A Concurrency Control in Hardware Transactional Memory Considering Execution Path Variation

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

***Abstract*—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, conflicts can degrade the performance of HTM because of the overhead for re-execution of transactions. To address this problem, various transaction scheduling algorithms for avoiding conflicts have been proposed. However in the existing algorithms, execution path variation is not considered at all. Some transactions have branch instructions, and they cause execution path variations of transaction, resulting in poor efficacy of the scheduling algorithms. In this paper, we propose a novel concurrency control based on the execution time of transactions with considering execution path variation. The result of the experiment shows that the execution time of HTM is reduced 61.6% at a maximum, and 13.8% on average with 16 threads.**

I. Introduction

On multi-core processors, multiple threads can run in parallel for speed-up. Therefore, parallel programming becomes more important for programmers to utilize processor performance. In order to run 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 they lead to poor scalability. In addition, *lock* is not so convenient because it is difficult to adjust locking granularity for each program.

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 in the shared memory, TM saves both new and old values (*version management*). TM also keeps tracks of each memory access, checking whether each requested datum has been accessed yet by another transaction or not (*conflict detection*). *Hardware Transactional Memories* (HTMs) have hardware mechanisms for version management and conflict detection. Therefore, each of version management and conflict detection costs only a small delay overhead.

In general, transactions, which have conflicted on a shared variable once each other, tend to conflict repeatedly on the same shared variable if they are executed in parallel again. This conflict repetition will bring severe performance degradation of HTMs. Transaction scheduling algorithms considering conflict patterns or conflict frequencies will avoid some of such repetitive conflicts. However, the execution path and the execution time of each transaction can vary, and this will deteriorate efficiency of the scheduling algorithm. To address this problem, we propose a concurrency control mechanism based on a transaction scheduling considering execution path variation. With this control mechanism, before starting a transaction, a thread estimates the execution time of the transaction by predicting execution paths, and delays starting the transaction so that the expected conflict will come slightly after the commit of competing transactions.

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

1) We propose a novel concurrency control mechanism for effectively avoiding conflicts on HTM. The mechanism precisely predicts execution time of each transaction, and adjust the execution timing of the transaction.
2) We consider execution path variation in the transactions for precisely estimating execution time of each transaction.
3) We evaluate the transaction scheduling. The results show that the execution cycles can be reduced 61.6% at a maximum and 13.8% on average.

II. A Concurrency Control Considering Execution Path Variation

In this section, we describe overviews of HTM, and point out a problem of general HTM. After that, we propose a transaction scheduling for avoiding conflicts based on the execution time which is estimated considering execution path variation.

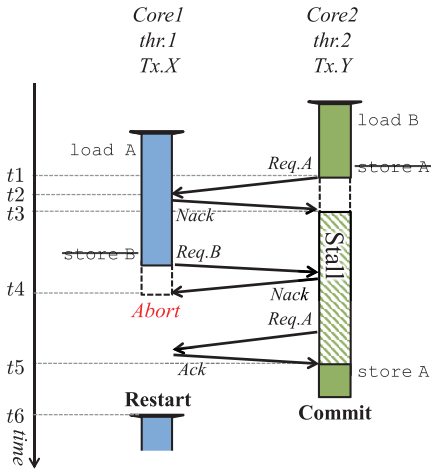
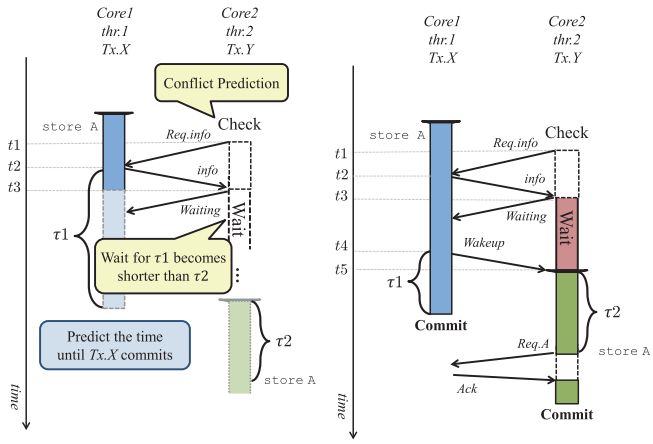


Fig. 1. Conflict resolution on general HTM.



(a) Predicting a conflict. (b) Avoiding a conflict.

Fig. 2. Conflict prediction based on execution time.

A. Conflict Resolution on HTM and its Problem

In this section, we describe how conflicts are resolved on LogTM[2] which is the most general HTM system. Figure 1 shows an example where the thread *thr.1* executes the transaction *Tx.X* and *thr.2* executes *Tx.Y*. Now, assume that *thr.1* has issued `load A` and *thr.2* has issued `load B`. First, when *thr.2* tries to issue `store A`, *thr.2* sends *thr.1* a request for accessing the address A (at $t1$). After *thr.1* receives the request from *thr.2*, *thr.1* detects a conflict because *thr.1* has already accessed A. Then, *thr.1* sends *thr.2* a *Nack* ($t2$). When *thr.2* receives the *Nack* from *thr.1*, *thr.2* gives up issuing `store A` and stalls *Tx.Y*, waiting for *thr.1* to commit ($t3$). After that, when *thr.1* tries to issue `store B`, *thr.2* detects another conflict and *thr.2* sends a *Nack* to *thr.1*. If *thr.1* stalls at this time, *thr.1* and *thr.2* will be waiting for each other, and a deadlock will be caused. To avoid causing the deadlock, *thr.1* aborts *Tx.X* ($t4$). Thereby, *thr.2* is now allowed to issue `store A` ($t5$). After a while, *thr.1* restarts *Tx.X* ($t6$). In this way, threads avoid deadlocks on HTM. However, stalling and restarting transactions may degrade the performance of HTM.

B. Conflict Prediction based on Execution Time

As mentioned in Section II-A, the performance of HTM can be degraded by stalls of transactions originating from conflicts. Especially, transactions which have conflicted once each other tend to conflict repeatedly, because the threads often access the same shared variables when they are executed again. In this paper, we propose a concurrency control mechanism for avoiding conflicts based on this knowledge. Before a thread starts a transaction, the thread predicts whether the transaction will cause a conflict or not. To this end, the thread examines whether the transaction had conflicted with some running transactions on the other threads or not in the past. If the transaction had conflicted with one of the running transactions, the thread compares two temporal parameters $\tau 1$ and $\tau 2$, where $\tau 1$ is the predicted remaining time until the competing transaction commits, and $\tau 2$ is the predicted remaining time until the conflict will be caused. If the thread will access the conflicted address after the competing thread committed, namely $\tau 1 < \tau 2$, the thread predicts that a conflict will not be caused this time, and starts the transaction. On the contrary, if $\tau 1 > \tau 2$, the thread waits for $\tau 1$ becomes shorter than $\tau 2$ without starting the transaction. In order to implement this conflict prediction, two temporal data of each transaction should be remembered. 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 after the transaction starts. Before starting a transaction, a thread predicts a conflict using these temporal data.

Figure 2 (a) shows an example where a thread predicts that a conflict will be caused, and Fig. 2 (b) shows an example the thread avoids the conflict by waiting before starting its transaction. Assume that *Tx.X* had already conflicted with *Tx.Y*, and each thread remembers the temporal data of the two transactions for conflict prediction. First, when *thr.2* tries to execute *Tx.Y*, *thr.2* sends all the other threads a *Req.Info* which is a request acquiring for running transaction ID and remaining time until the competing transaction commits ($t1$). After *thr.1* receives the *Req.Info*, *thr.1* sends *thr.2* the transaction ID '*X*' and $\tau 1$ the predicted remaining time of *Tx.X* ($t2$). Then, *thr.2* compares $\tau 1$ sent back from *thr.1* with $\tau 2$ the time until the conflict will be caused between *Tx.X* and *Tx.Y* ($t3$). If $\tau 1$ is shorter than $\tau 2$, *thr.2* starts to execute *Tx.Y*. On the other hand, as Fig. 2 (a) shows, if $\tau 2$ is shorter than $\tau 1$, *thr.2* waits for being allowed to start *Tx.Y*. At this time, *thr.2* sends a *Waiting* message to *thr.1*. After a while, as Fig. 2 (b) shows, when $\tau 1$ becomes shorter than $\tau 2$, *thr.1* sends a *Wakeup* message to *thr.2* for prompting *thr.2* to execute *Tx.Y* ($t4$). When *thr.2* receives this *Wakeup* message, *thr.2* starts to execute *Tx.Y* ($t5$). In this way, threads can avoid causing a conflict.

Stall is also a 'waiting' mechanism for conflict resolution. In contrast to stall, the waiting mechanism before starting transactions will not cause any other new conflicts, because the thread waits without accessing any addresses.

C. Predicting Execution Time Considering Execution Path Variation

Although threads can avoid many conflicts by the conflict prediction as mentioned in Section II-B, threads may still cause conflicts if threads fail the conflict prediction. Especially, when the execution path of a transaction varies because of conditional branches, the past temporal data for the transaction become unreliable, and the accuracy of the conflict prediction will largely decline. Therefore, the transaction execution time should be predicted precisely considering such execution path variation. However, unlike branch prediction algorithms, local history of branch instructions in transaction can not be used, because the execution path of a transaction need to be predicted before the transaction starts. Hence, we introduce a conflict prediction mechanism which exploits the idea of *global branch prediction*[3]. Global branch prediction manages a shared history of all conditional branches, and predicts the direction of a branch instruction based on the pattern history of other recent branch instructions. The idea of global branch prediction can also be considered as that the execution path after a branch instruction is predicted from the execution path just before the branch instruction.

To apply this idea to conflict prediction, we employ pattern history of load/store accesses as an execution path expression. We define the pattern of load and store '*global load/store history.*' In the conflict prediction mechanism, the execution time of each transaction is remembered associated with the global load/store history just before the transaction. When the transaction is executed again, the remembered execution time associated with the current global load/store history is acquired and used for conflict prediction. In this way, our conflict prediction method can consider execution time variation of a transaction by using global load/store history as a key.

III. Implementation

In this section, we will show how threads predict conflicts and execute their transaction considering execution path variation.

A. How to Remember Historical Data

In this section, we will explain how threads remember historical data for conflict prediction. 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 because of cache misses or stalls, even if its execution path does not change. Hence, we use the number of memory accesses as an approximate expression of execution time.

In the example shown in Fig. 3, *thr.1* first gets the global load/store history before starting *Tx.X* ($t1$). In this example, assume that the length of global load/store history pattern being used for conflict prediction is two. Now, the latest two memory accesses before starting *Tx.X* are load and load, and the set 'load-load' is remembered as global load/store history. Then, when *thr.2* tries to issue `store A` ($t2$), *thr.1* detects a conflict and sends *thr.2* a *Nack* ($t3$). After receiving the *Nack*, *thr.2* remembers '3' as the approximate value representing the time from the start of *Tx.Y* to the conflict between *Tx.X* and *Tx.Y* ($t4$). After that, *thr.2* stalls *Tx.Y*. When each thread commits its transaction, the thread remembers the approximate total execution time of its transaction associated with global load/store history which was refered to before the thread starts to execute the transaction. In this example, the total number of memory accesses in *Tx.X* is '5' because *thr.1* issues `load B`, `load C`, `load A`, `store C`, and `store B` during the execution of *Tx.X*. Then, *thr.1* remembers the number '5' as the approximate total execution time associated with global load/store history of *Tx.X* 'load-load' ($t5$). In this way, each threads remember the data which are required for the conflict prediction.

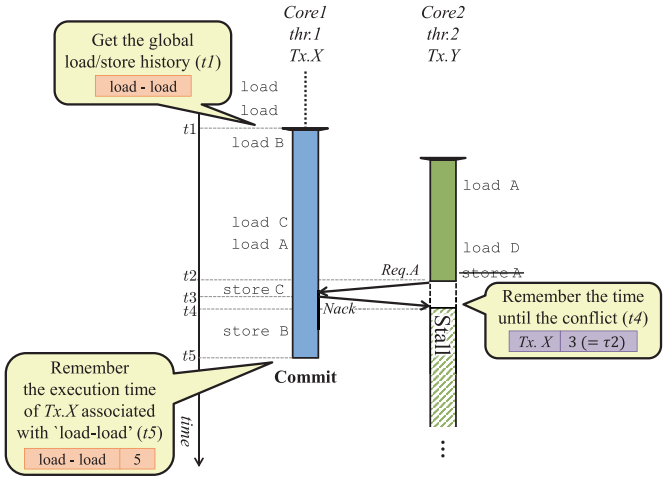


Fig. 3. How to remember Historical Data.

B. How to Predict and Avoid Conflicts Considering Execution Path Variation

In this section, we explain how a thread predicts a future conflict with two examples. Assume that some temporal data for conflict prediction are collected for a while after the situation shown in Fig. 3. Then *thr.1* and *thr.2* try to execute *Tx.X* and *Tx.Y* respectively again. This situation is illustrated in Fig. 4. When *thr.1* tries to execute *Tx.X*, *thr.1* acquires the global load/store history, and estimates the execution time of *Tx.X* as the remembered time associated with the global load/store history. In this example, the global load/store history just before *Tx.X* is 'store-load.' Therefore, *thr.1* predicts that the execution time of *Tx.X* will be '3' which is remembered associated with 'store-load' ($t1$). After that, when *thr.2* tries to execute *Tx.Y*, *thr.2* sends all the other threads a *Req.Info* for predicting a conflict ($t2$). Receiving the *Req.Info*, *thr.1* calculates the remaining time until *Tx.X* commits. In this example, *thr.1* gets the value '2' as the remaining time by subtracting '1' as the number of memory accesses which *thr.1* has already issued from '3' as the predicted execution time of *Tx.X*. Therefore, *thr.1* sends *thr.2* the transaction ID '*X*' and '2' as the remaining time until *Tx.X* commits ($t3$). Then, *thr.2*

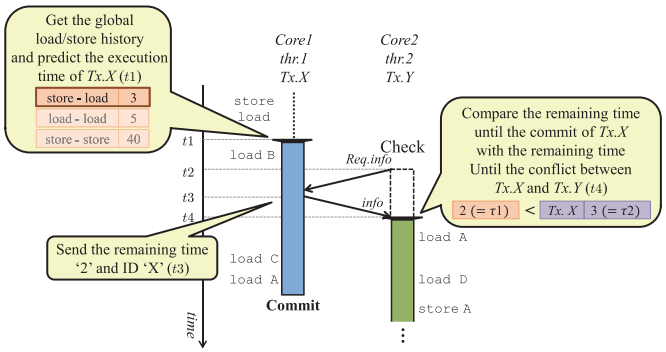


Fig. 4. Execution flow when *thr.2* predicts that a conflict will not caused.

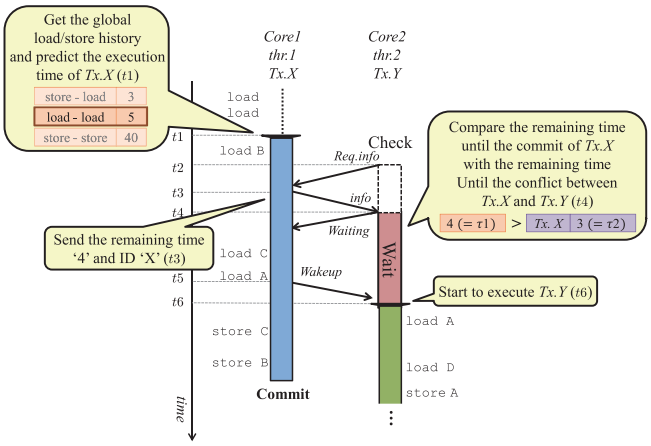


Fig. 5. Execution flow when *thr.2* predicts that a conflicts will be caused.

compares '2' which is sent back from *thr.1* with '3' as the time until the conflict between *Tx.X* and *Tx.Y* ($t4$). *Thr.2* predicts that a conflict will not be caused and starts to execute *Tx.Y*, because '3' as the time until the conflict between *Tx.X* and *Tx.Y* is longer than '2' as the time until *thr.1* commits.

In contrast to Fig. 4, Fig. 5 shows an example where a thread predicts that a conflict will be caused. In this example, the execution path of *Tx.X* varies although *thr.1* executes the same transaction as Fig. 4. When *thr.1* tries to execute *Tx.X*, *thr.1* gets the global load/store history for predicting the execution time of *Tx.X* ($t1$). In this example, the global load/store history just before *Tx.X* is 'load-load,' and *thr.1* predicts the execution time of *Tx.X* as '5' which is associated with 'load-load.' After that, when *thr.2* tries to execute *Tx.Y*, *thr.2* sends a *Req.Info* ($t2$). Receiving this, *thr.1* gets the value '4' as the remaining time until *Tx.X* commits, and *thr.1* sends *thr.2* '4' and the transaction ID 'X' ($t3$). Now, the value '3' which *thr.2* remembers associated with *Tx.X* is smaller than the value '4' which *thr.2* receives from *thr.1*. Therefore, *thr.2* predicts that a conflict will be caused, and waits for being allowed to start *Tx.Y*. At this time, *thr.2* sends *thr.1* a *Waiting* message with '3' as the time until the conflict between *Tx.X* and *Tx.Y* ($t4$). After a while, as *Tx.X* issues `load C` and `load A`, the remaining time of *Tx.X* becomes smaller than '3' which is sent from *thr.2* as the time remaining until the conflict. Therefore, *thr.1* predicts that a conflict will not be caused this time, and sends *thr.2* a *Wakeup* message ($t5$). When *thr.2* receives it, *thr.2* starts to execute *Tx.Y* ($t6$).

In this way, threads predict a conflict before starting its transaction and avoid the conflict by waiting for being allowed to execute its transaction. If a thread does not consider execution path variation, the thread may start to execute its transaction too early to avoid the conflict or the waiting time can be longer than the minimum necessary time to avoid the conflict. To address this problem, in our transaction scheduling, threads get the global load/store history just before executing the transaction. Hence, even if the execution time of the transaction varies because of execution path variation, threads can rather precisely predict the execution time of its transaction and avoid conflicts.

IV. Performance Evaluation

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

A. Evaluation Environment

We used a full-system execution-driven functional simulator *Wind River Simics*[4] in conjunction with customized memory simulators built on *Wisconsin GEMS* [5] 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 I. The topology and the link latency of interconnect network are defined based on LogTM. We have evaluated the execution cycles of 11 workloads from GEMS microbench, SPLASH-2 benchmark suite [6], and STAMP benchmark suite [7] with 16 threads. We configured the length of global load/store history as eight, and the latest eight memory accesses are used for conflict prediction.

B. Evaluation Results

The evaluation results with following four HTM configurations are shown in Fig. 6.

- (**B**) LogTM (baseline)
- (**R**) Reference model; predicts conflicts by using the past shortest execution time of each transaction, without considering execution path variation
- ($\mathbf{P_S}$) Proposal #1; predicts conflicts by using the past *shortest* execution time of each transaction considering execution path variation.
- ($\mathbf{P_L}$) Proposal #2; predicts conflicts by using the past *longest* execution time of each transaction considering execution path variation.

Fig. 6 shows the total sum of execution cycles of all 16 threads and its breakdown. Each bar in both figures is normalized to the baseline (**B**). For simulating multi-threaded execution on a full-system simulator, the performance variability [8] 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. 6.

TABLE I
SPECIFICATIONS OF THE SIMULATED PROCESSOR

Processor	SPARC V9
#cores	32 cores
clock	4 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	3 cycles
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

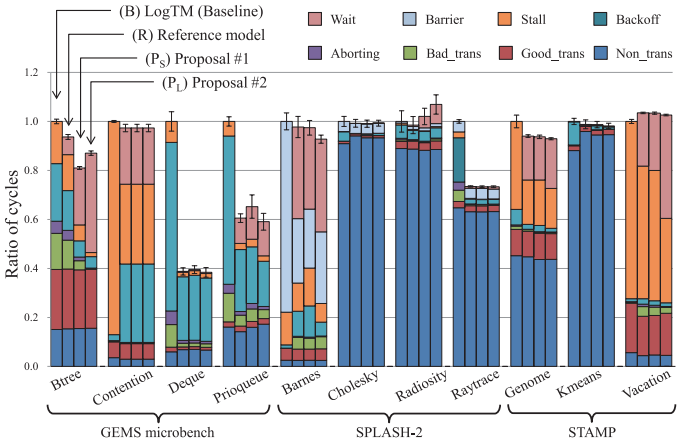


Fig. 6. The sum of the total execution cycles ratio.

When the global load/store history and the execution path of the transaction do not correlate to each other, the total number of memory accesses which the thread issues can be different from the remembered value, even though the thread has the same global load/store history pattern. Therefore, we have evaluated two variants of the proposal. With (P_S), if the number of memory accesses in the current execution is smaller than the remembered value, the value is updated with the current value. Hereby, the past smallest number of memory accesses is used for conflict prediction. On the other hand, with (P_L), the past largest number of memory accesses is used for conflict prediction. As a result, with (P_S), because threads underestimate the remaining time until the transaction commits, the wasteful waiting time can be reduced. On the other hand, because threads overestimate the remaining time until the transaction commits with (P_L), the waiting time will increase compared with (P_S), while the number of conflicts can be reduced than (P_S). Also with respect to (R), two variations can be assumed as same as the proposal models. We preliminarily evaluated the two variants for (R), and it is found that the model which uses the past shortest time achieves considerably better performance than the model which uses the past longest time. Hence, we show only the former model for (R) in Fig. 6. The legend in Fig. 6 shows the breakdown items of the total sum of cycles. They represent the waiting cycles before starting transactions by the proposed conflict prediction (*Wait*), the barrier synchronization cycles (*Barrier*), the stall cycles (*Stall*), the exponential backoff cycles (*Backoff*), the aborting overheads (*Aborting*), the execution cycles in the transactions which are aborted/committed (*Bad_trans*/*Good_trans*), and the execution cycles out of transactions (*Non_trans*).

As shown in the figures, both (P_S) and (P_L) achieve better performance than (B) with most of all programs. In summary, the execution cycles are reduced 61.6% at a maximum, and 13.8% on average with (P_L). In addition, either (P_S) or (P_L) achieves better performance than (R) with all benchmark programs, except for Radiosity.

We estimate the additional hardware cost required for the conflict prediction mechanism. For a 16-core processor, the total additional hardware cost is estimated at only 1.6 KBytes per core, or 26 KBytes per processor. The required size is calculated as for remembering historical data of the transactions which appear in the benchmark programs, but it will be enough also for many practical applications because STAMP programs deal with real-world problems. Even when it is insufficient for a certain program, the program can correctly run with slightly deteriorated prediction accuracy.

C. Detailed Examination

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

a) Prioqueue: The performance of Prioqueue with (P_S) declines compared with (R) by increasing *Bad_trans*, *Aborting*, and *Stall*. We examined the program, and it is found that even with the same global load/store history, the execution time of a transaction largely varies in each execution. Therefore, with (P_S), the conflict prediction often fails and conflicts are caused. On the other hand, the performance with (P_L) is better than (R). Although *Wait* with (P_L) is larger than (R) and (P_S), threads can avoid many conflicts. Thereby, the performance with (P_L) is improved over about 40% by reducing *Bad_trans*, *Aborting*, *Backoff*, and *Stall*.

b) Btree: *Aborting*, *Stall*, and *Backoff* are largely reduced with Btree on (P_S) and (P_L), although *Wait* is larger than other programs. 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, many conflicts are caused between *Tx.Insert* and *Tx.Insert*, but *Tx.Insert* rarely conflicts with *Tx.Lookup*. In *Tx.Insert*, whether a conflict is caused or not depends on its execution path. Figure 7 shows a digest code inside *Tx.Insert*. The transaction *Tx.2* in the function BTreeNode_split() includes an if-then-else statement, and the execution path inside *Tx.2* varies depending on whether the 'node' is a leaf or a non-leaf in the binary tree. When the function is called at line 5, the first argument passed to the formal argument 'node' is not a leaf but always the root, as shown in the figure, and the condition at line 22 becomes

```
1  void BTree_insert(BTree* tree, key_type_t key, char* value, int tid){
2    :
3    if(BTreeNode_isFull(tree->m_root)){
4      BEGIN_TRANSACTION(3);
5      median = BTreeNode_split(tree->m_root, tree, tid);
6      :
7      COMMIT_TRANSACTION(3);
8    }
9    :
10   while(!node->isLeaf){
11     :
12     if(BTreeNode_isFull(child)){
13       median = BTreeNode_split(child, tree, tid);
14       :
15     }}
16   :
17 }
18 :
19 key_ptr_pair BTreeNode_split(BTreeNode *node, BTree* tree, int tid)
       {
20   :
21   BEGIN_TRANSACTION(2);
22   if(node->isLeaf){
23     :
24   }else{
25     :
26   }
27   COMMIT_TRANSACTION(2);
28   :
29 }
```

Fig. 7. A digest code of Btree.

always false. On (P_S) and (P_L), this can be predicted based on the load/store pattern before BTreeNode_split() being called, and threads can effectively avoid conflicts compared with on (R). On the other hand, the performance of (P_L) is lower than (P_S). This is because wasteful wait is caused and *Wait* increases.

c) Deque: The performance of all models is improved with Deque. We examined this program and it is found that the transaction in this program is composed of a few instructions and the execution time of the transaction is very short. Therefore, waiting time for avoiding a conflict is very short and *Wait* is very small. As a result, the total execution cycles are considerably reduced by avoiding conflicts with few waiting overhead.

d) Barnes: The performance of both (P_S) and (P_L) with Barnes is better than (B), and the performance gain mainly results from decrease of *Barrier*. This is because many conflicts and aborts are avoided, and no threads are largely delayed by conflicts and aborts. Besides, the performance of (P_L) is not much different from (P_S). This is because Barnes has a transaction whose number of memory accesses is exceptionally very large and the time until the conflict is very small. Therefore, there is not much room to reduce *Wait* with both (P_S) and (P_L).

e) Cholesky, Kmeans: *Bad_trans*, *Aborting*, *Backoff*, and *Stall* of these programs slightly decrease. However, total performance gain is small. This is because *Non_trans* occupies most of the total cycles of these programs. Especially, Kmeans has transactions whose execution time do not vary, even though their execution paths vary. Therefore, the performance of Kmeans is improved only 1% at a maximum compared with (R).

f) Radiosity, Vacation: In these programs, the execution time of a transaction can vary even though the thread has the same global load/store history in each execution. As a result, conflict prediction often fails. We examined these programs and it is found that Vacation has a transaction whose execution time varies based on a random input value. On the other hand, Radiosity has a transaction whose execution time becomes shorter and shorter as being executed repeatedly. Our proposal can not precisely predict conflicts for such types of transactions. One possible approach to address this problem is that if huge variation is observed on the execution time of a transaction with the same global load/store history, the transaction is excluded from the target of conflict prediction.

g) Contention, Raytrace, Genome: The performance of both (P_S) and (P_L) with these programs does not much differ from (R), even though *Stall* is considerably reduced. We examined these programs, and it is found that transactions whose number of memory accesses does not change occupy more than 80% of total number of transactions. Thereby, the ratio of the performance improvement is just a little smaller than the other programs.

V. Related Work

So far, various techniques for improving HTM have been proposed. To reduce the re-execution overhead of transactions, some techniques for optimizing rollbacks have been proposed [9], [10], [11]. Besides, many thread scheduling techniques for reducing conflicts by controlling transactional sequences have also been proposed [12], [13], [14].

To improve the performance of parallel execution, Yoo et al. [15] 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 transaction scheduling, the execution cycles is maximally reduced 61.6% with Deque. This means that the throughput is improved to about 2.60x with Deque. In addition, the execution cycles of four programs are reduced over about 20% with our proposal, or the throughput is improved to 1.25x. Especially, the performance of Raytrace is not improved at all with ATS, while it is improved about 30% with our method.

Blake et al. [16] have proposed a method focusing on common memory locations which are accessed in multiple transactions. In this method, locality of memory access on a transaction executed consecutively is called 'similarity' and the similarity is calculated by using Bloom filter. If the similarity exceeds a threshold, the transactions are executed sequentially. The performance of this method is evaluated

with STAMP benchmark suite [7] 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 [16] should be quite unfair, because only serializing transactions can increase performance.

Akpinar et al. [17] have proposed some novel ideas for conflict resolution policies on HTMs, such as alternating priorities of transactions in many various ways based on the execution time or the total number of stalled transactions and so on.

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 transaction scheduling, 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 larger than the cost for our method.

All these methods, anyway, do not consider execution path variation in transactions which is caused by branch instructions. Hence, performance can not be improved significantly in case the execution path in a transaction varies. On the other hand, our novel transaction scheduling can improve the performance of many practical programs because it can avoid causing conflicts even if the execution path in a transaction varies.

VI. Conclusions

In this paper, we propose a concurrency control mechanism considering execution path variation. With this mechanism, threads estimate the execution time of transactions based on the idea of global branch prediction, and predict conflicts before starting a transaction by using remembered data associated with the global load/store history patterns. We have evaluated the method by comparing with both LogTM and a reference model, through experiments with GEMS microbench, SPLASH-2 benchmark suite, and STAMP benchmark suite. The evaluation results show that HTM with the conflict prediction mechanism decreases the total execution cycles 61.6% at a maximum, and 13.8% on average, with 16 threads.

However, the conflict prediction sometimes fails when the execution time of a transaction varies with the same global load/store history pattern. Therefore, we need to improve the prediction accuracy by considering another data of transactions such as load/store addresses. In addition, we also should consider whether to exclude transactions, whose execution time can not be predicted precisely, from the target of this concurrency control mechanism.

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. 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 (HPCA'06)*, Feb. 2006, pp. 254–265.

[3] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proc. 24th Annual IEEE/ACM Int'l Symp on Microarchitecture(MICRO-24)*. ACM, 1991, pp. 51–61.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] E. Moss and T. Hosking., "Nested Transactional Memory: Model and Preliminary Architecture Sketches." in *Science of Computer Programming*, 2006, pp. 186–201.

[10] 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," in *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2006, pp. 1–12.

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

[12] A. Shriraman, S. Dwarkadas, and M. L. Scott., "Flexible Decoupled Transactional Memory Support," in *Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA'08)*, 2008, pp. 139–150.

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

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

[15] 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.

[16] 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.

[17] 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.

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