Fine-Grain Conflict Management for Hardware Transactional Memory Systems Employing Eager Version Management

Shoichiro HORIBA*, Hiroki ASAI*†, Masamichi ETO*, Tomoaki TSUMURA* and Hiroshi MATSUO*

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

†Currently with DENSO CORPORATION
1-1, Showa-cho, Kariya, Aichi, Japan

***Abstract*—On hardware transactional memory systems, false sharing on the shared cache brings conflicts between some transactions even if the transactions do not share any data. This paper proposes a method for managing transaction conflicts not on each cache block but on each octet or such a small unit. The method can be implemented with low hardware costs, and the evaluation results show that the method can improve the performance of transactional memory 89.4% in maximum, and 26.3% in average.**

I. Introduction

High-speed processor technologies have been achieved by multiplying clock frequency. However, as electric power consumption and thermal dissipation are increasing, it becomes difficult to raise clock frequencies of microprocessors. Complying with this trend, multi-core processors have been widely prevalent. When the shared memory programming is used on the processors, independent cores share a single address space. Hence, *locking* is generally used as a concurrency control mechanism. However, lock-based methods can cause deadlocks, and they lead to poor scalability. To solve these problems, **Transactional Memory** [1] has been proposed as a lock-free synchronization mechanism. In general HTMs, which are the hardware implementations of transactional memories, the fields called read bit and write bit are installed on each cache block. These bits keep track of the read/write accesses in transactions. However, even when multiple threads try to access to different data on a same cache block, a conflict is detected. This is because conflicts are managed on each cache block. Hence, the thread which detects such a conflict by receiving *NACK* must stall its transaction until the opponent transaction is committed. In addition, such a conflict can occur outside of transactions, because non-shared data will not be protected in transactions by programmers. To solve this problem, this paper proposes a fine-grain transactional conflict management for eager version management HTM systems. Using this conflict management method, memory accesses, which potentially have no conflict, can avoid being accidentally detected as transactional conflicts.

II. Related Work

On shared memory multi-core processors, when multiple threads access to different data on the same cache block, a well-known problem called *false sharing*[2] occurs. A general solution for this problem is locating such data on different cache blocks by padding useless data[3]. However, padding leads to high memory usage and low spatial locality.

Under the situation where multiple transactions speculatively run in parallel, false sharing can cause *false conflicts*. In this case, the thread which finds a false conflict should stall its transaction, and the transaction will be serialized.

A method for solving the problem of false conflicts by decoupling transactional conflicts from cache coherence conflicts is proposed[4]. In the model, even if a thread detects a conflict, the thread keeps running its transaction speculatively on the assumption that the conflicted datum has not been modified by any other transactions. On commit of the transaction, the conflicted datum is checked whether it has been modified or not, and if modified, the transaction is aborted.

This method can increase the number of parallel transactions. However, when a speculative execution of a transaction fails and the transaction is aborted, all the speculative execution of the transaction is discarded because the method is based on *lazy conflict detection*. Incidentally, the method can not be applied to *eager version management* transactional memory systems, because it can not restore a part of cache block. It is also a problem that the hardware implementation and the additional hardware cost are not discussed in detail, and the feasibility is not clear.

On the other hand, a fine-grain conflict management method proposed in this paper can detect transactional conflicts more strictly, by dividing a cache block into several subsets and checking conflicts on each subset. This can prevent false conflicts. Moreover, this method can be used not only with lazy

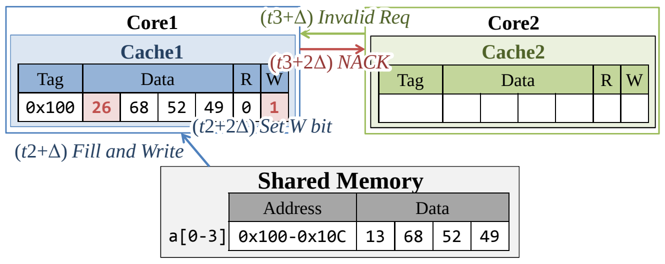

Fig. 1. False conflict on accessing different variables.

Time	Thread1	Thread2
$t1$	`BEGIN_XACT;`	`BEGIN_XACT;`
$t2$	`a[0]=26;`	`...`
$t3$	`...`	`a[2]=70;`
$t4$	`...`	`COMMIT_XACT;`

Fig. 2. Sample codes executed on the threads.

version management but also with eager version management, supporting partial restore of a cache block on aborts.

III. Fine-Grain Conflict Management

In this section, the problem which general HTMs have is described, and a fine-grain conflict management is proposed.

A. *Problem of Cache Block Based Conflict Management*

General HTMs have a problem that a false conflict is detected when multiple transactions access to different data on a certain cache block, because transactional conflicts are managed on each cache block. Fig. 1 shows the problem on LogTM[5]. LogTM is a hardware transactional memory system employing eager conflict detection and eager version management. In this figure, two unique threads are accessing different data on the same cache block. Now, Thread1 runs on Core1, and Thread2 on Core2. Each thread executes the codes shown in Fig. 2. Here, `BEGIN_XACT` and `COMMIT_XACT` indicate the beginning and the end of a transaction respectively. The column of *Time* in Fig. 2 represents the time when each statement is executed, and $t1 < t2 < t3 < t4$ here.

First, two threads starts their own transactions ($t1$), and then, Thread1 stores a value to `a[0]` ($t2$). Now, `a[0]` is not on the cache, and the block at 0x100 on the shared memory is transferred onto the cache, and `a[0]` is updated on the cache ($t2$+Δ in Fig. 1). On eager version management HTM systems, the address 0x100 and the previous value of the cache block are kept in the area called *log*, but the behaviour around the log is omitted in this figure. Then, the write bit of the cache block is set, because the block is modified ($t2$+2Δ).

Next, Thread2 tries to store a value to `a[2]`, which is on the same cache block with `a[0]` ($t3$). Thread2 sends an invalidation request to the block 0x100 ($t3$+Δ). Receiving this request, Thread1 refers the read and write bits of the block 0x100. Now, the write bit is set and a conflict is detected, and Thread2 receives *NACK* ($t3$+2Δ).

In this way, when multiple threads are going to access to different data on the same cache block, a conflict is detected. This can cause a significant performance degradation, but it is hard for programmers to prospect which data will be located in the same cache block and which data are not.

Generally, different variables which are defined as thread-local storage will not be located in the same cache block. However, when the spaces for the variables are allocated on heap area by malloc(), different thread-local variables can be located in the same cache block. For the same reason, a thread-local variable and a global variable can share a cache block.

Especially in eager version management HTM systems, all *NACK*ed threads must be stalled, and false conflicts can cause severe performance degradation. In this paper, we call the stall which is derived from a false conflict as a *false stall*.

Incidentally and interestingly, false stalls can occur outside of transactions. In other words, threads on HTM systems can be *NACK*ed and stalled even when they are processing the code outside of its transactions. The reason is that, programmers will protect shared data by enclosing them in transactions, but false stalls can occur on exclusive data, even on thread-local variables, and such data will not be protected in transactions on any account.

To solve this problem, a fine-grain transactional conflict management for eager version management HTM systems is proposed in this paper. By managing conflicts on subsets of the cache blocks, false stalls can be avoided.

B. *Outline of Proposed Model*

1) Conflict Detection: The behaviour of conflict detection on the method proposed in this paper is explained with an example shown in Fig. 3. Now, two threads are executing the transactions shown in Fig. 2. First, each of the threads starts its own transaction ($t1$), and Thread1 on Core1 stores a value to `a[0]` ($t2$). Then, the block 0x100 is cached and the value of `a[0]` is updated ($t2$ in Fig. 3).

After that, Thread2 on Core2 tries to store a value to `a[2]` ($t3$), and an invalidation request for the block 0x100 is sent to Core1. In the existing HTM systems, the invalidation request is labeled with the block address 0x100, but not labeled with the address of the concerned datum `a[2]`. Hence, Thread1 can not know whether the access requested from Thread2 surely conflicts with the own previous access to `a[0]`, or not.

On the other hand, the new conflict management system, which is proposed in this paper, piggybacks the target address `a[2]` along with the invalidation request to Thread1 as shown ($t3$+Δ) in Fig. 3(a). Beforehand, Thread1 keeps track of the read/write accesses in its own transaction in a fine-grained manner. Hence, Thread1 can know whether there surely is a conflict or not, by receiving the address of `a[2]`.

In this example, Thread1 has not accessed to `a[2]`, and it is considered that the access by Thread2 does not lead to a conflict. Therefore, complying with the cache coherence protocol, Thread1 invalidates the cache block of 0x100 ($t3$+2Δ), sends *ACK* to Thread2 ($t3$+3Δ), and writes back the block

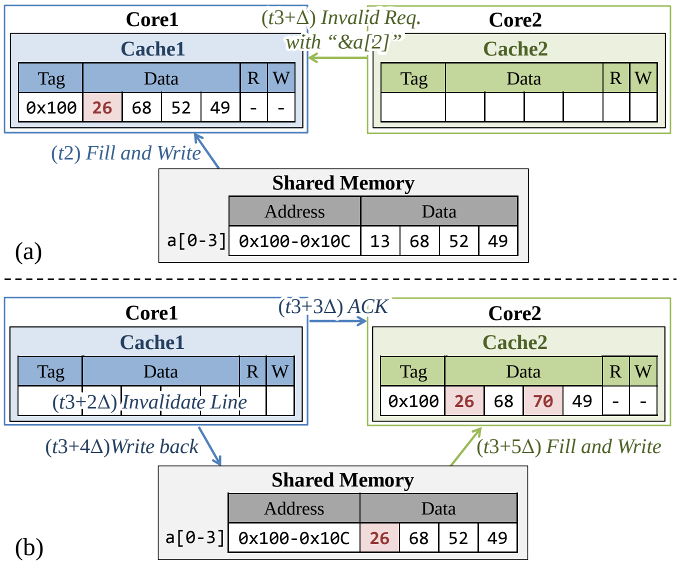

Fig. 3. Accesses to different data on the same cache block are granted in the proposed model.

$(t3{+}4\Delta)$, as shown in Fig. 3. After that, Thread2 caches the block of 0x100, and updates the value of `a[2]` $(t3{+}5\Delta)$.

2) Aborting: When aborting a transaction, LogTM writes back the old values stored in the log to the memory, and the state of the memory is rolled back to the beginning of the transaction. Here, the whole cache block which includes the old values is stored in the log and is written back to the memory. However in the new conflict management model, there may be some values modified by other threads in the cache block, and such values should not be restored to their past states. Hence, we propose a new cache coherent mechanism for our conflict management model. Cache coherency is maintained by restoring not the whole cache block but only a necessary part of the cache block. Detailed implementation of this mechanism is described in the next section.

IV. Hardware Implementation

In this section, additional hardware for achieving the fine-grain conflict management and its behaviour is described.

A. Additional Hardware

For implementing the fine-grain conflict management, a small RAM named *R/W Table* is installed to each core of the HTM system. This table is used for keeping the fine-grained information about read/write accesses on specific cache blocks. A field is also installed to each cache block. The field, which is represented as *Ptr* in this paper, is used for storing an index of R/W Table entry associated with the cache block.

Fig. 4 shows the structure of the proposed HTM system. In the proposed system, it is assumed that there are N subsets in a cache block, and transactional conflicts are managed on each subset. In this way, false conflicts are avoided. Each processor core has one R/W Table, which has four fields; *Tag* for cache tag, R for managing read accesses, W^{self} for

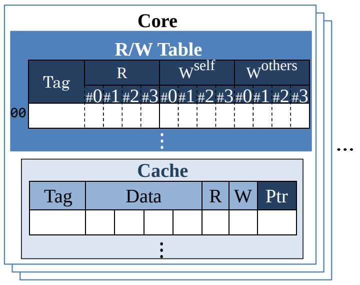

Fig. 4. Processor structure of proposed HTM.

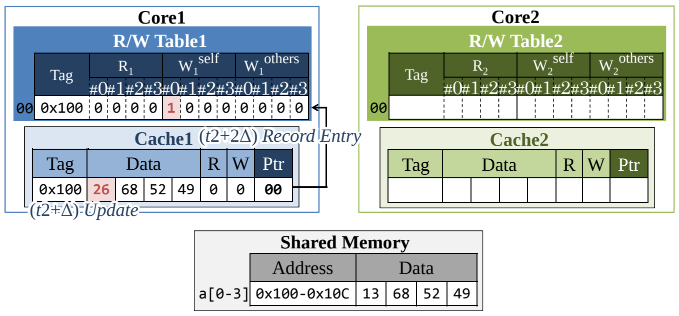

Fig. 5. Registering conflict information to R/W Table.

managing write accesses issued by the transaction on the own core, and W^{others} for managing write accesses issued by the transactions on the other cores. Each of R, $\mathrm{W}^{\mathrm{self}}$ and $\mathrm{W}^{\mathrm{others}}$ has N-bit width for managing the accesses on N subsets of each cache block individually.

The depth of R/W Table is arbitrary, but we have confirmed through performance evaluation that not so large depth is required for R/W Table. The consideration on the depth of R/W Table is shown in section V-C later.

B. Registering to R/W Table

In the proposed find-grain conflict management method, R/W Table is designed to manage only the cache blocks on which a transactional conflict has occurred. However, it is difficult to know in advance which cache block, or which variable, will be concerned in transactional conflict.

Hence, in the implementation of this method, each cache block is originally managed by its read bit and write bit, same as on existing HTM systems. Once a transactional conflict occurs on a cache block, the cache block is managed by R/W Table thereafter. Hence, there is no need to statically analyze workload programs.

If the first transactional conflict on a cache block is a false conflict, the conflict will cause a futile false stall. However, the false stall will occur up to only once on each cache block in the worst case, and second and subsequent transactional conflicts on the cache block are managed by R/W Table in fine-grained manner. Hence, one false stall per core will have little impact on the total performance.

Now, an example where two different transactions are running concurrently is shown in Fig. 5. Core1 and Core2 in Fig. 5 are executing Thread1 and Thread2 shown in Fig. 2

respectively. In this example, N the number of subsets in each cache block is defined as four ($N = 4$), and each cache block has four integer values on it.

First, the threads start their transactions ($t1$). Then, Thread1 issues a store on `a[0]` ($t2$), and the value of `a[0]` is modified ($t2$+Δ). If Ptr associated with the cache block 0x100 does not have a valid value, an entry on R/W Table is reserved for the cache block and the index of the entry is stored in Ptr.

In this example, Ptr already has a valid value `00`. This means that a transactional conflict has already occurred on the cache block, and R/W Table is managing the block. Hence, the entry `00` of R/W Table is referred and modified ($t2$+2Δ). If it is the second transactional conflict on the cache block, the tag value of the block is registered to Tag field of the entry `00`, and the bits in R and $\mathrm{W^{self}}$ fields, which are associated with the accessed address, are set. In this example, $\mathrm{W_1^{self}}[\#0]$ is set because Thread1 accesses `a[0]`, while $\mathrm{W_1^{others}}$ is not set because the cache block 0x100 is accessed from no other transactions.

C. Conflict Detection by Referring R/W Table

In the proposed method, when an access request arrives, a transactional conflict is detected in the fine-grained manner if an R/W Table entry is already associated to the cache block. Now, return to the example described in section IV-B. After $t2$, Thread2 tries to update `a[2]` ($t3$). As shown in Fig. 6(a), an invalidation request which piggybacks the address of `a[2]` is sent from Thread2 ($t3$+Δ). R/W Table in Core1, which receives the invalidation request, has the entry associated to the block 0x100, and the entry is referred for conflict detection ($t3$+2Δ). In the entry, $\mathrm{R_1}[\#2]$, $\mathrm{W_1^{self}}[\#2]$ and $\mathrm{W_1^{others}}[\#2]$, which are associated to `a[2]`, are not set, and no conflict is detected. Then, Thread1 invalidates 0x100 ($t3$+3Δ in Fig. 6(b)), and clears Ptr of 0x100.

The thread which receives a memory access request is responsible for conflict detection. However, in this example, the block 0x100 is invalidated and Core1 never receives any request about 0x100 hereafter, and Core1 can not detect any conflict on 0x100. Hence, Core2 should take over the responsibility for conflict detection on 0x100 from Core1, by inheriting the past access information managed by Core1. To handle this, Core1 logically adds $\mathrm{W_1^{self}}$ and $\mathrm{W_1^{others}}$ in the R/W Table entry which is associated to 0x100, and piggybacks the result and $\mathrm{R_1}$ along the *ACK* reply to Core2 ($t3$+4Δ).

In this example, $\mathrm{R_1}$=`0000` and $\mathrm{W_1^{self}} \vee \mathrm{W_1^{others}}$=`1000` are biggybacked. In this paper, the logical addition of $\mathrm{W_1^{self}}$ and $\mathrm{W_1^{others}}$ is inscribed as $\mathrm{W_1}$. Sending the *ACK*, Core1 writes back 0x100 block ($t3$+5Δ).

Receiving the *ACK*, Core2 knows that now it can access the block, and caches it ($t3$+6Δ). Simultaneously, Core2 knows that there have occurred one or more transactional conflicts on the cache block, by receiving the piggybacked $\mathrm{R_1}$ and $\mathrm{W_1}$ values. Hence, Core2 registeres an R/W Table entry associated to 0x100, stores the index `00` to Ptr of the block, and modifies $\mathrm{W_2^{self}}[\#2]$ for memorizing its write access on `a[2]`. At this time, Core2 logically adds its own $\mathrm{R_2}$ and $\mathrm{R_1}$ received from

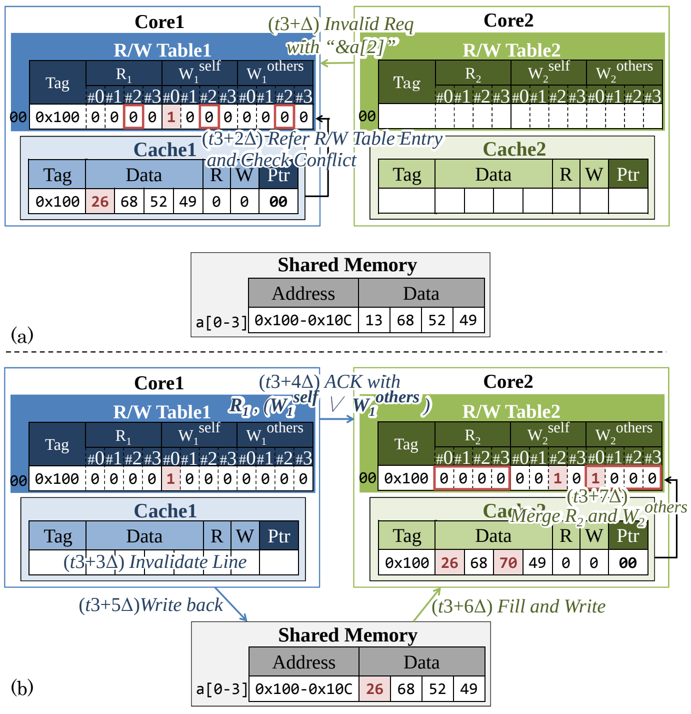

Fig. 6. Conflict detection using R/W Table.

Core1, and stores the result to $\mathrm{R_2}$, for inheriting the past access information which has been managed by Core1. $\mathrm{W_2^{others}}$ is also overwritten with the logical addition of $\mathrm{W_2^{others}}$ and $\mathrm{W_1}$, in the same way ($t3$+7Δ). In this way, Core2 can detect conflicts on `a[0]` which are resulting from accesses on other transactions.

D. Discarding Entries on R/W Table

Information about read and write accesses which is managed in R/W Table is specific to each transaction. Hence, when a transaction commits or aborts, all entries in R/W Table can be discarded.

As described in section III-B2, when a transaction is aborted, only the logged values, which are associated to the modified cache block subsets, should be written back to the memory. This is implemented by masking the logged cache block with $\mathrm{W^{self}}$.

For example in Fig. 6, consider that Thread1 aborts its transaction after $t4$ shown in Fig. 2. In this case, Thread1 can restore only the necessary values by masking the logged cache block with $\mathrm{W_1^{self}}$=`1000` as shown in Fig. 7.

V. Performance Evaluation

This section describes about the evaluation results, and the estimation of additional hardware cost and overhead.

A. Evaluation Environment

We implement the proposed method on LogTM[5], distributed as a module for GEMS[6], and evaluate it on Wind River Simics[7]. Simulation parameters are shown in TABLE I. The access latency for R/W Table is defined as three

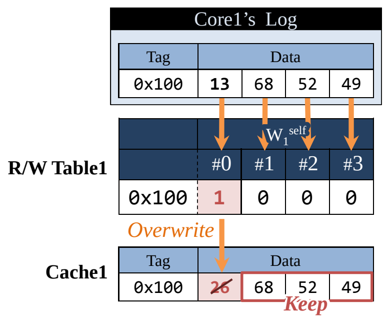

Fig. 7. Masking restoring values by W^{self}.

TABLE I
SIMULATION PARAMETERS

Processor	SPARC V9
#cores	16 cores
clock	4 GHz
issue width	single
issue order	in-order
non-memory IPC	1
L1 I&D cache	32 KBytes
ways	4 ways
latency	3 cycle
line size	64 Bytes
L2 cache	8 MBytes
ways	8 ways
latency	34 cycles
line size	64 Bytes
L2 Directory	Full-bit vector sharers list
latency	6 cycles
Memory	4 GBytes
latency	500 cycles
Interconnect network	2D mesh topology
link latency	3 cycles
link bandwidth	64 Bytes

cycles, which is same as L1 cache. The reason is that R/W Table can be implemented with much smaller RAM than L1 cache, as described in section V-C later. The workloads are Prioqueue and Slist from GEMS micro-benchmark suite, and Kmeans and Vacation from STAMP[8] benchmark suite.

B. Evaluation Results

Fig. 8 shows the evaluation results. The legend shows the breakdown items of the total cycles;

inTX	:cycles in transactions
outTX	:cycles out of transactions
FalseStall-inTX	:false stall cycles in transactions
FalseStall-outTX	:false stall cycles out of transactions

Each program was evaluated with five models as shown below, where N is the assumed number of subsets in each cache block, and executed with 16 threads. Each bar is normalized to the total cycle of the baseline (B).

(B)	LogTM (baseline)
(T_2)	proposed model ($N = 2$)
(T_4)	proposed model ($N = 4$)
(T_8)	proposed model ($N = 8$)
(T_{16})	proposed model ($N = 16$)

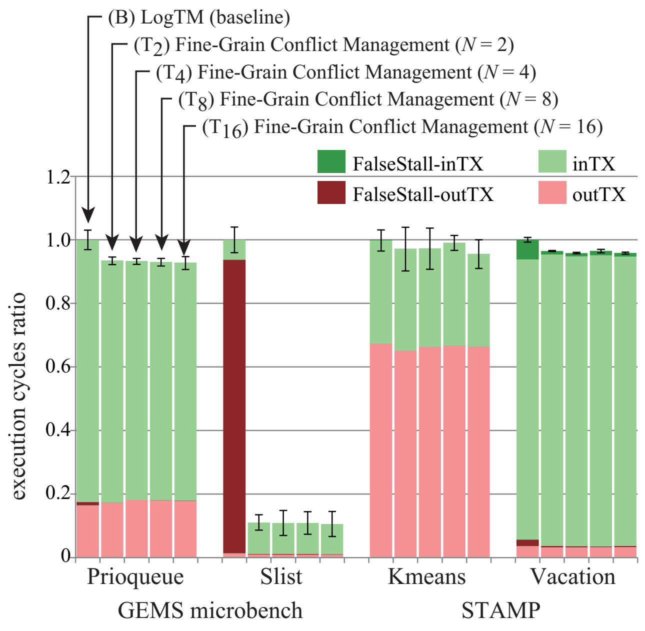

Fig. 8. Execution cycles ratio

For the simulation of multi-threading on a full-system simulator, the variability performance [9] must be considered. Hence, we tried 10 times on each benchmark, and measured 95% confidence interval. The confidence intervals are illustrated as error bars in Fig. 8.

All models reduced FalseStall-inTX and FalseStall-outTX in all programs. The result of the experiment shows that the model (T_{16}) can improve the performance 89.4% in maximum, and 26.3% in average.

First, FalseStall-outTX is reduced in Prioqueue, Slist and Vacation. Especially in Slist, in which FalseStall-outTX accounts for a large percentage, total cycles were reduced significantly by avoiding false stalls. Incidentally, FalseStall-outTX in Slist was reduced by the model (T_2) as much as (T_{16}). Investigating the cause, it is found that allocation for different two variables is greatly related to it. The tail of the allocated area for one variable is on the head of a cache block, and the head of the other variable is on the tail of the same cache block. Hence, in Slist, it is enough to divide a cache block into two subsets.

On the other hand, in Prioqueue and Kmeans, inTX is slightly reduced. The reason is that, by avoiding false stall, the number of stalled transactions is reduced and less transactions will abort than baseline (B). This makes the cycles for aborting and backoff smaller, and inTX decreases.

In Vacation, execution cycles are reduced by solving FalseStall-outTX and FalseStall-inTX, and there is no much difference among the performance of four proposed models, as same as Slist. In this program, different variables are defined as thread-local storages, and the variables are placed on a cache block at a distance each other. Thus, it is enough to divide the cache block into two subsets for avoiding false conflicts on the cache block.

TABLE II
EXECUTION CYCLES RATIO WITH PADDING MODEL

	(B)	(P)	(T_2)	(T_4)	(T_8)	(T_{16})
Slist	1.00	0.12	0.11	0.10	0.10	0.10
Vacation	1.00	1.63	0.96	0.96	0.97	0.96

TABLE III
REQUIRED R/W TABLE ENTRIES IN (T_{16})

GEMS	Max	Ave	STAMP	Max	Ave
Prioqueue	10	8.9	Kmeans	13	11.0
Slist	3	3.0	Vacation	21	16.4

Now, in both of Slist and Vacation, false stalls account for a large percentage, and naive padding method may also relieve the programs. Hence, we have examined the effect of padding method on these programs. TABLE II shows the cycles ratio of the four fine-grained conflict management models and the padding model (P). Each value is normalized to the total cycle of (B). In these two programs, FalseStall-inTX and FalseStall-outTX are reduced by the model (P) as much as the four proposed models. However, the total performance of Vacation declines no less than 63%. This is because that locating data on different cache blocks leads to low spatial locality and raises cache miss rate. Especially in Vacation, multiple threads share some tree structures, and sequential accesses to the tree nodes bring a lot of cold start misses in inTX and outTX. On the other hand, the proposed models could avoid false stall without raising cache miss rate very much.

The traditional problem caused by false sharing was only high cache miss ratio, and the performance was deteriorated by cache miss penalties. However on HTM systems, false sharing also brings false stall, because transactional conflicts are treated equally to cache conflicts. The overhead incurred from false stall is incomparably larger than the overhead from cache misses, and it can cause a substantial performance loss as in Slist. Hence, it is very important to reduce the overhead from false stall, and the proposed fine-grain conflict management can achieve this.

C. Hardware Costs

In the proposed HTM system, the depth of R/W Table should be as much as the number of conflicted cache blocks. We have evaluated how many R/W Table entries are required. The result is shown in TABLE III. As we can see, if R/W Table has 21 entries, overflow does not occur with these four programs. Here, Tag field is 58-bit width, and each of R, $\mathrm{W}^{\mathrm{self}}$ and $\mathrm{W}^{\mathrm{others}}$ needs 16-bit width when $N = 16$. Consequently, one R/W Table can be implemented with a RAM which has 106-bit width and 21 rows. Hence, for 16 core processor, the hardware cost is about 4.5KBytes in total. If $N = 2$, the hardware cost is only about 2.7KBytes.

D. Overhead for Referring R/W Table

In this section, the overhead for referring R/W Table is estimated. The overhead can simply expressed as $C \times T$, where C is how many times R/W Table is referred, and T is the access latency for R/W Table. TABLE IV shows the average count of R/W Table reference in executing the benchmark programs with model (T_{16}). In Prioqueue, which has the most reference count, the overhead cycles can be calculated as 4542.6×3, and is about 13 thousands. On the other hand, the total cycles of Prioqueue is about six millions. Hence, the overhead ratio is only 0.2 percent.

TABLE IV
AVERAGE COUNT OF REFERRING R/W TABLE IN (T_{16})

GEMS		STAMP	
Prioqueue	4542.6	Kmeans	187.8
Slist	97.4	Vacation	769.1

VI. CONCLUSION

In this paper, we proposed a fine-grain transactional conflict management for eager version management HTM systems. Through an evaluation with microbench in GEMS and STAMP benchmark programs, it is found that the proposed method can improve the performance 89.4% in maximum, and 26.3% in average. Our future work includes the improvement of R/W Table usability. Under the case where conflicts occur on many cache blocks, R/W Table can be overflowed. An easy way to break the situation is to randomly clear some entries by aborting transactions. However, considering the contribution of each entry may provide more efficient R/W Table management.

ACKNOWLEDGMENT

This work was supported in part by MEXT KAKENHI Grant Number 24500113.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proc. 20th Annual Int'l Symp. on Computer Architecture*, May. 1993, pp. 289–300.

[2] J. Torrellas, M. S. Lam, and J. L. Hennessy, "False sharing and spatial locality in multiprocessor caches," *IEEE Transactions on Computers*, vol. 43, pp. 651–663, 1994.

[3] T. L. Harris, K. Fraser, and I. A. Pratt, "A Practical Multi-word Compare-and-Swap Operation," in *Proc. 16th Int'l Conf. on Distributed Computing (DISC'02)*, 2002, pp. 265–279.

[4] F. Tabba, A. W. Hay, and J. R. Goodman, "Transactional Conflict Decoupling and Value Prediction," in *Proc. Int'l Conf. on Supercomputing (ICS'11)*. ACM, 2011, pp. 33–42.

[5] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Proc. 12th Int'l Symp. on High-Performance Computer Architecture*, Feb. 2006, pp. 254–265.

[6] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood., "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[7] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[8] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)*, Sep. 2008.

[9] A. R. Alameldeen and D. A. Wood, "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.