Yet Another Waiting Mechanism based on Conflict Prediction for Hardware Transactional Memory

Keisuke MASHITA*, Sho MIYAKE*, Ryohei YAMADA* and Tomoaki TSUMURA*
*Nagoya Institute of Technology
Gokiso, Showa, Nagoya, Japan
Email: camp@matlab.nitech.ac.jp

***Abstract*—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 be declined. To address this problem, in this paper, we propose a conflict prediction to avoid conflicts in advance based on historical data of conflicts. The result of the experiment shows that the execution time of HTM is reduced 63.5% in maximum, and 19.6% in average with 16 threads.**

I. Introduction

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. On *Hardware Transactional Memories* (HTMs), which are the hardware implementations of TM, the mechanisms for version management and conflict detection are implemented in hardware.

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, in this paper, 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.

II. A Conflict Prediction

In this section, we describe overviews of HTM, and point out a problem of general HTMs. After that, we propose a conflict prediction to address the problem.

A. Conflict Detection and Resolution of the General HTM

TM must resolve conflicts, because it executes transactions speculatively. Here, Fig.1 shows an example where conflicts are detected with LogTM [2] which is one of the most general HTM systems. In this example, *thr.1* executes *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 because *thr.1* has already accessed to address A (*t2*). In this case, as *thr.2* receives a *Nack* from *thr.1*, *thr.2* stalls *Tx.Y*, waiting for *thr.1* to commit (*t3*). 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 address C. In this case, *thr.3* receives a *Nack* from *thr.2*, and *thr.3* stalls *Tx.Z*, waiting for *thr.2* to commit (*t4*). 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.

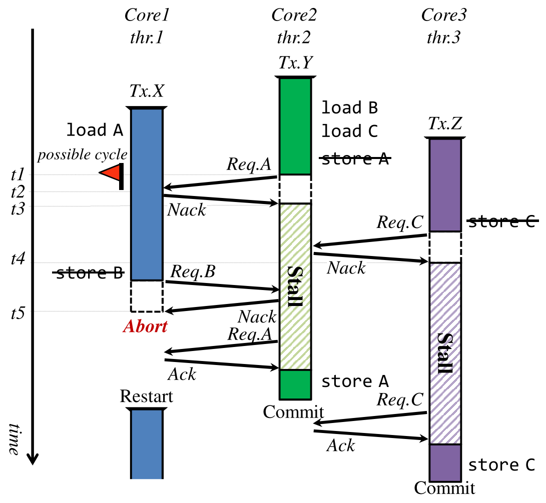

Fig. 1. Conflict resolution on a general HTM.

B. Avoiding Conflicts by a Conflict Prediction

As mentioned in section II-A, threads can resolve conflict. However, transactions which have conflicted once each other

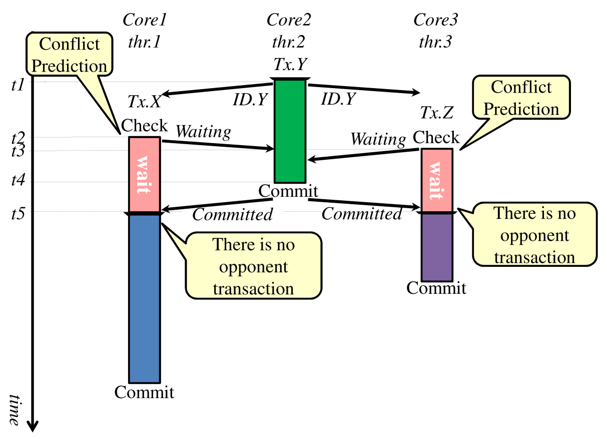

Fig. 2. How to predict conflicts.

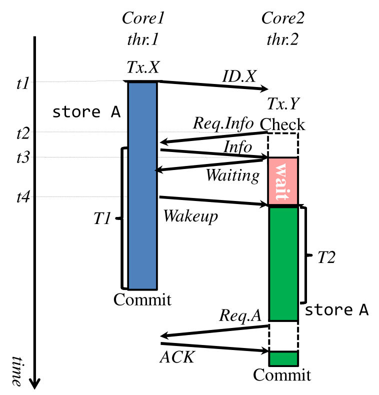

Fig. 3. A conflict prediction by using opponent transaction IDs and temporal data of a transaction.

tend to conflict repeatedly because their execution paths do not change and the threads access to the same shared variables when they are executed again. 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. Before starting a transaction, a thread predicts whether a conflict will be caused or not during the transaction, by referring to 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 section II-A, stall is also a 'waiting' mechanism for conflict resolution. In contrast to stall, a thread does not cause a new conflict with this conflict prediction, because the thread waits without accessing any addresses.

Fig.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* has already conflicted with *Tx.X* and *Tx.Z*, and each thread remembers it. 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*, *thr.1* predicts that *thr.1* will conflict with *thr.2*. Therefore, *thr.1* waits for *thr.2* to commit *Tx.Y*. 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 *thr.1* and *thr.3* (*t4*). When *thr.1* and *thr.3* receive this *Committed* message, they start to execute transactions (*t5*).

C. Avoiding Conflicts with Minimum Waiting Time

To avoid conflicts with minimum waiting time, we adopted additional two temporal data about 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.

Here, Fig.3 shows an example where a thread does not wastefully wait 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 for inquiring the remaining time until the commit of the opponent transaction to each opponent thread (*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 execute *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, a *Waiting* and a *Wakeup*, on the coherence protocol for accurate conflict prediction.

III. Implementation

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

A. Additional Hardware Units

To implement the conflict prediction described in section II-C, we have installed five 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 because of cache misses or stalls. Hence, we use the number of memory accesses as an approximation of execution time. For a 16-core processor,

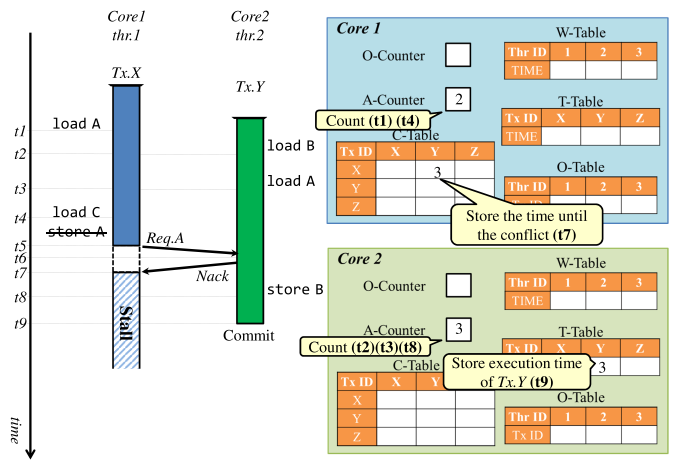
Fig. 4. How to remember historical data of conflicts.

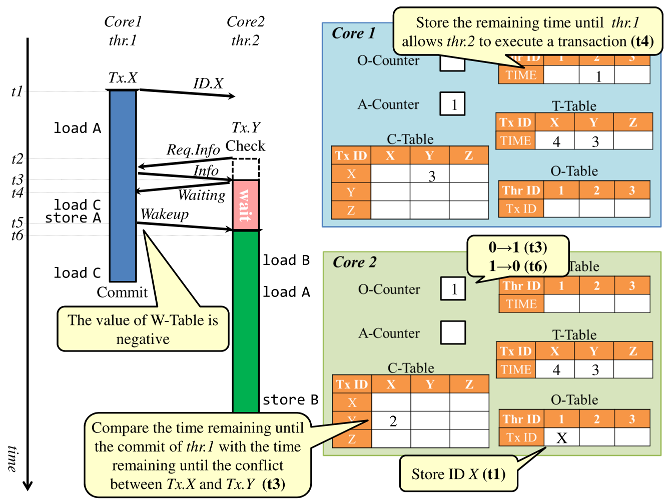
Fig. 5. How to predict and avoid conflicts.

the total size of the additional hardware units is estimated at 11.8 Kbytes.

B. How to Remember Historical Data of Conflicts

In this section, we will describe how threads remember historical data for predicting conflicts. Here, Fig.4 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*).

C. How to Predict and Avoid Conflicts

In this section, we will describe how a thread predicts a future conflict and avoid it. Here, Fig.5 shows an example where a thread predicts that a conflict will be caused. In Fig.5, assume that each of *thr.1* and *thr.2* remembers its execution time and the time until the conflict between *Tx.X* and *Tx.Y* for the conflict prediction. 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 *thr.2* should 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*. Receiving '*X*' as the transaction ID of *thr.1* and the calculated answer '3' from *thr.1*, *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 a transaction. 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*).

IV. Performance Evaluation

In this section, we describe the evaluation results.

A. Evaluation Environment

We used a full-system execution-driven functional simulator *Wind River Simics*[3] in conjunction with customized memory

TABLE I
SIMULATION PARAMETERS

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

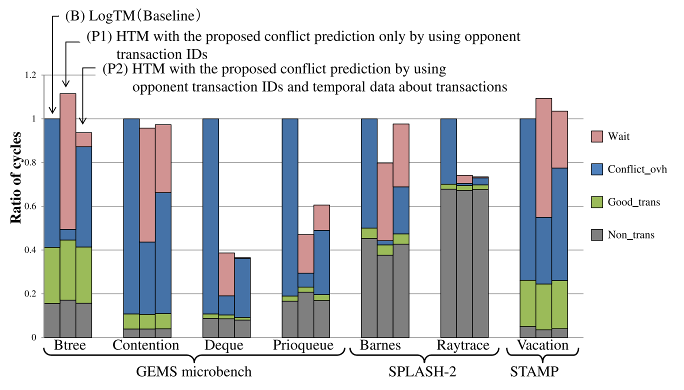

Fig. 6. Execution cycles ratio.

simulators built on *Wisconsin GEMS* [4] for evaluation. The detailed configuration of the simulated processor is shown in TABLE I. We have evaluated the execution cycles of 7 workloads from GEMS microbench, SPLASH-2 benchmark suite [5], and STAMP benchmark suite [6] with 16 threads.

B. Evaluation Results

The evaluation results with following three HTMs are shown in Fig.6.

- (B) LogTM (baseline)
- (P1) HTM with the proposed conflict prediction only by using opponent transaction IDs
- (P2) HTM with the proposed conflict prediction by using opponent transaction IDs and temporal data

Fig.6 shows the execution cycles of each HTM. Each bar is normalized to the total execution cycles of the baseline (B). For simulating multi-threaded execution on a full-system simulator, the performance variability [7] 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.

The legend shows the breakdown items of the total cycles. They represent the execution cycles out of transactions (*Non_trans*), the execution cycles in the transactions which are committed (*Good_trans*), the execution cycles of conflict resolution (*Conflict_ovh*), the waiting cycles before starting transactions by the proposed conflict prediction (*Wait*).

As a result of the evaluation, HTM with the conflict prediction (P2) reduces the execution cycles 63.5% in maximum, and 19.6% in average with 16 threads. Next, we go to the detailed examination of these results in the following.

C. Detailed Examination

As shown in Fig.6, the performance of many programs is improved with (P1) and (P2). Especially, *Wait* with (P2) is smaller than (P1). However, performance of some programs with (P1) is better than (P2). To investigate the origin of this result, we examined these programs in detail, and it is found that the number of memory accesses tends to change in these programs, because the execution path varies due to branch instructions. Hence, with (P2), conflict predictions sometimes fail and *Conflict_ovh* of some programs are larger than with (P1). However, Contention and Raytrace have transactions whose number of memory accesses does not change. Therefore, the conflict prediction in (P2) is more accurate than (P1) because it uses temporal data of transactions, and the performance of (P2) outperforms that of (P1).

V. CONCLUSIONS

In this paper, we propose a conflict prediction for HTM to avoid conflicts in advance. 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 HTM with 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 63.5% in maximum, and 19.6% in 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 works is to improve the prediction accuracy by considering variation of execution paths in transactions.

ACKNOWLEDGMENT

This research was partially supported by the grant from the 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 *et al.*, “LogTM: Log-based Transactional Memory,” in *Proc. 12th Int’l Symp. on High-Performance Computer Architecture (HPCA’06)*, Feb. 2006, pp. 254–265.
[3] P. S. Magnusson *et al.*, “Simics: A Full System Simulation Platform,” *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[4] 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.
[5] S. C. Woo *et al.*, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proc. 22nd Int’l. Symp. on Computer Architecture (ISCA’95)*, 1995, pp. 24–36.
[6] C. C. Minh *et al.*, “STAMP: Stanford Transactional Applications for Multi-Processing,” in *Proc. IEEE Int’l Symp. on Workload Characterization (IISWC’08)*, Sep. 2008.
[7] 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.