Priority-Based Conflict Resolution for Hardware Transactional Memory

Ryohei YAMADA*, Koshiro HASHIMOTO* 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 scalabilities, and Transactional Memory (TM) has been proposed and studied for lock-free synchronization. On TMs, transactions are executed speculatively as long as there is no conflict on shared variables. On HTMs, which are the hardware implementations of TM, if a speculative execution of a transaction fails, the re-execution of the transaction should wait a period prescribed by a backoff algorithm to avoid further conflicts. However, the performance of HTM may be decreased drastically by wastefully long backoff periods. To address this problem, in this paper, we propose a new algorithm to set a value called Priority on each transaction, and the transaction which should be aborted is selected according to Priority instead of the initiated time of transactions. The result of the experiment shows that the execution time of HTM is reduced 59.9% in maximum, and 11.2% in average with 16 threads.

I. INTRODUCTION

On multi-core processors, multiple threads run in parallel for speed-up. In order that multiple threads may 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 these problems, Transactional Memory (TM) [1] has been proposed as a lock-free synchronization mechanism.

On TM, transactions are executed speculatively as long as there is no conflict on shared variables. Furthermore, the interim results of transactions may be discarded because transactions are executed speculatively. Hence, when a transaction modifies a value on memory, TM generally needs to save both new and old values (version management). Moreover, TM keeps track of memory accesses, checking whether each requested datum has been accessed yet by another transaction or not (conflict detection). On Hardware Transactional Memories (HTMs) [2], [3], which are the hardware implementations of TM, 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.

In general HTMs, when a conflict is detected, only a logically elder transaction can continue its execution. Consequently, if a younger transaction is aborted and re-executed immediately, the transaction will mostly conflict with the elder transaction again, and will result in another abort. Accordingly, the thread which the aborted transaction is assigned waits a backoff period using a backoff algorithm before re-executing the transaction to avoid bringing a conflict again. In many cases, TM systems adopt Exponential Backoff algorithm. This algorithm defines backoff period as increasing exponentially according to how many times the transaction is aborted repeatedly. However, this algorithm may define a backoff period much longer than necessary. If wasteful waiting time is caused frequently, the performance of HTM will be decreased drastically. To address this problem, in this paper, we propose a new effective criterion for selecting the transaction which should be aborted according to Priority by considering the transaction execution time and the number of transactional loads and stores.

II. RESEARCH BACKGROUND

In this section, we describe overviews of TM and HTM.

A. Transactional Memory

Transaction mechanism has been used for achieving data consistency on database systems. TM is an implementation of the transaction mechanism for shared memory synchronization. On TM, a transaction is defined as an instruction sequence which covers a critical section, and the transaction needs to satisfy atomicity and serializability. To ensure atomicity and serializability, TM keeps track of memory accesses, checking whether each requested datum on the shared memory has been accessed by another transaction yet or not. Specifically, when a transaction tries to access the same memory address which has been accessed by another transaction, TM detects it as a conflict between the transactions. If TM detects a conflict, TM selects transactions among the conflicted transactions and stalls the selected transactions. Then, if one of the conflicted transaction is aborted to avoid deadlocks, the aborted transaction will be re-executed later. On the other hand, if there occurs no conflict through a transaction, TM commits the transaction.

As far as there is no conflict among some transactions, the transactions can concurrently run under the TM without any blocking. Therefore, compared with lock-based methods, TM provides generally better scalability. The mechanisms for version management and conflict detection can be implemented in hardware or software. Some TM systems operate completely in
software (STMs) [4], [5]. However, STM has more overheads than HTMs.

B. Conflict Detection

HTM generally uses signatures inspired by Bulk [6] to summarize transactions’ load and store accesses, and detects conflicts on coherence requests. Each processor core has two signatures for Read and Write addresses. These signatures are updated by using a logical sum of the present signature and a decoded Read/Write address. Therefore, the updated signature holds not only the currently accessed address, but also the addresses accessed in the past. Then, if a logical product of the current signature and a decoded Read/Write address requested by another processor may be same as the decoded address, a conflict is detected.

Two policies for conflict detection are defined as follows. They differ in respect of when conflicts are detected.

**Eager Conflict Detection:** When a memory address is accessed in a transaction, it is checked whether other transactions already have accessed to the same address or not.

**Lazy Conflict Detection:** When a transaction tries to commit, it is checked whether other transactions accessed to addresses which are accessed by the transaction or not.

With Lazy conflict detection policy, it takes much time to detect a conflict after the conflict is caused. Consequently, more transactional execution time will be wasted than Eager conflict detection. Hence, we adopt Eager Conflict Detection.

Here, we explain Eager conflict detection and its conflict resolution. Fig. 1 shows an example where Thread1 executes Tx.X and Thread2 executes Tx.Y, and Thread1 has issued load A before Thread2 has issued load A. First, when Thread1 tries to issue store A (t1), a conflict is detected (t2) because Thread2 has already accessed to address A. In this case, Thread1 receives NACK from Thread2, and stalls Tx.X, waiting for Thread2 to commit (t3). Afterwards, when Thread2 tries to issue store A (t4), another conflict is detected because Thread1 has already accessed to the same address. In this case, as Tx.X is elder than Tx.Y, Thread2 aborts its Tx.Y (t5).

While Thread2 waits a backoff period, Thread1 receives ACK and resumes Tx.X (t6). After that, Thread2 restarts Tx.Y (t8). Incidentally, the backoff period is generally defined by an algorithm called Exponential Backoff. This algorithm defines backoff period as increasing exponentially according to how many times the transaction is aborted repeatedly.

C. Version Management

On TM, interim results of transactions may be discarded because transactions are executed speculatively. Hence, when a transaction modifies a value on memory, HTM generally needs to save both new and old values. Two policies for version management are defined as follows;

**Eager Version Management:** Old values and kept in the area called log which is in a virtual memory and new values are stored into cache blocks. Therefore, the time required for committing transactions is relatively short. On the other hand, the time for aborting transactions is relatively long because the old values must be restored into cache blocks.

**Lazy Version Management:** New values are kept in the area called write buffer which is in a virtual memory and old values are kept in cache blocks. Therefore, the time required for aborting transactions is relatively short because the old values are kept in cache blocks. On the other hand, the time for committing transactions is relatively long because the new values must be stored into cache blocks.

Every transaction definitely includes one commit and the commit cannot be omitted. Therefore, there is almost no room to reduce the overheads for commit on Lazy version management. On the other hand, the number of aborts could be reduced by improving transaction scheduling. Therefore, there is room to reduce the overheads for abort on Eager version management TMs. Hence, we adopt Log-based Transactional Memory Signature Edition (LogTM-SE) [3] which uses Eager Conflict Detection and Eager Version Management.

III. PRIORITY-BASED CONFLICT RESOLUTION

In this section, we point out a problem of the traditional HTM, and propose a new criterion for selecting one of conflicted transactions as the preferential transaction.

A. Problem of the Traditional Backoff Algorithm

We have proposed a transaction scheduling for relieving specific conflict patterns [7]. This transaction scheduling can prevent the performance deterioration caused by the specific
In this case, Thread1 commits using these parameters. Here, (where time is caused. In this figure, after a long period, a transaction may continue to wait wastefully prescribed by Exponential Backoff algorithm. However, with this algorithm, a transaction may continue to wait wastefully long. 

Here, Fig. 1 shows an example where such wasteful waiting time is caused. In this figure, after Thread1 receives ACK, Thread1 commits Tx.X (t7). Therefore, Thread2 can re-execute Tx.Y without a conflict between itself and Thread1. However, as Thread2 waits a backoff period, Tx.Y executed by Thread2 may be drastically delayed from the commit of Thread1 (t8). In this case, Thread2 wastefully waits after Thread1 commits Tx.X until the end of a backoff period. As a result, the performance of HTM may severely decline.

B. Solution of Wasteful Waiting Time by using Priority

As the number of transactions which are executed in parallel increases, wasteful waiting time may be caused more frequently. In this paper, we propose a new criterion for selecting transactions which should be continued or aborted. The criterion is called Priority, and calculated for each transaction. Priority is defined as proportional to a progress of a transaction. When a conflict is detected, the Priorities of both transactions which have conflicted are compared. At this time, a transaction which has a higher Priority can continue its execution preferentially. Thereby, the transaction which has short remaining time until the transaction commits will continue its execution preferentially.

Here, we define following parameters for calculating Priority.

Transaction age (T): This represents how much time has passed since the start of the transaction.

Issued load/store instructions (L, S): This is the number of issued load/store instructions in the current transaction.

Past load/store instructions (L0, S0): This is the number of load/store instructions which are issued in the transaction, when it is executed in the past.

Using these parameters, Priority (P) is defined as follows;

\[ P = \frac{1}{\alpha/T + \beta(S_0 - S) + \gamma(L_0 - L)} \]  

where \( \alpha, \beta \), and \( \gamma \) represent weight factors of parameters. Here, \((S_0 - S)\) and \((L_0 - L)\) represent the predictive number of load/store instructions which will be issued hereafter until the transaction is committed. Thus, a transaction which has smaller the predictive number of load/store instructions should be executed preferentially. Incidentally, if an earlier transaction waits a commit of later one, the earlier one will keep shared variables accessed. Hence, the earlier transaction will conflict with other transactions. Therefore, an earlier transaction should be committed preferentially for releasing shared variables. Accordingly, Priority is defined as inversely proportional to the predictive number of load/store instructions which will be issued until the transaction is committed, and proportional to transaction age. When a conflict is detected, Priorities of both conflicted transactions are compared and the transaction which has higher Priority is selected as the preferential transaction.

C. Control for Selecting the Preferential Transaction by Using Priority

In this section, we explain a control for selecting the preferential transaction according to the proposed criterion. In Fig. 2, Thread1 executes Tx.X and Thread2 executes Tx.Y, and these threads have already issued load A. In this situation, Thread1 tries to issue store A and req.A which piggybacks Priority for comparing with an opponent transaction’s Priority is sent from Thread1 (t1). After that, a conflict is detected because Thread2 has already accessed to address ‘A.’ Then, Thread1 receives NACK from Thread2, and stalls Tx.X (t2). Afterwards, when Thread2 tries to issue store A, another conflict is detected because Thread1 has already accessed to the same address (t3). At this time, Thread1 which receives req.A calculates Priority, and Priorities of Tx.X and Tx.Y are compared. In this example, assume that Priority of Tx.X and Tx.Y are calculated as 1/2 and 1/5. Accordingly, Tx.Y which has lower Priority is aborted (t4), and Thread1 can continue Tx.X (t5). At this time, Core2 which executes Thread2 stores the pair of the opponent transaction ID ‘X’ and the conflicted
address ‘A.’ When a thread receives a request, the thread predicts whether a conflict will be caused or not. If the transaction ID of the transaction executed by the request sender and the address to which the sender tries to access are the same as the stored pair of the transaction ID and the address, the thread predicts that a conflict will be caused. Depending on this prediction, the transaction which should continue its execution preferentially is selected before the conflict is caused. Thereby, aborts can be avoided. Here, after Thread1 commits Tx.X, Thread2 needs to restart Tx.Y. Hence, we define Wakeup message by extending coherence protocol, and use this Wakeup message for prompting the opponents to continue their execution. In this example, Thread1 sends Wakeup message to Thread2 after Thread1 commits Tx.X (t6).

D. Control for Avoiding Further Conflicts Between the Same Transactions

As mentioned above, each core stores conflicted transaction IDs and conflicted addresses for selecting a preferential transaction. Fig. 3 shows a situation a little while after the situation shown in Fig. 2. At the situation shown in Fig. 3, assume that Core1 remembers that the thread which runs on Core1 has conflicted with Tx.Y at address ‘A’ in the past. Incidentally, Thread1 executes Tx.X and Thread2 executes Tx.Y.

First, Thread2 tries load A and sends req.A to Thread1 (t1), after Thread1 issues load A. At this time, Core1 finds that a transaction which was executed by Thread1 has conflicted with Tx.Y at address ‘A’ in the past. Then, it is predicted that these threads may conflict with each other again if Tx.Y continues. Accordingly, Priorities of Tx.X and Tx.Y are compared after Thread1, which receives an access request, calculates Priority, and one of the transactions is selected as the preferential transaction. In this example, assume that Priorities of Tx.X and Tx.Y are calculated as 1/2 and 1/5, and Thread1 continues Tx.X which has higher Priority. Accordingly, Thread1 sends Wait request to Thread2. Here, Wait request is defined by extending coherence protocol as same as Wakeup message described in section III-C. When Thread2 receives Wait request, Thread2 waits for being allowed to issue load A (t2). Hence, when Thread1 tries store A, Thread1 does not conflict with Thread2 and can continue Tx.X. After that, when Thread1 commits Tx.X, Thread1 sends Wakeup message to Thread2. Then, Thread2 can resume the execution of Tx.Y (t3). In this way, as transactions can wait the minimum required time, wasteful waiting is avoided.

IV. IMPLEMENTATION

In this section, we describe additional hardwares for implementing the proposed HTM and the execution flow of the HTM.

A. Additional Hardware

To implement the proposed HTM, we have installed following hardware units in each core.

**Load Counter (L-Counter)**: This counter records the number of load instructions which are issued in the current transaction.

**Store Counter (S-Counter)**: This counter records the number of store instructions which are issued in the current transaction.

**Prior Core ID (Pr-CoreID)**: This register stores the core ID of the core from which Wait request is sent.

**Posterior Core ID bits (Po-CoreID)**: This bitmap records the core IDs of the cores to which the own core sends Wait requests. When the total number of cores is n, this bitmap has n-bit width. When the thread which runs on the own core commits its
transformation, *Wakeup* messages are sent to the cores whose core IDs are recorded on this bitmap.

**Past Access Table (Past-Access):** This table stores the total number of load/store instructions which are issued in the past.

**Conflict Address Table (Conflict-Addr):** This table stores pairs of conflicted transaction IDs and conflicted addresses. When the thread which runs on the own core conflicts with other threads and aborts its transaction, the core stores the opponent transaction ID and the conflicted address.

Here, transaction age can be calculated by using *Timestamp* of each core on traditional LogTM-SE.

### B. Execution Flow

We describe an execution flow on the proposed HTM, using Fig. 5 and Fig. 6. In Fig. 5, when *Thread1* and *Thread2* issue *load A*, *L-Counters* are incremented for recording the number of the load accesses (t1, t2). After that, when *Thread2* tries *store A*, *Priority* is calculated using the number of counts stored in *L-Counter*, *S-Counter* and *Past-Access*. Then, *Thread2* sends a request which piggybacks *Priority* to *Thread1* (t3). Afterwards, the pair of conflicted transaction ID ‘X’ and conflicted address ‘A’ is stored in *Conflict-Addr*, and the core ID of *Core1* whose thread executes *Tx.X* is stored in *Pr-CoreID* (t4). Simultaneously, *Core2* sets a bit of *Core2* in *Po-CoreID* because *Thread1* on *Core1* is responsible for making *Thread2* on *Core2* resume. After *Thread1* commits *Tx.X*, the values of *L-Counter* and *S-Counter* are stored in *Past-Access*. Now, if a transaction includes branch instructions, the count of load/store instructions may drastically change according to the selected execution path. Accordingly, when the number of load/store instructions has already stored in *Past-Access*, *Past-Access* is updated by the arithmetic average of the present value of *Past-Access* and the values in *L-Counter/S-Counter*. In this way, deterioration of prediction accuracy for load/store instruction counts can be restrained. After the number of load/store instructions is stored, the committing thread sends *Wakeup* messages to the cores whose core IDs are stored in *Po-CoreID*, and clears *Po-CoreID*. In this example, *Thread1* sends *Wakeup* message to *Core2* (t5). Receiving this message, *Thread2* restarts *Tx.Y* and clears *L-Counter*, *S-Counter* and *Pr-CoreID* (t6).

Fig. 6 shows a situation a little while after the situation shown in Fig. 5. In Fig. 6, assume that *Core1* has already stored an entry for the pair of conflicted transaction ID ‘Y’ and conflicted address ‘A’ on *Conflict-Addr*. At first, *Thread2* tries *load A* after *Thread1* issues *load A*. At this time, as transaction ID ‘Y’ and address ‘A’ match the entry on *Conflict-Addr* of *Core1*, *Thread1* sends *Wait* request to *Thread2* and sets *Po-CoreID* (t1). On the other hand, *Thread2* which receives *Wait* request waits for being allowed to issue *load A*, and the core ID of *Core1* is stored in *Pr-CoreID* (t2). After that, the committing thread sends *Wakeup* messages to the cores whose core IDs are stored in *Po-CoreID*. In this example, when *Thread1* commits *Tx.X*, *Thread1* sends *Wakeup* messages to *Core2* (t3).

### V. Performance Evaluation

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

#### A. Evaluation Environment

We used a full-system execution-driven functional simulator *Wind River Simics*[8] in conjunction with customized memory simulators built on *Wisconsin GEMS* [9] 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. We have evaluated the execution cycles of four workloads from GEMS microbench and two workloads from SPLASH-2[10] with 8 and 16 threads. Incidentally, each of weight factors (\(\alpha\), \(\beta\), \(\gamma\)) in equation (1) described in section III-B is equally defined as “1.”
B. Evaluation Results

The evaluation results with following two HTMs are shown in Fig. 7.

(B) LogTM-SE (baseline)
(P) Proposed HTM

Fig. 7 shows the execution cycles of each HTM. Each bar is normalized to the total execution cycles of the baseline LogTM-SE (B). The legend shows the breakdown items of the total cycles. They represent the executed cycles out of transactions (Non-trans), the executed 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 for the proposed HTM (Wait). For the simulation of multi-threading on a full-system simulator, the variability performance [11] 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 this figure.

As shown in Fig. 7, the performance is improved with the proposed HTM (P) in many programs, as we expect. The proposed HTM (P) reduces the execution cycles 59.9% in maximum, and 11.2% in average with 16 threads. Next, we go to the detailed examination of these results in the following.

C. Detailed Examination

First, the performance with Deque and Prioqueue are improved by the decrease of Backoff cycles. To consider this reason, we examined these two programs and found that these two programs have transactions which cause aborts repeatedly with LogTM-SE (B). However, with the proposed algorithm, when a transaction is aborted, the opponent transaction ID and the conflicted address are stored. Hence, repetitive aborts can be avoided because a transaction which has already accessed to shared variables is selected as the preferential transaction before other transactions access to the same variables. Besides, the number of Wait cycles with the proposed HTM (P) are smaller than Backoff cycles with LogTM-SE (B). Consequently, it is confirmed that wasteful waiting is avoided because transactions can wait the minimum required time.

On the other hand, the performance with Btree declines as the number of thread increases. We examined this program, and found that Btree has two noteworthy transactions. The one (Tx.A) includes both load and store instructions, and the other (Tx.B) includes only load instructions. Therefore, with the proposed HTM (P), when Tx.A which has a lower Priority is aborted by Tx.B which has a higher Priority, the
core whose thread executes Tx.A remembers the conflict with Tx.B. Afterwards, if some Tx.B’s are executed in parallel, these transactions which do not conflict essentially are executed sequentially because the entry for Tx.B is stored in Conflict-Addr until the program terminates. Here, assume that Tx.B has a higher Priority than Tx.A, and Tx.B tries to load from a shared variable which has already overwritten by Tx.A. In this case, Tx.A will be aborted. Such situations were caused frequently in this program. Hence, Wait cycles and Aborting cycles increase, and the performance of Btree declines. Therefore, one of our future work is considering an algorithm which can take account of the contexts of transactions. Specifically, if a transaction has only load instructions, the entry for this transaction should be cleared from Conflict-Addr.

The performance with Cholesky and Raytrace (8thr) is not improved. This is because Non-trans occupies most of the total cycles in these programs. Therefore, with the proposed HTM (P), the ratio of the performance improvement is smaller than the other programs. In contrast, with Contention, Stall cycles are decreased with the proposed HTM (P), but Wait cycles are increased almost the same as the amount of decreased cycles. With LogTM-SE (B), when a conflict is detected, the transaction is stalled. On the other hand, with the proposed HTM (P), the transaction waits before conflicts are caused. Therefore, Stall cycles are decreased. Incidentally, a transaction in Contention has great amount of instructions in the transaction. Hence, when transactions conflict, one of the conflicted transaction which has lower Priority will wait a long time. Consequently, Wait cycles are increased with the proposed HTM (P).

D. Additional Hardware Cost

In the proposed HTM, each of L-Counter and S-Counter should have an enough bit width for counting as many as the maximum number of load/store instructions in transactions. Then, we have measured how many instructions are executed in the benchmark programs. As a result, if L-Counter can count to 470 and S-Counter can count to 944, these counter do not overflow with all programs. Hence, 10-bit width is enough for each of L-Counter and S-Counter. Moreover, for a 16-core processor which can execute 16 threads, the size of Pr-CoreID is 4-bit and Po-CoreID is 16-bit. Next, we have measured how many transactions are included in each program for investigating the size of Past-Access. As a result, 19 transactions are included in maximum. Besides, Past-Access needs 25-bit width because transaction ID field should have 5-bit width and each of past load/store instructions field should have 10-bit width. Therefore, Past-Access can be implemented with 25-bit width and 19 rows. On the other hand, the depth of Conflict-Addr should be as many as the number of conflicted addresses. As the result of the measurement, if Conflict-Addr has 36 entries, this table does not overflow with all programs. Furthermore, the width of Conflict-Addr is 37-bit, and Conflict-Addr can be implemented with 37-bit width and 36 rows. Hence, the total additional hardware cost is only about 3.7 Kbytes, and it is confirmed that the additional hardware cost is very small.

VI. Conclusion

In this paper, we propose a new criterion called Priority for avoiding wasteful waiting. One of conflicted transactions is selected as the preferential transaction according to Priority instead of the initiated time of transactions. As a result, wasteful waiting time becomes shorter than using Exponential Backoff algorithm. We have evaluated the proposed HTM by comparing with LogTM-SE, through experiments with GEMS microbench and SPLASH-2 benchmark suites. The evaluation results show that the proposed HTM decreases the total execution cycles 59.9% in maximum, and 11.2% in average with 16 threads. However, with the proposed HTM, transactions may wait even if they can run in parallel. Incidentally, if a transaction tries to load from a shared variable and another transaction has already stored to the same variable, the transaction which has already stored and has lower Priority than the other is aborted. Such situation was caused frequently in the proposed HTM. Therefore, one of our future works is considering an algorithm which can take account of the contexts of transactions.

REFERENCES