This is the accepted manuscript of a paper published in
Proc. 3rd Int'l Symp. on Computing and Networking (CANDAR'15), pp.147-153

Eliminating Cascading Stall on Hardware Transactional Memory

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

***Abstract*—Multi-core processors are equipped in almost every computer systems from smartphones to high-end server machines, and shared memory programming becomes increasingly important for programmers to utilize the multi-core systems. Lock-based thread synchronization techniques have been commonly used in parallel programming on multi-core processors. However, lock can cause deadlocks and this leads to poor scalability. To make up for the shortcomings of lock, transactional memory (TM) is proposed and widely studied. On TMs, transactions are executed speculatively while any conflicts do not occur on shared variables. However, wasteful re-executions and waits can cause low concurrency and drastic performance degradation. In this paper, we propose a method for resolving Cascading Stall which is one of the main factors of low concurrency on TM. The result of the experiment shows that the method can reduce execution time 56.5% in maximum and 11.1% in average with 16 threads.**

I. Introduction

Nowadays, multi-core processors are equipped in almost every computer systems, from smartphones that we are carrying around, to server-class machines in data centers which provide us cloud services. On multi-core processors, multiple threads can run in parallel for speed-up. Therefore, shared memory programming becomes increasingly important for programmers to utilize the multi-core systems. When multiple threads run in parallel on shared memory systems, lock-based thread synchronization techniques have been commonly used for mutual exclusion. However, lock can cause deadlocks, and this leads to poor scalability. Then, *Transactional Memory* (TM) [1] has been proposed as a lock-free synchronization mechanism. With TM, programmers simply declare code blocks as *transactions* which include load/store instructions on shared memory values. On TM, as long as any conflicts do not occur on shared variables, transactions are executed speculatively. However, the interim execution results may be discarded due to speculative execution. Thus, when a transaction modifies a value on shared memory, TM should save both new and old values (*version management*). Besides, TM should keep track of memory accesses, and check whether or not each requested datum has been already accessed by another transaction (*conflict detection*). *Hardware Transactional Memories* (HTMs), which are the hardware implementations of TM, achieve both version management and conflict detection on hardware for small delay overhead. As mentioned above, transactions are executed speculatively for speed-up on TM, thus the execution concurrency with TM is higher than with lock. However, the performance may decline drastically when the number of concurrently executed transactions decreases due to wasteful re-executions and waits. In such a situation, some of wasteful waits can be caused cascadingly, and the problem is called *Cascading Stall*[2]. In this paper, we propose a method to detect Cascading Stall by observing the dependency among the transactions, and resolve Cascading Stall at appropriate time to prevent concurrency from becoming low.

II. Research Background

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 has to satisfy *atomicity* and *serializability*. In order to ensure atomicity and serializability, TM should keep track of memory accesses, and check whether or not each requested datum has been accessed yet by another transaction. When a transaction tries to access the same memory address which has already been accessed by another transaction, a conflict is detected between the transactions. In case that a conflict is detected, TM selects a transaction among the conflicted transactions and stalls the selected transaction. When the conflicted transactions are aborted to avoid deadlocks, the aborted transactions will be re-executed later. On the other hand, if any conflicts do not occur through a transaction, TM commits the transaction. As far as any conflicts do not occur through a transaction, the transactions can run speculatively under TM in parallel. Thus, TM generally makes execution concurrency higher than lock-basesd systems. The mechanisms for version management and conflict detection can be implemented in hardware or software. Some TM systems operate completely in software (STMs) [3]. However, STM has more overheads than HTMs.

B. Conflict Detection and Version Management

With HTMs, conflict detection and version management are implemented in hardware. Each of both policies of conflict detection and version management is classified into Eager/Lazy.

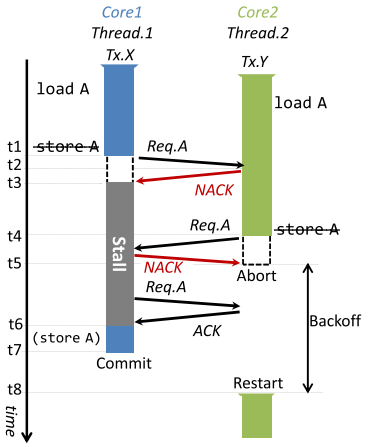


Fig. 1. Eager Conflict Detection and Resolution.

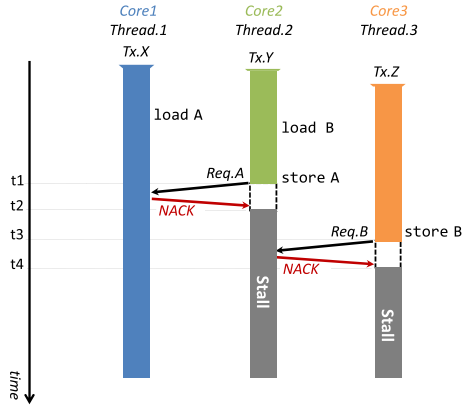


Fig. 2. *Cascading Stall*.

With Eager conflict detection policy, just when a memory access occurs in a transaction, it is checked that whether the access causes a conflict or not.

Here, we explain Eager conflict detection and the conflict resolution. Fig.1 shows an example where *Thread1* executes *Tx.X* and *Thread2* executes *Tx.Y*, and *Thread1* and *Thread2* have already issued `load A`. First, when *Thread1* tries to issue `store A` (t1), a conflict is detected (t2) because *Thread2* has already accessed to the address A. In this case, *Thread1* receives a *NACK* from *Thread2*, and stalls *Tx.X* until *Thread2* commits *Tx.Y* (t3). Afterwards, when *Thread2* tries to issue `store A` (t4), another conflict is detected because the same address has already been accessed by *Thread1*. In this case, as *Tx.Y* is younger than *Tx.X*, *Thread2* aborts its *Tx.Y* (t5). At this time, *Thread2* waits for restarting *Tx.Y* to prevent a conflict with *Thread1* from being repeated, and the period from aborting a transaction to restarting the transaction is called Backoff. The Backoff period is generally defined based on an algorithm called Exponential Backoff. This algorithm makes Backoff increase exponentially as a transaction is aborted repeatedly. While *Thread2* waits a Backoff period, *Thread1* receives an *ACK* and resumes *Tx.X* (t6) and commits *Tx.X* (t7). Then, *Thread2* re-executes *Tx.Y* (t8) without a conflict with *Thread1* in this example.

Next, we explain version management. On TM, interim results of transactions may be discarded because transactions are executed speculatively. Hence, when a transaction modifies a value on the shared memory, HTM generally needs to save both new and old values. With Eager version management, old values are stored to *log* in a thread-private virtual memory and new values are stored into cache blocks. It makes commits faster than aborts because TM commits transactions by only discarding the old values with Eager version management.

With Lazy conflict detection policy, it takes much time from causing a conflict to detecting the conflict. Consequently, more execution time of transaction will be wasted than Eager conflict detection. In this paper, we adopt *Log-based Transactional Memory (LogTM)*[4] which uses combination of Eager conflict detection and Eager version management.

III. Resolution of Cascading Stall

We refer to a problem called *Cascading Stall* on TM and propose a resolution of the problem.

A. Cascading Stall Resulted from a Chain of Conflicts

As illustrated in Fig.1, a transaction can make another transaction be stalled to avoid conflicts on HTMs. When some threads execute transactions in parallel, one transaction may conflict with an already stalled transaction. Then, the transaction will be stalled as well. On TMs, such a chain of conflicts causes cascaded stalls, and the problem is called *Cascading Stall*. *Cascading Stall* can lead to poor concurrency of the running threads. Fig.2 shows an example where *Cascading Stall* is caused. In this example, *Thread1* executes *Tx.1*, *Thread2* executes *Tx.2* and *Thread3* executes *Tx.3*. First, *Thread2* tries to issue `store A` (t1), and *Thread1* sends a *NACK* to *Thread2* because *Thread1* has already issued `load A`. Then, *Thread2* receives the *NACK* from *Thread1*, and stalls *Tx.2* (t2). Subsequently, when *Thread3* tries to issue `store B` (t3), *Thread2* sends a *NACK* to *Thread3* because *Thread2* has already issued `load B`. Then, *Thread3* also stalls *Tx.3* (t4). This results in *Cascading Stall*. In this case, *Thread3* is waiting for *Thread2* to commit and *Thread2* is waiting for *Thread1* to commit. Thus, if *Thread2* aborts *Tx.2*, *Thread3* can resume *Tx.3* without conflicting with *Thread1* in this case. Generally, as more transactions are related to the *Cascading Stall*, more and more transactions can stall wastefully.

B. Definition of Conflict Depth

In order to detect and resolve *Cascading Stall*, we need to know how much conflicts are cascaded. Therefore, we define "*depth*" of conflicts. A thread where a transaction is running and not stalled has 0-*depth* conflict. When another thread is stalled by conflicting with a 0-*depth* thread, the stalled thread is defined as 1-*depth*. Hence, if a thread is stalled by conflicting with an N-*depth* thread, the thread is defined as $(N+1)$-*depth*. Besides, in case the maximum depth among the threads whose transactions are cascadingly stalled is M, we call the situation

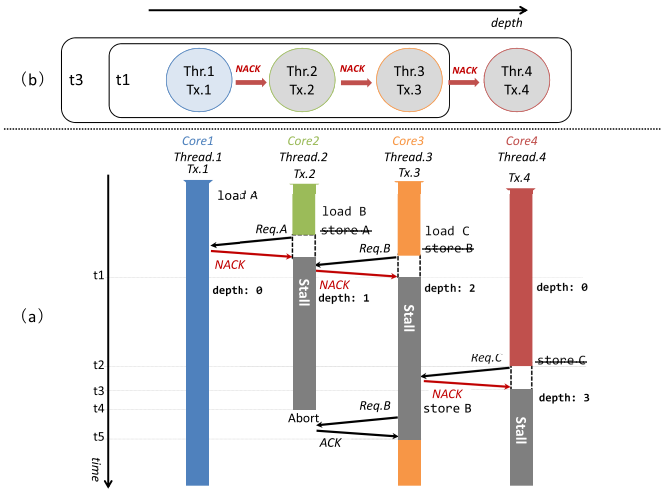


Fig. 3. Definition of *depth*.

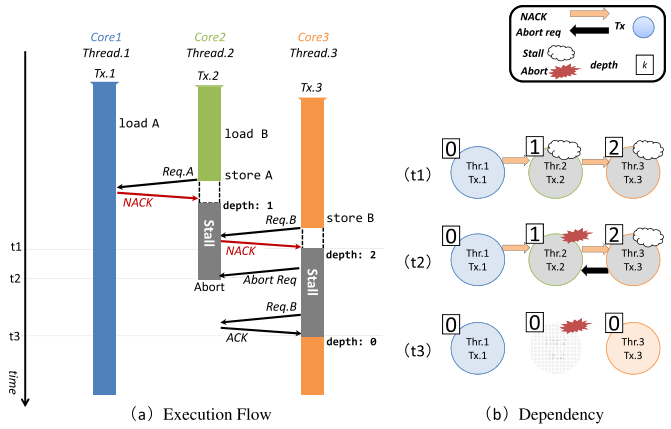


Fig. 4. Aborting a transaction whose thread is 1-depth to resolve 2-depth Cascading Stall.

"*M-depth Cascading Stall*," and we also use the expression "the depth of *Cascading Stall* is *M*." To resolve *Cascading Stall*, we define a threshold T for the depth of *Cascading Stall*, and when M the depth of *Cascading Stall* exceeds T, it is judged that the stalls are too much cascaded and the involved threads should be separated by aborting some of the transactions.

Here, we explain an example of how to detect and resolve *Cascading Stall*. Fig.3(a) shows an example where *Thread1* executes *Tx.1*, *Thread2* executes *Tx.2*, *Thread3* executes *Tx.3* and *Thread4* executes *Tx.4*. Fig.3(b) shows the dependency among the transactions at (t1) when *2-depth Cascading Stall* is caused and (t3) when *3-depth Cascading Stall* is caused on Fig.3(a). In this example, assume that the threshold T is defined as '3.' Thus, if a thread reaches *3-depth*, *Cascading Stall* is detected. First, as well as Fig.2, *Thread1* is executing *Tx.1*, and *Thread2* has *Tx.2* stalling because of the conflict with *Thread1*, and *Thread3* has *Tx.3* stalling because of the conflict with *Thread2*, and *2-depth Cascading Stall* is caused (t1). At this time, *Thread1*, *Thread2*, *Thread3* and *Thread4* is 0-, 1-, 2- and 0-depth, respectively. After that, *Thread4* tries to issue `store C` (t2), and a conflict is detected because *Thread3* has already issued `load C`. Consequently, *Thread4* receives a *NACK* from *Thread2* which is *2-depth*, and the *depth* of *Thread4* comes up to 3 (t3). In this example, if *Tx.2* or *Tx.3* is aborted, *Tx.3* or *Tx.4* can be resumed. Here, assume that *Tx.2* is selected, and *Thread2* aborts *Tx.2* to resolve Cascading Stall (t4). Then, *Thread3* can continue *Tx.3* because *Thread3* receives an *ACK* from *Thread2* (t5). In this way, Cascading Stall is resolved.

C. How to Select Victim Transactions

We refer to following two important points to resolve *Cascading Stall*.

- How much the threshold value T should be.
- Which transactions should be aborted to resolve *Cascading Stall*.

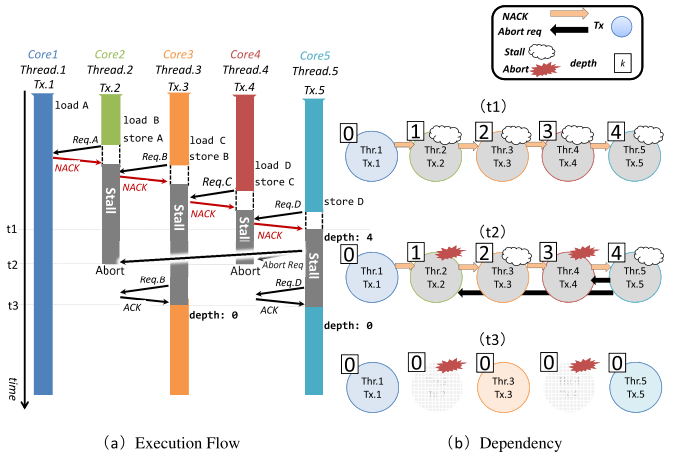
First, we will discuss the former point and consider two thresholds from different perspectives. In order to resolve *Cascading Stall* as soon as possible and not to increase the number of threads relating to *Cascading Stall*, we define the minimum *depth* of *Cascading Stall* as one of the thresholds. In short, if 1-*depth* thread *ThreadX* conflicts with another thread, the *depth* of *ThreadX* comes up to 2, and this is the smallest *Cascading Stall*. When such a *2-depth Cascading stall* is caused, it is assumed that the *Cascading Stall* should be resolved. Hence, we define '2' as one of the thresholds. On the other hand, *Cascading Stall* which has small *depth* may be resolved soon by only stalling transactions. Thus, we define the relatively large *depth* of *Cascading Stall* as the other threshold. Through some experiments, we found that conflict *depth* seldom reaches 5. Thus, we define 4 as the threshold.

Second, we will discuss the latter point. When one or more stalled transactions which are related to *Cascading Stall* are aborted, one or more conflicts are resolved and some of the transactions will be resumed. Accordingly, more transactions can run in parallel than before abort. Thus, we suggest that transactions are selected to be aborted so that the threads involved in the *Cascading Stall* are completely separated. In this paper, we call the transactions, which are selected to be aborted, "victim transactions." We propose following two ways in consideration of the two thresholds and how to select the victim transactions.

(P1) When a certain thread reaches *2-depth*, *1-depth* threads which are stalled by conflicting with the *2-depth* thread abort their transactions.

(P2) When a certain thread reaches *4-depth*, *1-depth* threads and *3-depth* threads which are involved in the *Cascading Stall* abort their transactions so that all five threads involved in the *Cascading Stall* are completely separated.

Here, we explain these two ways in detail. Fig.4(a) shows an example where *Thread1*, *Thread2* and *Thread3* execute *Tx.1*, *Tx.2* and *Tx.3*, respectively. In this example, assume that (P1) mentioned above is adopted. Fig.4(b) shows the



(a) Execution Flow (b) Dependency

Fig. 5. Aborting transactions whose threads are 1-depth and 3-depth to resolve 4-depth Cascading Stall.

dependency among the transactions at (t1), (t2) and (t3). First, *Thread3* conflicts with *Thread2* which is *1-depth*, and the *depth* of *Thread3* reaches the threshold 2 (t1). Thus, *Thread3* sends *Thread2* an *Abort Req* which is a request to let *Thread2* abort *Tx.2*. On the other hand, *Thread2* receives this request and aborts its *Tx.2* (t2). Then, *Thread2* sends an *ACK* as a reply for the *Req.B*, and *Thread3* resumes *Tx.3* after *Thread3* receives the *ACK* (t3).

Subsequently, Fig.5(a) shows an example where *Thread1*, *Thread2*, *Thread3*, *Thread4* and *Thread5* executes *Tx.1*, *Tx.2*, *Tx.3*, *Tx.4* and *Tx.5*, respectively. In this example, assume that (P2) mentioned above is adopted. Fig.5(b) shows the dependency among the transactions at (t1), (t2) and (t3). First, *Thread5* conflicts with *Thread4* which is *3-depth*, and the *depth* of *Thread5* reaches the threshold 4 (t1). Thus, *Thread5* sends an *Abort Req* to *Thread2* and *Thread4*. On the other hand, *Thread2* and *Thread4* receive this request and abort their own *Tx.2* and *Tx.4*, respectively (t2). Then, each of the *Thread2* and *Thread4* sends an *ACK* as replies for *Req.B* and *Req.D*, and each of the *Thread3* and *Thread5* resumes *Tx.3* and *Tx.5* after receiving the *ACK*. (t3).

IV. Implementation

In this section, we describe additional hardware units for implementing the resolution of *Cascading Stall* on HTM and the execution flow with the additional units.

A. Additional Hardware Units

With the proposed method on HTM, each core has to manage *depth* of the own thread so as to detect the *Cascading Stall*. Besides, each core also has to retain opponent core IDs because it is need to know the dependency among the transactions which are involved in *Cascading Stall*. To implement the proposed method on HTM, we have installed following hardware units in each core.

Depth-counter (D-counter)

This counter records *depth* of the thread which is executed in own core. This counter has 3-bit width to record the number from 0 to 4 because we defined '4' as a threshold.

Core-ID-bitmap (C-bitmap)

This bitmap records opponent core IDs. When a thread receives a *NACK* from another thread, the core records the opponent core ID on this bitmap. On HTMs, a thread may conflict with more than two other threads, and the thread should record the opponent core IDs whose threads sent *NACKs*. Therefore, when the total number of cores is n, this bitmap should have $(n-1)$bit width.

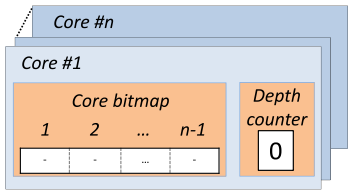


Fig. 6. Additional hardware units.

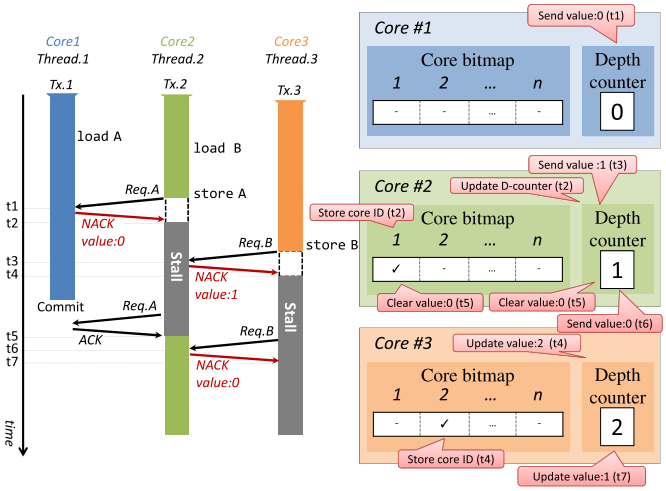


Fig. 7. Execution flow and additional hardware in the cases 1) and 2).

B. Management of the Depth

We describe how to manage *depth* of threads using the additional hardware. When *depth* of each thread changes, recorded values on the additional hardware need to be updated. Situations where *depth* of each thread changes are divided into following three cases;

1) When a thread receives a *NACK*.
2) When a thread which is stalling its own transaction receives an *ACK*.
3) When a transaction is aborted.

We describe how D-counter and C-bitmap are updated in the cases 1) and 2), using Fig.7. Fig.7 shows an example where *Thread1* executes *Tx.1*, *Thread2* executes *Tx.2* and *Thread3* executes *Tx.3*. First, each value of D-counter and C-bitmap is 0, and *Thread1* has already issued `load A`, and *Thread2* has already issued `load B`. When *Thread2* tries to issue `store A`, a conflict is detected because *Thread1* has already accessed

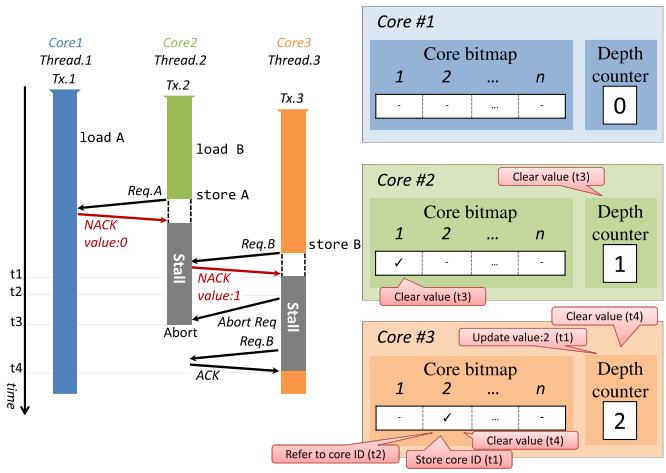


Fig. 8. Execution flow and additional hardware in the case 3).

to address A. Thus, *Thread1* replies a *NACK* with the value '0' which is stored in D-counter (t1). Then, *Thread2* receives the *NACK* and *Core2* stores the opponent core ID '1' on C-bitmap, and *Core2* increments the value which is sent from *Thread1* and updates D-counter with the calculated value '1' (t2). After that, as *Thread3* tries to issue `store B`, another conflict is detected because *Thread2* has already accessed to address B. Therefore *Thread2* sends a *NACK* with the value '1' which is in D-counter (t3). *Thread3* receives the *NACK* and *Core3* stores the opponent core ID '2' on C-bitmap, and *Core3* increments the value which is sent from *Thread2* and updates D-counter with the calculated value '2' (t4). Then, after *Thread1* commits *Tx.1*, *Thread2* receives an *ACK* from *Thread1* (t5). In this case, *Thread2* clears the values on both D-counter and C-bitmap as *Thread2* restarts *Tx.2*. After that, *Thread2* sends a *NACK* with the value '0' which is in D-counter (t6). Thus, *Thread3* receives the *NACK* and *Core3* increments the value which is sent from *Thread2* and updates D-counter with the new value '1' (t7). In this way, each thread keeps track of each own depth.

Next, we describe how D-counter and C-bitmap are updated in the case 3), using Fig.8. Fig.8 shows the same situation shown in Fig.4. Values of D-counter and C-bitmap on *Core1* and *Core2* are 0, and *Core2* has the opponent core ID '1' on C-bitmap and has the value '1' due to a conflict with *Thread1* which is *0-depth* on D-counter. First, *Thread3* conflicts with *Thread2* which is *1-depth*, and *Thread2* replies a *NACK* with the value '1' which is in D-counter. Therefore, *Thread3* receives the *NACK* and *Core3* stores the opponent core ID '2' on C-bitmap, and *Core3* increments the value which is sent from *Thread2* and updates D-counter with the new value '2.' Then *depth* of *Thread3* reaches the threshold 2 (t1). Thus, *Thread3* refers to C-bitmap on *Core3*, and *Thread3* sends an *Abort Req* to *Thread2* which is *2-depth* (t2). When *Thread2* receives this request, *Thread2* aborts its *Tx.2* and *Core2* clears the values on both D-counter and C-bitmap (t3). Afterwards, as *Thread3* receives an *ACK* from *Thread2*, *Core3* clears the values on both D-counter and C-bitmap (t4).

TABLE I
SIMULATION PARAMETERS.

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

TABLE II
REDUCED CYCLE RATE.

		GEMS	SPLASH-2	STAMP	All
(P1)	ave	25.2%	0.9%	-3.2%	11.1%
	max	56.5%	8.0%	-0.5%	56.5%
(P2)	ave	5.1%	-2.5%	-5.4 %	0.3 %
	max	15.2%	1.1%	0.3%	15.2%

V. PERFORMANCE EVALUATION

In this section, we show the evaluation results, and consideration of the results.

A. Evaluation Environment

We used a full-system execution-driven functional simulator *Wind River Simics*[5] in conjunction with customized memory simulators built on *Wisconsin GEMS* [6] for evaluation. Simics provides a SPARC-V9 architecture and boots Solaris 10, and GEMS provides a detailed timing simulation for the memory subsystem. The detailed configuration of the simulated processor is shown in TABLE I. We have evaluated the execution cycles of 9 workloads from GEMS microbench, SPLASH-2 benchmark suite [7], and STAMP benchmark suite [8] with 16 threads.

B. Evaluation Results

The evaluation results with following three HTMs are shown in TABLE II and Fig.9.

(B) LogTM (baseline)

(P1) When a certain thread reaches *2-depth*, *1-depth* threads which are stalled by conflicting with the *2-depth* thread abort their transactions.

(P2) When a certain thread reaches *4-depth*, *1-depth* threads and *3-depth* threads which are involved in the *Cascading Stall* abort their transactions so that all five threads involved in the *Cascading Stall* are completely separated.

Fig.9 shows the execution cycles of the slowest thread in each HTM. Each bar is normalized to the total execution cycles of the baseline LogTM (B).

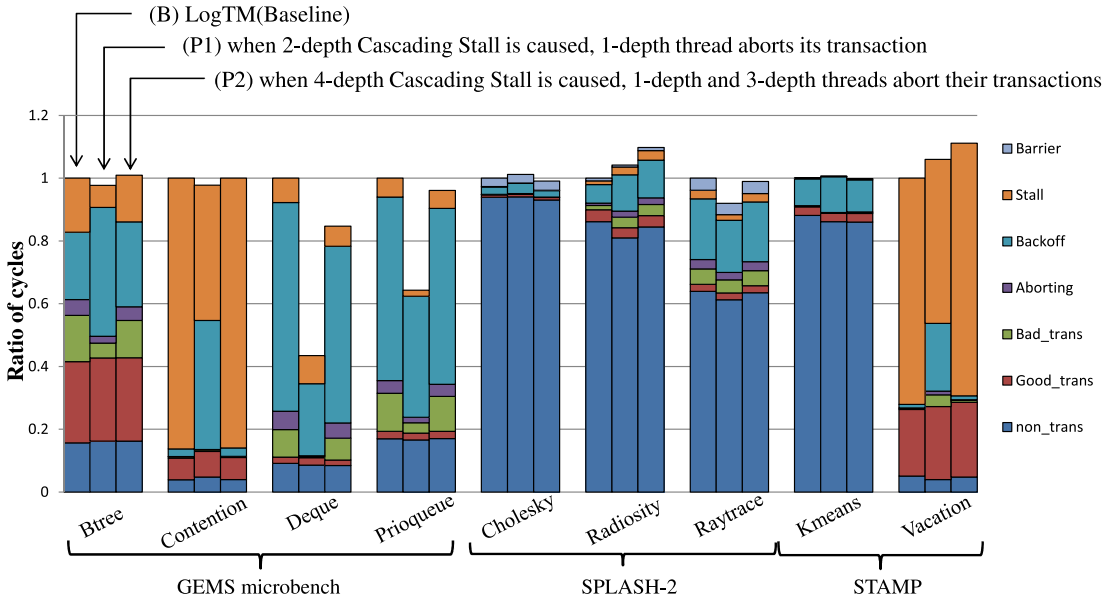


Fig. 9. Execution cycles ratio.

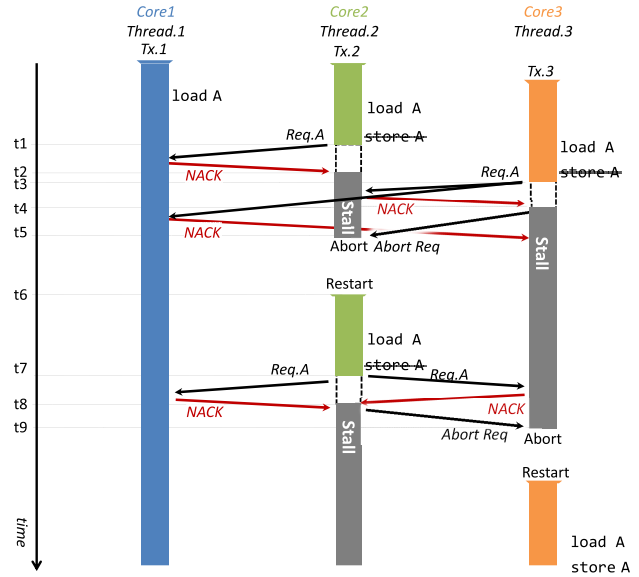


Fig. 10. Example of the case that *Backoff* cycles increase.

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*).

As a result of the evaluation, the execution cycles with the proposed HTM (P1) is reduced 56.5% in maximum and 11.1% in average with 16 threads.

C. Detailed Examination

As shown in Fig.9, the performance is improved with (P1) in many programs. However, the performance is improved slightly with (P2) in most programs. In particular, *Stall* cycles decrease in all programs with (P1). However, *Backoff* cycles increase in most programs. Here, we explain the case that *Backoff* cycles increase with (P1), using Fig.10. In this example, *Thread1* executes *Tx.1*, *Thread2* executes *Tx.2* and *Thread3* executes *Tx.3*, and assume that *Thread1* has already issued `load A` and *Thread2* has also already issued `load A`. When *Thread2* tries to issue `store A` (t1), *Thread1* sends a *NACK* to *Thread2* because *Thread1* has already issued `load A`. Then, *Thread2* receives the *NACK* from *Thread1* and stalls *Tx.2* (t2). Simultaneously, *Thread3* issues `load A`, and tries to issue `store A` (t3). As address A has already been accessed by *Thread1* and *Thread2*, each of the *Thread1* and *Thread2* sends a *NACK* to *Thread3*. Accordingly, the *depth* of *Thread3* comes up to the threshold 2 as *Thread3* receives the *NACK* from *Thread2* which is *1-depth*. Thus *Thread3* sends an *Abort Req* to *Thread2* (t4), and *Thread2* receives the *Abort Req* and aborts its *Tx.2* (t5). Then, *Thread3* receives a *NACK* from *Thread1* which is *0-depth*, therefore *Thread3* keeps stalling although *Thread3* receives an *ACK* from *Thread2*, and *Thread3* keeps to be *1-depth*. After that, *Thread2* restarts *Tx.2* (t6), and *Thread2* issues `load A` and tries to issue `store A` again (t7). Accordingly, the *depth* of *Thread2* comes up to the threshold 2 as *Thread2* receives a *NACK* from *Thread3* which is *1-depth*, and *Thread2* sends an *Abort Req* to *Thread3* (t8). Therefore, *Thread3* aborts its *Tx.3* (t9). Then, *Thread3* restarts its *Tx.3* and a conflict is caused again, as long as *Thread1* does not commit *Tx.1*. In this case, *Thread2* and *Thread3* wait each other despite one of the threads aborts its transaction, and *Thread2* and *Thread3* restart repeatedly. Thus *Backoff*

cycles exponentially increase as the conflict is repeated unless *Thread1* commits *Tx.1*.

Next, the performance of Btree, Contention and Vacation with (P1) is improved. In detail, the number of *Stall* cycles is reduced to about a half, however, the number of *Backoff* cycles increases to ten times at most. We examined these programs, and it is found that these programs include long transactions, and a situation as shown Fig.10 is caused frequently. Thus, *Cascading Stall* is caused repeatedly, and *Backoff* cycles exponentially increase as mentioned above. On the other hand, the performance of Contention and Vacation with (P2) is not improved. This is because *4-depth Cascading Stall* rarely occurs.

In contrast, the performance of Deque and Prioqueue with (P1) and (P2) is improved. In these programs, *Backoff* occupies most of the total cycles. With (P1), not only *Stall* but also *Backoff* is reduced drastically. This is because wasteful stalls and aborts are decreased drastically by resolving *Cascading Stall* as soon as possible. As a result, the number of aborts is reduced more than a half. On the other hand with (P2), few threads reach the threshold 4 because only one small transaction is included in each program. In short, *Cascading Stall* is resolved by stalling transactions before threads reach the threshold 4.

However, the performance of Radiosity with (P1) and (P2) declines. We examined this program and it is found that Radiosity has a transaction which causes conflicts frequently. Thus, although *Cascading Stall* is resolved, conflicts due to the transaction are caused repeatedly. As a result, the number of aborts increases to about three times, and *Backoff*, *Stall* and *Bad_trans* cycles increase.

Next, the performance of Raytrace with (P1) is improved. In this program, wasteful aborts are repeated frequently with (B), thus, not only *Non_trans* but also *Backoff*, *Bad_trans* and *Aborting* occupy most of the total cycles in this program. However, wasteful stalls and aborts are avoided by resolving *Cascading Stall* as soon as possible with (P1). Therefore, *Backoff*, *Bad_trans* and *Aborting* are reduced. On the other hand, the performance is not improved with (P2). The reason is that few threads reach the threshold 4 because transactions are aborted due to frequent conflicts.

In contrast to the other programs, the performance of Cholesky and Kmeans with the (P1) and (P2) is not improved. This is because *Stall* occupies just a little of the total cycles of these programs. Therefore, with the proposed HTMs (P1) and (P2), the ratio of the performance improvement is smaller than the other programs.

D. Abort Request Exchange Overhead

In this section, we examine the overhead for additional messages for resolving *Cascading Stall*. The overhead of additional message *Abort Req* can be calculated as a multiplication of a number which is how many times *Abort Reqs* are sent and the link latency between cores. As shown in TABLE.I, the link latency is assumed to be 14 cycles in this paper. We examined all benchmark programs, and we found that Vacation has the largest ratio of the overhead in all programs. In Vacation, *Abort Reqs* are sent about four thousand times. Therefore, the total overhead cycles of all 16 threads is about $4000 \times 14 = 56000$. Besides, the total execution cycles of Vacation is about 160 millions. Thus, the overhead ratio is only about 0.035% in maximum.

VI. Conclusions

In this paper, we propose a resolution of *Cascading Stall* to avoid decline of concurrency. In order to detect *Cascading Stall*, we define "*depth*" of conflicts, and *Cascading Stall* is resolved when the *depth* of a thread reaches a threshold. We evaluated the proposed HTM by comparing with LogTM, through experiments with GEMS microbench, SPLASH-2 benchmark suite, and STAMP benchmark suite. The evaluation results show that the proposed HTM decreases the total execution cycles 56.5% in maximum and 11.1% in average with 16 threads. However, in the case we define '2' as the threshold for the depth of *Cascading Stall*, conflicts can be caused frequently and repeatedly at almost every time when victim transactions are aborted in some programs. As a result, such repeated conflicts make the number of *Backoff* cycles increase exponentially. On the other hand, in the case we define '4' as the threshold, the number of times that *depth* of each thread reaches the threshold is smaller than in the case that we define '2' as the threshold. Based on the above, our future work is to consider which addresses are the factor of conflicts when victim transactions are selected.

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] 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.

[3] N. Shavit *et al.*, "Software Transactional Memory," in *Proc. 14th ACM Symposium on Principles of Distributed Computing*, 1995, pp. 204–213.

[4] K. E. Moore *et al.*, "LogTM: Log-based Transactional Memory," in *Proc. 12th Int'l Symp. on High-Performance Computer Architecture*, Feb. 2006, pp. 254–265.

[5] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.

[6] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood., "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[7] 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.

[8] 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, pp. 35–46.