# Initial Study of a Phase-Aware Scheduling for Hardware Transactional Memory

Tomoki TAJIMI[*], Anju HIROTA[*], Ryota SHIOYA[†], Masahiro GOSHIMA[‡] and Tomoaki TSUMURA[*]

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

[†]Nagoya University
Furo, Chikusa, Nagoya, Japan
Email: shioya@nuee.nagoya-u.ac.jp

[‡]National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda, Tokyo, Japan
Email: goshima@nii.ac.jp

***Abstract*—Transactional memory is a promising paradigm for shared memory parallel programming model. Effective transaction scheduling is very important for transactional memory systems, and a substantial body of work has been conducted. We have proposed a transaction scheduling which considers execution path variation in transactions, and it goes well with many types of programs, but some programs still can not gain performance. In this paper, we focus on such programs and investigate the reason for low performance by analyzing conflict prediction accuracy and typical conflict patterns. Then, we propose a novel phase-aware transaction scheduling for resolving one of the harmful conflict patterns. Evaluation result shows that the phase-aware scheduling can largely improve the performance of one of the benchmark programs, and indicates its potential superiority.**

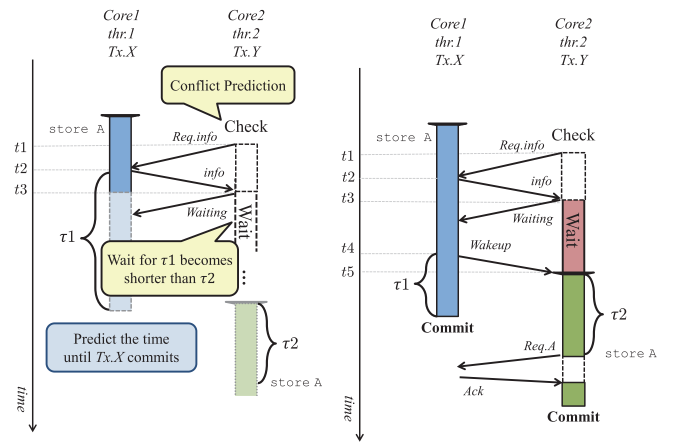
## I. Introduction

Parallel programming on shared memory model is now very important while multi-core processors have become popular. In shared memory parallel programming, *lock* has been commonly used for arbitrating access conflicts on shared resources, especially the shared memory space. However, lock has some drawbacks; lock can cause deadlocks, the overhead for locking/unlocking may degrade the total performance, it is hard for programmers to find appropriate locking granularity, and so on.

*Transactional memory* (TM) is a promising paradigm for replacing and/or complementing traditional lock-based conflict management. On TM-managed systems, multiple transactions running on separate threads can speculatively access the shared memory in parallel, as long as any access conflict is not detected. Hardware transactional memories (HTMs) are implemented on recent commodity processors, such as Intel Haswell[1], IBM Power8[2], and so on.

Effective transaction scheduling is very important for HTM performance, and many scheduling algorithms have been proposed and studied. However, any previous algorithms do not consider execution path variation in transactions. In a previous work[3], we found that the execution path variation inside transactions is very important for transaction scheduling, and proposed a novel conflict prediction algorithm. It works well with the most of the benchmark programs, but we found that some programs still can not gain performance.

In this paper, we analyze its reason in detail, and we propose



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

Fig. 1. Conflict prediction based on execution time.

an optimistic conflict detection algorithm and a transaction scheduling with the algorithm. The contribution of our work can be summarized as follows:

- We estimate the accuracy of our previous conflict prediction algorithm, and discuss insufficiencies of the algorithm.
- We disclose the access conflict patterns which can hardly be resolved by previous transaction scheduling algorithms including our conflict prediction algorithm, and analyze the transaction structure which will cause such conflicts.
- We propose a novel *phase-aware* transaction scheduling based on an optimistic conflict detection, and show its potential by evaluating it with a benchmark program.

## II. Conflict Prediction Algorithm

### *A. Outline*

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. Hence, we have proposed a transaction scheduling algorithm[4] for reducing conflicts by a conflict prediction. The algorithm is as follows.

Before a thread starts a transaction, the thread predicts whether the transaction will cause a conflict or not (Fig. 1, t1). 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 other 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 (t3).

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.

*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.

### B. Execution Path Variation

Although threads can avoid many conflicts by the conflict prediction as mentioned in Section II-A, 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 a transaction can not be used for conflict prediction, because the execution path of a transaction needs to be predicted *before* the transaction starts. Hence, we have proposed a conflict prediction mechanism[3] which exploits the idea of *global branch prediction*[5].

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 call 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.

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 |

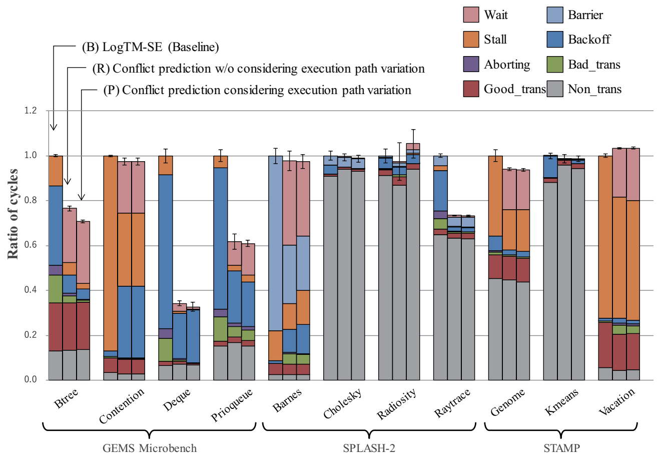


Fig. 2. The sum of the total execution cycles ratio.

### C. Evaluation Results

To evaluate the transaction scheduling based on conflict prediction, we used a full-system execution-driven functional simulator *Wind River Simics*[6] in conjunction with customized memory simulators built on *Wisconsin GEMS* [7]. The detailed configuration of the simulated processor is shown in Table I. The topology and the link latency of interconnect network are defined as same as LogTM-SE[8]. We have evaluated the execution cycles of 11 workloads from GEMS microbench, SPLASH-2 benchmark suite [9], and STAMP benchmark suite [10] 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.

The evaluation results with following three HTM configurations are shown in Fig. 2.

(**B**) LogTM-SE (baseline)

(**R**) Reference model; predicts conflicts by using the past shortest execution time of each transaction, without considering execution path variation.

(**P**) Previous method; predicts conflicts by using the past execution time of each transaction considering execution path variation.

Fig. 2 shows the total sum of execution cycles of all 16 threads and its breakdown. Each bar in both figures is normalized to LogTM-SE the baseline (**B**). For simulating multi-threaded execution on a full-system simulator, the performance variability must be considered[11]. Hence, we tried 10 times on each benchmark, and measured 95% confidence interval, which is illustrated as error bar in Fig. 2.

The legend in Fig. 2 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 figure, (P) achieves 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. However, the performance is not improved enough with some programs. Especially with Contention and Vacation, *Stall* still occupies a large part of the total execution time. In the following, we investigate its reason and introduce an idea for resolving it.

## III. Analysis of Harmful Conflict Patterns

In this section, we focus on two benchmark programs: Vacation and Contention, and investigate the reason why their performance is not improved by our previous conflict prediction.

### A. Vacation

Fig. 3 shows the accuracy of conflict prediction on one of the transactions in Vacation. The transaction is executed more times than any other transactions in Vacation, and it can largely impact the total performance of Vacation. The figure shows the error distribution of the transaction. The horizontal axis of the graph shows the error values, and the vertical axis shows the frequency of that the error value is measured. The error value is calculated by subtracting actual execution time of the transaction from predicted execution time of the transaction.

When the error value is positive, as plotted in the right half area of the Fig. 3, predicted value of execution time is larger than the actual value, and the transaction will wait futilely long. On the other hand, when the error value is negative, as plotted in the left half, waiting time should be not long enough and the transaction will not avoid a conflict. Both of large positive and negative errors are frequently observed as shown in this figure.

Fig. 4 shows the simplified code of the transaction in Vacation. It includes a switch case statement (line 4..14), and the key value for the statement is randomly selected (line 3). Hence, the execution path in the transaction can not be accurately predicted from global load/store history patterns. This is the reason of the low performance of Vacation. This type of transaction must be difficult to be adequately scheduled.

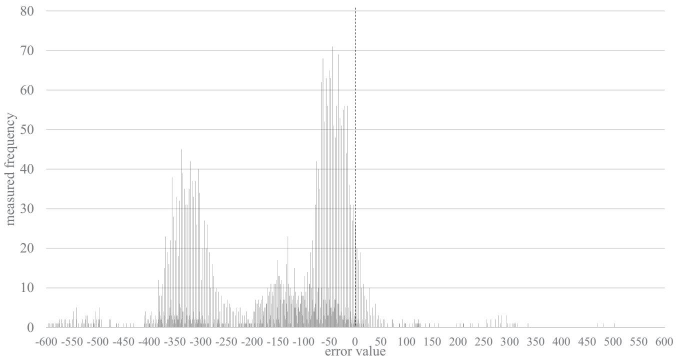


Fig. 3. Error distribution of conflict prediction with Vacation.

```
1  BEGIN_TRANSACTION();
2  for( n = 0; n < numQuery; n++ ){
3    long t = types[n];     // t is a random value.
4    switch( t ){           // execution path is randomly selected.
5    case RESERVATION_CAR;
6      :
7    break;
8    case RESERVATION_FLIGHT;
9      :
10   break;
11   case RESERVATION_ROOM;
12     :
13   break;
14   }
15 }
16
17 if( isFound ){
18   :
19 }
20                         // execution path is randomly selected.
21 if( Id[RESERVATION_CAR] > 0 ){
22   :
23 }
24 if( Id[RESERVATION_FLIGHT] > 0 ){
25   :
26 }
27 if( Id[RESERVATION_ROOM] > 0 ){
28   :
29 }
30 COMMIT_TRANSACTION();
```

Fig. 4. Simplified code of a transaction in Vacation.

### B. Contention

Contention is another program whose performance is not improved enough by our previous conflict prediction. We measured the error distribution of the sole transaction in Contention, and found that the error value was zero on almost all prediction tries. However, the performance is not improved as shown in Fig. 2.

Now Fig. 5 shows the simplified code of the transaction in

```
int A[1];
int B[1024];

BEGIN_TRANSACTION();
// phase 1
for( i = 0; i < 10; i++ ){
  if( access_type[i] == READ )
    var = A[index[i]];
  else
    A[index[i]] = 0;
}

// phase 2
for( i = 10; i < 90; i++ ){
  if( access_type[i] == READ )
    var = B[index[i]];
  else
    B[index[i]] = 0;
}
for( i = 90; i < 100; i++ ){
  if( access_type[i] == READ )
    var = B[index[i]];
  else
    B[index[i]] = 0;
}
COMMIT_TRANSACTION();
```

Fig. 5. Simplified code of the transaction in Contention.

Contention. Two array variables `A[]` and `B[]` are accessed in the transaction. The transaction consists of two phases, namely $phase1$ and $phase2$, and they have different memory access patterns. Specifically, array `A[]` is accessed in $phase1$, but is never accessed in the succeeded phase $phase2$, as shown in Fig. 5.

We investigated the behavior of this transaction, and found that all conflicts in this transaction occurred on `A[]` and no confilct on `B[]`, and 95.7% of access requests on `A[]` was denied.

When threads execute this transaction in parallel, after a thread accesses `A[]` in $phase1$, other threads can not execute $phase1$ until the preceding thread commits the transaction because of a conflict on `A[]`, although the preceding thread never accesses to `A[]` again in the transaction. This will completely serialize the execution of this transaction on multiple threads, and leads to the low performance of Contention.

In such a case, granting access requests on `A[]` from other threads will not cause any inconsistency on `A[]` if certain requirements are met. Hence, we propose a novel *phase-aware* transaction scheduling based on an *optimistic* conflict detection algorithm. We introduce it in the next section.

## IV. Phase-Aware Transaction Scheduling

As described in Section III-B, some programs including Contention have multiple phases in its transaction, and memory access pattern may be different between such phases. Considering this difference can improve the performance of HTMs. In this section, we propose a *phase-aware* transaction scheduling based on an optimistic conflict detection.

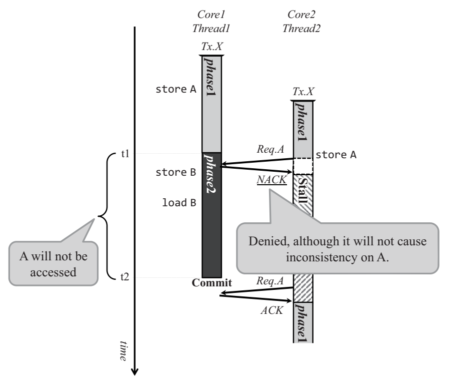


Fig. 6. A frequent conflict pattern in Contention.

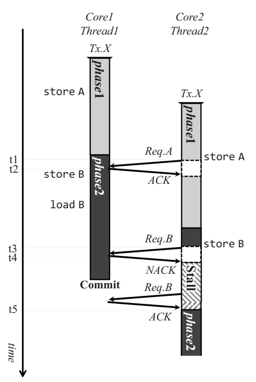


Fig. 7. Phase-aware transaction scheduling by speculatively granting an access request.

### A. Execution Model

Now assume that transaction $Tx.0$ consists of two phases, $phase1$ and $phase2$. The transaction accesses address `A` in $phase1$, but will never access the address after finishing $phase1$.

Fig. 6 illustrates a frequent conflict pattern in such programs. When two threads $Thr.1$ and $Thr.2$ running in parallel executes $Tx.0$, their access on `A` will cause a conflict. However, $Thr.1$ will never access `A` again after finishing $phase1$ (t1..t2), and granting the access request on `A` from $Thr.2$ (at $t1$) will not cause any inconsistency. As shown in Fig. 7, if the access request from $Thr.2$ can be speculatively granted (t2), the stall cycles of $Thr.2$ will be reduced and the total performance will increase.

To safely grant the access request from other threads, each thread should remember when, or in which phase, each shared variable is lastly accessed in the transaction. We at the moment assume that phases will be indicated by programmers using nested transactions.

### B. Additional Required Modifications

We must consider additional modification on transaction management, because speculatively granting access requests as shown in Fig. 7 is somewhat *optimistic* on conflict detection. When $Thr.1$ speculatively grants an access request on shared variable `A` from $Thr.2$, it is supposed that $Thr.1$ already finished modifying the variable `A` and the modification will be committed. However, for some reasons such as execution path variation in the transaction, $Thr.1$ may access the shared variable again, and the presupposition may be overthrown. In such a case, $Thr.2$ whose access request was speculatively granted should abort its transaction for consistency on the shared variable.

Another case is that a thread aborts its transaction after speculatively granting access requests from others. In the example shown in Fig. 8, $Thr.1$ speculatively grants an access request on variable `A` from $Thr.2$, and then it faces a deadlock on another shared variable with another thread (t1). In this

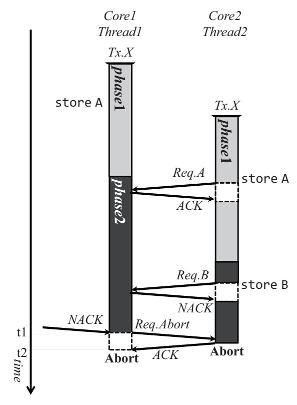


Fig. 8. Propagating abort.

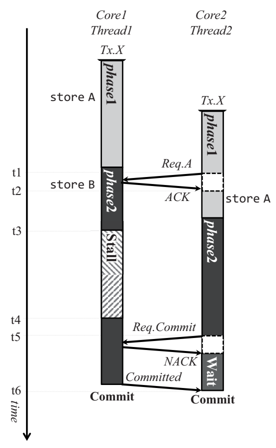


Fig. 9. Required wait and related message exchange.

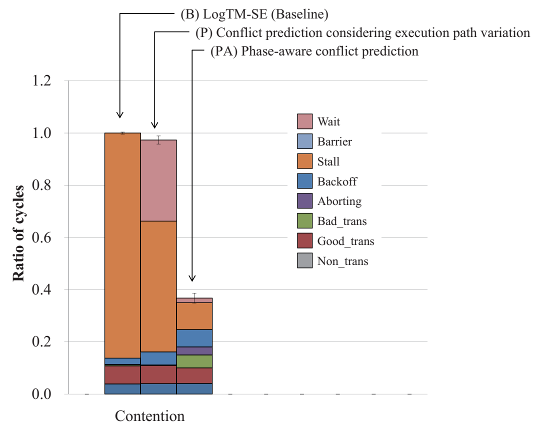


Fig. 10. The sum of the total execution cycles ratio of Contention.

case, $Thr.1$ must abort its transaction, and the presupposition that the modification on A by $Thr.1$ will be committed is overthrown. Hence, $Thr.2$ also must abort its transaction.

To ensure this Domino effect, as shown in the example of Fig. 8, $Thr.1$ should remember the ID of $Thr.2$ when it speculatively grants the access request from $Thr.2$, and send an *abort request* message to $Thr.2$ as required (t1). In addition, $Thr.2$ can not commit its transaction until $Thr.1$ commits, even if $Thr.2$ reaches to the end of transaction earlier than $Thr.1$, because $Thr.1$ may abort its transaction, as shown in Fig. 9. Hence, before committing, $Thr.2$ must ask $Thr.1$ whether $Thr.1$ already committed the transaction (t5), and wait for $Thr.1$ committing the transaction (t6) if not.

## V. Experimental Evaluation

In this section, we conduct experiments to evaluate the proposed *phase-aware* transaction scheduling. The evaluation setup such as platform configuration is as same as shown in Section II-C. The evaluation results with following three HTM configurations are shown in Fig. 10.

(**B**) LogTM-SE (baseline)

(**P**) Previous method; predicts conflicts by using the past execution time of each transaction considering execution path variation.

(**PA**) Proposed transaction scheduling with phase-aware conflict prediction.

Breakdown items except for *Wait* are as same as Fig. 2. *Wait* in Fig. 10 represents the waiting cycles before committing transactions shown in Fig. 9.

The result shows that the proposed transaction scheduling (PA) can reduce total execution cycles of Contention 63.2% compared with LogTM-SE (B) the baseline, and 60.0% compared with our previous work[3] (P). Especially *Stall* is largely reduced, and it mainly contributes the total performance gain. This indicates that transaction execution is well overlapped and futile stalling time is reduced, by speculatively granting access requests. On the other hand, *Aborting* and *Bad_trans* become larger. This will be caused by Domino effect on aborts explained in Fig. 8.

Contention has been one of the most difficult programs to gain performance with HTM, and most of the previous work on HTM scheduling has not able to improve its performance. This evaluation result shows that scheduling transactions with considering phases in transactions can improve the performance of such type of programs.

## VI. Related Work

So far, various techniques for improving the performance of HTMs have been proposed. Especially, many thread scheduling techniques for reducing conflicts by controlling transactional sequences have also been proposed [12], [13], [14].

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 a benchmark program 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[3], the execution cycles are 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 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 previous work[3].

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 [10] 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. Hence, the baseline performance can be unreasonably underestimated.

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[3], [4], 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 of our method.

All these methods 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 transaction scheduling[3] can improve the performance of many practical programs because it can avoid causing conflicts even if the execution path in a transaction varies.

However, there still are some types of programs whose performance can not be improved even by our previous transaction scheduling[3] which is superior to other previous work as mentioned above. In response to this, in this paper, we focused on such programs, and analyze the harmful conflict patterns which will cause the low performance. Especially, for transactions who have multiple phases with different memory access patterns, we proposed a novel *phase-aware* scheduling.

## VII. Conclusion

In this paper, we estimated the accuracy of our previous conflict prediction algorithm[3], and investigated some programs which can not gain performance with the conflict prediction algorithm. Through the investigation, we found two typical access conflict patterns which can hardly be resolved by many of previous transaction scheduling algorithms.

Then we focused on one of the conflict patterns of the transaction in Contention benchmark program. The transaction has two phases in it, and the phases have different memory access pattern. We proposed a *phase-aware* transaction scheduling which speculatively allows some memory accesses which are denied on general HTM systems. Evaluating the proposal with Contention disclosed that the *phase-aware* transaction scheduling goes very well with a type of program.

Our future work includes a detailed evaluation of the *phase-aware* scheduling with much more benchmark programs, and finding a methodology for dynamically detecting phases in transactions. Designing and implementing a portable protocol for applying this scheduling are also left for our future work.

## Acknowledgment

This research was partially supported by JSPS KAKENHI Grant Numbers JP17H01711, JP17H01764, JP17K19971 and the grant from Tatematsu Foundation.

## References

[1] Intel Corporation, *Intel Architecture Instruction Set Extensions Programming Reference, Chapter 8: Transactional Synchronization Extensions.*, 2012.

[2] International Business Machines Corporation, "Power ISA® Version 2.07," https://www.power.org/documentation/power-isa-version-2-07/, 2013.

[3] A. Hirota, K. Mashita, and T. Tsumura, "A Concurrency Control in Hardware Transactional Memory Considering Execution Path Variation," in *Proc. 4th Int'l Symp. on Computing and Networking (CANDAR'16)*, Nov. 2016, pp. 77–83.

[4] K. Mashita, M. Tabuchi, R. Yamada, and T. Tsumura, "A Waiting Mechanism with Conflict Prediction on Hardware Transactinal Memory," *IEICE Trans. on Information and Systems*, vol. E99-D, no. 12, pp. 2860–2870, Dec. 2016.

[5] 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.

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

[7] 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.

[8] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Proc. 13th Annual Int'l Symp. on High Performance Computer Architecture (HPCA-13)*, Feb. 2007, pp. 261–272.

[9] 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.

[10] 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.

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

[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.