor architecture with speculative multithreading," IEEE Transactions on Computers, pp. 866-880, 1999.

[13] Leis, V., Kemper, A., and Neumann, T., "Exploiting hardware transactional memory in main-memory databases," in Proceedings of the 2014 IEEE 30th International Conference on Data Engineering, pp. 580-591, 2014.

[14] Odaira, R., Castanos, J. G., and Nakaike, T., "Do C and Java programs scale differently on hardware transactional memory?," in Proceedings of the 2013 IEEE International Symposium on Workload Characterization, pp. 34-43, 2013.

[15] Odaira, R., Castanos, J. G., and Tomari, H., "Eliminating global interpreter locks in Ruby through hardware transactional memory," in Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 131-142, 2014.

[16] Packirisamy, V., Zhai, A., Hsu, W., Yew, P., and Ngai, T., "Exploring speculative parallelism in SPEC2006," in Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software, pp. 77-88, 2009.

[17] Packirisamy, V., Zhai, A., and Yew, P., "Exploring speculative parallelism in SPEC2006," Technical report TR 08-036, Department of Computer Science and Engineering, University of Minnesota, 2008.

[18] Porter, L., Choi, B., and Tullsen, D. M., "Mapping out a path from hardware transactional memory to speculative multithreading," in Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques, pp. 313-324, 2008.

[19] Ritson, C. G., Ugawa, T., and Jones, R. E., "Exploring garbage collection with haswell hardware transactional memory," in Proceedings of the 2014 International Symposium on Memory Management, pp. 105-115, 2014.

[20] Sohi, G. S., Breach, S. E., and Vijayukumar, T. N., "Multiscalar processors," in Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 414-425, 1995.

[21] Steffan, J. G., Colohan, C., Zhai, A., and Mowry, T. C., "The STAMPede approach to thread-level speculation," ACM Transactions on Computer System, Vol. 23, No. 3, pp. 253-300, 2005.

[22] Tsai, J., Huang, J., Amlo, C., Lilja, D. J., and Yew, P., "The superthreaded processor architecture," IEEE Transactions on Computers, Vol. 48, No. 9, pp. 881-902, 1998.

[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," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, pp. 127-136, 2012.

[24] Wang, Z., Qian, H., Li, J., and Chen, H., "Using restricted transactional memory to build a scalable in-memory database," in Proceedings of the Ninth European Conference on Computer Systems, pp. 26:1-26:15, 2014.

[25] Yoo, R. M., Hughes, C. J., Lai, K., and Rajwar, R., "Performance evaluation of Intel transactional synchronization extensions for high-performance computing, " in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp. 19:1-19:11, 2013.