This is the accepted manuscript of a paper published in
Proc. 2nd IEEE Nordic Circuits and Systems Conf. (NorCAS 2016)

Exclusive Control for Compound Operations on Hardware Transactional Memory

Keisuke MASHITA*, Anju HIROTA* and Tomoaki TSUMURA*
*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—Transactional memory (TM) is a lock-free synchronization mechanism for shared memory systems, and it is a promising paradigm for complementing or replacing conventional lock-based techniques. On TM, read-after-read (RaR) accesses cause no conflict and do not prevent parallel speculative execution of transactions. However, quite a few of read accesses are followed by write accesses to the same variables or addresses, for example in compound operations such as increment or decrement. We found that granting such RaR access requests causes futile stalls which impact TM performance seriously. In this paper, we propose a novel effective transaction scheduling for hardware transactional memories by controlling such RaR accesses with very small additional hardware costs. If an RaR access to an address is expected to be followed by a write access to the same address, the transactions concerned with the access are serialized. The result of the experiment shows that the execution time of HTM is reduced 72.2% at a maximum and 17.5% on average with our transaction scheduling.**

I. Introduction

As electric power consumption and calorific power are increasing, and semiconductor devices keep downscaling, it becomes difficult to raise clock frequencies of microprocessors. In response to this distress, multi-core processors now attract a great deal of attention. On multi-core processors, multiple threads run in parallel for speed-up.

For running multiple threads in parallel on shared memory systems, mutual exclusion is required, and *lock* has been commonly used. However, lock-based methods can cause deadlocks, and this leads to poor scalability and high complexity. To solve these problems, *transactional memory* [1] has been proposed as a lock-free synchronization mechanism.

On HTM, hardware implementation of transactional memory, transactions are executed speculatively as long as there is no conflict on shared variables, and read-after-read (RaR) access requests are always granted because they do not bring any conflict. However, a read access is often followed by a write access to the same variable, or address. Hence a conflict will be brought by the write access after all, and some execution or stall cycles turn into completely futile. We have raised this problem and proposed a primitive solution to address this problem [2]. In this paper, we discuss this problem more concretely, analyzing some programs, and propose a new practical method and its light-weight implementation to solve this problem. We propose a novel effective transaction scheduling for HTMs by detecting such RaR accesses and serializing the transactions that include such RaR accesses.

In this paper, we aim to make the following contributions:

1) We disclose that not a few read accesses are followed by write accesses to the same shared variables, and explain RaR requests on such read accesses can bring serious futile stalls and aborts.
2) We propose a novel transaction scheduling for eager/eager HTMs to solve the problem. It is the first *practical* solution which can be implemented with a very simple mechanism, and we show that the additional hardware cost for the implementation is reasonably small.
3) We evaluate the transaction scheduling, and show the results that the execution cycles can be reduced 72.2% at a maximum and 17.5% on average with the proposed transaction scheduling.

II. Related Work

So far, various scheduling techniques for HTM have been proposed. To improve the performance of parallel executions, Yoo et al. [3] proposed a method which brings the concept of adaptive transaction scheduling (ATS) in TM. ATS can improve the performance of workloads that lack for parallelism because of frequent contentions by dynamically dispatching transactions and controlling the total number of concurrent transactions using runtime feedbacks.

Geoffrey et al. [4] proposed a method that focuses on common memory location accessed in multiple transactions. In the method, locality of memory access on each consecutive execution is called similarity, and the similarity is calculated with a Bloom filter. If the similarity exceeds a threshold value, the transactions are serialized. Akpinar et al. [5] have proposed some novel ideas for conflict resolution policies in HTMs such as alternating priorities of transactions in various ways based on the total number of stalled or aborted transactions.

To reduce energy consumption, Gaona et al. [6] proposed a method that serializes transactions when a conflict arises. Then, if a conflicted transaction has finished, the transaction wakes up a transaction of the highest priority among all of the stalled transactions.

In contrast to these methods, we focus on what type of access pattern to shared variables will have a bad influence

on HTM performance, and have proposed a scheduling for controlling harmful read-after-read accesses.

Bobba et al. [7] took notice of access ordering on each shared variable, and the solution is partially similar to our proposal. They proposed *Store Predictor*, a mechanism for managing addresses that are written after being read. When reading an address which is managed by *Store Predictor*, each thread sends a *write* request instead of a read request. Thereby, a thread that has already accessed to the address detects an ostensible write-after-read (WaR) or write-after-write (WaW) conflict, and sends back a NACK. In this way, the bad influence of RaR accesses can be partially lightened. However, *Store Predictor* has two serious problems. One problem is that the mechanism implementation for managing target addresses is almost not discussed. Both the structure and access overhead of a lookup table for the target addresses are not explained, and the practicality is not clear. The other problem is that *Store Predictor* does not consider execution path variation of transactions. Hence, even once an address is written after being read, any RaR access to the address is stalled even when they will not bring conflicts. This may cause severe performance degradation. We will compare the performance between *Store Predictor* and our proposal in Section V.

III. Exclusive Control for Compound Operations

In this section, we argue a problem of HTM, and propose a method for solving the problem.

A. Futile Stalls

A read access on a shared variable will be followed by a write access on the same variable in many cases. For example, the statements with increment/decrement, and compound assignment operators can cause such situations. When a transaction running in parallel with other transactions has such a *compound operation*, even if a read access on a shared variable is allowed by other transactions, the counterpart write access will cause a conflict with other transactions afterwards. This can degrade the performance of HTM severely.

Figure 1 shows an example where two threads *Thr.1* and *Thr.2* execute the same transaction *Tx.X* in parallel. Assume that *Tx.X* has a compound operation. After both the threads execute `load A`, *Thr.2* tries `store A` and a conflict is detected. At this time, in transactional memories employing *eager conflict detection*, such as LogTM [8], *Thr.2* receives NACK and stalls its *Tx.X* (t1). Afterwards, when *Thr.1* tries `store A` (t2), another conflict is detected because *Thr.2* has already accessed the address, and *Thr.1* receives NACK. In this case, because *Tx.X* being executed by the NACK sender *Thr.2* is elder than *Tx.X* by *Thr.1*, *Thr.1* aborts its *Tx.X* (t3), and *Thr.2* can continue *Tx.X*. In this case, *Thr.2* has stalled a while, but *Tx.X* on *Thr.1* was aborted and did not proceed at all. In other words, the stall of *Thr.2* becomes completely *futile*. Such stall is called *futile stall* [7], and one of the major reasons of the performance degradation.

When a transaction that reads a shared variable and afterwards writes to the same variable is executed by multiple threads in parallel, all the threads can read the shared variable early in the transactions. It is because read-after-read (RaR) accesses never cause conflicts, and are always allowed by HTM. However, the following write accesses cause conflicts after all. This is the major cause of futile stalls.

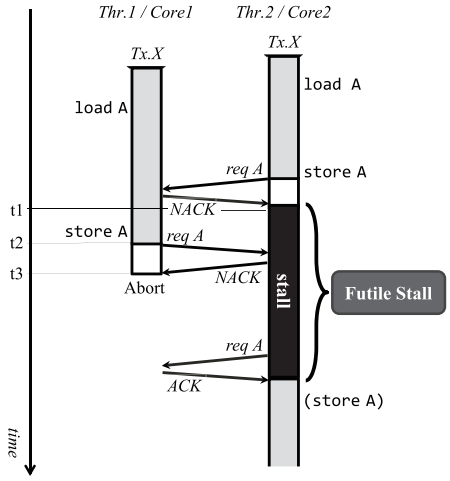

Fig. 1. Futile stall caused by a compound operation.

TABLE I
Percentage of the aborts caused by compound operations.

GEMS microbench			
Btree	Contention	Deque	Prioque
39.6%	80.9%	92.9%	99.3%
SPLASH-2			
Barnes	Cholesky	Radiosity	Raytrace
52.2%	92.8%	97.6%	99.4%

We simulated some programs from benchmark suites; GEMS microbench [9] and SPLASH-2 [10], which are commonly used for evaluating transactional memory systems, and examined how harmful to the performance of HTM such RaR accesses are. The simulation environment is the same as described in Section V later.

We investigated how much aborts are caused by write requests after read accesses to the same variables. Table I shows the percentage of such aborts in the total number of aborts, when each benchmark program is executed in 16 threads. The configuration of the processor is shown in Table III later. As shown in the table, the percentages are large for all the programs, and even over 90% for five programs out of eight.

Table II shows three representative addresses, where many aborts are caused by compound operations, for each programs from SPLASH-2 benchmark suite. It is found that even one address can be implicated in the large number of aborts, when the address is written after read in a transaction.

These results imply that, when a read access on a shared variable will be followed by a write access to the same shared variable, granting RaR requests to such a variable can cause heavily repeated aborts. Repeated aborts will raise backoff cycles, and significantly degrade the performance of HTM. Therefore, such RaR accesses should be controlled.

TABLE II
THE RANKINGS FOR THE NUMBER OF ABORTS, WHICH ARE CAUSED BY COMPOUND OPERATIONS, BY TARGET ADDRESS. (SPLASH-2)

program	Tx. ID	address	#aborts
Barnes	*Tx.2*	0xb4265ac0	361
	Tx.2	0xb4265a80	134
	Tx.3	0xd9af4e00	56
	total #aborts		1,568
Cholesky	*Tx.17*	0xd51d6240	99
	Tx.16	0xd51d6240	66
	Tx.16	0xd51d6280	45
	total #aborts		348
Radiosity	*Tx.17*	0xe6ddf140	13,628
	Tx.0	0xe1534940	140
	Tx.3	0xe1470900	109
	total #aborts		14,454
Raytrace	*Tx.19*	0xe144a300	45,257
	Tx.16	0xe144a300	36,368
	Tx.18	0xe144a300	11,296
	total #aborts		93,556

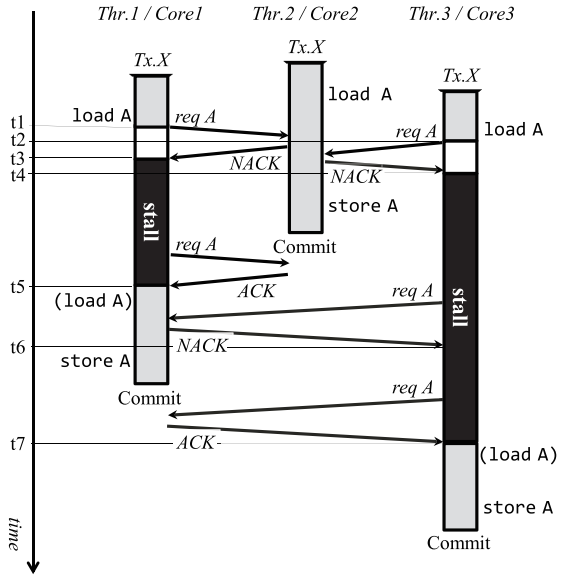

Fig. 2. Exclusive control for a compound operation.

B. Exclusive Control Model

To address this problem, we propose a novel transaction scheduling in this paper for controlling RaR accesses. Specifically, an RaR access request will be answered with NACK, if the access is expected to be followed by a write request to the same address. This can achieve exclusive control for any instruction region from a read access on a shared variable to a write access on the same variable, and prevent futile stalls.

Figure 2 shows an execution model of our proposal. In this example, three threads (*Thr.1* to *3*) are executing the transaction *Tx.X* speculatively in parallel. *Tx.X* includes a compound operation, namely a read access and a subsequent write access to an address A.

First of all, *Thr.2* reads from A, and then, *Thr.1* and *Thr.3* also try to read from A by sending read requests for A (t1, t2). In traditional eager HTM models, these requests are granted by the directory, and both *Thr.1* and *Thr.3* can read from A. In contrast, in our HTM model, if the directory predicts that the read access on A in *Tx.X* will be followed by a write access on A, NACKs are sent back to *Thr.1* and *Thr.3*. Receiving NACKs (t3, t4), the read accesses are deferred, and both of *Thr.1* and *Thr.3* are stalled

As a result, when *Thr.2* tries to write on A, no conflict is brought and *Thr.2* can advance its execution of *Tx.X* without any futile stall. After *Thr.2* commits its *Tx.X*, the read request from *Thr.1* is granted (t5), and after *Thr.1* commits its *Tx.X*, *Thr.3* can return from its stall (t7).

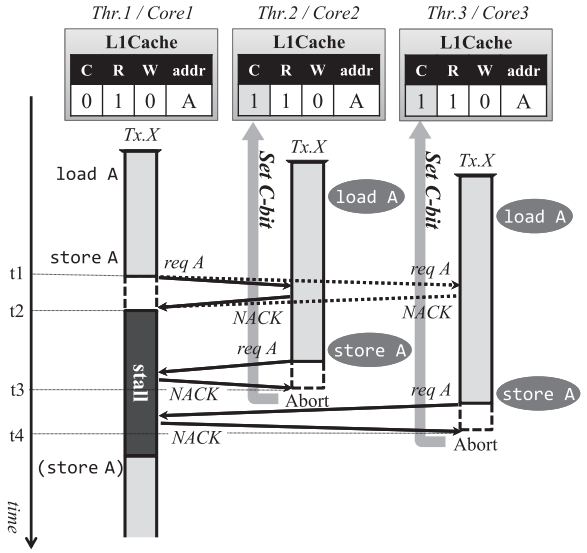

Fig. 3. How to set C-bit.

IV. IMPLEMENTATION

In this section, we explain the implementation of the transaction scheduling proposed in Section III, and its detailed execution model. We adopt LogTM [8] as the base model for simplicity in this paper, but our model can be implemented on any eager/eager HTM in a similar way.

A. Additional Hardware

In LogTM, each cache line in the L1 cache has R-bit and W-bit field for detecting conflicts. Besides these two bit fields, we use additional 1-bit field called *C-bit*. The value of C-bit in a cache line represents whether the line has been read and afterwards written from the same thread. When a thread writes on a cache line that the thread has read in the current transaction and whose R-bit is set, the thread sets the C-bit of the cache line. Receiving a read request, a thread tests C-bit and R-bit of the requested cache line, and decides whether it should send NACK or ACK.

Now, we discuss the additional hardware cost for C-bits. C-bit is the 1-bit field for each cache line. Assume that the number of cores is 32, L1 cache is 32 kB, cache line size is 64 Bytes, and the number of ways is 4. In this case, the additional hardware cost for C-bits is $32 \times 1024/(4 \times 64) = 128$ bits for one core, and only 512 Bytes for whole a 32-core processor.

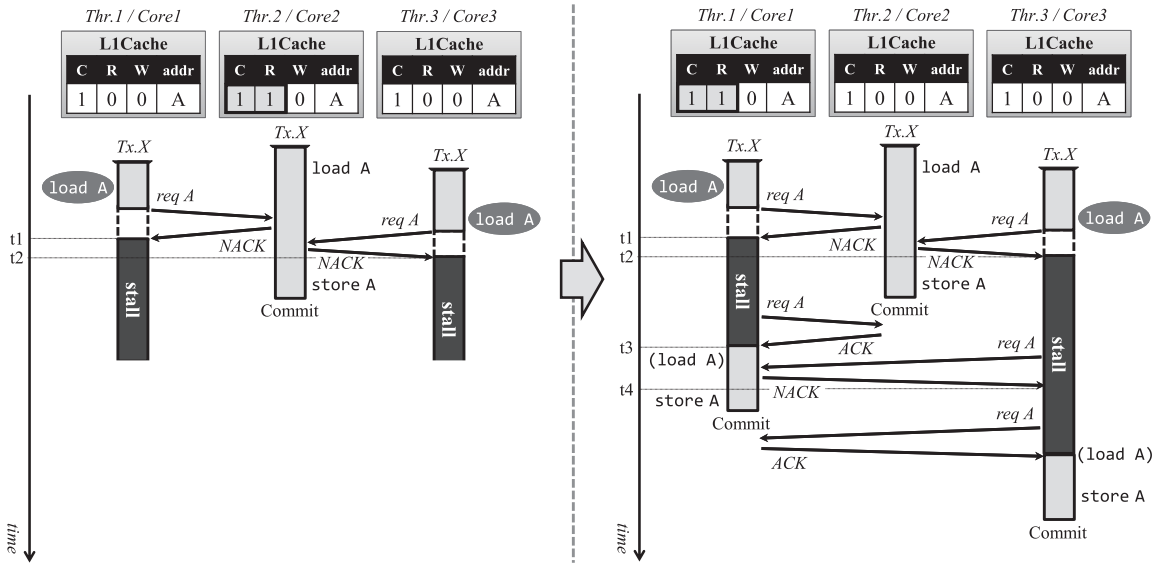

Fig. 4. Testing C-bits.

B. Execution Model

1) How to set C-bits: First, we explain how to set C-bit by using an example shown in Fig. 3, where three threads *Thr.1* to *3* are executing the same transaction *Tx.X*.

After three threads execute `load A`, *Thr.1* tries `store A` (t1). This write access is write-after-read (WaR) and causes a conflict. Hence, *Thr.2* and *Thr.3* send NACKs to *Thr.1*, and *Thr.1* stalls its *Tx.X* (t2). After that, each of *Thr.2* and *Thr.3* tries `store A`, detects a WaR conflict, and aborts its own *Tx.X*. When aborting its own *Tx.X*, each of *Thr.2* and *Thr.3* tests R-bit of the cache line including A. If R-bit is set, the thread finds that the thread itself has read the cache line before writing on the same cache line, and set C-bit of the line (t3 and t4).

2) How to test C-bits: As explained in IV-B1, C-bits are set. Next, we explain how to use the value of C-bits for controlling harmful RaR accesses.

Figure 4 shows an example, where three threads *Thr.1*$\sim$*3* are executing the same transaction *Tx.X*, and C-bit for the address A on each cache has already been set. First, *Thr.2* executes `load A`, and after that, each of *Thr.1* and *Thr.3* tries `load A` and sends a read request for A to *Thr.2*. Receiving this request, *Thr.2* tests C-bit and R-bit of the cache line of A. When both bits are set, or $C_A \wedge R_A = 1$, *Thr.2* predicts that, even if it grants the read request now, the request sender will send a new *write request* soon, and a conflict will be brought after all. Therefore, *Thr.2* replies to the read request with NACK. Receiving NACK, each of *Thr.1* and *Thr.3* stalls its *Tx.X* (t1 and t2), and *Thr.2* can commit *Tx.X* without conflicts and stalls. After *Thr.2* commits *Tx.X*, in this example, *Thr.1* receives ACK (t3), and *Thr.3* keeps stalling (t4).

3) How to clear C-bits: In this section, we describe how and when C-bits are cleared. As described in the Section IV-B2, each C-bit represents whether its corresponding cache line is read first and written afterwards, and tested for deciding whether the transactional operations to the cache line should be serialized or not. However, an address, which is read and afterwards written in a transaction, may only be read in another transaction. Furthermore, such an address may only be read and not written even in the same transaction, when the execution path in the transaction varies because of branch instructions. Consequently, a transaction can only read a certain cache line whose C-bit is set, and this leads to performance degradation because of unnecessary serialization.

Hence, it is very important to clear C-bits appropriately and periodically. On the proposed HTM model, when a thread commits a transaction, the thread tests the W-bit of a cache line whose C-bit is set. If the W-bit is not set, the thread can find that the cache line has been only read in this transaction execution, and clear the C-bit for future transaction control. Clearing C-bits as mentioned can prevent unnecessary serialization and resulting performance degradation.

V. Performance Evaluation

A. Simulation Environments

We used a full-system execution-driven functional simulator *Wind River Simics* [11] in conjunction with customized memory models built on *Wisconsin GEMS* [9] for evaluation. Simics provides a SPARC-V9 architecture and boots Solaris 10. GEMS provides a detailed timing model for the memory subsystem. Illinois-based directory protocol maintains cache coherence over a high-bandwidth switched interconnect. The detailed configuration of the simulated processor is shown in Table III. We have evaluated the execution cycles of 10 workloads from GEMS microbench, SPLASH-2 benchmark suite [10], and STAMP benchmark suite [12].

B. Evaluation Results

Figure 5 shows the evaluation results with three models: (B) LogTM the baseline, (S) Store Predictor, and (P)

TABLE III
CONFIGURATION OF THE SIMULATED PROCESSOR.

Processor	SPARC V9, 32 cores, 4 GHz, single issue, in-order, non-memory IPC = 1
L1 cache	32 KBytes, 4 ways, latency = 3 cycles
L2 cache	8 MBytes, 8 ways, latency = 34 cycles
Memory	4 GBytes, latency = 500 cycles
Interconnect	hierarchical switch topology, link latency = 14 cycles

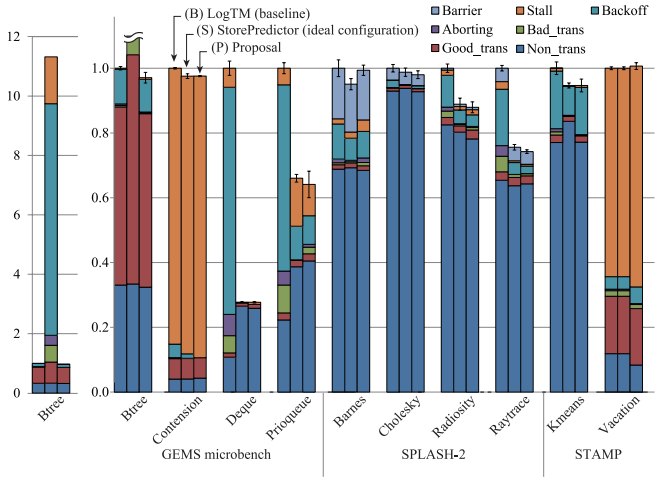

Fig. 5. Execution cycles ratio (16 threads).

the proposal. Each bar is normalized to that of the baseline LogTM (B). The legend shows the breakdown items of total cycles. They represent the executed cycles out of transactions (*Non_trans*), the executed cycles in committed/aborted transactions (*Good_trans*/*Bad_trans*), aborting overhead (*Aborting*), stall cycles (*Stall*), barrier synchronization cycles (*Barrier*), and exponential backoff cycles (*Backoff*).

For the simulation of multi-threading on a full-system simulator, the variability of performance [13] must be considered. Therefore, we measured 95% confidence interval of 10 times trials for each benchmark program, which are illustrated as error bars in this figure.

As described in Section II, *Store Predictor* (S) should manage addresses which are written after being read. To estimate the ideal performance of *Store Predictor*, we assume that the mechanism can register all the addresses written after being read with no overhead, although its implementation is not discussed originally in [7].

As shown in Fig. 5, the performance is remarkably improved with our transaction scheduling (P) in the most of the benchmark programs. This result implicates that each of many programs has a transaction which includes a read access followed by a write access, and is liable to bring futile stalls frequently. In contrast, the performance of Btree is significantly deteriorated with *Store Predictor* (S), although the other results with (S) are similar to (P).

The proposed scheduling reduces the execution cycles 72.2% at a maximum and 17.5% on average, by effectively solving these futile stalls. We go to detail examination of these results in the next section.

```
1  BEGIN_TRANSACTION( I );
2  Btree_insert (*node, insert, key, *ptr) {
3    if (node->isLeaf) {
4      node->key_num++;
5      node->child[node->key_num] = node->child[node->key_num-1];
6      for (i = node->key_num; i >= insert+1; --i) {
7        if (i != NODE_NUM_PAIRS)
8          node->key[i] = node->key[i - 1];
9        node->child[i] = node_child[i - 1];
10     }
11     if (insert < NODE_NUM_PAIRS)
12       node->key[insert] = key;
13     node_child[insert] = ptr;
14   }
15 }
16 COMMIT_TRANSACTION( I );
```

Fig. 6. The transaction for insertion (*Tx.I*) in Btree.

```
1  BEGIN_TRANSACTION( L );
2  Btree_lookup (*node, key) {
3    if (node->isLeaf) {
4      for (i = 0; i < node->key_num; ++i) {
5        if (key <= node->key[i])
6          index = i;
7      }
8    } else {
9      for (i = 0; i < node->key_num; ++i) {
10       if (key > node->key[i])
11         index = i;
12     }
13   }
14   node = node->child[index];
15 }
16 COMMIT_TRANSACTION( L );
```

Fig. 7. The transaction for lookup (*Tx.L*) in Btree.

C. Detailed Examination

1) GEMS microbench: The execution cycles of all the programs from GEMS microbench are reduced with the proposed scheduling (P). Especially, *Backoff* is drastically reduced in Deque and Prioque. We examined these two programs, and confirmed that each of them has a transaction with a compound operation for a shared variable, and each of many read accesses is followed by a write access to the same address. The proposed scheduling prevents the futile stalls proceeding from such transactions, and *Backoff* is reduced.

The performance of Btree is slightly improved with (P). Btree has two noted transactions. The one is for inserting data to a tree data structure (*Tx.I* shown as Fig. 6), and the other is for looking up the tree (*Tx.L* shown as Fig. 7). *Tx.I* has a read access to the shared variable `node->key_num`, and it is followed by a write access to the variable (at line 4 in Fig. 6). On the other hand, *Tx.L* also has read accesses to the same variable (at line 4 and 9 in Fig. 7), but no write access follows the read accesses. In this case, the proposed scheduling works well for *Tx.I* by serializing it, and this can be implicated from the reduction of *Backoff* in Btree. On the other hand, if the execution of *Tx.L* is also serialized, the total performance will degrade. The proposed scheduling effectively

```
1  BEGIN_TRANSACTION( 18 );
2   if (gm.lookup_array[key] != NOPE) {
3     /* some processing on each element in the queue */
4     gm.array[index].element += 1;
5     /* some processing on each element in the queue */
6     if (gm.lookup_array[tmp_l] != index) {
7       gm.lookup_array[tmp_l] = index;
8     }
9   }
10 COMMIT_TRANSACTION( 18 );
```

Fig. 8. A transaction in Prioque.

```
1 BEGIN_TRANSACTION( 16 );
2   ray->id = gm->rid++;
3 COMMIT_TRANSACTION( 16 );
```

Fig. 9. A transaction in Raytrace.

prevents this performance degradation by clearing C-bits as described in Section IV-B3. In contrast, *Store Predictor* (S) severely suffers from this problem. Stalling *Tx.L* causes many deadlocks and aborts, and its performance is degraded over 11x because *Stall* and *Backoff* are remarkably increased.

In the result of Prioque, *Stall* with (S) costs more cycles than (P). The reason is that Prioque has a transaction in which access pattern to a shared variable varies because of branch instructions. Figure 8 shows the transaction. In this transaction, the variable `gm.array[].element` is incremented at line 4. Proposal (P) and *Store Predictor* (S) can deal with this compound operation on `gm.array[].element`, and improve performance. Now, the array `gm.lookup_array[]` is read and written at line 2, 6 and 7, when the conditional expression at line 6 is true. However, when it is false, `gm.lookup_array` is only read and not overwritten. As with `node->key_num` in Btree, all the read accesses to `gm.lookup_array` are stalled with (S) even when it is not overwritten, and this makes *Stall* longer than (P).

2) SPLASH-2: The execution cycles of most of all the programs from SPLASH-2 benchmark suite are reduced with (P). Especially for Radiosity and Raytrace, *Backoff* is drastically reduced. Figure 9 shows a transaction that includes a compound operation in Raytrace.

In this transaction shown in Fig. 9, the variable `gm->rid` is incremented, and the increment is a compound operation. This increment must be atomically executed to generate unique IDs, and traditional HTMs cause many futile stalls as shown in Fig. 1 when this transaction is executed by multiple threads in parallel. As a result, *Backoff* increases because most of the conflicted transactions are aborted after futile stalls. The transaction shown in Fig. 9 is serialized by the proposed scheduling, and *Backoff* is drastically reduced.

3) STAMP: The performance of Kmeans is improved with the proposed scheduling (P). Kmeans has a transaction in which a shared variable is read and written, and serializing the transaction works well with our proposal. On the other hand, the performance of Vacation is not improved. This is because there is no transaction that includes a read access to a shared variable followed by a write access to the same variable.

VI. Conclusion

In this paper, we show that not a few read accesses to shared variables are followed by write accesses to the same variables, and granting read-after-read requests on such variables will cause conflict before long and result in futile stalls and repeated aborts.

To address this problem, we proposed a novel transaction scheduling for hardware transactional memories, and show a practical implementation of the scheduling on LogTM .

We have evaluated our proposal by comparing it with LogTM and *Store Predictor*, through experiments under GEMS microbench, SPLASH-2, and STAMP benchmark suites. The evaluation results show that the proposed scheduling can reduce futile stalls and resulting aborts, and improve the performance 72.2% at a maximum and 17.5% on average.

Acknowledgment

This research was partially supported by the grant from Tatematsu Foundation.

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] K. Hashimoto, S. Horiba, M. Eto, T. Tsumura, and H. Matsuo, "A Speed-up Technique for Hardware Transactional Memory by Controlling Read-after-Read Accesses," *IPSJ Trans. on Advanced Computing Systems*, vol. 6, no. 3(ACS44), pp. 58–71, Oct. 2013, (in Japanese).

[3] R. M. Yoo and H.-H. S. Lee, "Adaptive Transaction Scheduling for Transactional Memory Systems," in *Proc. 20th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, Jun. 2008, pp. 169–178.

[4] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom Filter Guided Transaction Scheduling," in *Proc. 17th Int'l Conf. on High-Performance Computer Architecture (HPCA-17)*, 2011, pp. 75–86.

[5] E. Akpinar, S. Tomić, A. Cristal, O. Unsal, and M. Valero, "A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory," in *Proc. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)*, 2011.

[6] E. Gaona, R. Titos, M. E. Acacio, and J. Fernández, "Dynamic Serialization Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems," in *Proc. Parallel, Distributed and Network-Based Processing 2012 20th Euromicro International Conference (PDP'12)*, 2012, pp. 221–228.

[7] J. Bobba *et al.*, "Performance Pathologies in Hardware Transactional Memory," in *Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA'07)*, 2007, pp. 81–91.

[8] 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.

[9] 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.

[10] 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.

[11] P. S. Magnusson *et al.*, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[12] 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.

[13] 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.