This is the accepted manuscript of a paper published in
Proc. 25th IEEE Int'l Conf. on Electronics Circuits and Systems (ICECS 2018)
Copyright (C) 2018 IEEE

Isolation-Safe Speculative Access Control for Hardware Transactional Memory

Tomoki TAJIMI*, Masaki HAYASHI*, Yuki FUTAMASE*,
Ryota SHIOYA†, Masahiro GOSHIMA‡ and Tomoaki TSUMURA*

*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

†The University of Tokyo
7-3-1, Hongo, Bunkyo, Tokyo, Japan
Email: shioya@ci.i.u-tokyo.ac.jp

‡National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda, Tokyo, Japan
Email: goshima@nii.ac.jp

***Abstract*—On shared-memory systems such as multi-core processors, lock-based mechanisms have been used in parallel programs to exclude conflicting accesses on shared variables. Transactional memory (TM) is a new paradigm to arbitrate conflicting accesses and is expected to be an alternative to the traditional lock-based mechanisms. Transactional memory speculatively executes transactions in parallel and cancels the execution when access conflicts are detected between them. Because canceling the speculative execution has an impact on TM performance, it is advisable to reduce conflicts. We found that some of the accesses, that are judged as conflicting accesses by TM, need not be denied and can be granted. In this paper, we propose a novel execution control for TMs for speculatively granting a certain type of access requests. The result of the evaluation shows that the execution cycles can be reduced by 63.6% at a maximum and 38.8% on average.**

I. Introduction

Transactional memory (TM)[1] is a promising mechanism for improving both of productivity and performance of parallel processing programs on shared-memory systems such as multi-core processors. It can be complementary and/or alternative to traditional lock-based mechanisms that are awkward and intractable for programmers. Hardware transactional memories (HTMs), hardware implementations of TM, are now widely installed to the several latest processors, such as IBM zEC12, Power8, and Intel Haswell. Programmers define critical sections as *transactions*, and the transactions can be speculatively executed in parallel by HTMs, while traditional lock-based systems completely serialize them.

HTMs dynamically detect access conflicts on shared variables between transactions and cancel speculative execution of transactions. Because the canceled execution comes to nothing and causes some performance overhead for rolling back the transaction execution, how to avoid and reduce access conflicts is very important for the performance of HTMs.

Many studies have been conducted on improving HTM performance. Some[2], [3] of them are on partial rollback that reduces the re-execution overhead of transactions, and others[4], [5], [6] are on adaptive transaction control that follows application behavior. We also proposed some transaction scheduling based on conflict predictions[7], [8], [9]. However, some type of programs still suffer many access conflicts and large performance overhead, and they should be relieved.

Transactions must guarantee several properties including Atomicity and Isolation, and access requests that may violate them are denied by HTMs. However, we found that some of the requests can be granted without violating Isolation, and this can improve the HTM performance. In this paper, we aim to make the following contributions:

1) We carefully examine a typical benchmark program and disclose that some access requests can be granted without violating Isolation.
2) We propose a novel transaction control for speculatively granting some access requests that may violate Isolation, and canceling the accesses when the speculation fails.
3) We evaluate our proposal and show that the execution cycles can be reduced by 63.6% at a maximum and 38.8% on average.

II. Transaction Control on HTM

HTM guarantees the following two properties for transactions by detecting memory accesses that violate them as *conflicting accesses*.

Atomicity: Each transaction must be executed completely and must not be executed partially.

Isolation: The result of the concurrent execution of transactions must be the same as that of sequential execution. Hence, values modified in a transaction should not be visible to other transactions before it is completely executed.

Figure 1 shows an example where how conflicting accesses are detected and resolved on LogTM [10], a typical HTM implementation. In this figure, *thr.*0 signifies a thread whose ID is '0' and *Tx.X* signifies a transaction whose ID is '*X*.'

Now, assume that *Tx.X* and *Tx.Y* have issued `store A` and `store B`, respectively, and *thr.*0 sends an access request for address B when *Tx.X* tries `store B` (at *t*1) in accordance with the coherence protocol. In this example, *thr.*0 receives *NACK* (at *t*3), because *Tx.Y* has already accessed on B and Isolation may be violated if the access is permitted. Receiving *NACK*, *Tx.X* stalls until the conflict is resolved. After then, assume that *Tx.Y* tries to access A and receives *NACK* (at *t*4). Since this situation where two transactions receive *NACK*

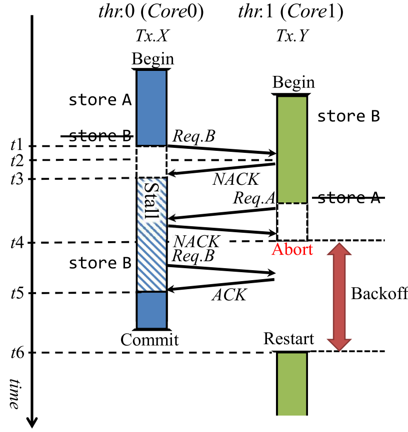

Fig. 1. Conflict detection and resolution on a typical HTM

from each other is a kind of deadlock, *thr.*1 aborts *Tx.Y* for avoiding deadlock (at *t*4), cancels its issued `store B`, and restores its state of before the beginning of *Tx.Y* (*roll-back*).

Because `store B` of *Tx.Y* is canceled, now `store B` of *Tx.X* is not a conflicting access and *Tx.X* can come out of stall and issue `store B` (at *t*5). In contrast, *Tx.Y* should restart after waiting a certain period (called *backoff*) to prevent from conflicting with *Tx.X* again (at *t*6). This is an outline of transaction control on general HTMs. As you can see in this example, conflicting access brings futile stalled cycles, and a stalled transaction can cause another conflict that leads to performance overhead resulting from abort, roll-back, and backoff. Hence, it is desirable for HTM performance to reduce conflicting accesses.

III. Permitting Conflicting Accesses

A. Conflicting Accesses that can be Permitted

With traditional HTM systems, an access request on a shared variable is denied for Isolation if the variable has been already accessed by another concurrent transaction. However, if the transaction never accesses the variable again until its commit, the request can be granted and the access can be permitted without violating Isolation. We explain it with the code shown in Fig. 2; this is a simplified code of Deque from GEMS microbench suite[11].

The portion between `BEGIN_TRANSACTION()` and `COMMIT_TRANSACTION()` is defined as a transaction. In this transaction, after the variable `count` is incremented (line 2), an element is enqueued or dequeued on the right or left end of the deque (double-ended queue). After a transaction accessed `count`, all accesses on `count` by other transactions are judged as *conflicting accesses* and denied for Isolation until the transaction commits or aborts. However, in this case, the transaction will never access `count` again before its commit, and other transactions can access `count` without violating Isolation because they can observe the same value with the committed value of `count` before the commit.

```
1  BEGIN_TRANSACTION(0);
2  count++;
3
4  switch(op){
5   case 0:   enqueue_right();    break;
6   case 1:   dequeue_right();    break;
7   case 2:   enqueue_left();     break;
8   case 3:   dequeue_left();     break;
9  }
10   :
11 COMMIT_TRANSACTION(0);
```

Fig. 2. Simplified code of Deque

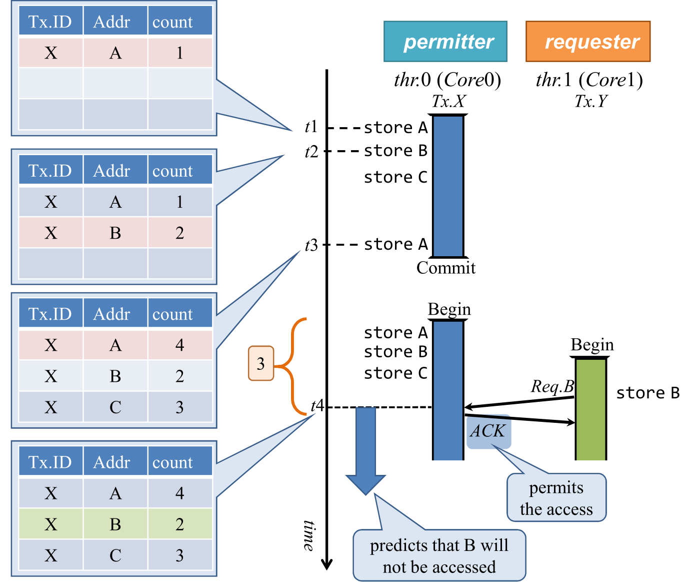

Fig. 3. How to permit a conflicting access

B. Speculative Control for Conflicting Accesses

As explained in Sec. III-A, if a transaction has completed its modification on a shared variable, or after the *last-touch* on the variable by the transaction, access requests on the variable from other transactions can be granted without violating Isolation. We propose some extensions for HTM to support such novel access control between transactions.

To exploit this idea, the *last-touch* on each shared variable in each transaction should be managed and maintained. We installed a counter and a small dedicated table to each processor core for this. When a transaction starts, the counter is reset. After then, on each memory access (load/store), the transaction increments the counter value and registers the tuple of the transaction ID, the accessed address, and the counter value on the dedicated table.

We explain the execution model with Fig. 3. After *thr.*0 starts *Tx.X*, it issues `store A` (at *t*1). Then, the counter is incremented to 1, and the tuple {X,A,1} is registered to the table. The transaction registers {X,B,2} when it issues `store B` (at *t*2), and updates the registered {X,A,1} with {X,A,4} at the second access to A (*t*3).

After *Tx.X* is committed once and the *last-touch* of each accessed addresses are registered on the table, *thr.*0 can predict whether *Tx.X* has completed its modification on each address.

As shown in the example of Fig. 3, assume that *thr.*0 receives an access request on B through the next execution of *Tx.X* (at *t*4). In this case, *Tx.X* has already issued three memory accesses and *thr.*0 can tell that the *last-touch* on B is the second memory access in *Tx.X* from the table. Hence, *thr.*0 knows that it can grant the request without violating Isolation.

This is the concept of our novel transaction execution control for HTM. On HTM, execution of a transaction is naturally speculative, and our proposal boosts its speculativeness by speculatively permitting some conflicting accesses. We call a transaction that speculatively grants a conflicting access request *permitter*, and the sender of the request *requester*.

C. Control for Isolation when Speculation Fails

A *requester* is speculatively permitted its conflicting access on the presupposition that the *permitter* has completed its modification on the shared variable, will commit, and the modified value of the shared variable will be persistent. If this speculation fails, or the *permitter* accesses the variable again or aborts, the execution of the *requester* must be canceled for Isolation. We implement two coherence messages for it:

Req.Abort for requesting receiver to abort its transaction.
Committed for announcing that sender's transaction has committed.

When a *permitter* touches a shared variable after the predicted *last-touch* or the *permitter* aborts, it needs to send *Req.Abort* to the *requester*. In contrast, the *requester* must postpone its commit until the corresponding *permitter* commits even if it instantly can, because the *requester* may receive *Req.Abort* before the *permitter*'s commit. Hence, the *permitter* ought to send *Committed* to the *requester* when committing, and the *requester* can commit after receiving it.

Another control is required for roll-back. As explained, when the *permitter* aborts, the corresponding *requester* also should abort, and they should roll-back together. In that case, the state that should be restored is one of before the beginning of not *requester* but *permitter*. Hence, the *requester* should roll-back first, and after then, the *permitter* should roll-back.

IV. Performance Evaluation

A. Evaluation Setup

We used a full-system execution-driven functional simulator Wind River Simics [12] in conjunction with customized memory simulators built on Wisconsin GEMS [11] for evaluation. The detailed configuration of the simulated processor is shown in TABLE I. The topology and the link latency of the interconnect network are defined as same as LogTM [10]. We used Btree, Contention, and Deque from GEMS microbench suite, Raytrace from SPLASH-2 benchmark suite [13], and Kmeans from STAMP benchmark suite[14] as the workloads. We have evaluated the execution cycles of them with 16 threads.

B. Evaluation Results

The evaluation results with the following two HTM configurations are shown in Fig. 4. It shows the total sum of execution cycles of all 16 threads and its breakdown. Each bar is normalized to the result of baseline (B).

(B) LogTM (baseline)
(P) HTM with the proposed execution control; speculatively grants conflicting accesses that may not violate Isolation.

When simulating multi-threaded execution on a full-system simulator, performance variability must be considered [15]. Hence, we tried 10 times on each benchmark and measured 95% confidence interval, which is illustrated as an error bar.

The legend in Fig. 4 shows the breakdown items of the total sum of cycles. They represent the waiting cycles for receiving *Committed* message (*Wait*), the barrier synchronization cycles (*Barrier*), the stalled cycles (*Stall*), the exponential backoff cycles (*Backoff*), the aborting overheads (*Aborting*), the execution cycles in the transactions that are aborted/committed (*Bad_trans/Good_trans*), and the execution cycles out of transactions (*Non_trans*).

As a result of the evaluation, (P) reduces the execution cycles 63.6% at a maximum, and 38.8% on average with respect to (B). We go to the detailed examination of some results in the following section.

TABLE I
SPECIFICATIONS OF THE SIMULATED PROCESSOR

Processor	SPARC V9
#cores	32 cores
clock	4 GHz
issue width	single
issue order	in-order
non-memory IPC	1
L1 cache	32 KBytes
ways	4 ways
latency	3 cycles
L2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

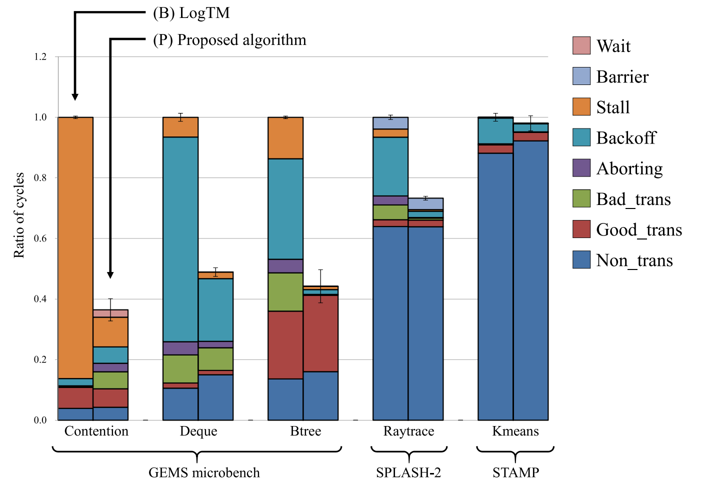

Fig. 4. The normalized total sum of execution cycles

```
int A[1];
int B[1024];

BEGIN_TRANSACTION(0);
for( i = 0; i < 10; i++ ){
  if( access_type[i] == READ )
    var = A[0];
  else
    A[0] = 0;
}
// A[0] is never accessed after here.
for( i = 10; i < 100; i++ ){
  if( access_type[i] == READ )
    var = B[index[i]];
  else
    B[index[i]] = 0;
}
COMMIT_TRANSACTION(0);
```

Fig. 5. Simplified code of Contention

```
BEGIN_TRANSACTION(0);
while(!node->isLeaf){
  child = BtreeNode_findChild();
    :
    :
  node = child;
}
    :
COMMIT_TRANSACTION(0);
```

Fig. 6. Simplified code of Btree

C. Detailed Examination

Contention has a transaction that has a similar structure to the transaction of Deque shown in Fig. 2. Figure 5 shows a simplified code of the transaction in Contention. In this transaction, `A[0]` is accessed in the former `for` loop, and then, never accessed in the latter `for` loop. On general HTMs, a transaction can not access `A[0]` if a concurrent transaction is executing the latter `for` loop. In contrast, such an access can be speculatively granted with our proposed execution control, and this largely improves the performance of (P).

Btree's performance is also drastically improved with our proposal as shown in Fig. 4. Figure 6 shows a simplified code of a transaction in Btree; the transaction is for searching leaf nodes in a B-tree. In this transaction, nodes are traced from a parent to a child until a leaf node is found by the `while` loop. In this loop, each node is accessed in a certain iteration and then never accessed again. The proposed execution control mechanism counts memory accesses dynamically, and can properly manage the *last-touch* on each node. This allows the conflicting accesses on each node to be speculatively granted and the performance of (P) should be improved

Raytrace has a transaction that costs much execution cycles. In such a transaction, many cycles can be reduced by speculatively granting conflicting accesses, and Fig. 4 shows that the sum of execution cycles except Non_trans is reduced to nearly a fourth with (P).

The total performance of **Kmeans** is not much improved because Non_trans occupies large, but the sum of execution cycles except Non_trans is reduced to less than half with (P).

D. Additional Hardware Cost

This section discusses the hardware cost of the dedicated table for managing *last-touch*. Each table entry has 3-bit, 26-bit, and 13-bit width fields for storing the transaction ID, accessed cache line address, and the counter value, respectively. In a transaction of Btree, 15,018 cache lines are accessed, and the table needs $(3+26+13)\text{bits} \times 15{,}018 \fallingdotseq 77.1\text{KBytes}$ per thread for managing them all. This is rather large and we should examine the trade-off between the capacity and performance.

V. Conclusion

In this paper, we disclosed that some conflicting accesses do not violate Isolation on hardware transactional memory, and proposed an execution control for speculatively allowing such accesses. We also designed a mechanism to guarantee Isolation when the speculation fails. We implemented it on a simulator and the evaluation results show that the execution cycles can be reduced by 63.6% at a maximum. Evaluation with a large variety of programs and finding the optimum table configuration are left for our future work.

Acknowledgment

This work was partially supported by JSPS KAKENHI Grant Numbers JP17H01711, JP17H01764, and JP17K19971.

References

[1] M. Herlihy *et al.*, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in *Proc. 20th Int’l Symp. on Computer Architecture (ISCA’93)*, May. 1993, pp. 289–300.
[2] M. J.Moravan *et al.*, “Supporting Nested Transactional Memory in LogTM,” in *Proc. 12th Int’l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006, pp. 1–12.
[3] A. McDonald *et al.*, “Architectural Semantics for Practical Transactional Memory,” in *Proc. 33rd Annual Int’l Symp. on Computer Architecture (ISCA’06)*, 2006, pp. 53–65.
[4] A. Shriraman *et al.*, “Flexible Decoupled Transactional Memory Support,” in *Proc. 35th Annual Int’l Symp. on Computer Architecture (ISCA’08)*, 2008, pp. 139–150.
[5] S. Tomic *et al.*, “Eazyhtm, Eager-lazy Hardware Transactional Memory,” in *Proc. 42nd Annual IEEE/ACM Int’l Symp. on Microarchitecture (MICRO-42)*, 2009, pp. 145–155.
[6] M. Lupon *et al.*, “A Dynamically Adaptable Hardware Transactional Memory,” in *Proc. 43rd Annual IEEE/ACM Int’l Symp. on Microarchitecture (MICRO-43)*, 2010, pp. 27–38.
[7] A. Hirota *et al.*, “A Concurrency Control in Hardware Transactional Memory Considering Execution Path Variation,” in *Proc. 4th Int’l Symp. on Computing and Networking (CANDAR’16)*, Nov. 2016, pp. 77–83.
[8] K. Mashita *et al.*, “A Waiting Mechanism with Conflict Prediction on Hardware Transactional Memory,” *IEICE Trans. on Information and Systems*, vol. E99-D, no. 12, pp. 2860–2870, Dec. 2016.
[9] T. Tajimi *et al.*, “Initial Study of a Phase-Aware Scheduling for Hardware Transactional Memory,” in *Proc. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing (PacRim 2017)*, 2017.
[10] K. E. Moore *et al.*, “LogTM: Log-based Transactional Memory,” in *Proc. 12th Int’l Symp. on High-Performance Computer Architecture (HPCA’06)*, Feb. 2006, pp. 254–265.
[11] M. M. K. Martin *et al.*, “Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.
[12] P. S. Magnusson *et al.*, “Simics: A Full System Simulation Platform,” *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[13] S. C. Woo *et al.*, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proc. 22nd Int’l. Symp. on Computer Architecture (ISCA’95)*, 1995, pp. 24–36.
[14] C. C. Minh *et al.*, “STAMP: Stanford Transactional Applications for Multi-Processing,” in *Proc. IEEE Int’l Symp. on Workload Characterization (IISWC’08)*, Sep. 2008.
[15] A. R. Alameldeen *et al.*, “Variability in Architectural Simulations of Multi-Threaded Workloads,” in *Proc. 9th Int’l Symp. on High-Performance Computer Architecture (HPCA’03)*, Feb. 2003, pp. 7–18.