# Performance Evaluation of Thread-Level Speculation in Off-the-Shelf Hardware Transactional Memories

Juan Salamanca[1(✉)], José Nelson Amaral[2], and Guido Araujo[1]

[1] Institute of Computing, UNICAMP, Campinas, SP, Brazil
{juan,guido}@ic.unicamp.br
[2] Computing Science Department, University of Alberta, Edmonton, AB, Canada
amaral@cs.ualberta.ca

**Abstract.** Thread-Level Speculation (TLS) is a hardware/software technique that enables the execution of multiple loop iterations in parallel, even in the presence of some loop-carried dependences. TLS requires hardware mechanisms to support conflict detection, speculative storage, in-order commit of transactions, and transaction roll-back. There is no off-the-shelf processor that provides direct support for TLS. Speculative execution is supported, however, in the form of Hardware Transactional Memory (HTM)—available in recent processors such as the Intel Core and the IBM POWER8. Earlier work has demonstrated that, in the absence of specific TLS support in commodity processors, HTM support can be used to implement TLS. This paper presents a careful evaluation of the implementation of TLS on the HTM extensions available in such machines. This evaluation provides evidence to support several important claims about the performance of TLS over HTM in the Intel Core and the IBM POWER8 architectures. Experimental results reveal that by implementing TLS on top of HTM, speed-ups of up to 3.8× can be obtained for some loops.

**Keywords:** Thread-Level Speculation · Transactional memory

## 1 Introduction

Loops account for most of the execution time in programs and thus extensive research has been dedicated to parallelize loop iterations [2]. Unfortunately, in many cases these efforts are hindered when the compiler cannot prove that a loop is free of *loop-carried* dependences. However, sometimes when static analysis concludes that a loop has a *may* dependence—for example when the analysis cannot resolve a potential alias relation—the dependence may actually not exist or it may occur in very few executions of the program [12]. *Thread-Level Speculation* (TLS) is a promising technique that can be used to enable the parallel execution of loop iterations in the presence of *may* loop-carried dependences. TLS assumes that the iterations of a loop can be executed in parallel—even in


F.F. Rivera et al. (Eds.): Euro-Par 2017, LNCS 10417, pp. 607–621, 2017.
DOI: 10.1007/978-3-319-64203-1_44

the presence of potential dependences—and then relies on a mechanism to detect dependence violations and correct them. The main distinction between TLS and HTM is that in TLS speculative transactions must commit in order.

Recently hardware support for speculation has been implemented in commodity off-the-shelf microprocessors [3,4]. However, the speculation support in these architectures was designed with *Hardware Transactional Memory* (HTM) in mind and not TLS. The only implementation of hardware support for TLS to date is in the IBM Blue Gene/Q (BG/Q), a machine that is not readily available for experimentation or usage. HTM extensions, available in the Intel Core and in the IBM POWER8 architectures, allow for the speculative execution of atomic program regions [3–5]. Such HTM extensions enable the implementation of three key features required by TLS: (a) conflict detection; (b) speculative storage; and (c) transaction roll-back.

Until now, the majority of the attempts to estimate the performance benefits of TLS were based on simulation studies [10,11]. Unfortunately, studies of TLS execution based on simulation have serious limitations. The availability of speculation support in commodity processors allowed for the first study of TLS on actual hardware and led to some interesting research questions: (1) can the existing speculation support in commodity processors, originally designed for HTM, be used to support TLS? and (2) if it can, what performance effects would be observed from such implementations? Earlier work has provided a cautiously positive answer to the first question, i.e. supporting TLS on top of HTM hardware is possible, but it requires several careful software adaptations [9]. To address the second question, this paper presents a careful evaluation of the implementation of TLS on top of the HTM extensions available in the Intel Core and in the IBM POWER8. This evaluation uses the same loops from an earlier study by Murphy *et al.* [6] and led to some interesting discoveries about the relevance of loop characterization to predict the potential performance of TLS. The experimental results indicate that: (1) small loops are not amenable to be parallelized with TLS on the existing HTM hardware because of the expensive overhead of: (a) starting and finishing transactions, (b) aborting a transaction, and (c) setting up loop for TLS execution; (2) loops with potential to be successfully parallelized in both Intel Core and IBM POWER8 architectures have better performance on the POWER8 because TLS can take advantage of the ability of this architecture to suspend and resume transactions to implement *ordered transactions*; (3) the larger storage capacity for speculative state in Intel TSX can be crucial for loops that execute many read and write operations; (4) the ability to suspend/resume a transaction is important for loops that execute for a longer time because their transactions may abort due to OS context switching; and (5) the selected size of the strip can be critical for the increase of aborts due to order inversion.

The remainder of this paper is organized as follows. Section 2 describes the relevant aspects of the implementation of HTM in both Intel Core and IBM POWER8 architectures. Section 3 details the related work. Benchmarks,

methodology and settings are described in Sect. 4. Finally, Sect. 5 shows experimental results and a detailed analysis.

## 2 How to Support TLS over HTM

This section reviews HTM extensions and discusses how they can be effectively used to support the TLS execution of hard-to-parallelize loops containing (*may*) loop-carried dependences.

### 2.1 Intel Core and IBM POWER8

Transactional memory systems must provide transaction *atomicity* and *isolation*, which require the implementation of the following mechanisms: *data versioning management*, *conflict detection*, and *conflict resolution* [9].

Both Intel and IBM architectures provide instructions to begin and end a transaction, and to force a transaction to abort. To perform such operations Intel Core's *Transactional Synchronization Extensions* (TSX) implements the *Restricted Transactional Memory* (RTM), an instruction set that includes `xbegin`, `xend`, and `xabort`. The corresponding instructions in the POWER8 are `tbegin`, `tend`, and `tabort`.

All data conflicts are detected at the granularity of the cache line size because both processors use cache mechanisms—based on physical addresses—and the cache coherence protocol to track transactional states. Aborts may be caused by: memory access conflicts, capacity issues due to excessively large transactional read/write sets or overflow, conflicts due to false sharing, and OS and micro-architecture events that cause aborts (*e.g.* system calls, interrupts or traps) [4,7].

The main differences between POWER8 and the Intel Core HTMs, summarized in Table 1, are: (1) transaction capacity; (2) conflict granularity; and (3)

**Table 1.** HTM implementations of Intel Core and IBM POWER [7].

| Processor type | Intel i7-4770 | POWER8 |
|---|---|---|
| Conflict-detection granularity (cache line) | 64 B | 128 B |
| Tx load capacity | 4 MB | 8 KB |
| Tx store capacity | 22 KB | 8 KB |
| L1 data cache | 32 KB, 8-way | 64 KB |
| L2 data cache | 256 KB | 512 KB, 8-way |
| SMT level | 2 | 8 |

**Table 2.** HTM Architectural Features.

| Features | TLS | Intel | P8 |
|---|---|---|---|
| Eager conflict detection | ✓ | ✓ | ✓ |
| Resolution conflict policy | ✓ | | |
| Ordered transactions | ✓ | | |
| Multi-versioned caches | ✓ | | |
| Suspend/resume | | | ✓ |
| Lazy conflict detection | | | |
| Data forwarding | ✓ | | |
| Word conflict detection | ✓ | | |

```
1  for(count = 0; count < WEIGHT; count++){
2    /* Start sequential segment 1 */
3    if (cond) glob++; /* Global scalar*/
4    /* End sequential segment 1 */
5
6    /* Start sequential segment 2 */
7    for(i = 0; i < factor; i++){
8      /* Global array, A */
9      int tmp = A[factor*(count%4) + i];
10     tmp += count*5;
11     if(tmp%2 == 0){
12       A[factor*(count%4) + i] = tmp;
13     }
14   }
15   /* End sequential segment 2 */
16 }
```

**Fig. 1.** A loop with two *may* loop-carried dependences. Adapted from [6].

```
1  d= STRIP_SIZE;
2  inc=(NUM_THREADS-1)*STRIP_SIZE;
3  count=param->count;
4
5  for(; count < WEIGHT; count += inc){
6    prev_count=count;
7  Retry:
8    if (!BEGIN()){
9      for (; count-prev_count < d &&
           count < WEIGHT; count++){
10       if(cond) glob++;
11     }
12     END();
13   }
14   else goto Retry;
15 }
```

**Fig. 2.** Code of each thread to parallelize Fig. 1's loop with TLS on ideal HTM system.

ability to suspend/resume a transaction. The maximum amount of data that can be accessed by a transaction in the Intel Core is much larger than in the POWER8. This speculative storage capacity is limited by the resources needed both to store read and write sets, and to buffer transactional stores.

In POWER8 the execution of a transaction can be paused through the use of suspended regions—implemented with two new instructions: `tsuspend` and `tresume`. As described in [9], this mechanism enables the implementation of an *ordered-transaction* feature in TLS [5].

### 2.2 Thread-Level Speculation

Thread-Level Speculation (TLS) has been widely studied [10,11]. Proposed TLS hardware systems must support four primary features: (a) data conflict detection; (b) speculative storage; (c) ordered transactions; and (d) rollback when a conflict is detected. Some of these features are also supported by the HTM systems found in the Intel Core and the POWER8, and thus these architectures have the potential to be used to implement TLS. Table 2 shows the necessary features required to enable TLS on top of an HTM-supporting mechanism, and its availability in some modern architectures. Neither Intel TSX nor the IBM POWER8 provide all the features necessary to carry out TLS effectively [9].

Lets examine how TLS can be applied to a simplified version of the loop example of Fig. 1 (the inner loop is omitted) when it runs on top of an ideal HTM system containing: (a) ordered transactions in hardware; (b) multi-versioning cache; (c) eager-conflict detection; and (d) conflict-resolution policy. Figure 2 shows the loop after it was strip-mined and parallelized for TLS on four cores. Assume that the `END` instruction implements: (a) ordered transactions, i.e., a transaction executing an iteration of the loop has to wait until all transactions executing previous iterations have committed, and (b) a conflict-resolution policy

that gives preference to the transaction that is executing the earliest iteration of the loop while rolling back later iterations. Multi-versioning allows for the removal of Write-After-Write (WAW) and Write-After-Read (WAR) loop-carried dependences on the `glob` variable. As shown in Fig. 3, in the first four iterations `cond` evaluates false and the iterations finish without aborts. Then, at iteration 4, the eager-conflict detection mechanism detects the RAW loop-carried dependence violation on variable `glob` between iterations 4 and 5, thus rolling back iteration 5 because it should occur after iteration 4. Subsequent iterations wait for the previous iterations to commit.

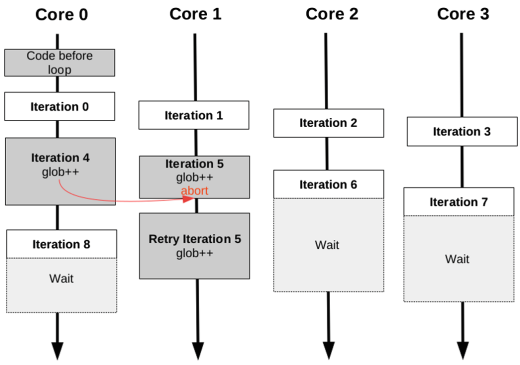


**Fig. 3.** Execution flow of Fig. 2's code with `STRIP_SIZE` = 1 and `NUM_THREADS` = 4.

## 3 Previous Research on TLS

Murphy *et al.* [6] propose a technique to speculatively parallelize loops that exhibit transient loop-carried dependences—a loop where only a small subset of loop iterations have actual loop-carried dependences. The code produced by their technique uses a TM hardware (TCC hardware) and software (Tiny STM) model running on top of the HELIX time emulator. They developed three approaches to predict the performance of implementing TLS on the HELIX time emulator: coarse-grained, fine-grained, and judicious. The *coarse-grained approach* speculates a whole iteration while the *fine-grained approach* speculates sequential segments and executes parallel segments without speculation. The *judicious approach* uses profile data at compile time to choose which sequential segment to speculate or synchronize so as to satisfy (may) loop-carried dependences. They conclude that TLS is not only advantageous to overcome limitations of the compiler static data-dependence analysis, but that performance might also be improved by focusing on the transient nature of dependences.

Murphy *et al.* evaluated TLS on emulated HTM hardware using `cBench` programs [1] and, surprisingly, predicted up to 15 times performance improvements with 16 cores [6]. They arose at these predictions even though they did not use strip mining to decrease the overhead of starting and finishing transactions

as previous work suggested [8,9]. Particularly, fine-grained speculation without strip mining can result in large overheads due to multiple transactions (sequential segments) per iteration, even larger than coarse-grained speculation. They parallelized loops in a round-robin fashion which can result in small transactions, large number of transactions, high abort ratio, bad use of memory locality, and false sharing. Their over-optimistic predictions are explained by the fact that their emulation study does not take into account the overhead of setting TLS up—which is specially high without strip mining. For instance, their emulation study predicted speed-ups even for small loops. However, when executing such loops in real hardware, the TLS overhead—setup, begin/end transactions, and aborts—would nullify any gain from parallel execution.

Odaira and Nakaike, Murphy *et al.* and Salamanca *et al.* use coarse-grained TLS to speculate a (strip-mined) whole iteration and perform conflict detection and resolution at the end of the iteration to detect RAW dependence violations [6,8,9]. The advantages of coarse-grained TLS are: (a) it is simple to implement because it does not need an accurate data dependence analyzer. (b) the number of transactions is smaller than or equal to the fine-grained or judicious approaches; and (c) there is no synchronization in the middle of an iteration. The downside is that even a single frequent actual loop-carried dependence will cause transactions to abort and serialize the execution. To illustrate this, assume an execution of the example of Fig. 1 where `cond` always evaluates true, and thus the `glob` variable is increased at each iteration of the outer loop. With coarse-grained TLS the execution of this outer loop would be serialized.

Salamanca *et al.* describe how speculation support designed for HTM can also be used to implement TLS [9]. They focused their work on the impact of false sharing and the importance of judicious strip mining and privatization to achieve performance. They provide a detailed description of the additional software support that is necessary in both the Intel Core and the IBM POWER8 architectures to support TLS. This paper uses that method to carefully evaluate the performance of TLS on Intel Core and POWER8 using 22 loops from `cBench` focusing on the characterization of the loops. This loop characterization could be used in the future to decide if TLS should be used for a given loop.

## 4 Benchmarks, Methodology and Experimental Setup

The performance assessment reports speed-ups and abort/commit ratios (Transaction Outcome) for the coarse-grained TLS parallelization of loops from the Collective Benchmark (`cBench`) benchmark suite [1] running on Intel Core and IBM POWER8. For all experiments the default input is used for the `cBench` benchmarks. The baseline for speed-up comparisons is the serial execution of the same benchmark program compiled at the same optimization level. Loop times are compared to calculate speed-ups. Each software thread is bound to one hardware thread (core) and executes a determined number of pre-assigned iterations. Each benchmark was run twenty times and the average time is used. Runtime variations were negligible and are not presented.

**Table 3.** Loops extracted from cBench applications.

| Class | Loop | Previous ID | Benchmark | Location | Function | %*Cov* | Invocations |
|---|---|---|---|---|---|---|---|
| I | A | 14 | automotive_bitcount | bitcnts.c,65 | main1 | 100% | 560 |
| | B | 18 | automotive_susan_c | susan.c,1458 | susan_corners | 83% | 344080 |
| | C | 22 | automotive_susan_e | susan.c,1118 | susan_edges | 18% | 165308 |
| | D | 24 | automotive_susan_e | susan.c,1057 | susan_edges | 56% | 166056 |
| | E | 28 | automotive_susan_s | susan.c,725 | susan_smoothing | 100% | 22050 |
| | F | 15 | automotive_bitcount | bitcnts.c,59 | main1 | 100% | 80 |
| II | G | 19 | automotive_susan_c | susan.c,1457 | susan_corners | 83% | 782 |
| | H | 23 | automotive_susan_e | susan.c,1117 | susan_edges | 18% | 374 |
| | I | 25 | automotive_susan_e | susan.c,1056 | susan_edges | 56% | 374 |
| | J | 29 | automotive_susan_s | susan.c,723 | susan_smoothing | 100% | 49 |
| III | K | 1 | consumer_jpeg_c | jfdcint.c,154 | jpeg_fdct_islow | 5% | 1758848 |
| | L | 2 | consumer_jpeg_c | jfdcint.c,219 | jpeg_fdct_islow | 5% | 1758848 |
| | M | 4 | consumer_jpeg_c | jcphuff.c,488 | encode_mcu_AC_first | 10% | 5826184 |
| | N | 6 | consumer_jpeg_d | jidcint.c,171 | jpeg_idct_islow | 14% | 7280000 |
| | O | 7 | consumer_jpeg_d | jidcint.c,276 | jpeg_idct_islow | 15% | 7280000 |
| | P | 13 | automotive_bitcount | bitcnts.c,96 | bit_shifter | 35% | 90000000 |
| | Q | 16 | automotive_susan_c | susan.c,1615 | susan_corners | 7% | 344080 |
| | R | 26 | automotive_susan_s | susan.c,735 | susan_smoothing | 96% | 198450000 |
| | S | 34 | security_rijndael_d | aesxam.c,209 | decfile | 7% | 31864729 |
| | T | 3 | consumer_jpeg_c | jccolor.c,148 | rgb_ycc_convert | 10% | 439712 |
| | U | 5 | consumer_jpeg_c | jcphuff.c,662 | encode_mcu_AC_refine | 17% | 5826184 |
| Others | V | 17 | automotive_susan_c | susan.c,1614 | susan_corners | 7% | 782 |

Loops from `cBench` were instrumented with the necessary code to implement TLS, following the techniques described by Salamanca *et al.* [9]. They were then executed using an Intel Core i7-4770 and the IBM POWER8 machines, and their speed-ups measured with respect to sequential execution. Based on the experimental results, the loops studied are placed in four classes that will be explained later. Table 3 lists the twenty two loops from `cBench` used in the study. The table shows (1) the loop class (explained later); (2) the ID of the loop in this study; (3) the ID of the loop in the previous study [6]; (4) the benchmark of the loop; (5) the file/line of the target loop in the source code; (6) the function where the loop is located; (7) %*Cov*, the fraction of the total execution time spent in this loop; and (8) the number of invocations of the loop in the whole program.

This study uses an Intel Core i7-4770 processor with 4 cores with 2-way SMT, running at 3.4 GHz, with 16 GB of memory on Ubuntu 14.04.3 LTS (GNU/Linux 3.8.0-29-generic x86_64). The cache-line prefetcher is enabled (by default). Each core has a 32 KB L1 data cache and a 256 KB L2 unified cache. The four cores share an 8 MB L3 cache. The benchmarks are compiled with GCC 4.9.2 at optimization level `-O3` and with the set of flags specified in each benchmark program.

The IBM processor used is a 4-core POWER8 with 8-way SMT running at 3 GHz, with 16 GB of memory on Ubuntu 14.04.5 (GNU/Linux 3.16.0-77-generic ppc64le). Each core has a 64 KB L1 data cache, a 32 KB L1 instruction cache, a 512 KB L2 unified cache, and a 8192 KB L3 unified cache. The benchmarks are compiled with the XL 13.1.1 compiler at optimization level `-O2`.

## 5 Classification of Loops Based on TLS Performance

The `cbench` loops were separated into four classes according to their performance when executing TLS on top of HTM. The following features, shown in Table 4, characterize the loops: (1) $N$, the average number of loop iterations; (2) *Tbody*, the average time in nanoseconds of a single iteration of the loop on Intel Core; (3) *Tloop*, $Tbody \times N$; (4) $\%lc$, the percentage of iterations that have loop-carried dependences for the default input; (5) the average (and maximum) size in bytes read/written by an iteration. The right side of Table 4 describes TLS execution: (1) the type of privatization within the transaction used in TLS implementation;[1] (2) *ss*, the *strip size* used for the experimental evaluation in Intel Core; (3) Transaction Duration in the Intel Core, which is the product $ss \times Tbody$; (4) the average speed-ups with four threads for Intel Core after applying TLS; (5) the *ss* for POWER8; (6) the speed-ups for POWER8; and (7) the predicted speed-up from TLS emulation reported in [6] for coarse-grained (C), fine-grained (F), and judicious (J) speculation using 16 cores.

For all the loops included in this study $N > 4$, thus they all have enough iterations to be distributed to the four cores in each architecture. When the duration of a loop, *Tloop*, is too short there is not enough work to parallelize and the performance of TLS is low—in the worst case, LoopS, TLS can be 100 times slower than the sequential version. Even a small percentage of loop-carried dependences, $\%lc$, materializing at runtime may have a significant effect on performance depending on the distribution of the loop-carried dependences throughout loop iterations at runtime; thus TLS performance for those loops is difficult to predict. The size of the read/write set in each transaction can also lead to performance degradation because of capacity aborts. For the Intel Core the duration of each transaction is important: rapidly executing many small transactions leads to an increase of order-inversion aborts[2]. The number of such

**Table 4.** Characterization and TLS Execution of Classes.

| Class | Loop ID | Loop Characterization | | | | | | | | TLS Execution | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $N$ | Intel's *Tbody* ($ns$) | Intel's *Tloop* ($ns$) | $\%lc$ | Read Size avg | Read Size max | Write Size avg | Write Size max | Privatization | Intel Core *ss* | Intel Core Duration ($ns$) | Intel Core Speed-up | IBM POWER8 *ss* | IBM POWER8 Speed-up | Speed-ups in [6] C | Speed-ups in [6] F | Speed-ups in [6] J |
| I | A | 1125000 | 5.0 | 5680000 | 0% | 12 B | 24 B | 0 B | 20 B | Reduction | 502 | 2600.0 | 2.20 | 502 | 3.80 | 14.0 | 14.3 | 14.3 |
| | B | 590 | 12.7 | 7500 | 0% | 48 B | 176 B | 0 B | 36 B | No | 59 | 749.0 | 1.20 | 59 | 1.59 | 10.2 | 12.0 | 12.0 |
| | C | 592 | 8.1 | 4810 | 0% | 14 B | 192 B | 0 B | 32 B | Array | 72 | 584.0 | 1.20 | 68 | 1.21 | 7.5 | 8.0 | 8.0 |
| | D | 594 | 14.1 | 8420 | 0% | 76 B | 176 B | 0 B | 28 B | Array | 88 | 1240.0 | 1.28 | 72 | 2.22 | 13.0 | 15.0 | 15.0 |
| | E | 600 | 198.0 | 118000 | 0% | 14 B | 192 B | 0 B | 32 B | Array | 15 | 2970.0 | 1.60 | 15 | 3.18 | 14.0 | 15.0 | 15.0 |
| | F | 7 | 5840000.0 | 40800000 | 0% | 48 B | 268 B | 155 B | 604 B | Array | 1 | 5840000.0 | 0.98 | 2 | 2.40 | 1.0 | 2.5 | 2.5 |
| II | G | 440 | 7710.0 | 3390000 | 0% | 2 KB | 3 KB | 29 B | 328 B | No | 1 | 7710.0 | 1.23 | 1 | 1.15 | 13.0 | 15.0 | 15.0 |
| | H | 442 | 4790.0 | 2120000 | 0% | 3 KB | 8 KB | 37 B | 260 B | Array | 1 | 4790.0 | 2.09 | 2 | 0.84 | 12.0 | 13.8 | 13.8 |
| | I | 444 | 8680.0 | 3850000 | 0% | 4 KB | 4 KB | 206 B | 1 KB | Array | 2 | 17300.0 | 1.76 | 1 | 1.05 | 13.0 | 15.0 | 15.0 |
| | J | 450 | 117000.0 | 52900000 | 0% | 3 KB | 8 KB | 37 B | 260 B | Array | 1 | 117000.0 | 1.89 | 1 | 0.73 | 0.5 | 1.0 | 1.0 |
| III | K | 8 | 8.7 | 69 | 0% | 16 B | 32 B | 16 B | 32 B | Array | 1 | 8.7 | 0.07 | 1 | 0.03 | 5.5 | 6.0 | 6.0 |
| | L | 8 | 8.5 | 68 | 0% | 16 B | 32 B | 16 B | 32 B | Array | 1 | 8.5 | 0.06 | 1 | 0.03 | 5.5 | 6.0 | 6.0 |
| | M | 38 | 5.4 | 205 | 100% | 12 B | 68 B | 4 B | 36 B | Scalar | 1 | 5.4 | 0.07 | 1 | 0.02 | 0.5 | 1.0 | 0.5 |
| | N | 8 | 8.1 | 65 | 0% | 23 B | 64 B | 16 B | 32 B | Array | 1 | 8.1 | 0.05 | 1 | 0.05 | 4.0 | 4.2 | 4.2 |
| | O | 8 | 9.4 | 75 | 0% | 24 B | 68 B | 5 B | 16 B | Array | 1 | 9.4 | 0.07 | 1 | 0.05 | 5.8 | 6.0 | 6.0 |
| | P | 23 | 1.1 | 26 | 0% | 4 B | 12 B | 4 B | 16 B | Reduction | 3 | 3.4 | 0.02 | 3 | 0.02 | 1.0 | 2.3 | 2.3 |
| | Q | 590 | 1.0 | 567 | 0.14% | 4 B | 212 B | 0 B | 36 B | Scalar | 118 | 113.0 | 0.46 | 95 | 0.49 | 9.0 | 8.5 | 8.5 |
| | R | 15 | 1.8 | 27 | 0% | 12 B | 68 B | 4 B | 56 B | Reduction | 10 | 18.2 | 0.05 | 10 | 0.04 | 4.0 | 4.0 | 4.0 |
| | S | 16 | 1.3 | 21 | 0% | 7 B | 8 B | 4 B | 16 B | Array | 2 | 2.6 | 0.02 | 2 | 0.01 | 1.0 | 3.0 | 3.0 |
| | T | 162 | 2.5 | 404 | 0% | 40 B | 44 B | 12 B | 24 B | Array & Scalar | 8 | 19.9 | 0.15 | 30 | 0.33 | 11.0 | 11.0 | 2.0 |
| | U | 63 | 4.6 | 289 | 30% | 7 B | 8 B | 4 B | 20 B | Scalar | 9 | 41.4 | 0.20 | 10 | 0.16 | 10.0 | 11.0 | 11.0 |
| Others | V | 440 | 511.0 | 225000 | 34% | 1 KB | 4 KB | 20 B | 196 B | Scalar | 1 | 511.0 | 1.25 | 1 | 1.34 | 2.5 | 2.5 | 1.0 |

[1] A Reduction privatization is a scalar privatization of a reduction operation.

[2] A order-inversion abort rolls back a transaction that completes execution out of order using an explicit abort instruction (`xabort`).

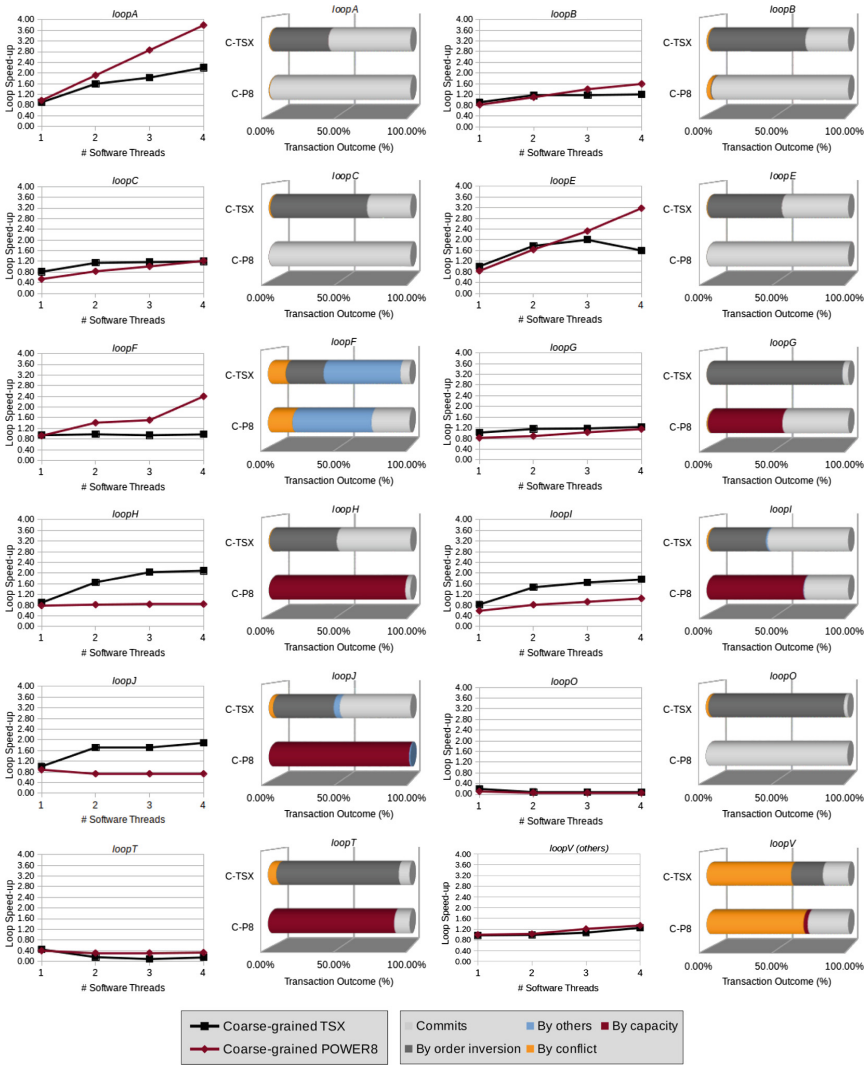


**Fig. 4.** Speed-ups and abort ratios for TLS execution on TSX and POWER8.

aborts is lowest for medium-sized transactions that have balanced iterations—when the duration of different iterations of the loop varies the number of order-inversion aborts also increases. Finally, long transactions in both architectures may cause aborts due to traps caused by the end of the OS quantum.

```
1 for (i=0; i < FUNCS; i++){//loopF
2   for (j=n=0, seed=1;
        j<iterations; j++, seed+=
        13)//loopA
3     n += pBitCntFunc[i](seed);
4   if (print)
5     printf("%-38s> Bits: %ld\n",
          text[i], n);
6 }
```

**Fig. 5.** `loopA` and `loopF`

```
1 for (is=0; is<FUNCS; is+=STRIP_SIZE){//loopF
2   for (i=is; i-is < STRIP_SIZE && i<FUNCS; i++)
3     for (j = n_arr[i] = 0, seed=1; j<iterations;
          j++,seed+=13)//loopA
4       n_arr[i] += pBitCntFunc[i](seed);
5   if (print)
6     for (i=is; i-is < STRIP_SIZE && i< FUNCS; i++)
7       printf("%-38s> Bits: %ld\n", text[i],
            n_arr[i]);
8 }
```

**Fig. 6.** `loopF` after applying strip mining and dividing into two components.

### 5.1 Class I: Low Speculative Demand and Better Performance in POWER8

The speculative storage requirement of loops in this class is below 2 KB and thus they are amenable for TLS, and see speed-ups, in both architectures. A sufficiently small speculative-storage requirement is more relevant for POWER8 which has smaller speculative-storage capacity (see Table 1). These loops also result in better scaling in POWER8, when compared to Intel Core, because they can take advantage of the `suspend` and `resume` instructions of POWER8 to implement ordered transactions in software. They do not scale much beyond two threads on Intel Core due to the lack of ordered transactions support.

Table 4 shows the characterization of Class I. These loops typically provide a sufficient number of iterations to enable their distribution among the threads. They also have a relatively moderate duration, as shown by the *Tloop* values, and thus they have enough work to be parallelized. TLS makes most sense when the compiler cannot prove that iterations are independent, but dependences do not occur at runtime, therefore most loops that are amenable for TLS (loops in Class I and II) have $\%lc$ of zero.

A typical example of a loop in Class I is `loopA`, shown in Fig. 5. This loop achieves speed-ups of up to 3.8× with four threads. This loop calls the same bit-counting function with different inputs for each iteration. Even though this loop has *may* loop-carried dependences inside the functions called, none of these dependences materialize at runtime. A successful technique to parallelize this loop relies on the privatization of variable `n` and partial accumulation of results to a global variable after each transaction commits. The successful parallelization of `loopA` stems from a moderate duration (*Tloop*), no actual runtime dependences, and a read/write set size that is supported by the HTM speculative-storage capacity. The large number of iterations of this loop allows increasing the strip size ($ss$), and thus the new *Tbody* (after strip mining)—$ss \times Tbody$—is longer; after that, order-inversion aborts decrease (`loopB` has more order-inversion aborts than `loopA`, although its *Tbody* is longer).

For most of the loops in this class the performance is directly related to the effective work to be parallelized, represented by *Tloop*. In the Intel Core the proportion of order-inversion aborts is inversely related to the transaction

```
for(j=mask_size;j<x_size-mask_size;j++){//loopE
  area = 0;
  total = 0;
  centre = in[i*x_size+j];
  ...// calulating area and total
  tmp = area-10000;
  if (tmp==0)
    *out++=median(in,i,j,x_size);
  else
    *out++=((total-(centre*10000))/tmp);
}
```

**Fig. 7.** `loopE`

```
n=0;
for(i=5;i<y_size-5;i++)//loopV
  for(j=5;j<x_size-5;j++){//loopQ
    x = r[i][j];
    if (x>0 &&(/*compare x*/)){
        corner_list[n].info=0;
        corner_list[n].x=j;
        corner_list[n].y=i;
        ...
        n++;
    }
  }
```

**Fig. 8.** `loopQ` and `loopV`

duration because very short transactions may reach the commit point even before previous iterations could commit. Another issue is that very long transactions may abort due to traps caused by the end of OS quantum. `loopF` has the longest $ss \times Tbody$ among all loops evaluated and thus many transactions abort due to traps caused by the end of the OS quantum, which explains this loop showing a high abort ratio by *other causes* in Fig. 4. Whole Coarse-grained TLS parallelization of `loopF` is not possible because each iteration has a `printf` statement that is not allowed within a transaction in either architecture. Therefore, each iteration of `loopF` must be divided into two components: `loopA` and the `printf` (as shown in Fig. 6), before applying TLS only to the first component. The second component is always executed non-speculatively.

The performance of `loopC` from one to three threads is higher on Intel Core than on POWER8 because the larger speculative store capacity in the Intel Core allows for the use of a larger strip size. With four threads, there is a small improvement in POWER8 due to the reduction of order-inversion aborts. The increment in the number of threads intensifies the effect of order inversion in performance. Therefore, for machines with a higher number of cores, better speed-ups should be achieved in POWER8 than in Intel Core.

In `loopC`, `loopD`, and `loopE` consecutive iterations write to consecutive memory positions leading to false sharing when these iterations are executed in parallel in a round-robin fashion. For instance, `loopE`, shown in Fig. 7, writes to `*out++` (consecutive memory positions) in consecutive iterations generating false sharing in a round-robin parallelization. The solution is privatization: write instead into local arrays during all the transaction and copy the values back to the original arrays after commit [9] (Fig. 8).

### 5.2 Class II: High Speculative Demand and Better Performance in Intel Core

These loops can scale better in the Intel Core compared to the POWER8 because of the larger transaction capacity of the Intel Core: the read/write sizes of these loops overflow the transaction capacity of the POWER8 (see Table 1) leading to a high number of capacity aborts.

Table 4 shows the characterization of loops in Class II. With more than 400 iterations and a loop execution time *Tloop* larger than 2 ms these loops have enough work to be parallelized. Also, no dependences materialize at runtime for the default inputs ($\%lc = 0$).

The smaller write size in `loopG` means that 50% of its transactions do not overflow the POWER8 speculative-storage capacity resulting in this loop showing speed-ups of up to 15% with four threads on POWER8. In the Intel Core this loop has a large number of order-inversion aborts because it has significant imbalance between its iterations [6]. A contrast is `loopH` that has better performance in the Intel Core even though its transactions are shorter. `loopH` results in much fewer order-inversion aborts because the durations of its transactions are balanced. `loopJ` has long transaction duration and suffer aborts due to OS traps.

### 5.3 Class III: Not Enough Work to Be Parallelized with TLS

These are loops where TLS implementation does not have enough work to be distributed among the available threads resulting in poor performance in any architecture. The overhead of setting up TLS for these loops is too high in comparison to the benefits of parallelization. Murphy *et al.* [6] reported speed-ups in these loops because their emulation of TLS hardware did not take into consideration these costs. The experiments in this section reveal that their emulated numbers overestimate the potential benefit of TLS for these loops. As shown in Table 4 the available work to be parallelized, *Tloop* in all the loops in this class is below 0.6 µs, which is too small to benefit from parallelization. For instance, `loopO` (and other loops as `loopP`) has no aborts in POWER8, but their performance is poor because of the overhead of setting TLS up.

Most of the loops in this category have many order-inversion aborts in Intel Core because their transaction duration is below 120 ns leading to a fast end of the transactions/iterations probably even before previous iterations could commit. `loopT` presents a high order-inversion abort rate in Intel Core because its transactions last less than 20 ns. In POWER8, the strip size needed to increase the loop body and the privatization of three arrays lead to aborts because the speculative capacity of the HTM is exceeded.

### 5.4 Others

They are a special case because of they are loops that have sufficient work to be parallelized but whose dependences materialize at runtime. For instance, `loopV` has 34% of probability of loop-carried dependences, but TLS can still deliver some performance improvement. As explained in [6], this loop finds local maxima in a sliding window, with each maximum being added to a list of corners, each iteration of `loopQ` processes a single pixel whereas a complete row is processed by each iteration of `loopV`. The input of this loop is a sparse image with most of the pixels set to zero, and the suspected corners (iterations with loop-carried dependences) are processed close to each other.

**Table 5.** Characterization of 6 loops from SPEC CPU 2006.

| Loop ID | Benchmark | Location | $\%Cov$ | $N$ | $Tbody$ (ns) | $Tloop$ (ns) | $\%lc$ | Iteration Size | Class |
|---|---|---|---|---|---|---|---|---|---|
| `mcf` | 429.mcf | pbeampp.c,165 | 40% | 300 | 20 | 6000 | 3% | 300 B | Others |
| `milc` | 433.milc | quark_stuff.c,1523 | 20% | 160000 | 94 | 15000000 | 0% | 1 KB | I |
| `h264ref` | 464.h264ref | mv-search.c,394 | 36% | 1089 | 156 | 170000 | 0% | 6 KB | II |
| `sphinx3` | 482.sphinx3 | vector.c,513 | 37% | 2048 | 29 | 60000 | 0% | 1 KB | I |
| `astar` | 473.astar | Way2_.cpp,100 | 60% | 1234 | 41 | 50000 | 20% | 1 KB | Others |
| `lbm` | 470.lbm | lbm.c,186 | 99% | 1300000 | 55 | 71000000 | 0% | 500 B | I |

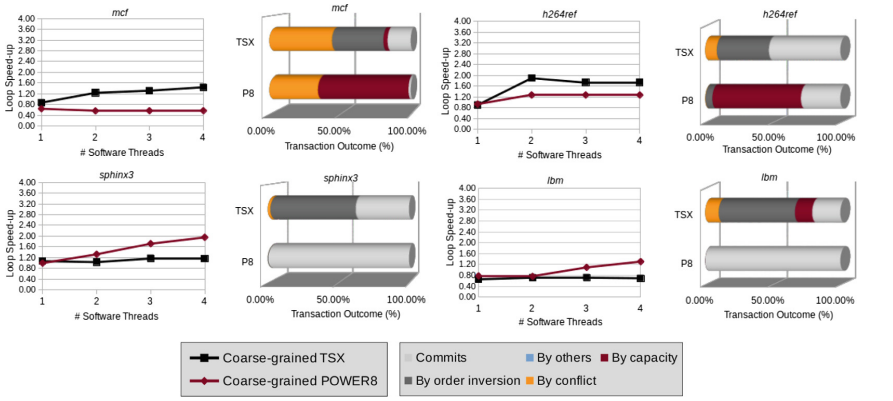


**Fig. 9.** Four SPEC2006 Loops. Speed-ups and abort ratios for coarse-grained TLS execution on TSX and POWER8.

### 5.5 Predicting the TLS Performance for Other Loops

The characterization of the loops given in Table 4 and the performance evaluation presented could also be used to predict the potential benefit of applying TLS for new loops that were not included in this study. For loops with short *Tloop*, such as those in class III, TLS is very unlikely to result in performance improvements in either architecture. For loops with small read/write sets and no dependences materializing at runtime, such as those in class I, TLS is likely to result in modest improvement for the Intel Core and more significant improvements for the POWER8. Loops that have sufficient work to be parallelized and no actual dependences but have larger read/write sets, such as those in Class II, are likely to deliver speed improvements in the Intel Core but will result in little or no performance gains in the POWER8 because of the more limited speculative capacity in this architecture. Finally, loops that have sufficient work to be parallelized but whose dependences materialize at runtime are difficult to predict—such as `loopV`. The distribution of loop-carried dependences among the iterations of such loops must be studied.

Six loops from the SPEC CPU 2006 suite are characterized (Table 5) to predict to which class they belong according to the classification described in Sect. 5. Loops `milc`, `sphinx3`, and `lbm` are classified as Class I; `h264ref` as Class II; and `mcf` and `astar` as Others. Based on this classification a prediction can be made about the relative performance of the loops on TLS over HTM for both

**Table 6.** TLS Execution for 6 loops from SPEC CPU 2006.

| Loop ID | *ss* | | Intel Tx Duration (*ns*) | Speed-up | | Loop Class |
|---|---|---|---|---|---|---|
| | Intel | P8 | | Intel | P8 | |
| `mcf` | 20 | 48 | 400 | 1.45 | 0.60 | Others |
| `milc` | 4 | 4 | 375 | 1.44 | 1.50 | I |
| `h264ref` | 16 | 6 | 2490 | 1.74 | 1.27 | II |
| `sphinx3` | 8 | 16 | 234 | 1.16 | 1.95 | I |
| `astar` | 128 | 256 | 5180 | 0.74 | 0.49 | Others |
| `lbm` | 33 | 17 | 1800 | 0.69 | 1.30 | I |

architectures. Results of TLS parallelization of these loops are shown in Table 6 and Fig. 9 and confirm the predictions.

## 6 Conclusions

This paper presents a detailed performance study of an implementation of TLS on top of existing commodity HTM in two architectures. Based on the performance results it classifies the loops studied and doing so provides guidance to developers as to what loop characteristics make them amenable to the use of TLS on the Intel Core or on the IBM POWER8 architectures. Future design of hardware support for TLS may also benefit from the observations derived from this performance study.

**Acknowledgments.** The authors would like to thank FAPESP (grants 15/04285-5, 15/12077-3, and 13/08293-7) and the NSERC for supporting this work.

## References

1. cTuning Foundation: cBench: Collective benchmarks (2016). http://ctuning.org/cbench
2. Hurson, A.R., Lim, J.T., Kavi, K.M., Lee, B.: Parallelization of doall and doacross loops-a survey. Adv. Comput. **45**, 53–103 (1997)
3. IBM: Power ISA Transactional Memory (2012). www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf
4. Intel Corporation: Intel architecture instruction set extensions programming reference. Intel transactional synchronization extensions, Chap. 8 (2012)
5. Le, H., Guthrie, G., Williams, D., Michael, M., Frey, B., Starke, W., May, C., Odaira, R., Nakaike, T.: Transactional memory support in the IBM POWER8 processor. IBM J. Res. Dev. **59**(1), 8:1–8:14 (2015)
6. Murphy, N., Jones, T., Mullins, R., Campanoni, S.: Performance implications of transient loop-carried data dependences in automatically parallelized loops. In: International Conference on Compiler Construction (CC), pp. 23–33, Barcelona, Spain (2016)

7. Nakaike, T., Odaira, R., Gaudet, M., Michael, M.M., Tomari, H.: Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In: International Conference on Computer Architecture (ISCA), pp. 144–157, Portland, OR (2015)
8. Odaira, R., Nakaike, T.: Thread-level speculation on off-the-shelf hardware transactional memory. In: International Symposium on Workload Characterization (IISWC), pp. 212–221, Atlanta, Georgia, USA, October 2014
9. Salamanca, J., Amaral, J.N., Araujo, G.: Evaluating and improving thread-level speculation in hardware transactional memories. In: IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 586–595, Chicago, USA (2016)
10. Steffan, J., Mowry, T.: The potential for using thread-level data speculation to facilitate automatic parallelization. In: High Performance Computer Architecture (HPCA), p. 2, Washington, DC, USA (1998)
11. Steffan, J.G., Colohan, C.B., Zhai, A., Mowry, T.C.: A scalable approach to thread-level speculation. In: International Conference on Computer Architecture (ISCA), pp. 1–12, Vancouver, BC, Canada (2000)
12. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In: Programming Language Design and Implementation (PLDI), pp. 177–187, PLDI 2009, ACM, Dublin, Ireland (2009)