Using Hardware Transactional Memory to Enable Speculative Trace Optimization

Juan Salamanca
Institute of Computing
UNICAMP
Email: juan@ic.unicamp.br

José Nelson Amaral
Computing Science Department
University of Alberta
Email: amaral@cs.alberta.ca

Guido Araujo
Institute of Computing
UNICAMP
Email: guido@ic.unicamp.br

***Abstract*—This paper describes a novel speculation technique for the optimization, and simultaneous execution, of multiple alternative traces of hot code regions. This technique, called Speculative Trace Optimization (STO), enumerates, optimizes, and speculatively executes traces of hot loops. It requires hardware support that can be provided in a similar fashion as that available in Hardware Transactional Memory (HTM) systems. This paper discusses the necessary features to support STO, namely multi-versioning, lazy conflict resolution, eager conflict detection, and transaction synchronization. A review of existing HTM architectures — Intel TSX, IBM BG/Q, and IBM POWER8 — shows that none of them have all the features required to implement STO. However, this work demonstrates that STO can be implemented on top of existing HTM architectures through the addition of privatization and pause/resume code. The evaluation of a prototype STO implementation, on top of Intel TSX, using benchmarks from Parboil, MediaBench, and SPEC2006, indicates that STO can yield whole-program speedups of up to 9%. This initial result is promising given that the prototype has significant overhead caused by the code that compensates for TSX absent features. An analysis, included in the paper, suggests that HTM mechanisms have the potential to considerably improve trace performance provided that they efficiently implement the suggested features.**

***Index Terms*—Hardware Transactional Memory; Speculative Optimization; Speculative Execution; Trace Optimization; TSX.**

I. Introduction

This paper proposes Speculative Trace Optimization (STO) to speculatively optimize and execute multiple alternative *traces* of a single iteration of a hot loop. The goal is to simultaneously execute speculative traces in hot loops to uncover hidden optimizations that could not be carried out at compile time because of program-flow indeterminism. STO **is not** a loop parallelization technique, rather it is a technique that speeds up both sequential and paralelizable loops. The discussion in this paper focuses on the exhaustive execution of inner-loop traces, but STO can be used in other code regions and it can also be used to selectively execute a subset of speculative traces. Contrary to whole-procedure traces, the number of inner-loop traces is reasonably small, making them good candidates for speculation in current HTM architectures that have limited capacity to store speculative state [1]–[4].

The use of STO described in this paper enumerates all possible traces, optimizes them, and executes each trace speculatively in a *transaction*, using a fork/join paradigm. All conditionals that select a specific trace are evaluated at the end of each transaction to determine if the trace should commit or abort. For each loop iteration, a single trace commits while the others miss-speculate and thus should be aborted.

To explain such approach, Program 1 lists the code of a simple `for` loop. Figure 1 shows the four possible traces that could be executed at each iteration of the loop, namely A, B, C and D. The loop has two consecutive `if` statements with conditional expressions that cannot be resolved until runtime and may depend on calculations performed earlier in the traces. In that case, it is not possible to select one amongst multiple traces before executing the traces. The instructions in lines 3-5 are dead code and can be eliminated when *Trace A* is executed because of the redefinition of `z` without prior use in line 18. Similarly, when *Trace B* is executed, the instructions in lines 9-11 are dead code because of the definition of `r` in line 24. Although some compilers may attempt to apply partial dead-code elimination [5] to this code, in general such techniques are not successful or come at high cost. The creation of longer traces of execution with STO enables many known compiler optimizations [6]. The simple program in this example is meant as a motivation for the ideas behind STO.

```
1   for (i=0; i<n; i++) {
2       if (cond1(z,r)){//condition calculated with z and r
3           z=z+y;
4           z=z*2;
5           z=y+z;
6           p=y*2;
7       }
8       else{
9           r=r+q;
10          r=r*2;
11          r=q+r;
12          x=q*2;
13      }
14      if (cond2(x,p)){//condition calculated with x and p
15          q=r-p; //z is not used here
16          p=p*q;
17          p=p+1;
18          z=p;
19      }
20      else {
21          y=z-x; //r is not used here
22          x=x*y;
23          x=x+1;
24          r=x;
25      }
26  }
```

Program 1. Example of code to optimize

The initial assessment of STO presented in this paper uses the prototype based on TSX and applies it to benchmarks from Mediabench, Parboil and SPEC2006 benchmarks. The results reveal speedups of up to 9% for four cores. This initial result is encouraging given that TSX lacks multi-versioning and lazy-conflict resolution, and it has a significant abort overhead [7]. To compensate for the missing features, extra code is inserted into the original program leading to additional overhead. Achieving speedups even in the presence of such overheads suggests that if an HTM architecture were to incorporate such features, significant speedups could emerge.

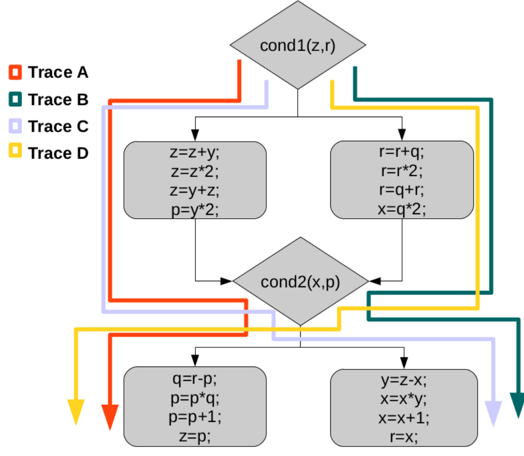

Fig. 1. Possible traces of execution

```
1.  Profile the program to identify the hottest loops.
2.  Collect the source code for all traces (when exhaustively
    speculating) of the selected loops identified in Pass 1.
    In the example, there are four traces --- A, B, C and D
    --- shown in Figure 1.
3.  Create a thread pool with sufficient threads to execute
    all traces. In the example, four threads will be created.
4.  Use the thread pool to dispatch a task for each trace.
    Each task executes all iterations of their corresponding
    trace as shown in Figure 2.
5.  In the source code transform each trace into a
    transaction enclosed by the begin and end instructions. At
    each iteration of the loop, traces must evaluate all
    their conditions at the end of the transaction (Figure
    2). If all conditions are true the trace must commit (and
    update the induction variable), otherwise it must wait for
    the correct transactional trace to commit.
6.  Activate compiler optimizations to be applied to
    speculative traces. Figure 3 shows the traces of Figure 2
    optimized by the compiler. As shown, Trace A and Trace B
    were optimized using a classic dead-code elimination.
    Other traces (C and D) were not optimized.
```

Algorithm 1. *STO* Algorithm

This paper offers the following contributions:

- It presents a novel technique to optimize and speculate exhaustive traces, called STO, that uses HTM (specifically TSX in the prototype) to execute these traces in transactions using a fork/join paradigm. STO does not parallelize loops, rather it accelerates the sequential execution of loops.
- It identifies the main features that an HTM mechanism should have to enable STO.

The remainder of this paper is organized as follows. Section II describes our technique. Section III shows experimental results. Related work is described in Section IV. Finally, Section V presents concluding remarks and future work.

II. Speculative Trace Optimization Supported by HTM

This section describes the main ideas behind STO and how to implement it on top of an HTM architecture. The unit of speculation is a *trace*. Each speculative trace can be also named a *task*. When the speculation in STO is supported by HTM, each trace is executed as a *transaction*.

A. STO on Ideal HTM

This prototype evaluation focuses on the use of STO to speculate traces found in frequently executed loops. However, STO can be also applied to other hot code regions such as frequently executed functions.

Figure 1 shows the possible traces of execution in the body of the `for` loop shown in Program 1. In this example, if Trace A is executed, lines 3-5 of the code shown in Program 1 are dead and can be eliminated. Similarly, if Trace B is executed, lines 9-11 are dead and can be also eliminated. However, without executing these traces, the value of the conditions are unknown and a compiler must preserve the full path in both cases.

The algorithm that leads to STO is described in Algorithm 1, using the code in Program 1 as a guideline.

Figure 2 shows each trace of Figure 1 as a transaction enclosed by the `begin` and `end` instructions of an ideal HTM system. Figure 4 shows an example of an execution sequence of the loop of Program 1 using STO. In this example, each trace of each loop iteration is executed in a single transaction by a thread. This ideal system has four hardware threads and an ideal HTM, which has a negligible abort overhead and the following features: eager conflict detection and lazy conflict resolution (Eager-Lazy HTM [8], [9]), multi-versioned cache memory addresses, ability to *pause* a transaction to *wait* for another commiting one (`wait` instruction), and large speculative capacity. For the sake of the this example, assume that there is no false sharing.

The conditionals of all `if` statements in each trace are converted into *predicates* for the execution of the trace and are evaluated at the end of the trace. Only one trace evaluates all its predicates to true and must commit when the `end` instruction is executed. The other traces either: (a) wait for the correct trace to commit and abort afterwards, or (b) will be aborted due to a conflict, by the transaction that commits, without having finished its execution. Figure 4 shows, in blue, the transactions (traces) that commit, in yellow the time spent by a transaction waiting for another one to commit and, in orange, the transactions that abort without finishing. Transactions that never complete are shown with dashed borders.

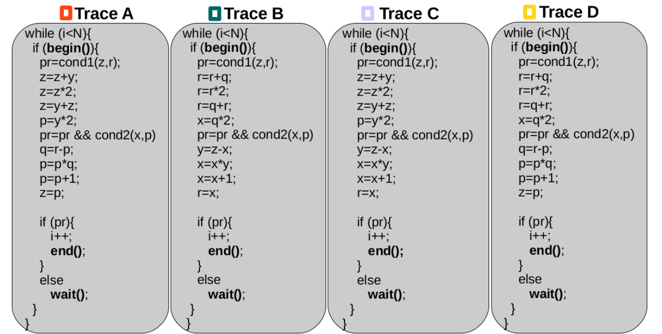

Fig. 2. Traces as transactions

As shown in Figure 4, all threads are created before the execution of loop iterations starts (Pass 3 of Algorithm 1). This preamble is executed by the hardware thread 1. In Figure 4, hardware threads 1, 2, 3, and 4 execute traces A, B, C, and D, respectively. At the first iteration, suppose that the if-conditions of Lines 2 and 14 of Program 1 are both true. Thus Trace A evaluates its predicates to true and commits. The other three traces B, C, and D — which read or write variables `p`, `q`, and `z` written by Trace A — abort without finishing.

In the second iteration, suppose that the conditions in Lines 2 and 14 of Program 1 are both false. This way Trace B evaluates all its predicates to true and commits writing to variables `x`, `y`, and `r`. On the other hand, traces A and C have to abort because

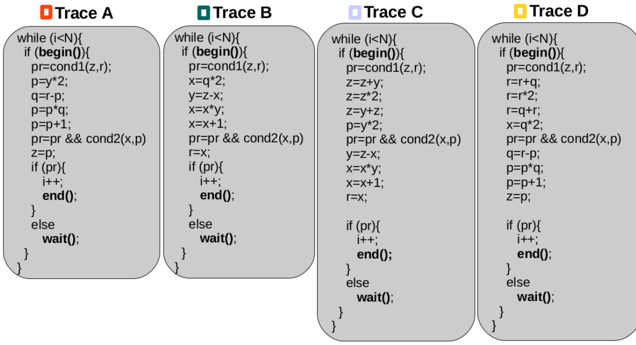

Fig. 3. Optimized traces of execution

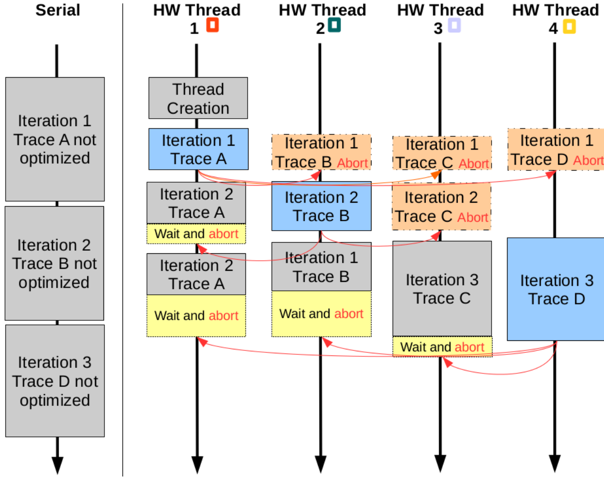

Fig. 4. Possible execution flow of STO Traces shown in Figure 3 on ideal HTM.

they read or write these variables. Trace A finishes before Trace B commits, and thus it has to wait and then abort. Trace C, on the other hand, aborts before completion. In some iterations not all traces are executed. Assume that, when the value of the induction variable is k, the trace that evaluates all its predicates to true finishes execution and updates the induction variable to $k+1$ before one of the traces with false predicates starts. Then the thread responsible for this later trace will skip the execution of a transaction for iteration k. In the example, Trace D skips iteration 2 entirely and resume execution in the iteration 3.

Finally, in the third iteration, Trace D evaluates all its conditions to true and commits writing to variables p, q, r, x, and z. The other three traces A, B, and C read or write these variables and finish before Trace D, thus they have to wait and then abort due to the commit of Trace D.

The left side of Figure 4 shows the serial execution of the above three iterations.

B. STO Prototype on Real-world HTM

The previous discussion assumed that STO would be executed using an ideal HTM. This sections explains how STO works in a real HTM (*i.e.* Intel TSX) and discusses the main features, which are lacking in TSX, that need to be addressed to enable speculative trace optimization.

1) Privatization to Simulate Multi-versioning and Lazy Conflict Resolution: Eager conflict resolution is a problem for STO because multiple traces write to the same variable, and thus each of those conflicting writes would cause a conflict abort. The aborted transaction could be the one that had to commit causing

```
1   i=&(param->i);
2   while ((*i) < n){
3      status = _xbegin();
4      if (status == _XBEGIN_STARTED){
5         iL=*i;
6         pr=cond1(z,r);
7         zL=z+y;
8         zL=zL*2;
9         zL=y+zL;
10        pL=y*2;
11        pr=pr && cond2(x,pL);
12        qL = r - pL;
13        pL = pL*qL;
14        pL = pL + 1;
15        zL = pL;
16        if (pr && (iL < n)){
17           _xend();
18           z=zL;
19           p=pL;
20           q=qL;
21           param->i= param->i + 1;
22        }
23        else if (iL < n)
24           while(1);
25     }
26  }
```

Program 2. Modified Source Code of Trace A

a significant retry overhead. To overcome this limitation, STO prototype privatizes all variables that have to be written within a transaction (trace) and the induction variable of the loop. In the example of Program 2, variable i is the induction variable and variables p, q, and z are written within Trace A. Variable i is copied at the beginning of the transaction executing Trace A. Variables that are written are also copied into their thread-local copies (*e.g.* zL is a local copy of z). Conflicts are detected and resolved after the commit (_xend), when the local copies are non-speculatively written back to the original variables by the trace with all predicates true. Thus, privatization implements multi-versioning and lazy-conflict resolution, necessary features for speculative trace optimization.

2) Non-speculative Writes to Simulate Conflict-Resolution Policy: The conflict-resolution policy used by TSX can interfere with the implementation of STO. Let T be the trace that evaluates all its predicates to true, and let F_0, …, F_k be the traces for which at least one of the predicates is false. When T writes to the variables that it modifies and attempts to commit, it is likely that T has a conflict with a transaction that is executing one of the other threads. If the conflict-resolution policy were allowed to abort T and allow the survival of some F_i trace, the intent of STO would be defeated. To overcome this limitation, once T commits, T non-speculatively writes the modified variables, including the loop-induction variable. Some F_i may be executing or have finished. If F_i finishes, STO forces F_i to spin on an infinite loop while only T proceeds to the commit phase. These non-speculative writes lead to the abortion of all F_i (spinning or not) because each trace has to read the induction variable. This mechanism is used to create the effect of a Conflict-Resolution Policy — a necessary feature for speculative trace optimization — in an HTM that does not have this feature.

3) Pausing to Simulate Trace Synchronization: A trace that evaluates any of its predicates to false is a miss-speculation and should abort. One way to abort such a trace, would be to issue an _xabort instruction whenever a predicate fails and to retry. An alternative is to keep the miss-speculation trace executing an idle loop until it is aborted **only once** because of a detection of conflict with the correct trace. A non-speculative write of the value of the induction variable — which is read at the start of all

traces — causes all incorrect traces to abort. These two approaches have been tested on Intel TSX for the traces of Program 1 and their impact on performance was measured. The use of the `_xabort` instruction to interrupt incorrect traces resulted in a 1.19× slowdown when compared to waiting for the correct trace to commit. This slowdown is due to the cost of recovery from `_xabort` in TSX, which is high (*150 cycles*) [7], resulting in a large performance penalty for issuing this instruction **many times** by retrying. Therefore, with the current implementation of TSX, waiting at the end of the transaction for the commit of the correct trace is a better approach to build a STO prototype. Lower-cost aborts in future architecture would change this tradeoff. This prototype implementation of STO on top of TSX uses an infinite loop `while (1);` statement, as shown in the Line 24 of Program 2. A trace that evaluates any of its predicates to false will wait until the thread executing the correct trace commits and then writes, non-speculatively, the new value of the induction variable (Lines 18 - 21 of Program 2). This non-speculative write will lead to the intended eager conflict detection and conflict resolution mechanisms of TSX to abort all the incorrect traces. In the current implementation of Intel TSX a transaction may also abort due to other reasons such as traps when the limit of OS quantum has been reached, interrupts, temporary capacity overflow, *etc*. Thus, the STO prototype retries the transaction when such spurious aborts occur to ensure that the correct trace is eventually executed to completion.

TABLE I
HTM ARCHITECTURAL FEATURES

Features/HTMs	TSX	BGQ	P8
Multi-version		✓	
Eager Detection	✓	✓	✓
Lazy Resolution		✓	
Ordered Txs		✓	
Suspend/Resume			✓
Lazy Detection		✓	
Forward		✓	
ROTs			✓

The above analysis suggests that multi-versioning, eager conflict detection, lazy conflict resolution, and transaction synchronization are central features to enable trace speculation. Unfortunately, as shown in Table I, none of the current HTM architectures (Intel TSX, IBM BG/Q and POWER8) have all the features required to implement trace speculation strategies like STO. Intel TSX does not allow for multi-versioning nor lazy-conflict resolution. On the other hand, although POWER8 has suspend/resume instructions, which could eventually implement transaction synchronization, it does not allow multi-versioning nor lazy conflict resolution. Moreover, in the current version of POWER8 the cost of suspend/resume is comparable to the cost of starting a transaction. Blue Gene/Q [4] is the architecture that is closest to implement all the features required for trace speculation. BG/Q features multi-versioning cache, ordered transactions in hardware (for transaction synchronization), and lazy conflict resolution; features that are useful to enable STO. However, the runtime system implemented on top of the best-effort HTM in BG/Q provides forward-progress guarantees that assume that each started transaction must eventually commit [10], [11]. This assumption does not fit well with the concept of speculation in STO, where all but one trace should abort.

C. Running STO on Intel TSX

Program 2 shows Trace A, after the code in Figure 2 is modified using Intel TSX; the other traces were modified accordingly. Algorithm 2 explains the implementation of the STO strategy when using TSX, again considering the example of Program 1.

```
1-5  Same as Passes 1-5 in Algorithm 1.
6.   At each trace, make a local copy of each variable that is
     written within the transaction (for example variables z,
     p, and q whose corresponding copies are zL, pL, and qL
     in Program 2) and replace the original variable by their
     private copies in the transaction.
7.   At each trace, after committing, write the value of the
     private variables to the original ones as in the Lines
     18-20 of Program 2.
8.   For each trace, read the induction variable at the
     beginning of the transaction into a local variable as
     shown in the Line 5 of Program 2; replace the original
     induction variable by the private copy in the whole
     transaction. After committing, update the induction
     variable as shown in the Line 21 of Program 2.
9.   At each trace, simulate waiting for the correct trace
     commit by putting a while(1) for the case when the
     predicates of the trace are false as shown in Line 24 of
     Program 2.
10.  Same as Pass 6 in Algorithm 1.
```

Algorithm 2. *STO* Strategy Using TSX

Figure 5 is an example of an execution sequence of the loop in Program 1 using STO on Intel TSX in a system with four hardware threads. As detailed below, the performance of this execution is worse than the execution shown in Figure 4 because Intel TSX lacks many features of an ideal HTM for STO, as mentioned in Section II-B.

The overhead to update (after commit) the induction variable and the written variables of the transaction (as in Passes 7 and 8 of Algorithm 2) are shown in red. The overhead to read the induction variable is shown at the beginning of each trace (transaction). The main differences between the execution of Figure 5 and the execution of Figure 4 are: (a) The insertion of the induction variable read in each transaction. (b) The insertion of the induction variable non-speculative write after committing the correct trace.

Hardware threads 1, 2, 3, and 4 execute traces A, B, C, and D, respectively. In the first iteration, suppose that all the predicates of Trace A evaluate to true and thus its transaction commits after testing the predicates. Immediately after committing, Trace A non-speculatively copies the values from the thread-local copies `pL`, `qL`, `zL`, and `iL` into variables `p`, `q`, `z`, and `i`, respectively. The other three traces — B, C, and D — read these variables in their respective transactions and have to abort due to a conflict. Every trace has the loop induction variable in its read set. Thus when the correct trace — Trace A in this example — non-speculatively writes to that variable, the eager-conflict detection and resolution in TSX forces all other traces to abort. Assume, in this example, that Trace B finishes before the non-speculative writes by the thread of Trace A. Thus Trace B has to wait (in the `while(1);` statement) until it is aborted by the eager conflict mechanism.

The left side of Figure 5 shows the serial execution of the program.

III. PERFORMANCE ASSESSMENT OF PROOF-OF-CONCEPT PROTOTYPE

This section presents performance assessment of the prototype of STO using TSX for benchmarks from Parboil, SPEC CPU2006, and Mediabench-II.

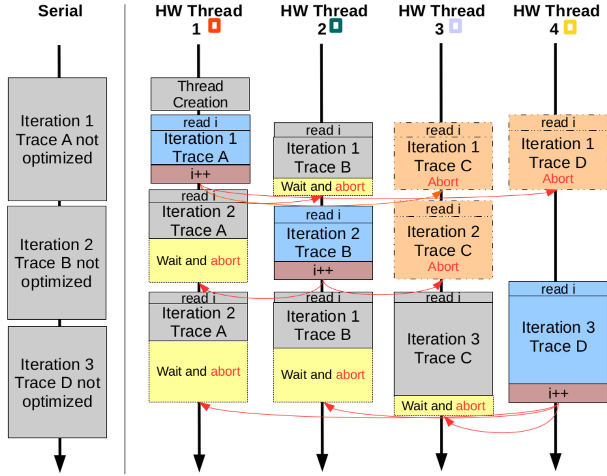

Fig. 5. Possible execution flow of STO Traces shown in Figure 3 on Intel TSX.

A. Benchmarks, Implementation, Settings, and Environment

To select programs for this initial experimental evaluation we searched for hot regions of code in the Parboil, SPEC CPU2006, and Mediabench benchmark suites to uncover code that could have potential to improve performance through STO. Profiling of **all benchmarks** from these three suites revealed hot loops that were then analyzed to determine if these loops are amenable to STO optimization. This analysis evaluates the number of traces in the loop and measures the trace hotness, execution probability and optimization potential.

Table II shows the benchmarks that contain loops for which STO is applicable at the moment. Except for `h263dec` that contains two STO loops, in all other benchmarks STO was applied to a single loop. The second column of the Table II indicates the Benchmark Suite that the program came from. The fourth column of Table II shows the locations/lines of the target regions in the source code. The fifth column shows the fraction of the total execution time ran by the hot code regions.

This initial evaluation of the prototype uses an Intel Core i7-4770 processor, running at 3.4 GHz, with 16 GB of memory on Ubuntu 12.04.3 LTS (GNU/Linux 3.8.0-29-generic x86_64). The Intel Core i7-4770 has 4 cores with 2-way SMT. 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`.

The following section discusses the performance evaluation comparing the STO execution with the serial execution of the same benchmark also compiled at level `-O3`. STO accelerates loops that may contain data dependences that prevent parallelization, thus comparison with the sequential code is appropriate. Whole-program executions are compared and not only the execution time of the region of the code to which STO is applied. Each benchmark was run 100 times.

B. Benchmark Results

Figure 6 shows the speedup of the selected benchmarks with respect to the sequential execution. The average performance improvement over 100 runs due to STO on TSX varies between 1% (for `458.sjeng`) and 9% (for `Sad`). Performance variability with a 95% confidence is also shown in Figure 6. The modest speedups are due to the overhead of privatization, padding, copying variables after commit (to simulate lazy conflict resolution),

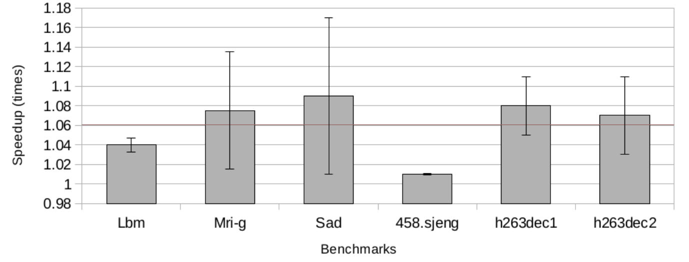

Fig. 6. Speedup of benchmarks with respect to serial execution.

and mainly due to the expensive cost of aborts in Intel TSX (an abort costs 150 cycles).

At each iteration only one transaction (trace) should commit and the others must abort, thereby the number of aborts by conflict at each iteration should be *number_of_traces-1*. Moreover, the cause of an abort should be, in most cases *memory conflicts*. Aborts due to other reasons, such as capacity and interruptions, occur occasionally. In such cases the transaction has to retry.

Figure 7 shows the abort ratio, computed as the number of aborted transactions divided by the number of started transactions, for the benchmarks. This ratio fluctuate between 35% and 68%. It also shows cause of the abort: conflict, waiting and others. Aborts due to *waiting* (while the transaction is executing `while(1)`) are due the OS quantum allocated to the thread and not due to memory conflicts. Almost all aborts are caused by memory conflicts, the other reasons are almost insignificant.

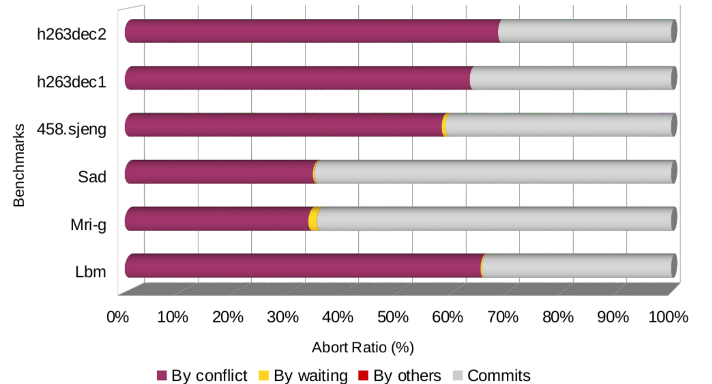

Fig. 7. Abort Ratio (%) of the benchmarks.

TABLE III
CONFLICT ABORTS AND COMMITS OF LBM TRACES

Conflict Aborts	Trace A (commits)	Trace B (commits)	Trace C (commits)	Total Commits	Factor
Trace A (1806K)		26K	1789K	26K+1789K =1815K	*1806/1815 =0.995*
Trace B (1995K)	343K		1789K	2132K	*0.936*
Trace C (258K)	343K	26K		369K	*0.700*

An interesting question is whether most conflict aborts are due to the commit of a transaction (lazy conflict resolution) or are they due to other causes (*e.g.*, false sharing). To provide some insight, Table III shows the number of commits and the number of conflict aborts for each trace in `Lbm`.

The number of conflict aborts of a given trace should be almost equal to the sum of commits of the other traces because each time that a trace with all predicates true commits, all other traces must abort. For example, consider the case of Trace A in `Lbm`

TABLE II
AMENABLE LOOPS TO STO

Benchmark	Origin	Description	Location in source code	% Coverage
`Lbm`	Parboil	A fluid dynamics simulation of an enclosed, lid-driven cavity.	`lbm.c`, 186	93%
`Mri-g`	Parboil	Computes a regular grid of data representing an Magnteic Resonance scan.	`CPU_kernels.c`, 174	71%
`Sad`	Parboil	Sum of absolute differences kernel, used in MPEG video encoders.	`sad_cpu.c`, 81	83%
`458.sjeng`	SPEC CPU 2006	Based on Sjeng 11.2, which is a program that plays chess and several chess variants.	`neval.c`, 493	23%
`h263dec`	Mediabench-II	A video decoder (h263dec) based on the ITU H.263 standard targeting video compression.	`store.c`, 400	45%
			`store.c`, 442	35%

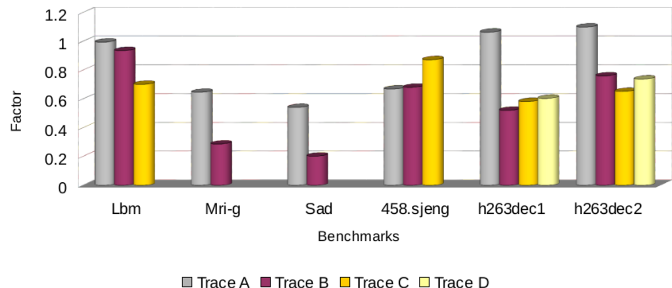

Fig. 8. Factor for each trace in the benchmarks.

shown in Table III. It should have aborted 1815K times (sum of commits of traces B and C) but it aborted 1806K times by conflict, resulting in a (real/expected) ratio of 0.995. This small difference is explained by the fact that some threads occasionally skip an iteration because they do not start before the correct thread commits and updates the induction variable, as explained in Section II-A. Table III shows similar results when considering the other traces of `Lbm`. Measurements for the other benchmarks reveal similar results as in the `Lbm` case. The ratios for each trace of each benchmark are shown in Figure 8. Most ratios are closer to or less than one, meaning that the number of conflict aborts for each trace is closer to or less than the number of commits of the other traces and thus the impact of other conflict abort causes is imperceptible, as expected.

IV. RELATED WORK

Traces have been used for traditional optimizations. Fisher was the first to introduce the concept of traces and to use it for instruction scheduling [12]. *Trace Scheduling* is a global compaction technique in contrast with local compaction techniques whose domain is a basic block of code. Static Trace Scheduling involves selecting traces and scheduling instructions on these traces trying to increase ILP, and improving the performance on a single processor. STO differs from this approach because it collects all traces and speculatively optimizes and executes them on an HTM system trying to improve the performance on multiple processors.

Neelakantam *et al.* proposed that microprocessors provide hardware primitives for atomic execution to increase the effectiveness of speculative compiler optimizations [13]. Thus, the compiler may speculatively optimize a program's hot path in isolation as a superblock. Atomic execution guarantees that if a miss-speculation is produced, the control is transferred to a non-speculative version of the code, relieving the compiler from generating compensation code. . These optimizations result in 10-15% average speedup. STO differs from this in that, to carry out the speculative compiler optimizations, it speculates in parallel all possible traces within a hot-loop iteration using a real HTM.

V. CONCLUSIONS AND FUTURE WORK

This paper introduces STO, a technique to enable speculative optimization and execution of traces from hot loops. It also describes the creation of a prototype for an initial evaluation of STO using Intel TSX. This evaluation uses six benchmarks, which were modified to enable STO under TSX. The initial performance results produced overall speed improvements varying from 1% to 9%. Another contribution of the paper is a discussion of the features that would be necessary in hardware to enable STO, such as multi-versioning cache, eager conflict detection, lazy conflict resolution, and pausing transaction. An HTM with such features would lead to significantly higher speed gains due to STO.

The focus of this paper is to present the idea of STO. Automatic transformation of code to use STO is future work. Impacts on power consumption, memory traffic, and chip area will require a detailed architectural simulation of the features required for STO.

REFERENCES

[1] *Intel architecture instruction set extensions programming reference. Chapter 8: Intel transactional synchronization extensions*, Intel Corporation, 2012.
[2] *Intel Xeon Processor E3-1200 v3 Product Family Specification Update August 2014 Revision 008*, Intel Corporation, 2014.
[3] *Power ISA Transactional Memory*, IBM, 2012. [Online]. Available: www.power.org/wp-content/uploads/2012/07/PowerISA_V2.06B_V2_PUBLIC.pdf
[4] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G.-T. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q compute chip," *IEEE Micro*, vol. 32, no. 2, pp. 48–60, March-April 2012.
[5] R. Bodík and R. Gupta, "Partial dead code elimination using slicing transformations," in *Programming Language Design and Implementation (PLDI)*, Las Vegas, Nevada, USA, 1997, pp. 159–170.
[6] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for vliw and superscalar compilation," *The Journal of Supercomputing*, vol. 7, no. 1-2, pp. 229–248, 1993.
[7] C. G. Ritson and F. R. Barnes, "An evaluation of intel's restricted transactional memory for cpas," *Communicating Process Architectures 2013*, pp. 271–291, 2013.
[8] S. Tomic, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm: Eager-lazy hardware transactional memory," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, New York, NY, USA, 2009, pp. 145–155.
[9] A. Shriraman, S. Dwarkadas, and M. Scott, "Flexible decoupled transactional memory support," in *International Conference on Computer Architecture (ISCA)*, Beijing, China, June 2008, pp. 139–150.
[10] A. Wang, M. Gaudet, P. Wu, M. Ohmacht, J. N. Amaral, C. Barton, R. Silvera, and M. M. Michael, "Evaluation of Blue Gene/Q hardware support for transactional memories," in *Parallel Architecture and Compilation Techniques (PACT)*, Minneapolis, MN, USA, September 2012, pp. 127–136, best Paper Award.
[11] ——, "Software support and evaluation of hardware transaction memory on blue gene/q," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 233–346, January 2015.
[12] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. 30, no. 7, pp. 478–490, 1981.
[13] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles, "Hardware atomicity for reliable software speculation," in *International Conference on Computer Architecture (ISCA)*, San Diego, California, USA, 2007, pp. 174–185.