PAPER *Special Section on Parallel and Distributed Computing and Networking*

A Waiting Mechanism with Conflict Prediction on Hardware Transactional Memory

Keisuke MASHITA[†], **Maya TABUCHI**[†], **Ryohei YAMADA**[†], ***Nonmembers**, **and*** **Tomoaki TSUMURA**[†a)], ***Member***

SUMMARY Lock-based thread synchronization techniques have been commonly used in parallel programming on multi-core processors. However, lock can cause deadlocks and poor scalabilites, and Transactional Memory (TM) has been proposed and studied for lock-free synchronization. On TMs, transactions are executed speculatively in parallel as long as they do not encounter any conflicts on shared variables. On general HTMs: hardware implementations of TM, transactions which have conflicted once each other will conflict repeatedly if they will be executed again in parallel, and the performance of HTM will decline. To address this problem, in this paper, we propose a conflict prediction to avoid conflicts before executing transactions, considering historical data of conflicts. The result of the experiment shows that the execution time of HTM is reduced 59.2% at a maximum, and 16.8% on average with 16 threads.

key words: *hardware transactional memory, conflict prediction, transaction scheduling, concurrency control*

1. Introduction

On multi-core processors, multiple threads can run in parallel for speed-up. Therefore, parallel programming becomes more important for programmers to achieve speed-up. When multiple threads run in parallel on shared memory systems, mutual exclusion is required, and *lock* has been commonly used. However, lock-based methods can cause deadlocks, and they lead to poor scalability. To solve this problem, *Transactional Memory* (TM) [1] has been proposed as a lock-free synchronization mechanism. On TMs, transactions are executed speculatively as long as they do not encounter any conflicts on shared variables. However, the interim results of transactions may be discarded because transactions are executed speculatively. Hence, when a transaction modifies a value on the shared memory, TM should save both new and old values (*version management*). TM also should keep tracks of memory accesses, checking whether each requested datum has been accessed yet by another transaction or not (*conflict detection*). On *Hardware Transactional Memories* (HTMs), which are the hardware implementations of TM, the mechanisms for version management and conflict detection are implemented in hardware. Therefore, each of version management and conflict detection costs only a small delay overhead. It is known that HTM can achieve higher scalability than lock-based mechanisms [2], and HTM is regarded as a promising paradigm.

Manuscript received January 6, 2016.
Manuscript revised May 9, 2016.
Manuscript publicized August 24, 2016.

[†]The authors are with Nagoya Institute of Technology, Nagoya-shi, 466–8555 Japan.
a) E-mail: tsumura@nitech.ac.jp

DOI: 10.1587/transinf.2016PAP0006

On general HTMs, transactions, which have conflicted on a shared variable once each other, will conflict repeatedly on the same shared variable if they will be executed in parallel again. This conflict repetition will bring severe performance degradation of HTMs. To address this problem, we propose a conflict prediction to avoid such a conflict in advance. Before a thread starts to execute a transaction, the thread predicts future conflicts based on historical data, including past conflicted pairs and temporal data about transactions. Thereby, the thread can avoid causing a conflict with transactions being executed by other threads.

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

1. We propose a novel waiting mechanism for avoiding future conflicts between transactions. The mechanism keeps a transaction waiting *before* the transaction starts. Hence, the waiting transaction will not cause another conflict, unlike *stalling* transactions.
2. We use two types of historical data: past conflicted transaction IDs and temporal data of transactions, as parameters for predicting future conflicts. We also propose a criterion for deciding temporal data should be considered for waiting periods or not.
3. We evaluate the prediction-based transaction scheduling. The results show that the execution cycles can be reduced 59.2% at a maximum and 16.8% on average.

2. Conflict Prediction for Transactions

In this section, we describe overviews of HTM, and point out a problem of general HTMs. After that, we describe our conflict prediction [3] to address the problem, and introduce an improvement of our conflict prediction.

2.1 Conflict Detection and Resolution on General HTMs

To detect a conflict, TM must keep track of whether each shared variable is accessed or not. To achieve this, each cache block has two additional bit-fields called *read bit* and *write bit* on general HTM. When a cache block is read during a transaction, the read bit of the cache block is set. In the same way, when a cache block is overwritten, the write bit of the cache block is set. When a transaction is committed or aborted, HTM resets all these bits which are set during the transaction. To handle these bits, HTM uses a modified cache coherence protocol. By managing the state of cache blocks and testing these bits, HTM keeps the caches coherent.

When a thread tries to access a shared variable, the thread sends a coherence request to detect a conflict. If the access will not cause any conflicts, the thread receiving a coherence request from another thread sends back an *Ack*. On the other hand, if a conflict is detected, a *Nack* is sent back. In *eager conflict detection* model, when a thread receives a *Nack*, it knows that a conflict with the thread which sent the *Nack* is detected, and *stalls*, waiting for the opponent thread to commit. The thread which stalls its transaction will keep sending the coherence request intermittently. If the opponent thread commits its transaction, the thread which has stalled its transaction finally receives an *Ack*, and can resume its transaction.

Figure 1 shows an example where conflicts are detected with LogTM [4]: the most general HTM system which adopts *eager conflict detection*. In this example, the thread *thr.1* executes the transaction *Tx.X*, *thr.2* executes *Tx.Y*, and *thr.3* executes *Tx.Z*. Now, assume that, *thr.1* has issued `load A` and *thr.2* has issued `load B` and `load C`. First, when *thr.2* tries to issue `store A` (at *t1*), a conflict is detected (*t2*) because *thr.1* has already accessed to the address A. In this case, as *thr.2* receives a *Nack* from *thr.1*, *thr.2* stalls *Tx.Y* (*t3*), waiting for *thr.1* to commit. To avoid causing deadlock, *thr.1* also sets a flag called *possible cycle*. After that, when *thr.3* tries to issue `store C`, another conflict is detected because *thr.2* has already accessed to the address C. In this case, *thr.3* receives a *Nack* from *thr.2*, and *thr.3* stalls *Tx.Z* (*t4*), waiting for *thr.2* to commit. Afterwards, when *thr.1* tries to issue `store B`, another conflict is detected because *thr.2* has already accessed to the address B. In this case, as *thr.1* has set *possible cycle* flag, *thr.1* aborts *Tx.X* (*t5*). As a result, *thr.1* and *thr.2* can avoid deadlock.

As *thr.2* in this example, a thread which stalls its transaction for a conflict can cause another new conflict. This is because a thread which stalls its transaction has already accessed some addresses and the thread does not reset read/write bits for the addresses until the thread commits or aborts.

2.2 Avoiding Future Conflicts

As mentioned in Sect. 2.1, a thread which stalls its transaction often causes a new conflict. To address this problem, in this paper, we propose a conflict prediction for avoiding conflicts *before a thread executes a transaction*, based on historical data of conflicts.

Transactions which have conflicted once each other tend to conflict repeatedly because their execution paths rarely change and the threads often access to the same shared variables when they are executed again. To achieve the conflict prediction, we make each thread remember the IDs of the past conflicted transactions as historical data of conflicts. Before starting a transaction, a thread predicts whether a conflict will be caused or not during the transaction, by referring to the historical data. If the thread predicts that a conflict will be caused, the thread waits for the opponent thread to commit *without starting the transaction*. As mentioned in Sect. 2.1, *stall* is also a ‘waiting’ mechanism for conflict resolution. In contrast to stall, a waiting thread without starting its transaction does not cause a new conflict, because the thread waits without accessing any addresses.

Figure 2 shows an example where threads try to execute same transactions as Fig. 1 and can avoid conflicts by conflict predictions. Assume that *Tx.Y* had already conflicted with *Tx.X* and *Tx.Z*, and each thread remembers that. First, *thr.2* sends the transaction ID ‘*Y*’ to all the other threads when *thr.2* starts to execute *Tx.Y* (*t1*). Receiving this, *thr.1* and *thr.3* remember the transaction ID ‘*Y*.’ After that, *thr.1* tries to execute *Tx.X* while *Tx.Y* is running on *thr.2*. At this time, *thr.1* predicts whether *Tx.X* will conflict with a transaction which is running on another thread or not by referring to historical data of conflicts (*t2*). As a result, *thr.1* knows that *Tx.X* has already conflicted with *Tx.Y* running on *thr.2*, and *thr.1* predicts that *thr.1* will conflict with *thr.2*. Therefore, *thr.1* waits for *thr.2* to commit *Tx.Y* without starting

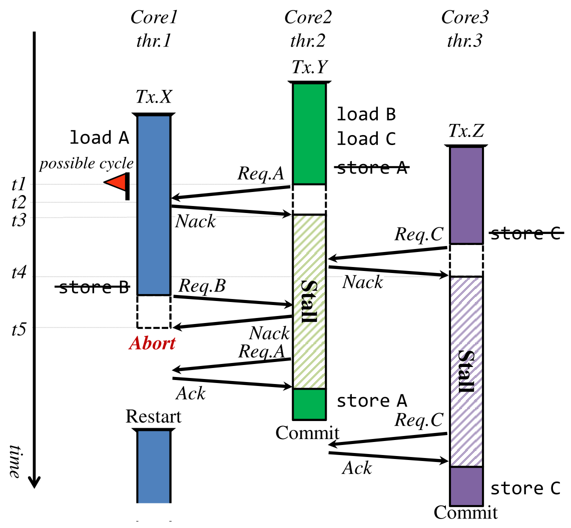


Fig. 1 Conflict resolution on a general HTM.

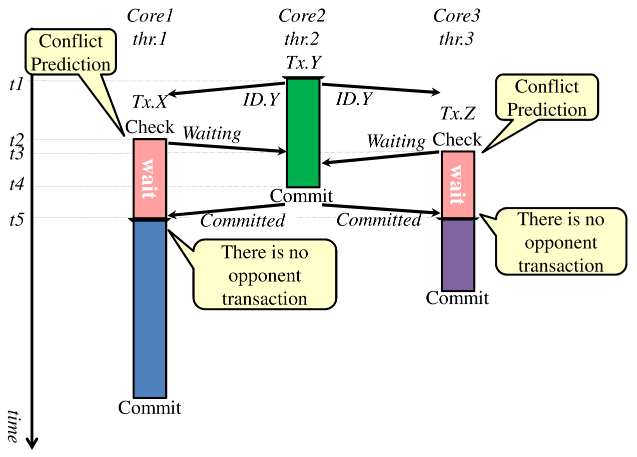
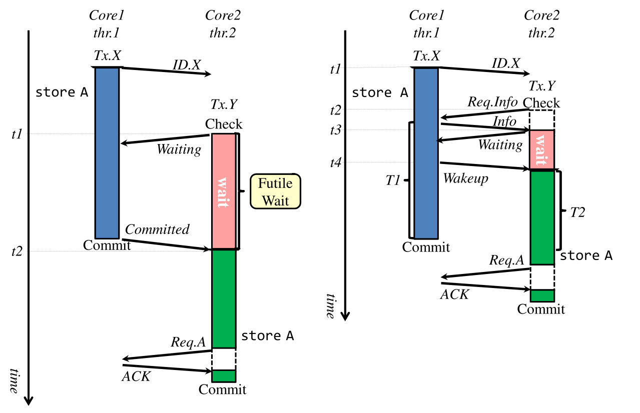


Fig. 2 Transaction scheduling with conflict predictions.



(a) A conflict prediction based on only opponent transaction IDs.

(b) A conflict prediction based on opponent transaction IDs and temporal data of a transaction.

Fig. 3 Comparison of two types of conflict predictions.

Tx.X. Thereby, *thr.1* sends a *Waiting* message to *thr.2*. In the same way, *thr.3* predicts that *thr.3* will conflict with *thr.2* before *thr.3* starts to execute *Tx.Z* (*t3*). Hence, *thr.3* waits for *thr.2* to commit, and it sends a *Waiting* message to *thr.2*. After *thr.2* commits *Tx.Y*, *thr.2* sends a *Committed* message to each of *thr.1* and *thr.3* (*t4*). When *thr.1* and *thr.3* receive the *Committed* messages, they start to execute transactions (*t5*). As mentioned above, threads can execute their transactions without conflicting with each other.

2.3 Minimizing Waiting Time

As mentioned in Sect. 2.2, threads can predict and avoid conflicts by remembering past opponent transaction IDs, but the waiting time can be longer than the minimum necessary time to avoid the conflict. If each thread knows when a conflict will be caused, the thread can wait minimum for avoiding the conflict.

Figure 3 (a) shows an example where a thread predicts a conflict by checking an opponent transaction is running or not, and waits for the opponent threads to commit. Assume that *thr.2* remembers that *Tx.Y* had already conflicted with *Tx.X*. First, before *thr.2* starts to execute *Tx.Y*, *thr.2* predicts that *thr.2* will conflict with *thr.1*, and *thr.2* waits for *thr.1* to commit *Tx.X* (*t1*). After a while, as *thr.2* receives a *Committed* message, it starts to execute *Tx.Y* (*t2*). As mentioned above, these two threads can execute their transactions without causing a conflict. However, for avoiding the conflict, `load A` in *Tx.Y* needs to be after the commit of *Tx.X*, but the start of *Tx.Y* does not need to be. Therefore, as in this example, if a conflict is caused after a long period than the transaction starts, the most of waiting time will become futile. If a thread can compare the predicted remaining execution time of the opponent transaction and the predicted remaining time until a conflict, such a futile wait can be avoided. To achieve this, we make each thread remember two temporal data of each transaction. One is how long the whole execution time of the transaction is, and the other is how much time later a conflict will be caused than the transaction starts.

Figure 3 (b) shows an example where a thread does not wait futilely when the thread predicts a conflict. Assume that *thr.2* has historical data used for conflict prediction. First, *thr.1* sends the transaction ID '*X*' to all the other threads when *thr.1* starts to execute *Tx.X* (*t1*). On the other hand, *thr.2* remembers the transaction ID '*X*.' In order to predict a conflict, *thr.2* sends a request to each opponent thread for inquiring the remaining time until the commit of the opponent transaction (*t2*). After that, *thr.2* compares *T1* the remaining time of *Tx.X* sent back from *thr.1* with *T2* the remaining time until the conflict will be caused between *Tx.X* and *Tx.Y* (*t3*). In this case, as *T2* is shorter than *T1*, *thr.2* waits for being allowed to start *Tx.Y* and sends a *Waiting* message to *thr.1*. After a while, when *T1* becomes shorter than *T2*, *thr.1* sends a *Wakeup* message to *thr.2* for prompting *thr.2* to execute its transaction (*t4*). When *thr.2* receives this *Wakeup* message, *thr.2* starts to execute *Tx.Y*. In this way, *thr.2* can avoid causing a conflict with the minimum waiting time. To implement this conflict prediction, we define two messages, *Waiting* and *Wakeup*, on the coherence protocol for accurate conflict prediction.

2.4 Usability of Temporal Data

As mentioned in Sect. 2.3, it is important to avoid futile waiting by estimating the minimum waiting time based on historical temporal data. However, the execution paths of a transaction can sometimes vary because of such as branch instructions. Hence, such the temporal values of a transaction can also vary on each execution, and the estimation accuracy for the minimum waiting time will severely decline. In such a case, a conflict or futile waiting will not be avoided and the total performance can not be improved.

To address this, we introduce a criterion for deciding temporal data should be considered or not. We define a parameter 'conflicts per commit,' and uses it as the criterion. Initially, the proposed mechanism estimates the minimum waiting time at each transaction beginning. Once, 'conflicts per commit' exceeds a given threshold, the mechanism cease to use temporal data and make a transaction wait for the opponent transaction to commit, because the estimation accuracy for the minimum waiting time should not be reliable. Detailed implementation and behavioral model will be explained in Sect. 3.4.

3. Implementation

In this section, we describe the additional hardware units required for implementing the conflict prediction on HTM and how threads execute their transactions.

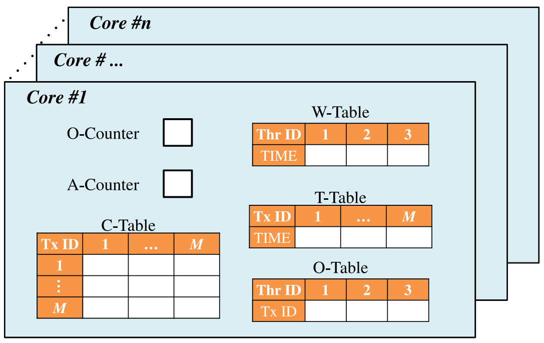


Fig. 4 Additional hardware units for the proposed conflict prediction.

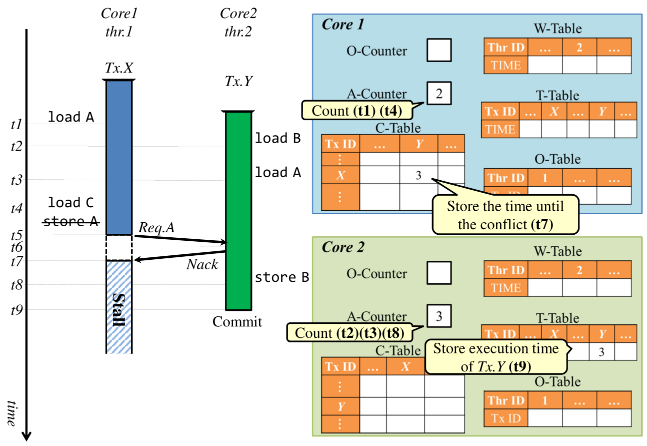


Fig. 5 The mechanism for remembering historical data.

3.1 Additional Hardware Units

To implement the conflict prediction described in Sect. 2, we have installed following hardware units in each core. For achieving the conflict prediction, some temporal data of transactions should be managed and used as parameters. However, the temporal data such as whole execution time of a transaction will vary at each execution even when its execution path does not change, because of cache misses or stalls. Hence, we use the number of memory accesses as an approximation of execution time.

Opponent-Counter (O-Counter): This counter records the number of threads which are executing opponent transactions.

Access-Counter (A-Counter): This counter records the number of memory accesses which are issued in the current transaction.

Conflict-Table (C-Table): This table stores the number of memory accesses. When a transaction which is running on the own core conflicts with another transaction, the core stores the number of memory accesses up to that time. It is used as the approximate value for the length of the period from the transaction start to the conflict.

Wait-Table (W-Table): This table stores the remaining number of memory accesses until allowing other threads to execute their transactions. This value is decremented at every memory access by the own thread. When the value becomes negative, the thread sends a *Wakeup* message to the thread which is remembered as a waiting thread on this table.

Time-Table (T-Table): This table stores the total number of memory accesses issued in the past execution of each transaction.

Opponent-Table (O-Table): This table stores IDs of transactions running on other threads.

The number of memory accesses in a transaction can vary because of execution path variation. Hence, when committing a transaction, the values for the transaction in *C-Table* and *T-Table* are updated if the number of memory accesses in the current execution is smaller than the stored past values. The reason for remembering and using the past minimum value is that wasteful waits and conflicts should be avoided as much as possible.

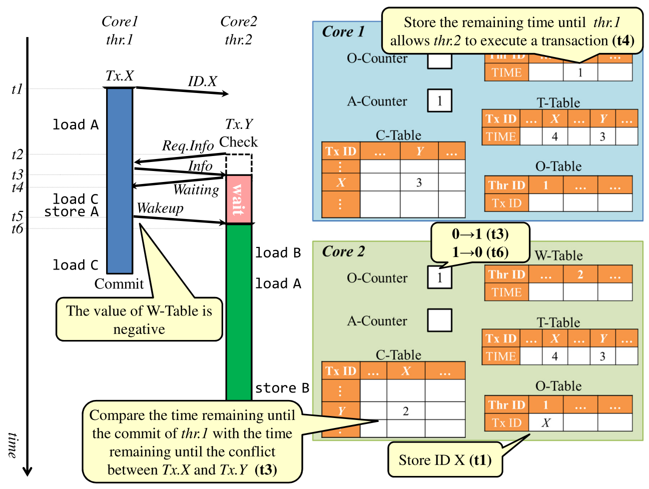
Some simple functional units for calculating temporal data are also required and the logic for them will cost some area overhead. However, the calculations are very simple and the area overhead should be negligible. We will discuss the hardware cost for these additional units in detail in Sect. 4.4.

3.2 How to Remember Historical Data of Conflicts

In this section, we will describe how threads remember historical data for predicting conflicts. Figure 5 illustrates an example. First, *Core1* increments the value of its *A-Counter* for remembering the number of memory accesses, when *thr.1* issues `load A` (*t1*). Similarly, *Core2* increments the value of its *A-Counter* when *thr.2* issues `load B` (*t2*) and `load A` (*t3*). After *thr.1* issues `load C` (*t4*), *thr.1* tries to issue `store A` (*t5*) and a conflict is detected (*t6*) because *thr.2* has already accessed to the address A. In this case, *thr.1* receives a *Nack* from *thr.2*, and stalls *Tx.X* (*t7*). At the time, as *thr.1* receives the *Nack*, *Core1* increments the value of its *A-Counter*, and copies the value to *C-Table*. In this example, *Core1* stores '3' on *C-Table* as the approximate value representing the time from the start of *Tx.X* to the conflict between *Tx.X* and *Tx.Y*. In this way, *Core1* remembers historical data about this conflict on *C-Table*. After that, *thr.2* issues `store B` and *Core2* increments the value of its *A-Counter* (*t8*). Finally, the value of its *A-Counter* is copied to *T-Table* as the approximate total execution time of *Tx.Y*, when *thr.2* commits *Tx.Y* (*t9*). As mentioned above, threads manage historical data for predicting conflicts.

3.3 How to Predict and Avoid Conflicts

In this section, we will describe how a thread predicts a future conflict and avoids it. Figure 6 shows an example where a thread predicts that a conflict will be caused. In Fig. 6, assume that *thr.1* and *thr.2* had already conflicted with each other, and each of them remembers execution time of its



Core1 thr.1
Core2 thr.2
Tx.X
ID.X
load A
Tx.Y Check
Req.Info
Info
Waiting
Wakeup
load C
store A
load C
Commit
Wait
load B
load A
store B
The value of W-Table is negative
Compare the time remaining until the commit of thr.1 with the time remaining until the conflict between Tx.X and Tx.Y (t3)
Store the remaining time until thr.1 allows thr.2 to execute a transaction (t4)
0→1 (t3)
1→0 (t6)
Store ID X (t1)
Core 1
Core 2
O-Counter
A-Counter
C-Table
T-Table
O-Table
W-Table
time

Fig. 6 The mechanism of the conflict prediction.

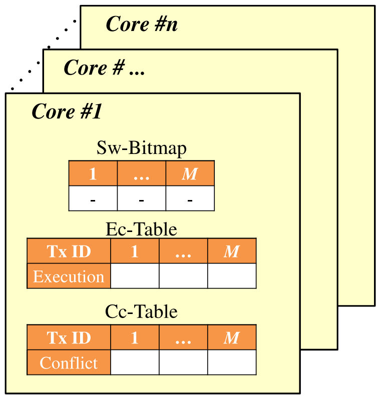


Fig. 7 Additional hardware units for accuracy validation.

transaction and the time until the conflict between *Tx.X* and *Tx.Y*.

First, *thr.1* sends the transaction ID '*X*' to all the other threads when *thr.1* starts to execute *Tx.X* (*t1*). *Core2* stores the ID '*X*' sent from *thr.1* on *O-Table*. After that, *thr.2* sends a request for inquiring the remaining time until the commit of the transaction to each opponent thread stored on *O-Table*, in order to predict when should *thr.2* start *Tx.Y* (*t2*).

Receiving this request, *thr.1* calculates the remaining time and gets the value '3,' by subtracting '1' the value of its *A-Counter* from '4' the execution time which is remembered on *C-Table*. Afterwards, *thr.1* sends '*X*' as the transaction ID and the calculated result '3' to *thr.2*. Receiving them, *thr.2* predicts whether a conflict will be caused or not by referring to the saved historical data. In this example, *thr.2* checks whether '*Y*' is stored as one of the opponent transaction of *Tx.X* on *C-Table* or not, and compares the time remaining until the commit of *Tx.X* and the time remaining until the conflict between *Tx.X* and *Tx.Y*. If the time remaining until the commit of *Tx.X* is shorter than the time remaining until the conflict between *Tx.X* and *Tx.Y*, *thr.2* can start to execute *Tx.Y*. In this case, the time until the commit of *thr.1* is longer than the time until the conflict between *Tx.X* and *Tx.Y*. Therefore, *thr.2* waits to start *Tx.Y* (*t3*).

As *thr.2* predicts that a conflict will be caused, *thr.2* stores '1' as the number of opponent threads on *O-Counter*, and sends a *Waiting* message, which piggybacks the time remaining until the conflict between *Tx.X* and *Tx.Y*, to *thr.1*. When *thr.1* receives this *Waiting* message, *thr.1* subtracts the time remaining until the conflict between *Tx.X* and *Tx.Y* from the time remaining until the commit *Tx.X* (*t4*). As a result, *Core1* stores '1' on *W-Table* as the remaining time until *thr.1* allows *thr.2* to execute *Tx.Y*.

Afterwards, the value of *W-Table* becomes negative when *thr.1* issues `store A` (*t5*). Therefore, *thr.1* sends a *Wakeup* message to *thr.2* for prompting *thr.2* to start *Tx.Y*. Receiving this *Wakeup* message, *Core2* decrements the value of its *O-Counter*. At this time, *thr.2* starts *Tx.Y* because the value of *O-Counter* becomes '0' and *thr.2* knows that no opponent transaction is running (*t6*). As mentioned above, *thr.1* and *thr.2* can avoid causing a conflict without wasteful waiting time.

3.4 Accuracy Validation for Waiting Time Estimation

As explained in Sect. 2.4, we introduce a criterion for deciding temporal data should be considered or not, because inappropriate waiting time may severely deteriorate the performance of HTM. For calculating 'conflicts per commit' which we adopt as the criterion, we have installed two tables and one bitmap in each core.

Switching-Bitmap (Sw-Bitmap): This bitmap records whether temporal data should be considered or not for each transaction.

Execution count-Table (Ec-Table): This table records the number of commits of each transaction.

Conflict count-Table (Cc-Table): This table records the number of conflicts of each transaction.

Remind that each thread estimates the minimum waiting time before starting a transaction, as long as 'conflicts per commit' does not reach a given threshold. For calculating 'conflicts per commit,' each thread counts the number of conflicts and commits on *Ec-Table* and *Cc-Table*.

When committing a transaction, if the associated value in *Ec-Table* reaches 128, 'conflicts per commit' for the transaction is calculated by using the associated values in *Ec-Table* and *Cc-Table*, and these values are reset to zero. If the calculated result is over a given threshold, the associated bit in *Sw-Bitmap* is set and it is broadcast to all cores. Afterwards, historical temporal data will be never used for scheduling the transaction because the reliability of historical temporal data is regarded as low.

4. Performance Evaluation

In this section, we describe the evaluation results, and estimate the additional hardware cost.

Table 1 Simulation parameters

Processor	SPARC V9
#cores	32 cores
clock	1 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	1 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	8 GBytes
latency	450 cycles
Interconnect network	hierarchical switch topology
link latency	14 cycles
Additional hardware	(discussed in Sect. 4.4)
per core	569 bytes
per processor	9.1 KBytes

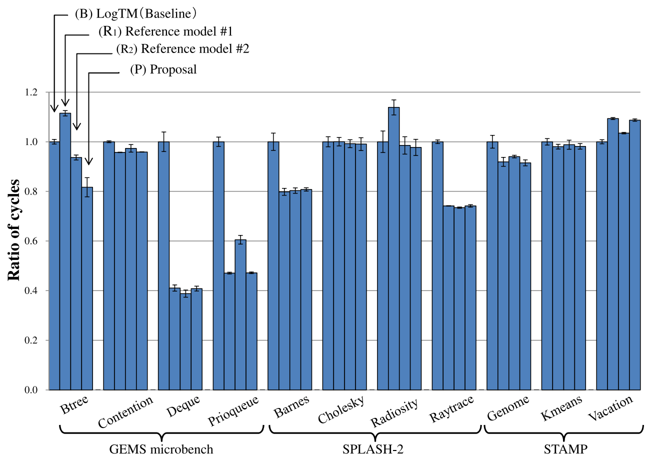


Fig. 8 Execution cycles ratio of each slowest thread.

4.1 Evaluation Environment

We used a full-system execution-driven functional simulator *Wind River Simics* [5] in conjunction with customized memory simulators built on *Wisconsin GEMS* [6] for evaluation. Simics provides a SPARC-V9 architecture and boots Solaris 10, and GEMS provides a detailed timing simulation for the memory subsystem. The detailed configuration of the simulated processor is shown in Table 1. The topology and the link latency of interconnect network are defined based on LogTM [4]. We have evaluated the execution cycles of 11 workloads from GEMS microbench, SPLASH-2 benchmark suite [7], and STAMP benchmark suite [8] with 16 threads.

4.2 Evaluation Results

The evaluation results with following four HTM configurations are shown in Fig. 8, Fig. 9 and Table 2.

(B) LogTM (baseline)

(R_1) Reference model #1; uses only opponent transaction IDs as historical data (as described in Sect. 2.2).

(R_2) Reference model #2; always uses both of opponent transaction IDs and temporal data as historical data (as described in Sect. 2.3).

(P) Proposal; initially uses both of opponent transaction IDs and temporal data, and ceases to use temporal data if they are regarded as unreliable with the mechanism described in Sects. 2.4 and 3.4.

Figure 8 shows the execution cycles of each benchmark program, or the execution cycles of the slowest thread of each benchmark program. On the other hand, Fig. 9 shows the sum total of execution cycles of all 16 threads. Each bar in both figures is normalized to the baseline (B). For simulating multi-threaded execution on a full-system simulator, the performance variability [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.

The legend in Fig. 9 shows the breakdown items of the sum total cycles. They represent the execution cycles out of transactions (*Non_trans*), the execution cycles in the transactions which are committed/aborted (*Good_trans*/*Bad_trans*), the aborting overheads (*Aborting*), the exponential backoff cycles (*Backoff*), the stall cycles (*Stall*), the barrier synchronization cycles (*Barrier*), and the waiting cycles before starting transactions by the proposed conflict prediction (*Wait*).

As a result of the evaluation, the performance of many programs is improved with both reference models (R_1) and (R_2). However, the performance of some programs with (R_2) is better than (R_1) and the the others with (R_1) is better than (R_2).

Figure 10 plots 'conflicts per commit.' As shown in Fig. 8 and Fig. 10, 'conflicts per commit' of each program whose performance declines with (R_2) is around or higher than 100%. Thus, we defined 100% as the threshold for (P). In other words, when committing a transaction, if its associated value in *Ec-Table* reaches 128 and its associated value in *Cc-Table* is over 128, the associated bit in *Sw-Bitmap* is set. No division logic is required but only a comparator is required for implementing it.

Thereby with (P), the performance of almost all programs is better than both of (R_1) and (R_2). As a result, the execution cycles are reduced 59.2% at a maximum, and 16.8% on average with (P).

We can get the following general conclusions from the evaluation result.

1. When the accuracy of the proposed conflict prediction is high, even the transactions which will bring conflicts can run in parallel as long as possible, and the performance is improved. The prediction accuracy will be high when the execution time of each transaction does not largely vary.
2. Even when the prediction accuracy is low, frequent conflicts are avoided and the conflict overhead can be

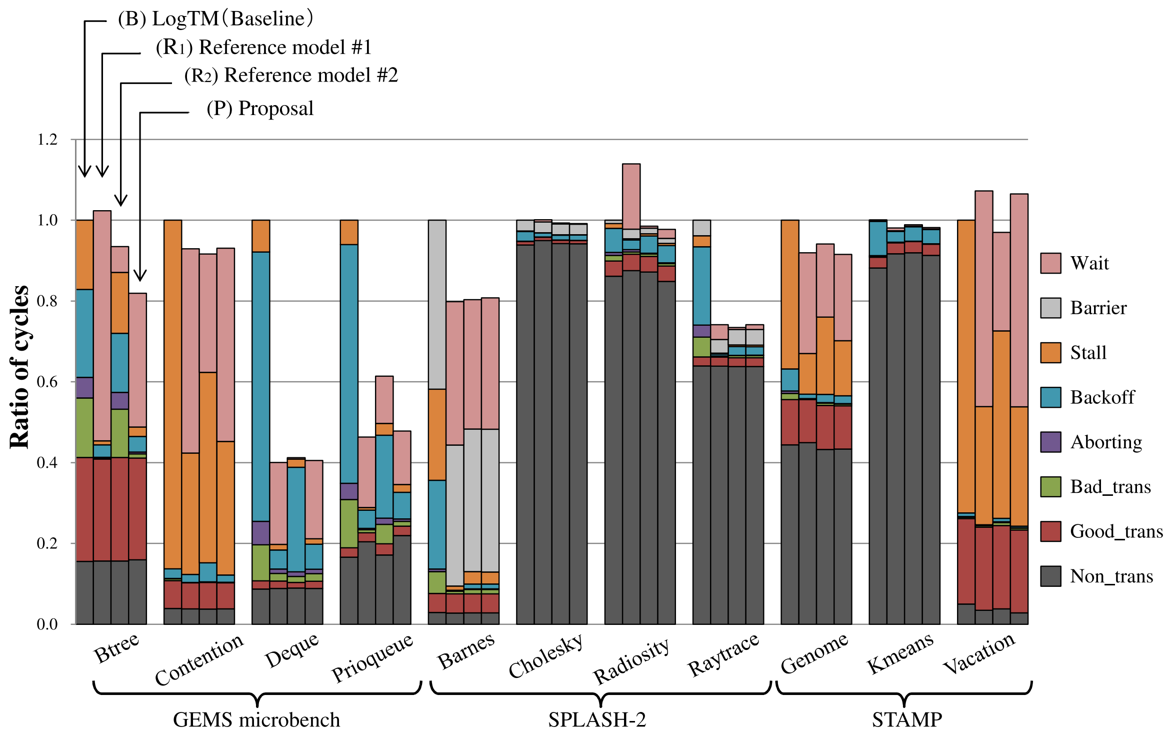


Fig. 9 The sum of the total execution cycles ratio.

Table 2 Reduced cycle rate.

	(R$_1$)	(R$_2$)	(P)
ave	12.5%	14.7%	16.8%
max	58.9%	61.2%	59.2%

reduced by delaying a transaction until its potential opponent transactions commit.

3. The reason for the low prediction accuracy with some programs is the execution path variation in the transactions, especially resulting from the branch instructions in the transactions.

We go to the detailed examination of these results in the following section.

4.3 Detailed Examination

In the following, we take up some characteristic programs and analyze them.

(1) Deque, Prioqueue

The performance of Deque and Prioqueue is improved with both (R$_1$) and (R$_2$). We examined these programs and it is found that each program has only one transaction. Hence, with (B), many threads access to the same address when the transaction is executed in parallel. Therefore, conflicts and aborts were caused repeatedly with (B). By contrast, the number of conflicts with (R$_1$) and (R$_2$) is smaller than that of (B), and *Stall*, *Aborting*, and *Bad_trans* are reduced.

However, the performance of these programs with (R$_1$) is better than (R$_2$). To investigate the origin of this result,

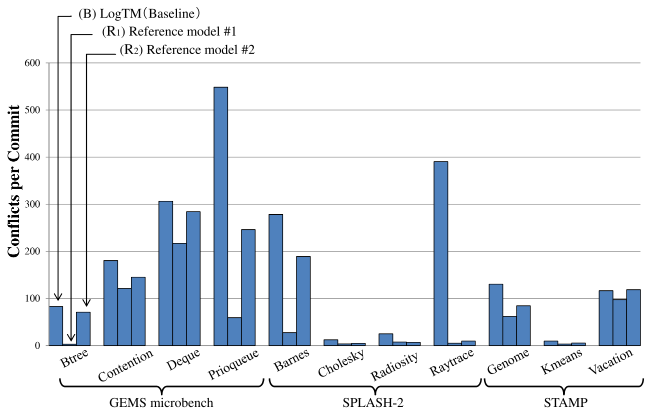


Fig. 10 Conflicts per commit.

we examined these programs in detail, and it is found that the number of memory accesses tends to change in the programs, because the execution path varies due to branch instructions. Hence, with (R$_2$), conflict predictions sometimes fail and *Stall*, *Aborting*, and *Bad_trans* of some programs are larger than with (R$_1$).

As shown in Fig. 10, 'conflicts per commit' is high and the reliability of historical temporal data should be low with these programs. Hence, the performance with (R$_1$) is better than with (R$_2$), and (P) can achieve almost equal performance with (R$_1$).

(2) Vacation

The performance of Vacation with (R$_1$) declines. With (R$_1$), threads can avoid conflicts, but the prediction sometimes results in false positive. This is because all transactions in Vacation include branch instructions, and threads rarely access the same addresses. As a result, conflict prediction fails and causes wasteful waits frequently.

On the other hand, the performance deterioration with (R_2) is smaller than that of (R_1) although *Stall* with (R_2) is larger than that of (R_1). This is because threads do not need to wait for opponent threads to commit, and can execute these transactions in parallel.

The proposal (P) can not achieve good performance with Vacation. This is because 'conflicts per commit' is slightly higher than 100% the threshold we have defined, as shown in Fig. 10, and historical temporal data are not used, although (R_2) can achieve better performance than (R_1). This increases *Wait* futilely. Hence, as one of our future works, we should adjust the threshold for each program considering its characteristics and behaviour.

Stall is also still large with (P). As mentioned above, in Vacation, transactions have some branch instructions, and the low prediction accuracy resulting from them brings many stalls. In addition, once a stall is caused, the stall lasts rather long because the transactions in Vacation include many instructions. Genome also has similar characteristics to Vacation although the total performance of Genome is slightly improved.

(3) Btree

The performance of Btree declines with (R_1) because wasteful waits are caused frequently. We examined this program, and it is found that Btree has two transactions. One (we call it *Tx.Insert*) includes both read and write accesses to a shared variable, and the other (we call it *Tx.Lookup*) includes only read accesses to the shared variable. Therefore, with (B), many conflicts are caused between *Tx.Insert* and *Tx.Insert*, but *Tx.Insert* rarely conflicts with *Tx.Lookup*.

If *Tx.Insert* conflicts with *Tx.Lookup* even once, *Tx.Insert* and *Tx.Lookup* are remembered as a pair of conflicted transactions with (R_1). As a result, these two transactions are serialized, and wasteful waiting time is increased with (R_1) because *Tx.Insert* rarely conflicts with *Tx.Lookup* practically.

On the other hand, with (R_2), a thread which predicts that a conflict will be caused does not need to wait for opponent threads to commit, and threads can execute these transactions in parallel.However, as same as Deque and Prioqueue, the transactions in Btree include branch instructions which lead conflict predictions to fail. This results in rather large *Stall*, *Aborting* and *Bad_trans*.

With (P), the performance of Btree is quite improved. To investigate the origin of this result, we measured and compared 'conflicts per commit' of *Tx.Insert* and *Tx.Lookup*. As shown in Fig. 11, it is found that *Tx.Lookup* rarely conflicts and 'conflicts per commit' of two transactions are very different. The number of commits of *Tx.Lookup* is indeed four times as many as *Tx.Insert*. Hence, by using historical temporal data only for *Tx.Lookup*, (P) can drastically reduces *Wait* compared with (R_1), while

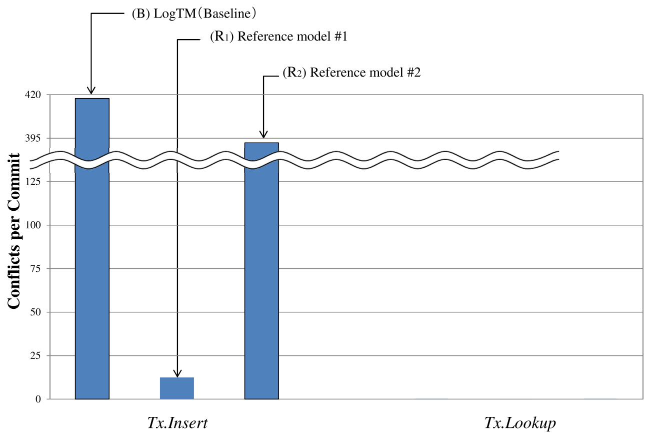


Fig. 11 Conflicts per commit of each transaction of Btree.

keeping *Backoff* and *Bad_trans* as small as (R_1).

(4) Barnes

The performance of Barnes with (R_2) does not much differ from with (R_1). This is because Barnes has a transaction whose number of memory accesses is very large exceptionally, and hardware units such as *A-Counter* sometimes overflow. Thereby, because threads can not use temporal data for conflict predictions, each thread which predicts that a conflict will be caused waits for the opponent thread to commit, and the performance with (R_2) is quite equal to (R_1).

(5) Cholesky, Kmeans

The performance of Cholesky and Kmeans is not improved with both (R_1) and (R_2). This is because *Non_trans* occupies most of the total cycles of these programs. Therefore, the ratio of the performance improvement is just a little smaller than the other programs.

4.4 Additional Hardware Cost

To implement the conflict prediction, *A-Counter* should have an enough bit width for counting as many as the maximum number of memory accesses in transactions. Then, we have measured how many load/store instructions are executed in each transaction. As a result, it is found that if *A-Counter* has 13-bit width, it does not overflow with almost all practical programs.

Similarly, we have measured how many transactions are included in each program for investigating the required size of *C-Table* and *T-Table*. As a result, it is found that 17 transactions are included at a maximum. Besides, each entry of *C-Table* and *T-Table* needs 13-bit width because it should remember the maximum number of memory accesses in a transaction. Therefore, *T-Table* costs 221 bits (17 entries × 13 bits) and *C-Table* costs 3,757 bits (17^2 entries × 13 bits).

In order to count up to the maximum number of threads, for a 16-core processor which can execute 16 threads, *O-Counter* should have 4-bit width. Incidentally, *W-Table* needs 13-bit width, and costs 195 bits for a 16-

core processor. Similarly, *O-Table* needs 5-bit width, and *O-Table* costs 75 bits for a 16-core processor.

In addition, each core needs further hardware units to implement the accuracy validation mechanism for historical temporal data, as mentioned in Sects. 2.4 and 3.4. First, to records the number of conflicts and commits of each transaction, each core needs *Ec-Table* and *Cc-Table*. Each entry of these tables needs 8-bit width because it should remember up to 128. Therefore, each table costs 136 bits (17 entries $\times$ 8 bits). Besides, each core needs a 17-bit width *Sw-Bitmap*.

Consequently, for a 16-core processor, the total additional hardware is estimated at only 569 bytes per core, or only 9.1 Kbytes per processor. Indeed this size is calculated based on the benchmark programs, but it will be enough also for many practical applications because STAMP programs deal with real-world problems. Even if it is insufficient for a certain program, the program can correctly run with slightly deteriorated prediction accuracy.

4.5 Message Exchange Overhead

In this section, we estimate the overhead for messages such as *Wakeup* and *Committed* implemented by enhancing coherence protocol for the conflict prediction. The overhead can be expressed as $M \times L$, where M is how many times messages are sent, and L is the link latency between cores. In this paper, we assumed that the link latency is 14 cycles as shown in Table 1, based on [4]. We examined all benchmark programs and it is found that Btree has the largest overhead ratio among all programs, and messages are sent about 300 thousands times. Therefore, the total overhead cycles across all 16 threads can be calculated as $300{,}000 \times 14$, and the result is about four millions. On the other hand, the total execution cycles of Btree is about 200 millions. Hence, even if the overhead across 16 threads is concentrated on only one thread, the overhead ratio of Btree which has the largest overhead ratio is only 2%. In addition, the average overhead ratio of the all programs is only 0.5%. Thus, the overhead will be quite negligible.

5. Related Work

So far, various techniques for HTM have been proposed. To reduce the cost for re-execution, some techniques for optimizing rollbacks have been proposed [10]–[12]. Besides, many thread scheduling techniques for reducing conflicts by controlling transactional sequences have been also proposed [13]–[15].

To improve the performance of parallel execution, Yoo et al. [16] have proposed a method based on the concept of adaptive transaction scheduling (ATS). ATS dynamically dispatches transactions and controls the number of concurrently executing transactions. Thereby, ATS can improve the performance of workloads which lack for parallelism because of high contentions. The throughput of Radiosity is improved 1.97x with ATS. However, the improvement with almost all programs other than Radiosity and Deque is quite small and lower than only 5%. On the other hand with our proposal, the execution cycles is maximally reduced 59.2% with Deque. This means that the throughput is improved to about 2.45x with Deque. In addition, the execution cycles of five programs are reduced over about 20% with our proposal, or the throughput is improved over about 25%. Especially, the performance of Raytrace is not improved at all with ATS, while it is improved about 30% with our proposal. As same as Btree, whose characteristic is illustrated in Fig. 11, Raytrace also has two different types of transaction; one rarely conflicts and the other frequently conflicts. With our proposal, the conflict overhead of Raytrace is reduced by serializing the latter type of transaction which conflicts frequently while the other type of transaction can still run in parallel. On the other hand, with ATS, all transactions are serialized when the overall conflict frequency becomes higher than a threshold.

Blake et al. [17] have proposed a method focusing on common memory locations which are accessed in multiple transactions. In the method, locality of memory access on each consecutive execution of a transaction is called 'similarity' and the similarity is calculated with bloom filter. If the similarity exceeds a threshold, the transactions are serialized. The performance of this method is evaluated with STAMP benchmark suite [8] and rather improved. However, the evaluation results are not practical because they are evaluated with 64 threads.It is known that the programs in STAMP benchmark suite bring so many conflicts and aborts when they are executed with many threads. The performance with 64 threads is drastically lower even than the performance of being executed serially with only one thread. Hence, the estimation of the baseline performance and the performance improvement in [17] should be quite unfair, because only serializing transactions can increase performance.

Armejach et al. [18] have proposed a prediction mechanism called HARP to avoid repetitive aborts. HARP is inspired by branch prediction and achieves high accuracy of a conflict prediction by considering the latest behavior of transactions and locality in conflicting memory references. The approach used in HARP is partly similar to our conflict prediction, but there are some distinct differences. Specifically, only the transactions, which are predicted not to conflict each other, can run in parallel on HARP. On the other hand with our proposal, even the transactions which will conflict each other can run partially in parallel, or their execution can be partially overlapped. In addition, HARP requires 2.06kB memory cells per core, and the hardware cost is about 3.6x as large as the cost for our conflict prediction. Incidentally, some benchmark programs seem to be arbitrarily modified for evaluating HARP in the paper [18]. For example, the paper describes that Vacation has only one kind of transaction, and the evaluation results shown in the paper illustrates that pretty large cycles for barrier synchronization are consumed in Vacation program. However, Vacation originally has three kinds of transactions and has no barrier synchronization. Hence, although the paper shows that

HARP can significantly improve the performance of Vacation unlike our proposal, their results should not be simply compared.

Akpinar et al. [19] have proposed some novel ideas for conflict resolution policies on HTMs, such as alternating priorities of transactions in many various ways based on the total number of stalled or aborted transactions. In addition, they have considered the most common performance pathologies [20] such as InactiveStall and FriendlyFire. However, all these methods can not efficiently avoid causing conflicts because they can control transaction behavior only after starting the transactions, and the performance gain is up to only 15%. On the other hand, our novel transaction scheduling can quite improve the performance of many practical programs, because it can avoid causing conflicts before starting transactions.

6. Conclusions

In this paper, we propose a conflict prediction for transaction scheduling on HTM to avoid conflicts in advance. The conflict prediction uses opponent transaction IDs and temporal data about transactions. When a thread predicts that a conflict will be caused, it waits for being allowed to start its transaction to avoid the conflict. We have evaluated the conflict prediction by comparing with LogTM, through experiments with GEMS microbench, SPLASH-2 benchmark suite, and STAMP benchmark suite. The evaluation results show that HTM with the conflict prediction decreases the total execution cycles 59.2% at a maximum, and 16.8% on average with 16 threads. However, the conflict prediction sometimes fails if a transaction includes branch instructions and its execution path changes. Thereby, a conflict may be caused or a thread may wait wastefully. Therefore, one of our future work is to improve the prediction accuracy by considering variation of execution paths in transactions.

Acknowledgments

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

References

[1] M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," Proc. 20th Annual Int'l Symp. on Computer Architecture (ISCA'93), pp.289–300, May 1993.

[2] V. Leis, A. Kemper, and T. Neumann, "Exploiting Hardware Transactional Memory in Main-Memory Databases," Proc. 30th Int'l Conf. on Data Engineering (ICDE'2014), pp.580–591, Jan. 2014.

[3] K. Mashita, S. Miyake, R. Yamada, and T. Tsumura, "Yet Another Waiting Mechanism for Hardware Transactional Memory," Proc. 3rd Int'l Workshop on Computer Systems and Architectures (CSA'15), pp.400–403, Dec. 2015.

[4] K.E. Moore, J. Bobba, M.J. Moravan, M.D. Hill, and D.A. Wood, "LogTM: Log-based Transactional Memory," Proc. 12th Int'l Symp. on High-Performance Computer Architecture (HPCA'06), pp.258–269, Feb. 2006.

[5] 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.

[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, Sept. 2005.

[7] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," Proc. 22nd Annual Int'l. Symp. on Computer Architecture (ISCA'95), pp.24–36, 1995.

[8] C.C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford Transactional Applications for Multi-Processing," Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08), pp.35–46, Sept. 2008.

[9] A.R. Alameldeen and D.A. Wood, "Variability in Architectural Simulations of Multi-Threaded Workloads," Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03), pp.7–18, Feb. 2003.

[10] J.E.B. Moss and A.L. Hosking, "Nested Transactional Memory: Model and Preliminary Architecture Sketches," Science of Computer Programming, vol.63, no.2, pp.186–201, 2006.

[11] M.J. Moravan, J. Bobba, K.E. Moore, L. Yen, M.D. Hill, B. Liblit, M.M. Swift, and D.A. Wood, "Supporting Nested Transactional Memory in LogTM," Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.359–370, Oct. 2006.

[12] A. McDonald, J. Chung, B.D. Caristrom, C.C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural Semantics for Practical Transactional Memory," Proc. 33rd Annual Int'l Symp. on Computer Architecture (ISCA'06), pp.53–65, 2006.

[13] A. Shriraman, S. Dwarkadas, and M.L. Scott, "Flexible Decoupled Transactional Memory Support," Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA'08), vol.36, no.3, pp.139–150, 2008.

[14] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero, "Eazyhtm, Eager-lazy Hardware Transactional Memory," Proc. 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-42), pp.145–155, 2009.

[15] M. Lupon, G. Magklis, and A. González, "A Dynamically Adaptable Hardware Transactional Memory," Proc. 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-43), pp.27–38, 2010.

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

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

[18] A. Armejach, A. Negi, A. Cristal, O. Unsal, P. Stenstrom, and T. Harris, "Harp: Adaptive abort recurrence prediction for hardware transactional memory," Proc. 20th Int'l Conf. on High Performance Computing (HiPC'13), pp.196–205, Dec. 2013.

[19] S. Tomić, E. Akpinar, A. Cristál, O. Unsal, and M. Valero, "A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory," Proc. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11), vol.18, pp.270–279, 2013.

[20] J. Bobba, K.E. Moore, H. Volos, L. Yen, M.D. Hill, M.M. Swift, and D.A. Wood, "Performance Pathologies in Hardware Transactional Memory," Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA'07), pp.81–91, 2007.

Keisuke Mashita received his B.E. degree from Nagoya Institute of Technology in 2015. Currently, he is a 1st year graduate student in the Department of Scientific and Engineering Simulation, Nagoya Institute of Technology, Japan. His current research interest is computer architecture. He is a student member of IPSJ.

Maya Tabuchi was admitted to Nagoya Institute of Technology in 2012. Currently, he is a 4th year undergraduate student in the Department of Computer Science, Nagoya Institute of Technology, Japan. His current research interest is computer architecture.

Ryohei Yamada received his B.E. degree from Nagoya Institute of Technology in 2014. Currently, he is a 2nd year graduate student in the Department of Scientific and Engineering Simulation, Nagoya Institute of Technology, Japan. His current research interest is parallel processing. He is a student member of IPSJ.

Tomoaki Tsumura received his M.E. and Ph.D. degree from Kyoto University in 1998 and 2004 respectively. After graduating from the Ph.D. candidate course of the Graduate School of Informatics, Kyoto University in 2001, he joined the university as a research associate. He joined Toyohashi University of Technology in 2004, and then joined Nagoya Institute of Technology as an associate professor in 2006. His current research interests are computer architecture, applications of parallel processing, and brain-type information processing. He is a member of ACM, IEEE-CS, IPSJ and IEICE.