This paper is author's private version of the paper published as follows:
*Proc. 2013 High Performance Computing & Simulation Conference (HPCS2013),* pp.374–381


# Reducing Wasteful Recurrence of Aborts and Stalls in Hardware Transactional Memory

Koshiro HASHIMOTO*, Masamichi ETO*†, Shoichiro HORIBA*,
Tomoaki TSUMURA* and Hiroshi MATSUO*

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

†Currently with Central Japan Railway Company
1-1-4, Meieki, Nakamura, Nagoya, Japan

***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. Hence, transactional memory has been proposed and studied for lock-free synchronization. However, the performance can decline with some conflict patterns in TM. Therefore, this paper proposes two methods to restrain the occurrence of very harmful conflicts. The one relieves starving writers who will keep stalling for a long time. The other serially executes highly conflicted transactions which tend to abort repeatedly. The result of the experiment shows that the merged model of these two methods improves the performance 72.2% in maximum and 28.4% in average.**

***Index Terms*—Hardware transactional memory, starving writer, futile stalls.**

## I. Introduction

As electric power consumption and calorific power are increasing, and semiconductor devices keep downscaling, it becomes difficult to raise clock frequencies of microprocessors. In response to this distress, multi-core processors now attract a great deal of attention. On multi-core processors, multiple threads run in parallel for speed-up. For running multiple threads in parallel on shared memory systems, mutual exclusion is required, and *lock* has been commonly used. However, lock-based methods can cause deadlocks, and this leads to poor scalability and high complexity. To solve these problems, *transactional memory* has been proposed as a lock-free synchronization mechanism.

If there is a potential of deadlock when several transactions are stalling, one of the transactions aborts to solve the deadlock on transactional memory systems. However, the victim transaction is determined without considering the conflicting pattern among the transactions. Hence, the performance can severely decline when some harmful conflict patterns occur.

To restrain the occurrence of such very harmful conflicts, this paper proposes two new methods. The one relieves starving writer which will keep stalling for a long time. The other serially executes highly conflicted transactions which tend to abort repeatedly. Both models can reduce the total number of aborts and recurrence of aborts.

## II. Research Background

In this section, we describe the overview of the **Transactional Memory (TM)** [1].

### A. Outline of Transactional Memory

TM is an application of transaction mechanism, which is originally for database consistency, to shared memory synchronization. In TM systems, a transaction is defined as an instruction sequence which covers a critical section, and the transaction satisfies *atomicity* and *serializability*. To ensure atomicity and serializability, TM keeps track of memory accesses, checking whether each requested datum has been accessed by another transaction yet or not. 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 a transaction between the conflicted transactions and stalls the transaction. Then, if one of the conflicted transaction is aborted to avoid deadlocks, the aborted transaction is restarted after its abort. 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. Hardware implementations of TM are called hardware transactional memories (HTMs).

### B. Conflict Detection and Resolution

To detect a conflict, TM must keep track of whether each shared datum is accessed or not. To achieve this, each cache block has two additional bit-fields called *read bit* and *write bit* in general HTM. When a cache block is read through a transaction, HTM sets the read bit for the cache block. In the same way, when a cache block is overwritten, its write bit is set. When a transaction is committed or aborted, HTM resets all these bits which are set through the transaction. To handle these bits, HTM uses an improved cache coherence protocol. To keep caches coherent, the states of cache blocks must be updated. When changing each state, the read bit and the write

bit of the cache block are tested on TM system. If one of the bits is set, a transaction finds that there may be a conflict with another transaction. The access patterns which regarded as conflicts are *read-after-write*, *write-after-read*, and *write-after-write*.

If there is no conflict, the transaction receiving a coherence request from another transaction sends back an *ACK*. On the other hand, if a conflict is detected, a *NACK* will be sent back. In *eager conflict detection* model, when the sender of a request receives a *NACK*, it knows that there is a conflict with the *NACK* sender, and stalls, waiting for the *NACK* sender to commit. The stalled transaction will keep sending the same coherence request. If the opponent transaction commits, the stalled transaction finally receives an *ACK*. In *lazy conflict detection* model, when a transaction tries to commits, all accessed data in the transaction are checked whether they bring conflict or not. In this model, it takes a long time until conflicts are detected, and more transaction executions will be wasted.

### C. Version Management

On TM systems, the interim results of the transactions may be discarded because transactions are executed speculatively. Hence, when a transaction modifies a value on memory, HTM needs to save its address and both new and old values. Now, every transaction definitely have one commit and the commit can not be omitted. Therefore, there is almost no room to reduce the overheads for commit, which are caused on *lazy version management* TMs. 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, which are caused on *eager version management* TMs. Hence, our study aims to improve transaction scheduling on **Log-based Transactional Memory (LogTM)** [2] which adopts eager conflict detection and eager version management.

## III. Related Works

So far, various scheduling techniques for HTM have been proposed. To improve the performance of parallel executions, Yoo et al. [3] have proposed a method which brings the concept of adaptive transaction scheduling (ATS) in TM. ATS can improve the performance of workloads, which lack for parallelism bacause of high contentions, by dynamically dispatching transactions and controlling the total number of concurrent transactions using runtime feedbacks.

Geoffrey et al. [4] have proposed a method which focuses on common memory location which is accessed in multiple transactions. In the method, locality of memory access on each consecutive execution is called similarity and the similarity is calculated with bloom filter. If the similarity exceeds a threshold value, the transactions are serialized. Akpinar etal. [5] have proposed some novel ideas for conflict resolution policies in HTMs such as alternating priorities of transactions in many various ways based on the total number of stalled or aborted transactions. In addition, they take into consideration the most common performance bottlenecks such as InactiveStall and FriendlyFire.

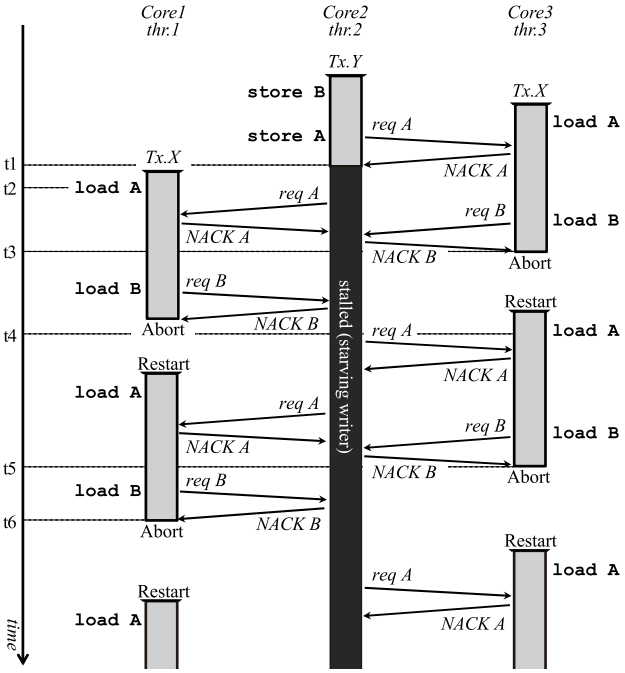


Fig. 1. An example of Starving Writer.

To reduce energy consumption, Gaona et al. [6] have proposed a method which serializes transactions when a conflict arises. Then, if a conflicted transaction has finished, the transaction wakes up the highest-priority transaction among all of the stalled transactions.

In contrast to these methods, we focus on the recurrence of aborts. Because, the restarted transaction which is aborted multiple times tends to be aborted again and this case makes large effect in performance. That is, the conflicts which cause recurrent aborts are very harmful. Therefore, we propose two methods which restrain the recurrence of conflicts.

## IV. Method for Relieving Starving Writers

In this section, we point out a drawback of TM which is called **starving writers**. After that we propose a method for relieving starving writers and describe implementation of the method.

### A. Problem of Starving Writers

When there runs a writer transaction which includes a store instruction to a certain address and multiple reader transactions each of which includes a load instruction from the same address, the writer transaction can be a starving writer [7]. Even if one of the readers is aborted, the writer can not restart because of other readers. Thereafter, the aborted reader is restarted and its load instruction is re-issued. This access is granted by other readers because the access pattern *read-after-read* (RaR) is not regarded as a conflict. Then, the reader also blocks the writer again. Therefore, the writer keeps stalling for a long time.

Fig.1 shows an example where three threads run concurrently. Here, *thr.1* and *thr.3* execute the same transaction

*Tx.X*, which includes `load A`, and *thr.2* executes *Tx.Y* which includes `store A`. Assume that *thr.3* has already executed `load A`, and now *thr.2* tries to execute `store A`. This access request by *thr.2* brings a conflict, and *thr.2* receives a *NACK* from *thr.3* and stalls *Tx.Y* (t1). In this case, *thr.2* can not execute `store A` until *thr.3* commits or aborts *Tx.X*. After that, *thr.1* tries to execute `load A` (t2). This is a RaR access, and brings no conflict. Now, *thr.2* can not continue unless both of *thr.1* and *thr.3* abort or commit their transactions. Hence, if *thr.3* aborts its transaction *Tx.X* by conflicting with *thr.2* (t3), *thr.2* can not continue because *thr.1* has already accessed to the address `A`. Then, *thr.3* restarts *Tx.X* and accesses the address `A` (t4), and *thr.2* can not continue even after *thr.1* aborts its *Tx.X*.

In this way, when there are multiple *reader* threads which try to read a location, a *writer* thread which tries to write to the same location will be starving. Such a writer thread is called a *starving writer*. The reader threads conflicting with the starving writer will abort their transactions repeatedly (t5, t6), and the starving writer will be starving over a long period of time. This will cause a large performance deterioration. When more transactions run concurrently, the number of reader threads will also increase. In such a case, a writer can not continue its execution unless all of the reader threads commit or abort their transactions.

On traditional HTMs, two methods can be used for alleviating the problem of recurrent conflicts. *Exponential backoff* is an algorithm for defining how long an aborted transaction should be wait before its restart. The backoff period for each transaction is initially defined short, and is increased exponentially as the transaction is aborted repeatedly, for avoiding useless aborts. However, under the starving writer situation, reader transactions will still abort repeatedly because backoff periods are initially defined as short.

*Magic waiting* is another algorithm for avoiding recurrence of conflicts. Following this algorithm, when a transaction is aborted, it will postpone its restart until the conflicted opponent transaction will be committed. With this algorithm, a transaction conflicts only once with another transaction, and recurrent aborts can be avoided. However, some transactions should be kept waiting even when they does not repeat aborts, and the transaction concurrency may be drastically reduced.

### B. Relieving Starving Writers

To relieve starving writers, we propose a method that if a reader satisfies two conditions mentioned below, magic waiting is applied to the reader and the reader waits for the conflicted writer to commit.

*Condition SW-I*

A reader transaction, which already issued its load instruction, sends a *NACK* to another writer transaction which tries to issue store instruction. In other words, a write-after-read conflict occurs, and a reader blocks a writer.

*Condition SW-II*

The last two aborts on a reader transaction were caused by conflicts on the same address.

If a reader transaction satisfies both of these two conditions, the writer transaction which is blocked by the reader transaction is assumed as a starving writer. Then, magic waiting is applied to the reader. As a result, the readers, blocking a starving writer, will not be aborted and the starving writer will be preferentially committed and relieved.

### C. Additional Hardware and Execution Model

To implement the mothod for relieving starving writers which is described in section IV-B, we have installed following three hardware units in each core.

*write-after-read flags (WaR flags)*

*WaR flags* are used for keeping track of whether a write-after-read (WaR) conflict occurs or not. When the total number of threads is *n*, these flags have *n*-bit width, and *i*-th bit *WaR[i]* is used for the opponent thread whose thread ID is *i*. When the own core executes a thread which already loaded from a certain address, and a thread on the *i*-th core tries to store to the same address, the associated *WaR[i]* is set.

*Conflicted Address Register (CA reg)*

*CA reg* is used for storing the conflicted address which caused an abort. When the own transaction is aborted due to the opponent transaction and the opponent thread ID is *i* while *WaR[i]* is set, *CA reg* is referred. Then, the current conflicted address is compared to the registered address. If the addresses are same, *M-W flag* which is explained below is set. In contrast, if the addresses are not same, the registered address is overwritten with the current conflicting address.

*Magic Waiting flag (M-W flag)*

Magic waiting is applied if an abort occurs while *M-W flag* is set.

When a processor has 32 cores and can execute 32 threads, and each cache block has 64-bit width, *WaR flags*, *CA reg* and *M-W flag* respectively have 32-bit, 64-bit and one-bit width. Thus, the total cost of these hardwares is only 388Bytes.

In Fig.2, thread *thr.1* runs transaction *Tx.X* on *Core1*, *thr.2* runs *Tx.Y* on *Core2* and *thr.3* runs *Tx.X* on *Core3*. This figure shows the execution model of relieving starving writers with the states of hardwares on *Core2*.

At first, when *thr.2* tries to store some value to the address `A` after *thr.3* loads from `A`, a *write-after-read* conflict is detected (t1). Then, *thr.1* loads from `A` and *thr.2* also conflicts with *thr.1* (t2). Thus, *Core1* and *Core3* set their own *WaR[2]* because the ID of opponent thread is 2.

After that, when each of *thr.1* and *thr.3* tries to load from the address `B`, another conflict is detected (t3, t4) and *CA reg* is referred because *WaR[2]* is already set. At this time, `B` is registered to *CA reg* because no address has been stored in *CA reg*. Then, *thr.1* and *thr.3* abort their *Tx.X* because they detect deadlock.

After they abort, all conflicts are solved, and *WaR flags* on *Core1* and *Core3* are reset. Next, *thr.1* and *thr.3* restart

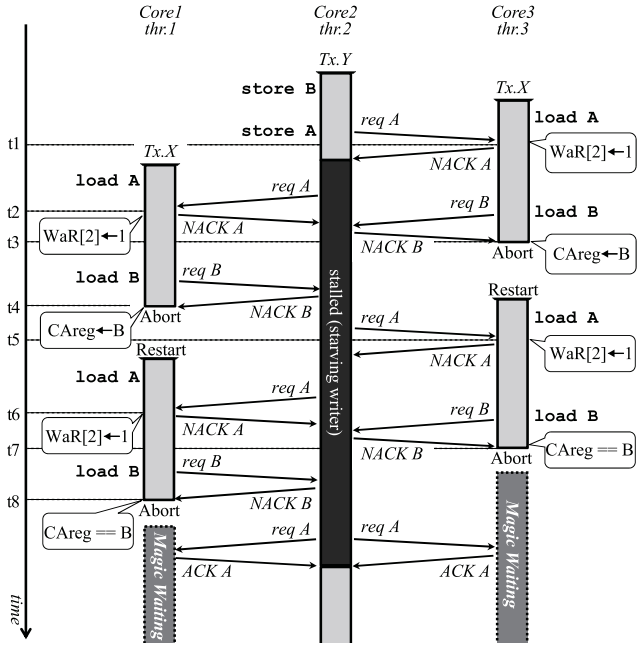


Fig. 2. Execution model for relieving Starving Writers.

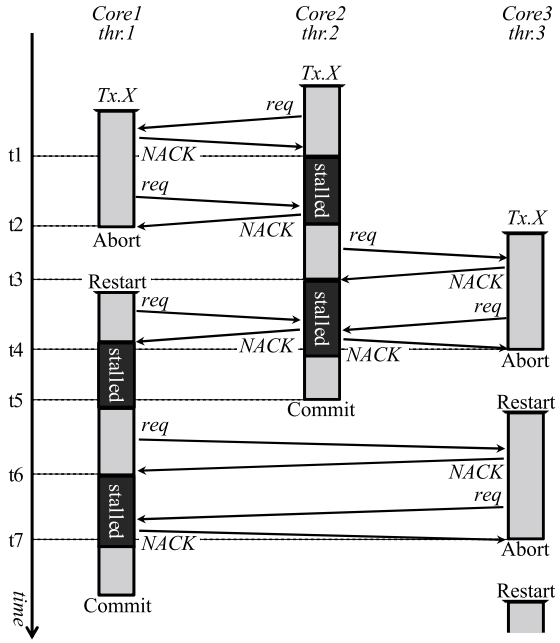


Fig. 3. An example of Futile Stalls

*Tx.X* and the conflicts on A are detected again (t5, t6). At this time, they set *WaR[2]* just like at t1 and t2. After that, the conflicts on B are also detected again (t7, t8). At this moment, on each of *Core1* and *Core3*, *CA reg* is referred and *M-W flag* is set because the current conflicted address B is same with the registered address. Therefore, *thr.1* and *thr.3* regard *thr.2* as a starving writer and applies *magic waiting* to themselves after they abort. Thus, the starving writer *Tx.Y* is relieved.

## V. Method for Avoiding Futile Stalls

This section describes **futile stalls**. Then, we propose a method for avoiding futile stalls and describe implementation of the method.

### A. Problem of Futile Stalls

When conflict-prone transactions run concurrently, the situation, where a logically earier transaction will be stalled by another logically later transaction, happens frequently. After such a conflict, if the transactions conflict with each other again, the later transaction will be aborted and the earier transaction can continue. In this case, the previous stall of the earier transaction becomes simply futile.

Fig.3 shows an example of futile stalls. In this figure, three threads (*thr.1*∼*thr.3*) execute the same transaction *Tx.X*. First, when *thr.2* tries to access an address which is already accessed by *thr.1*, *thr.2* stalls its *Tx.X* (t1). Next, *thr.1* tries to access another address which is already accessed by *thr.2*, *Tx.X* on *thr.1* is aborted (t2) because *Tx.X* on *thr.1* is logically later than *Tx.X* on *thr.2*. Then, *thr.2* can continue its *Tx.X*. In this case, *thr.2* has stalled its *Tx.X* and waited for the commit of *Tx.X* on *thr.1*, but *Tx.X* on *thr.1* is aborted. Hence, *thr.2* need not have stalled its *Tx.X*. Such a stall is called *futile stall*.

After then, if *thr.2* and *thr.3* conflict with each other (t3), the logically earier transaction *Tx.X* on *thr.2* stalls futilely again, and the opponent *Tx.X* on *thr.3* is aborted (t4). Even after *thr.2* commits its *Tx.X* (t5), *Tx.X* on *thr.1* and *Tx.X* on *thr.3* conflict each other, and *thr.1* stalls its *Tx.X* (t6), and *thr.3* aborts its *Tx.X* (t7). As you can see in this example, under the situation where the transactions, which potentially cause conflict frequently, run concurrently, many stalls become futile, and some transactions such as *Tx.X* on *thr.3* will be aborted repeatedly. If larger number of transactions run concurrently, more conflicts will occur, and consequently, the total performance will severely decline with frequent aborts and many futile stalls.

### B. Avoiding Futile Stalls

To avoid recurrence of aborts in highly conflicted transactions, we propose a method that if a transaction satisfies two conditions mentioned below, the transaction is serialized. As a result, the highest-priority transaction would not be stalled and its opponent transactions would not be aborted. Here, we use two criteria to determine whether each transaction should be serialized or not. The criteria are about the total number of aborts and executed instructions. Each criterion is defined using a certain threshold value $A(tx)$ or $L(tx)$, where $tx$ indicates transaction ID. The two threshold values are configured for every transaction. The reason for considering the total number of executed instructions is that when multiple transactions, which have a lot of instructions, are executed in serial order, the performance may drastically decline.

*Condition FS-1*

A transaction is more than $A(tx)$ times aborted repeatedly before commit.

*Condition FS-II*

The total number of the executed instructions in a transaction is less than the threshold value $L(tx)$.

If multiple transactions satisfy both of these two conditions, the transactions will be executed in serial order. Then, a lot of harmful conflicts are avoided. Incidentally, the number of executed instructions in a transaction can be measured much larger than the number of instructions the transaction actually contains. A major reason is switching on interrupts. Hence, some transactions should be executed in serial order even if the observed number of executed instructions is occasionally larger than $L(tx)$. To deal with this case, we configure another threshold value $S(tx)$, which is smaller than $L(tx)$, and this is used for the transactions which did not satisfy *Condition FS-II*. If the total number of the executed instructions in a transaction becomes less than $S(tx)$ after it is larger than $L(tx)$, the transaction should not be so large in fact, and is considered to satisfy *Condition FS-II*.

### C. Additional Hardware and Execution Model

To implement the method for avoiding futile stalls which is described in section V-B, we have installed following several hardware units in each core. To explain each hardware briefly, we assume that an ID of the transaction which is executed by the own core is *i*.

*Abort Counter (A-Counter)*

*A-Counter* is used for recording the total number of aborts in each transaction. The recorded value is reset on every commit.

*Recurrence flags (R-flags)*

*R-flags* are used for keeping track of whether aborts are repeated or not. When the value of *A-Counter* becomes equal to the threshold value $A(tx)$, the *i*-th bit in *R-flags* is set.

*Instruction Counter (I-Counter)*

*I-Counter* is used for recording the total number of executed instructions in each transaction. The recorded value is reset on every abort and commit.

*ID of Opponent Thread (O-id)*

*O-id* is used for recording the ID of the opponent thread which waits for commit of the own transaction. When the own thread commits its transaction, the thread refers *O-id* to know the opponent transaction who waits the commit, and clears *O-id*.

*Short Tx flags (Stx-flags)*

*Stx-flags* are used for keeping track of whether executed instructions are few or not. When a transaction is committed while the value of *I-Counter* is smaller than specific threshold value $L(tx)$, the *i*-th bit in *Stx-flags* is set.

*Long Tx flags (Ltx-flags)*

*Ltx-flags* are used for keeping track of whether executed instructions are many or not. When a transaciton is committed while the value of *I-Counter* is larger than specific threshold value $L(tx)$, the *i*-th bit in *Ltx-flags* is set. On the other hand, if the value of *I-Counter* is smaller than threshold value $S(tx)$, the *i*-th bit is reset.

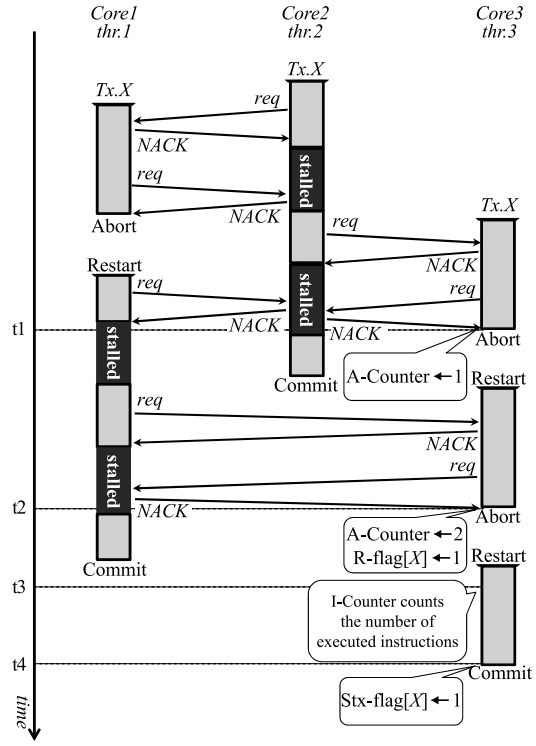


Fig. 4. Execution model for avoiding Futile Stalls.

*Serializing flag (S-flag)*

*S-flag* is used for controlling the order of serialized transactions. The last thread who tries to execute a serialized transaction sets this flag. Thus, this flag is set when the own thread starts to wait a commit of another transaction. This flag is reset when another thread starts to wait a commit of the own transaction.

Here, typical parallel programs seem to have less than 10 kinds of transactions. Thus, we estimate the cost for additional hardware units which can deal with such typical programs. When a processor has 32 cores and can execute 32 threads, *A-Counter*, *R-flags*, *I-counter*, *O-id*, *Stx-flags*, *Ltx-flags* and *S-flag* respectively have 4-bit, 16-bit, 10-bit, 5-bit, 16-bit, 16-bit and 1-bit width. Thus, the total cost is only 272Bytes.

Fig.4 shows an example of parallel execution on highly conflicted transactions. In this example, *Tx.X* on *thr.2* is futilely stalled by the other two transactions. Therefore, a commit of *Tx.X* on *thr.2* is delayed. To avoid this, highly conflicted transactions should be serialized. Serialization process consists of two steps. The first step is for deciding which transactions should be serialized and the second is for controlling the order of serialized transactions. In the first step, the value of *A-Counter* is incremented every abort (t1, t2). When the value reaches to the threshold $A(tx)$, *R-flag* is set (t2). If the transaction is executed while *R-flag* is set, the total number of instructions is counted on *I-Counter* (t3). If a transaction which has the thread ID *i* is committed while the value of *I-*

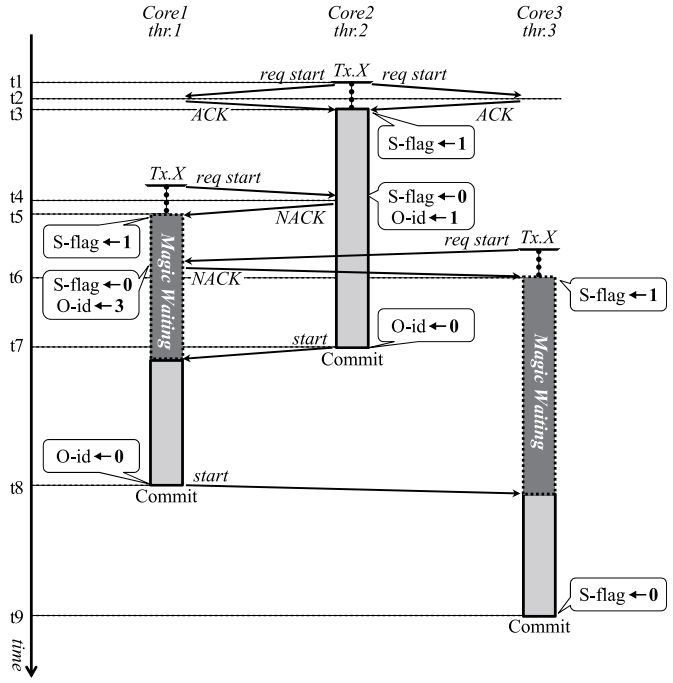


Fig. 5. Ordered wake-up mechanism.

*Counter* is smaller than the threshold $L(tx)$, *Stx[i]* is set (t4). On the other hand, if the transaction is committed while the value of *I-Counter* is larger than $L(tx)$, *Ltx[i]* is set. If *Ltx-flag* is not set and *Stx-flag* is set, the transactions are considered to be executed in serial order.

Fig.5 shows the execution model of the second step. Here, *Tx.X* is the transaction which is decided to be serialized in the first step, and three threads now try to execute *Tx.X* concurrently. First, when *thr.2* tries to execute *Tx.X*, *thr.2* sends *req start* to all threads to confirm that no other thread is executing this transaction (t1). At this time, no other thread executes *Tx.X* except for *thr.2* (t2). Therefore, *thr.2* starts to execute *Tx.X* as usual (t3). Then, *thr.2* sets its *S-flag* because the serialized transaction is executed only on *thr.2*. Next, when *thr.1* tries to execute *Tx.X*, *thr.2* already started *Tx.X*. Thus, *thr.1* does not start *Tx.X* and waits for the commit of *Tx.X* on *thr.2*. At this moment, *thr.2* clears *S-flag* and sets 1 on *O-id* because the ID of the opponent thread is 1 (t4). In addition, *thr.1* sets *S-flag* (t5). After that, when *thr.3* tries to execute *Tx.X*, *thr.3* sets its *S-flag* (t6) and waits for the commit of *Tx.X* on *thr.1* and *thr.1* resets its *S-flag* and sets 3 on *O-id*. Then, when *thr.2* commits *Tx.X* (t7), *thr.2* refers *O-id* in order to know who is waiting its commit. Because 1 is stored in *O-id*, *thr.2* wakes up *thr.1* to start *Tx.X* and clears its *O-id*. Thereafter, *thr.1* starts to execute *Tx.X*. Just like at t7, when *thr.1* commits *Tx.X*, *thr.1* wakes up *thr.3* and clears its *O-id* (t8). Finally, *thr.3* commits *Tx.X* and clear its *S-flag* (t9). As described above, serialized transactions are controlled.

TABLE I
SIMULATION PARAMETERS.

| | |
|---|---|
| Processor | SPARC V9 |
| total number of cores | 32 cores |
| frequency | 1 GHz |
| issue width | single-issue |
| 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 | 4 GBytes |
| latency | 450 cycles |
| Interconnect network latency | 14 cycles |

TABLE II
INPUT PARAMETERS FOR BENCKMARK PROGRAMS.

| | |
|---|---|
| GEMS | |
| Btree | priv-alloc-20pct |
| Contention | config 1 |
| Deque | 4096ops 128bkoff |
| Prioque | 8192ops |
| Slist | 500ops 64len |
| SPLASH-2 | |
| Barnes | 512BODIES |
| Cholesky | tk14.0 |
| Radiosity | -p 31 -batch |
| Raytrace | teapot |
| STAMP | |
| Genome | -g256 -s16 -n16384 -t16 |
| Kmeans | -m40 -n40 -t0.05 -p16 |
| Vacation | -n2 -q90 -u98 -r16384 -t4096 -c16 |

## VI. PERFORMANCE EVALUATION

### A. Simulation Environments

We used a full-system execution-driven functional simulator *Wind River Simics* [8] in conjunction with customized memory models built on *Wisconsin GEMS* [9] for evaluation. Simics provides a SPARC-V9 architecture and boots Solaris 10. GEMS provides a detailed timing model for the memory subsystem. This system has 32 processors, each with two levels of private caches. Illinois-based directory protocol maintains cache coherence over a high-bandwidth switched interconnect. The detailed simulation parameters are shown in TABLE I. We have evaluated the execution cycles of 12 workloads from GEMS microbench, SPLASH-2 benchmark suite [10], and STAMP benchmark suite [11]. The input parameters for the benchmark programs are shown in TABLE II.

Each workload was executed with 31 threads, because one of the 32 cores should be a default core which cannot be used for user programs. However, STAMP benchmark can run only if the total number of threads is power of 2. Therefore, only STAMP benchmark programs were evaluated with 16 threads.

### B. Evaluation Results

The evaluation results with following four models are shown in Fig.6 and TABLE III.

(B) LogTM (baseline)

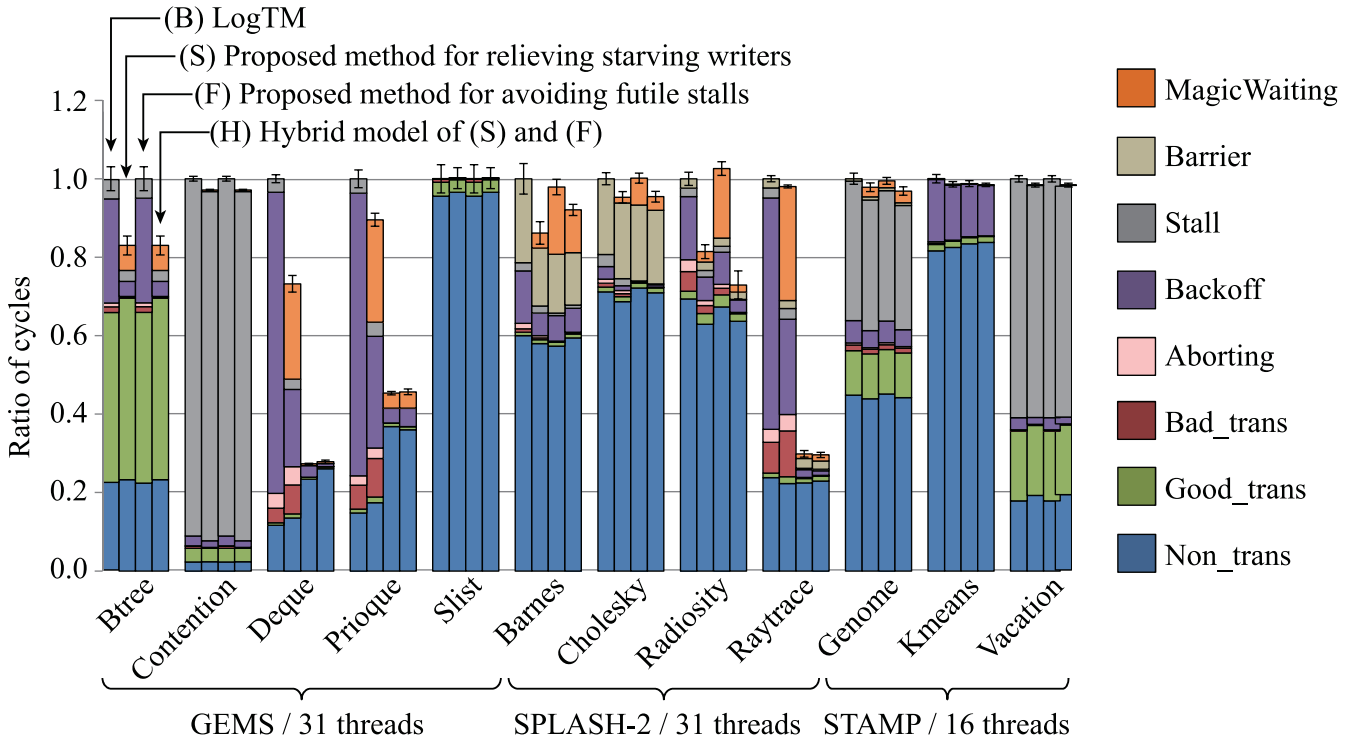


Fig. 6. Ratio of execution cycles w/ GEMS, SPLASH-2 and STAMP benchmark suites.

TABLE III
REDUCED CYCLES RATE.

| | | GEMS | SPLASH-2 | STAMP | all |
|---|---|---|---|---|---|
| (S) | Mean | 8.5% | 10.3% | 1.7% | 7.5% |
| | Max | 17.3% | 18.7% | 1.9% | 18.7% |
| (F) | Mean | 31.7% | 26.8% | 0.9% | 19.8% |
| | Max | 72.7% | 71.5% | 1.8% | 71.5% |
| (H) | Mean | 36.6% | 34.0% | 2.1% | 28.4% |
| | Max | 72.2% | 70.4% | 3.1% | 72.2% |

(S) Proposed method for relieving starving writers
(F) Proposed method for avoiding futile stalls
(H) Hybrid model of (S) and (F)

For model (F) and (H), we configured the threshold values $A(tx)$ as 4, $L(tx)$ as 512, and $S(tx)$ as 128. Fig.6 shows the execution cycles of each model. Each bar is normalized to the total number of executed cycles of the baseline LogTM (B). The legend shows the breakdown items of 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 overhead (**Aborting**), the stall cycles (**Stall**), the *magic waiting* cycles (**MagicWaiting**), the barrier synchronization cycles (**Barrier**), and the exponential backoff cycles (**Backoff**). For the simulation of multi-threading on a full-system simulator, the variability performance [12] 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.

First, the performance is improved with model (S) in the most of the programs. This model reduces Bad_trans, Aborting, Stall and Backoff in Barnes, Btree, Deque, Prioque, and Radiosity, because a lot of aborts are restrained. Especially, in Btree, the total number of aborts and the total number of recurrence of aborts are reduced to approximately 1/8 and 1/4 respectively. In addition, the model (S) slightly achieves speed-up in Contention, Cholesky Genome, Kmeans and Vacation because a lot of aborts are restrained, although ratio of Aborting is low. In these programs, the total number of aborts is reduced at most 72.9% (Kmeans) and at least 17.1% (Genome). Although model (S) increases MagicWaiting, the performance improvement by relieving starving writers more than offsets the bad effect in the most programs.

Meanwhile, model (F) achieves considerable performance gain in Deque (72.2%), Prioque (54.6%), and Raytrace (71.5%). In particular, the total number of aborts is reduced 99.8% in Deque, 99.6% in Prioque, and 99.8% in Raytrace. Nonetheless, MagicWaiting represents only 0.4%, 4.0% and 1.1% of the total cycles in each program. However, in several programs, the results of model (F) did not vary from those of traditional model (B). In addition, MagicWaiting takes up quite a lot of ratio in Barnes, Cholesky, and Radiosity. The reason will be that the threshold value $A(tx)$ is not appropriate for these programs and too much transactions are serialized. Therefore, serializing specific transactions leads to performance deterioration, and the good effect by serializing other transactions is cancelled out.

Finally, in each program except Barnes, hybrid model (H) reduces the same or more cycles than the better one of (S) and (F). In Barnes, transactions which achived speed-up with model (S) are serialized at early phase of the program. Therefore, the result is closed to the model (F). However, the results of Radiosity and Genome are more improved than those of other methods. In these programs, some transactions, which would be starving writers, are relieved with model (S) and the transactions would not be aborted. On the other hand, highly conflicted transactions, which are aborted many times with even model (S), are executed in serial order with model (F) and the transactions would not be aborted. Therefore, more appropriate method is applied automatically to each transaction.

## VII. Conclusions

In this paper, we proposed two methods to restrain the occurrence of very harmful conflicts. The one relieves starving writer, and the other serially executes highly conflicted transactions to avoid futile stalls. Through an evaluation with microbench in GEMS, SPLASH-2 and STAMP suite benchmark programs, it is found that the hybrid model of the two proposed methods improves the performance 72.2% in maximum and 28.4% in average.

Incidentally, performance slightly declines with the method for avoiding futile stalls in some programs. Therefore, one of our future work is dynamically and more appropriately configuring the threshold value of $A(tx)$, $L(tx)$ and $S(tx)$. How to utilize the idle cores with stalled transactions is also left for our future work.

## References


[1] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proc. 20th Annual Int'l Symp. on Computer Architecture*, May. 1993, pp. 289–300.

[2] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Proc. 12th Int'l Symp. on High-Performance Computer Architecture*, Feb. 2006, pp. 254–265.

[3] 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.

[4] G. Blake, R. G. Dreslinski, and T. Mudge, "Bloom Filter Guided Transaction Scheduling," in *Proc. 17th International Conference on High-Performance Computer Architecture (HPCA-17 2011)*, 2011, pp. 75–86.

[5] 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.

[6] E. Gaona, R. Titos, M. E. Acacio, and J. Fernández, "Dynamic Serialization Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems," in *Proc. Parallel, Distributed and Network-Based Processing 2012 20th Euromicro International Conference (PDP'12)*, 2012, pp. 221–228.

[7] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood, "Performance Pathologies in Hardware Transactional Memory," in *Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA'07)*, 2007, pp. 81–91.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.