2010 First International Conference on Networking and Computing

Proposition of Criteria for Aborting Transaction based on Log Data Size in LogTM

Hiroki ASAI*, Tomoaki TSUMURA* and Hiroshi MATSUO*

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

***Abstract*—Lock-based synchronization techniques are commonly used in parallel programming on multi-core processors. However, lock can cause deadlocks and poor scalabilities. Hence, LogTM has been proposed and studied for lock-free synchronization. LogTM is a kind of hardware transactional memory. In LogTM, transactions are executed speculatively to ensure serializability and atomicity. LogTM stores original values in a log before it is modified by a transaction. If a transaction accesses a shared datum which has been accessed by another transaction running in parallel, LogTM detects it as conflict and restores all data from the associated log and restarts the transaction. This is called aborting. On abort, the costs for restoring data from a log increases in proportion to the data size on the log. However, LogTM selects which transaction should be aborted by their initiated time. Hence, if conflicts occur frequently, it may degrades the performance. This paper proposes a criterion for selecting which transaction should be aborted taking account of data size in each log. In addition, another criterion which takes account of degree of conflict is also proposed. The result of the experiment with SPLASH-2 benchmark suite programs shows that the proposed methods improve the performance 2.7% in maximum.**

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. Multi-core processors consist of several independent cores on a chip. On multi-core processors, multiple threads run in parallel for speedup. For this multi-threaded parallel execution, shared memory programmings are commonly used. In shared memory programmings, independent cores share a single address space. Hence, an exclusive control is required. Lock-based synchronization methods have been commonly used for shared memory parallel programmings. However, lock-based method can cause deadlocks, and it leads to poor scalability and high complexity. Meanwhile, LogTM[1] is proposed as a lock-free synchronization mechanism.

LogTM is a kind of *Hardware Transactional Memory*. On a system with a transactional memory, transactions are executed speculatively to ensure the serializability and atomicity. A transaction is an instruction sequence which includes a certain critical section. LogTM stores original values into a log before they are modified by a transaction. When a transaction accesses a shared data which other transactions running in parallel have already accessed, it will be detected as a **conflict**. Then, LogTM **abort**s the transaction, restores all the original values from the log to the caches or the memory, and restarts the transaction.

On abort, restoring original values from the log costs in proportion to the stored data size on the log. However, the traditional LogTM selects the victim transaction which should be aborted only by comparing the timestamps of when the transactions started. Therefore, LogTM may abort the transaction which will cost more cycles for being aborted. If conflicts occur frequently, this may deteriorate the total performance.

Furthermore, when a lot of threads are executed in parallel, one transaction can block many other transactions. Such a transaction should not be aborted, because the transaction conflicted with many other transactions may cause a lot of conflict again after being aborted and restarted.

This paper describes two criteria for deciding which transaction should be aborted considering to the data size on the log and the number of transactions conflicted with. The methods with these criteria dynamically select a victim transaction by taking all of the data size on the log, the ages of the transactions and the number of transaction conflicted into account.

II. Research Background

A. Transactional Memory

Shared memory programmings are common for parallel programmings on multi-core processors. In shared memory programmings, several independent cores share a single address space and the shared resource must be kept synchronized between threads. Lock-based synchronization has been commonly used for shared memory parallel programming. However, lock-based methods can cause deadlocks, poor scalability and higher complexity. When we use a lock, we must consider the granularity of transactions. With a coarse granularity, lock mechanism is easy to use for programmers, but it will reduce the parallelism as the number of threads increases. On the other hand, with a fine granularity, lock mechanism increases parallelism but there should be much difficulty in its programming.

978-0-7695-4277-5/10 $26.00 © 2010 IEEE
DOI 10.1109/IC-NC.2010.51

Meanwhile, **Transactional Memory**[2] is proposed for lock-free synchronization. Transactional memory is an application of transaction mechanism which is originally for database consistency to shared memory synchronization. On the transactional memory mechanism, a transaction is a instruction sequence which covers a critical section, and the transaction satisfies the following properties:

Serializability
: The results of multiple transactions must not depend on whether they have been executed parallely or serially.

Atomicity
: Transactions must be guaranteed not to be executed partly, but either to be completed or to be left unexecuted.

To ensure atomicity and serializability described above, transactional memory keeps track of memory accesses checking whether each accessed datum has been accessed yet by another transaction or not. When a transaction accesses the same memory address which has been accessed by another transaction, transactional memory detects it as a **conflict** between the transactions. To solve the conflict, transactional memory selects a victim transaction among the two transactions concerned, discards all updates by the victim transaction and restarts it. On the other hand, if there occurs no conflict through a transaction, transactional memory make all updates by the transaction visible to other threads (called **commit**).

As far as there is no conflict between transactions, transactions can be concurrently executed under the transactional memory without any blocking. Moreover, it is easy to use because there needs not to consider granularities.

B. LogTM

LogTM is a kind of transactional memory implemented with specialized hardware support. Fig. 1 shows the structure of LogTM. Each core has two levels of private caches and a cache controller, and shares a memory with other cores.

1) Version Management:

On LogTM system, transactions are executed speculatively. Since the results of the transactions may be discarded, through the speculative executions, transactions must save data in per-thread **log** space on cachable virtual memory before the data was updated on the shared memory. This is called **version management**. LogTM appends the current value and its virtual address to the log when a store operation occurs within the transaction.

On abort, the transaction restores data on its log to the shared memory. After then, the transaction can restart. If the log has a lot of entries (i.e. pairs of value and address), the restore process will costs many cycles and the restart will be delayed. On the other hand, on commit, the transaction should only discard all the data on its log.

Now, let us see how the version management works. In Fig. 2, the `Thread` executes a transaction and has its `Log` and a shared `Memory`.

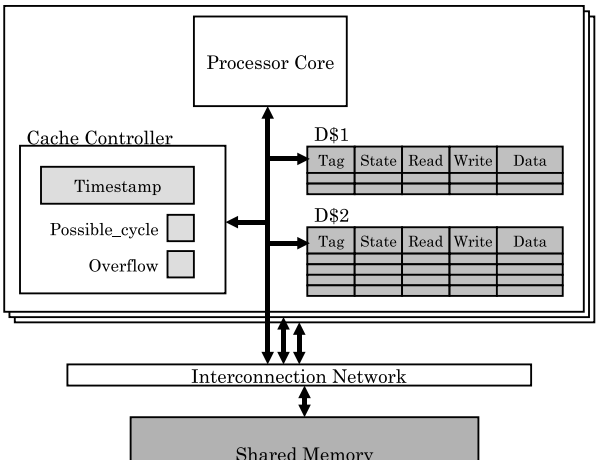

Fig. 1. Structure of LogTM.

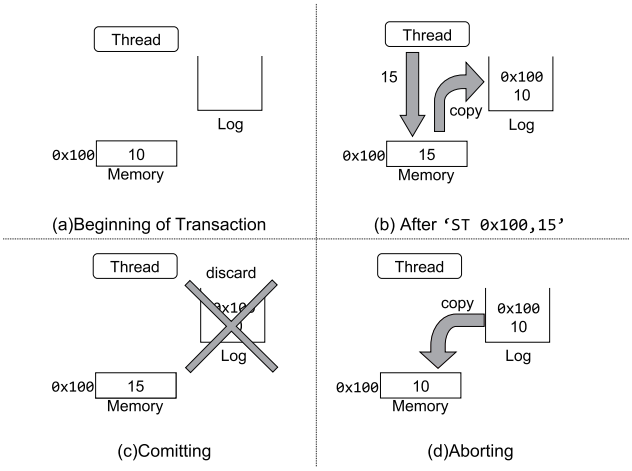

Fig. 2. Management of data versions.

First, the shared memory has a value 10 at `0x100` (a). When the thread issues the instruction `ST 0x100,15`, the value `15` and the address `0x100` are being stored into shared memory, and the previous value `10` and the address `0x100` which will be overwritten by the instruction are stored into the log (b).

If the speculative execution of the transaction succeeds, *commit* is required. On LogTM, all updates have already been stored in the memory, and the thread should only discards all backups in the log (c). On the other hand, if the speculative execution fails, the thread must *abort* to discard all updates and restart the transaction. This can be done by restoring the backup values in the log to the memory and discarding the log contents (d). The register values also should be restored as they had been as the begging of the transaction.

As a result, the memory access overhead increases in proportion to the log data size which will be written back on *abort*, but there occurs no memory access on *commit*. This behavior of LogTM is designed considering the fact that any

transaction will commit in the end. By making indispensable *commit*s faster, LogTM tries to improve total performance. When *abort* is rare, its overhead does not matter.

2) Conflict Detection:

If a transaction accesses a shared data which another transaction running in parallel has accessed, LogTM detects it as a conflict between the transactions (**conflict detection**). To detect a conflict, LogTM keeps track of cache accesses by other transactions. Each cache block has **read bit (R-bit)** and **write bit (W-bit)**. When a read access occurs through a transaction, LogTM sets the R-bit for the cache block. As well as R-bit, W-bit is set when a write access occurs.

To notify the conflict to other transactions, LogTM extends cache coherence protocol which was a combination of directory[3] and Illinois protocol[4]. To keep caches coherent, the states of cache blocks must be updated. When changing the state, R-bit and W-bit of the cache block will be tested on LogTM system. If one of the bits is set, the transaction finds that there may be a conflict with another transaction. There are following three cases that cause conflict:

read-after-write
: The case where a transaction reads the value which has been written by another transaction. The transaction may access the value before another transaction commits.

write-after-read
: The case where a transaction writes the value which has been read by another transaction. That is, through an execution of an transaction, another transaction may change the value which is used by the former transaction.

write-after-write
: The case where a transaction writes the value which has been written by another transaction. As well as *write-after-read*, a transaction may change the value before another transaction commits.

If there is no conflict, the transaction receiving a coherence request from another transaction sends back an *ack* reply. On the other hand, if a conflict is detected, a *nack* reply will be sent back. If the sender of the request receives a nack, it knows there is a conflict with the *nack* sender, and waits for the *nack* sender to commit. This is called **stall**. The stalled transaction will keep reissueing the same coherence request. If the other transaction commits and is completed, the stalled transaction finally receives an *ack* reply.

Now, we explain the process of finding a conflict between transactions. In Fig. 3, the transaction `trans1` is executed speculatively in the thread `Thread1`. Likewise, `trans2` is executed speculatively in `Thread2`.

First, let us see the case that no conflict occurs (a). `trans1` sends a coherence request to `trans2` before `trans1` issues `LD 0x100` instruction. At time `t1`, `trans1` can issue the load instruction because `trans2` has not accessed the address `0x100` yet.

After that, `trans2` sends a coherence request to `trans1` before `trans2` issues `LD 0x100` at `t2`. Actually, the coherence request is sent not to `trans1` but to the directory associated with `trans1`. Then, `trans1` knows that `trans2` is going to access at the address `0x100` by receiving the request. In this case, `trans1` does not detect a conflict because this memory access is *read after read*. Therefore, `trans2` receives an *ack* reply and can load value from `0x100`.

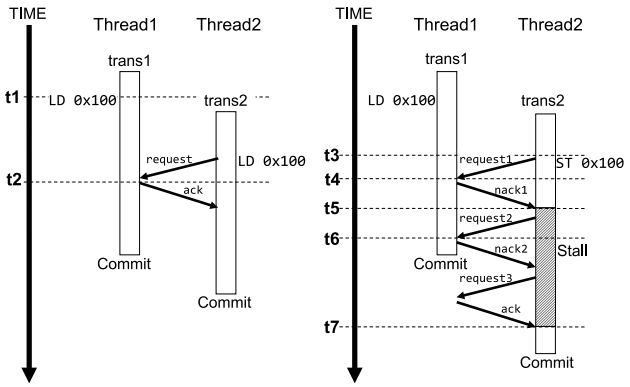

Fig. 3. Conflict detection.

Next, let us see the case that a conflict occurs (b). As well as the case (a), `trans1` issues the instruction `LD 0x100`. After that, `trans2` sends the coherence request `request1` at `t3` before `trans2` issues `ST 0x100`. Now, `trans1` detects a conflict because this access is a *write after read*. Therefore, `trans1` sends the nack reply `nack1` to `trans2` at `t4` to notify a conflict. Receiving `nack1`, `trans2` stalls at `t5`, and keeps sending requests (`request2` and `request3`) until `trans1` commits. When `trans1` commits, `trans2` receives an ack reply from `trans1` at `t6`. Therefore, `trans2` finds that it is now able to access the address `0x100` and returns from stall at `t7`.

If a lot of transactions stall, there comes some risks of their deadlocking. For example, assume that a transaction `trans1` sends a *nack* to another transaction `trans2` and `trans2` sends a *nack* to `trans1`. This makes a deadlock because `trans1` waits for `trans2`'s commit and vice versa. To dissolve this deadlock, LogTM should abort either transaction. On abort, the traditional LogTM selects the transaction which has started later than the other as the victim, because the earlier transaction should have possibly accessed more memory values than the other. Hence, it is regarded that the earlier transaction should be committed as soon as possible to avoid further conflicts.

For detecting deadlocks, LogTM uses *possible_cycle* flag of TLR's distributed timestamp method[5]. Each transaction has its own *possible_cycle* flag, and sets the flag when it sends a *nack* reply to another elder transaction. Then, if the transaction whose *possible_cycle* has been set receives a *nack* reply from another transaction, the transaction detects a deadlock and is aborted.

Now, let us see an example of detecting a conflict and

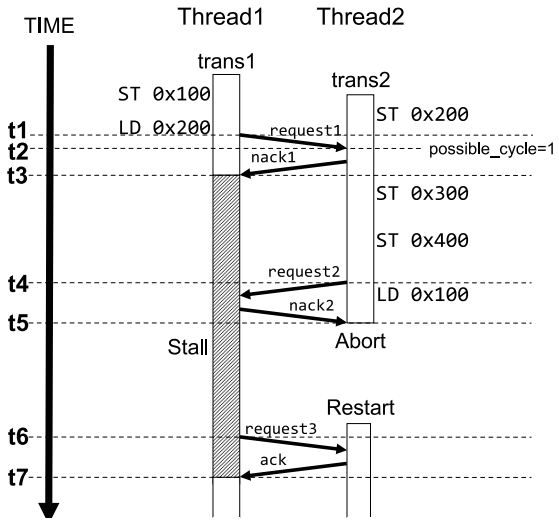

Fig. 4. Conflict Resolution.

restarting a transaction. In Fig. 4, the transaction `trans1` is initiated in `Thread1` before the transaction `trans2` starts in `Thread2`. When an address in the shared memory is accessed, LogTM first checks whether the *R/W-bit* for the cache block is set or not through the cache coherence protocol. If the *R-* or *W-bit* is set and the memory access is one of the *read-after-write*, *write-after-read* or *write-after-write*, a conflict is detected.

The transaction `trans1` executes `ST 0x100` before `trans2` issues `ST 0x200`. After that, `trans1` sends a coherence request `request1` to `trans2` at `t1` before `trans1` executes `LD 0x200`. Now, `trans2` detects a conflict between `trans1` and `trans2` because `trans2` already has stored into `0x200`. So, `trans2` sends a nack reply `nack1` to `trans1` for notifying a conflict and the *possible_cycle* of `trans2` is set at `t2`. At `t3`, `trans1` receives `nack1` and it stalls. On the other hand, `trans2` can execute instructions `ST 0x300` and `ST 0x400` because they do not conflict with other transactions. After that, when `trans1` receives the coherence request `request2` about the address `0x100` from `trans2` at `t4`, `trans1` detects a conflict with `trans2` and sends `nack2` to `trans2` at `t5`. In this case, `trans2` aborts since `trans2` has been initiated after `trans1` and has its *possible_cycle* flag set.

If `trans2` does not abort in this example, `trans1` will wait for `trans2`'s commit and `trans2` will wait for `trans1`'s commit. This is a deadlock between `trans1` and `trans2`. Hence, one of the transactions must be aborted to dissolve this conflict. After `trans2` is aborted, it returns to the checkpoint of its beginning and restarts at `t6`. After the restart, if `trans2` receives a coherent request from `trans1`, it replies back an ack this time to notify the conflict has been dissolved. Then, `trans1` can continue.

III. Proposition of New LogTM Models

In this section, we point out a drawback of LogTM, and propose two new models which will improve total performance of LogTM by selecting a victim transaction based on log data size.

A. Problem with LogTM

One of the drawbacks of LogTM is the high cost of transaction abort. The cost appears when a transaction writes back the backup values from the log to the shared memory for restoring the memory state as the transaction started. This *write back cost* increases in proportion to the number of entries in the log (*log data size*). Whenever a transaction issues a store operation, the log data size increases. In general, memory access latency is expensive. As a result, the more log data size grows, the more write back cost increases. Therefore, the performance may be reduced when a conflict and abort occur frequently.

In Fig. 4, either transaction has some entries in its log, and its number of entries is same as the number of store operations which have been issued in the transaction. In this case, `trans1` has one entry in the log and `trans2` has three. Therefore, if `trans2` is aborted, three blocks have to be restored into the memory. On the other hand, if `trans1` is aborted, restoring only one block is enough. As a result, `trans1` may be more suitable for a victim to be aborted rather than `trans2`, although `trans1` has been initiated before `trans2`.

B. Selecting Victim Transaction based on Log Data Size

First, we propose a new LogTM model for reducing the overheads by selecting a victim transaction with a certain criterion. The criterion considers the log data size, since aborting cost depends on how many data are stored in the log.

Of course, aborting the transaction which has smaller log data size does not always derive good performance. As we have seen in II-B, the transaction which has been initiated earlier should be committed for preventing frequent conflicts. The age of a transaction is also an important criterion. Both of the log data size and the transaction age should be considered.

However, a traditional LogTM only considers the transaction age. This can lead to the situation where a transaction, whose aborting overhead is expensive because of its huge log data size, is selected as a victim. Essentially, the transaction which costs higher aborting cost should be a victim transaction. Hence, we introduce a new criterion for selecting a victim transaction considering both the log data size and the age of the transactions.

Constructing a new criterion, how much cycles will be overhead or how much cycles will come to nothing through an abort should be considered. The cycles consist of two factors. The one is the how many log data should be written back to the shared memory. The other is how much cycles does it take until the aborted transaction replays from the beginning and comes back to the state just before the abort. Letting $L(tr)$

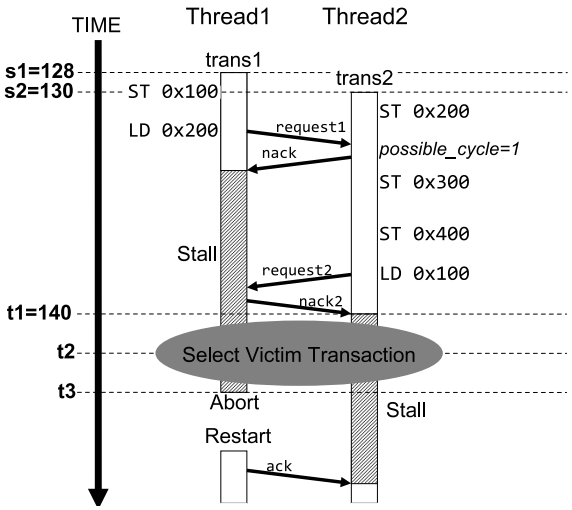

Fig. 5. Conflict resolution based on log data size.

be the former and $T(tr)$ be the latter where tr is transaction ID, the new criterion which represents the total cycle overhead can be calculated as follows:

$$C(tr) = k \cdot L(tr) + T(tr) \quad (1)$$

where k is the system-specific constant cost for writing one log entry back to the caches or the shared memory.

Now, we show an example where the new LogTM model selects the transaction which has lower cost. In Fig. 5, `trans2` receives a nack `nack2` from `trans1` at `t1` while `trans2`'s *possible_cycle* flag is set. Therefore, `trans2` detects a deadlock at `t2`. Then, $C(tr)$ is calculated in both transactions `trans1` and `trans2`.

Now, assume that the value of k the write back cycle cost for one log entry is 20. In `trans1`, since log data size is $L(1) = 1$ and its age is $T(1) = t1 - s1 = 12$, $C(1) = 20 \cdot 1 + 12 = 32$. Likewise, $C(2) = 70$ since $L(2) = 3$ and $T(2) = 10$.

At `t1`, `trans2` compares $C(1)$ and $C(2)$. In this case, $C(1) < C(2)$ and `trans1` will be selected as a victim. In this example, the difference of the ages is small and is assumed to affect total performance little. On the other hand, the difference of log data size or its write back overhead is relatively large and is assumed as a dominant factor. In other words, the large log data size of `trans2` outweigh its youngerness. As shown above, the new criterion $C(tr)$ can consider both the abort overhead and transaction age, and unify their effects.

C. Another Model Considering Conflict Chains

When a lot of threads are being executed in parallel, a lot of transactions may be blocked by one transaction. Now, we show another example Fig. 6 where a transaction blocks multiple other transactions. Each transaction is executed in other threads in parallel.

Here are four transactions. First, `trans3` detects a conflict with `trans4` at `t1`, and sends a nack `nack1` to `trans4`

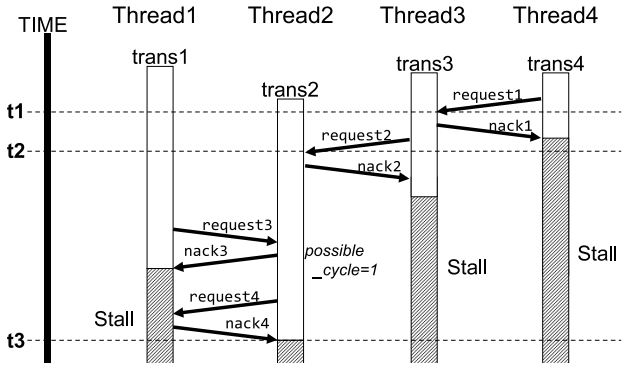

Fig. 6. An example of conflict chains.

blocking `trans4`. After that, `trans2` blocks `trans3` at `t2`. Now, as a result, `trans4` is indirectly blocked by `trans2` through `trans3`.

At `t3`, `trans1` detects a deadlock with `trans2`, so either `trans1` or `trans2` must be aborted. The number of transactions who are blocked by a transaction should affect total performance, and we propose another new criterion for selecting a victim transaction being aborted. Let $D(tr)$ the degree of conflict of the transaction tr, or how many transactions are blocked by the transaction tr. Now, $D(1) = 1$ since only `trans2` is blocked by `trans1`. On the other hand, $D(2) = 3$ since `trans1`, `trans3` and `trans4` are essentially blocked by `trans2`.

A transaction tr with larger $D(tr)$ should not be aborted but should be continued, because after restarting the transaction, it may cause many conflicts again. Hence, we propose another new criterion which can consider the conflict degree $D(tr)$ by extending Formula (1). The new criterion, the transaction tr's priority $P(tr)$ of being continued, can be defined as follows:

$$\begin{aligned} P(tr) &= w_C \cdot C(tr) + w_D \cdot D(tr) \quad (2) \\ &= w_C \cdot (k \cdot L(tr) + T(tr)) + w_D \cdot D(tr) \end{aligned}$$

bacause a transaction with small $C(tr)$ and small $D(tr)$ should be aborted.

Since the estimated overhead cycles $C(tr)$ and the conflict degree $D(tr)$ have different dimensions, each factor should have its own coefficient (or weight). The constant w_C is the weight for $C(tr)$ and w_D is for $D(tr)$. These values should be defined appropriately.

IV. Implementation

In this section, how to implement the two new LogTM models shown in the previous section will be explained.

A. Hardware Extension

For considering $L(tr)$, the number of log entries or the log data size, as a factor of the new criteria, the hardware should keep track of $L(tr)$ of each transaction. We have installed counters on the cache controllers. This is shown in Fig. 7. Each

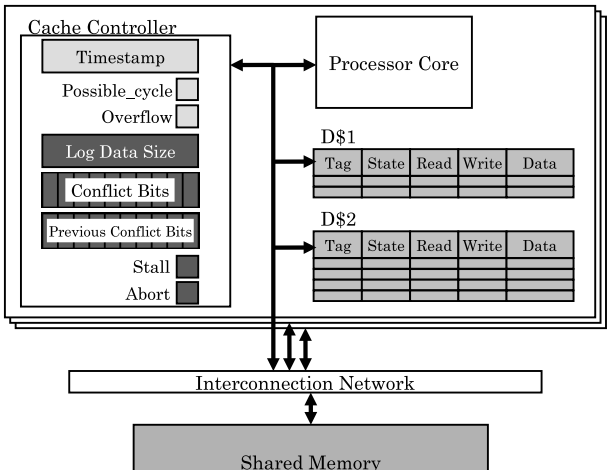

Fig. 7. Structure of Proposed LogTM.

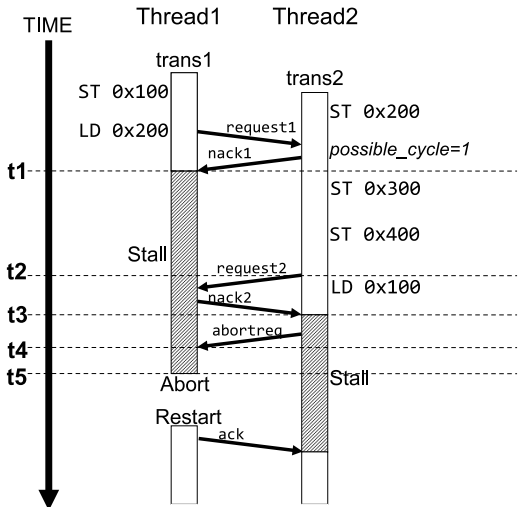

Fig. 8. Conflict Resolution of Proposed LogTM.

thread has its own conter, and a transaction in a thread can refer its counter for calculating $C(tr)$ defined by Formula (1).

To get a conflict degree $D(tr)$, a transaction must keep how many other transactions it blocks. For this purpose, we introduce **conflict bits** and have installed a register for storing it called **conflict bits register** into the each cache controller. If the number of threads running in parallel is n, the *conflict bits* has n-bit width. The m-th bit in $CB(i)$ the i-th thread's *conflict bits* is associated with the m-th thread, and the bit represents whether the m-th thread is blocked by the i-th or not. The $CB(i)$ is initialized with $0^{i-1}10^{n-i-1}$ and stored into the conflict bits Register of i-th thread. When the m-th thread is blocked by the i-th, the m-th bit in $CB(i)$ is set to 1. Then, a transaction can know which other transactions are blocking it by checking its conflict bits register. The value of $D(tr)$ is the number of 1's in the $CB(tr)$.

We have also installed a register called **previous conflict bits register** into the each cache controller. It temporarily stores the previous value of conflict bits register. When a deadlock occurs, the values of *previous conflict bits* are used for comparing conflict degrees before the deadlock.

B. Message Extension

1) Carrying Log Data Size:

For comparing log data sizes between two transactions, a transaction needs to know not only its own log data size but also the log data size of the other transaction. Therefore, information about log data size should be exchanged between threads by some messages. Hence, we have extend *nack* messages to include information about log data size.

Now, let us see how the first new model described in III-B works with an example shown in Fig. 8. Both transactions `trans1` and `trans2` issue same instruction sequences as in an example of Fig. 4. The transaction `trans1` sends its current log data size $L(1) = 2$ through the nack message `nack2` to `trans2`. At `t3`, `trans2` receives `nack2` and detects a deadlock. Then, `trans2` calculates $C(1)$ and $C(2)$ defined by Formula (1). Since $C(1) < C(2)$, aborting `trans1` will be cheaper than `trans2` and `trans1` is selected as a victim transaction.

In this case, the judging thread and the victim thread are different. Hence, `trans2` should tell `trans1` that `trans1` is selected as a victim. Therefore, a new message **abortreq**, which requests its receiver to be aborted, is introduced. In this example, `trans2` sends an *abortreq* to `trans1` and the receiver `trans1` is aborted at `t5`.

Now, with a traditional architecture of LogTM, a transaction can be aborted only when it receives *nack* message from another transaction. Therefore, we have installed a flag called **abort bit** into each cache controller for managing whether a transaction has received an *abortreq* or not. The *abort bit* is set when a transaction receives an *abortreq* message. In the example of Fig. 8, `trans1`'s abort bit is set at `t4`. After that, when `trans1` receives a *nack* reply from other transaction, it is aborted at `t5`. As a result, a transaction is aborted not as soon as it receives an *abortreq* message.

With the traditional LogTM model, the transaction whose *possible_cycle* flag is set is always selected as a victim. However, with our new model, it is not always true. Hence, a transaction, which only has rejected an access request from another transaction, may incidentally receives an *abortreq* message although the transaction is not stalled.

To avoid such a situation, we have also installed a flag called **stall bit** into each cache controller as shown in Fig. 7 for managing whether the transaction is stalled or not. The *stall bit* is set when the transaction becomes stalled and is reset when the transaction continues. Each transaction sends its current *stall bit* with *nack* message to other transactions. If the *stall bit* in the received *nack* message is not set, the receiver does not send *abortreq* back.

In the example of Fig. 8, `trans1` stalles when it receives `nack1` and sets its *stall bit* at `t1`. After a while, `trans1` sends `nack2` with its log data size and its state of *stall bit*

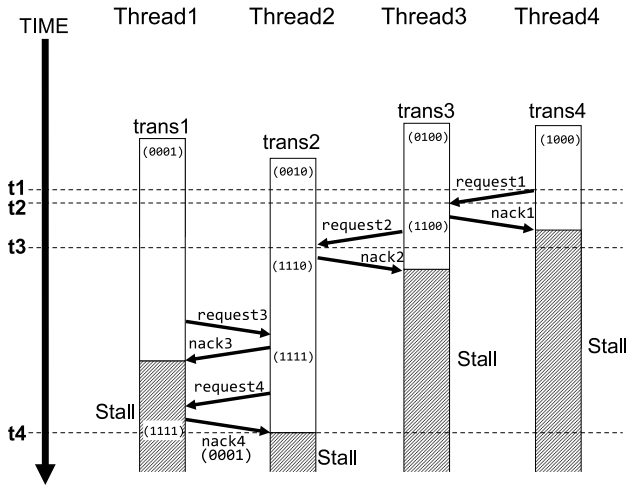

Fig. 9. Exchange conflict bits.

TABLE I
SIMULATION PARAMETERS

Processor	
number of cores	32 cores
frequency	1 GHz
issue width	1
issue order	in-order
IPC (non-memory)	1
D1 cache	
size	16 KBytes
ways	4 ways
latency	1 cycle
D2 cache	
size	4 MBytes
ways	4 ways
latency	12 cycles
Memory	
size	4 GBytes
latency	80 cycles
Interconnect Network	
topology	Hierarchical switching topology
link latency	14 cycles
LogTM	
Write back latency per log entry	20 cycles

to `trans2`. Then, `trans2` can select `trans1` as a victim because the received *stall bit* is set, and sends an *abortreq* to `trans1`.

2) Carring conflict bits:

As well as the log data size, the *conflict bits* should be carried by messages between transactions. Now, let us see how the second new model described in III-C works with an example shown in Fig. 9.

When a transaction starts, its *conflict bits* are initialized as described in IV-A. For example, the *conflict bits* of `trans1`, ..., `trans4` in Fig. 9 are initialized with 0001, 0010, 0100 and 1000 respectively.

Each transaction sends request messages with its current *conflict bits*. At `t1`, `trans4` sends `request1` to `trans3` with its *conflict bits* 1000. If a transaction which receives a request message detects a conflict with the sender, the receiver stalls the sender and logically adds the received *conflict bits* to its own *conflict bits*. In this example, `trans3` adds received 1000 to its own 0100 and gets 1100 after sending `nack1` to `trans4`. As a result, in the `trans3`'s *conflict bits*, the bit associated with `trans4` is set. This represents that `trans4` is blocked by `trans3`.

The *previous conflict bits* are carried not by request messages but by *nack* messages, because *nack* receivers should calculate transaction priorities $P(tr)$ using Formula (2) when a conflict occurs. The *previous conflict bits* carried by a *nack* message should have the value of *conflict bits* before the conflict.

At `t3` in Fig. 9, `trans2` receives `request2` with `trans3`'s current *conflict bits* 1100, sends `nack2` back to `trans3`, and gets new *conflict bits* $1110 = 0010 \vee 1100$. Now, not only the bit associated with `trans3` but also the bit associated with `trans4` is set in 1110. This means that `trans2` blocks not only `trans3` but also blocks `trans4` indirectly.

At `t4`, there occurs a deadlock between `trans1` and `trans2`, and one of them should be aborted. Hence, the *conflict degree*s of both transactions are required. Now, `trans1` has 1111 as its *conflict bits*, but they do not represent `trans1`'s *conflict degree* correctly because `trans1` does not block `trans3` and `trans4`. Hence, `trans1` should send its *previous conflict bits* `0001` to `trans2`. As a result, $D(1)$ is 1 and $D(2)$ is 3 because `trans2`'s previous conflict bits are `1110`.

If $P(1) > P(2)$ and after selecting `trans2` as a victim, the conflict bits of `trans1` should be modified. The bits associated with `trans2` and other transactions, which are blocked by `trans2`, should be cleared. Hence, `trans2` sends its correct conflict bits `1110` by *modify bits message* to `trans1` for notifying which bits should be cleard. As a result, `trans1` is released from the dependency with `trans2`.

V. PERFORMANCE EVALUATION

A. Simulation Environments

We used a full-system execution-driven functional simulator *Virtutech Simics*[6] in conjunction with customized memory models built on *Wisconsin GEMS (version 1.4)*[7], 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. An Illinois directory protocol maintains cache coherence over a high-bandwidth switched interconnect. This section discusses the performance of two proposed LogTM models. The simulation parameters are shown in TABLE I.

B. Results with SPLASH-2

We evaluated our new two LogTM models proposed in this paper. Workloads are three benchmark programs from SPLASH-2[8] suits and are executed with inputs shown in

TABLE II
SPLASH-2 BENCHMARK PROGRAMS AND THEIR INPUTS

Benchmark	Input
Barnes	512 bodies
Raytrace	small image (teapot)
Cholesky	14

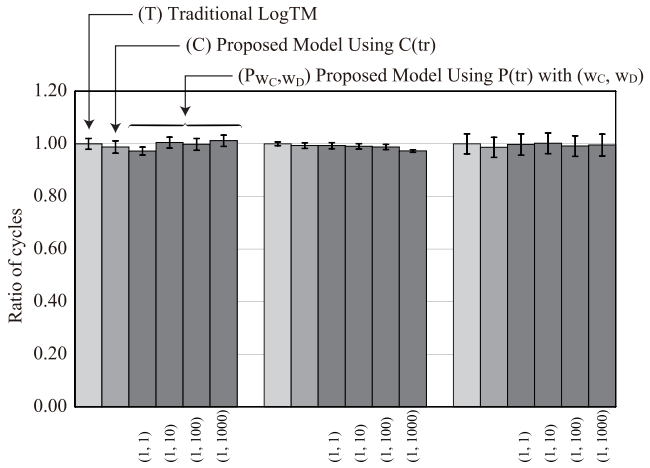

Fig. 10. Ratio of Execution Cycles

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.

For the simulation of multithreading on a full-system simulator, the variability performance[9] must be considered. Hence, we tried 10 times on each benchmarks, and measured 95% confidence interval.

The evaluation results are shown in Fig. 10. We have evaluated following six models,

- (T) Traditional LogTM
- (C) Proposed model using $C(tr)$ described in III-B
- ($P_{1,1}$) Proposed model using $P(tr)$ described in III-C with $(w_C, w_D) = (1, 1)$
- ($P_{1,10}$) with $(w_C, w_D) = (1, 10)$
- ($P_{1,100}$) with $(w_C, w_D) = (1, 100)$
- ($P_{1,1000}$) with $(w_C, w_D) = (1, 1000)$.

Fig. 10 shows the normalized execution cycles of each model and the confidence intervals are illustrated as error bars. Each bar is normalized to the number of executed cycles of Traditional LogTM (T). We have used 1 for w_C, and 1, 10, 100, and 1000 for w_D.

As we can see in Fig. 10, the proposed model (C) gets the average speed up 1.1% and the maximum speedup 1.3% with Barnes. TABLE III shows the difference of log data size in average between deadlocked transactions with the traditional LogTM model (T). As we can see, the differences of log data size with Raytrace and Cholesky are very small and they will not affect much the transaction selection for aborting. On the other hand, for Barnes, the difference considerably large.

TABLE IV shows the average of the number of entries which have been written back at aborts, and shows how much the average is reduced with the proposed model (C). As we can see, the number of entries with Raytrace or Cholesky is not reduced because there are not so much log entry differences originally with (T) as shown in TABLE III. However, the number of entries is reduced about 13% with Barnes. This result meets the purpose of the proposed new criterion $C(tr)$, and should contribute to the good performance shown in Fig. 10.

TABLE III
DIFFERENCE OF LOG DATA SIZE BETWEEN DEADLOCKED TRANSACTIONS IN (T)

	Barnes	Raytrace	Cholesky
Max.	44.7	1	2
Min.	0	0	0
Ave.	6.71	0.01	0.48

TABLE IV
NUMBER OF LOG ENTRIES WRITTEN BACK PER ABORT

	Barnes	Raytrace	Cholesky
(T)	5.58	1	1.01
(C)	4.83	1	1.01

Next, let us see the result of (P) models. There are four models with different weight parameter set (w_C, w_D). TABLE V shows the average of the number of stalls occured in transactions with the traditional model (T) and the proposed model (P). With almost all the (P) models, the number of stalls is reduced . The performance is improved 2.7% in maximum. The criterion $P(tr)$ proposed in III-C considers how many transactions are blocked by the transaction tr. The results shown in TABLE V meets the purpose of $P(tr)$. A transaction with large conflict degree $D(tr)$ blocks many other transactions and it should not be aborted because it may block many transactions again after being aborted and restarted. The proposed model (P) tends to select the transaction with small $D(tr)$ as the victim transaction which should be aborted. This should lead to the small number of stalls in total.

Now, as we can find in Fig. 10, the value of parameter set (w_C, w_D) will rather affect the performance. Hence, how to define the parameter set appropriately should be very important. Finding an algorithm for defining the parameter set is left for our future work.

VI. RELATED WORKS

Our proposition selects a victim transaction by considering the data size on the log, the ages of the transaction and the degree of conflict. Meanwhile, other various speedup techniques for LogTM have been proposed.

FASTM[10] extends the cache coherence protocol for reducing the abort overhead itself. FASTM stores the values which were modified in the transaction to the first level cache, while other values are kept in higher levels of memory hierarchy. This approach allows large transactions to recover from aborts very fast.

To improve the performance of parallel executions, Yoo et al.[11] have proposed a method which applies the concept

TABLE V
AVERAGE NUMBER OF STALLS IN TRANSACTIONS

	Barnes	Raytrace	Cholesky
(T)	2027	10938	21845
($P_{1,1}$)	2074	10820	21491
($P_{1,10}$)	2079	10777	21571
($P_{1,100}$)	2024	10628	21374
($P_{1,1000}$)	2224	10464	21283

of adaptive transaction scheduling (ATS) to LogTM. ATS can increase the performance of workloads, which lack for parallelism bacause of frequent contentions, by dynamically dispatching transactions and controlling the number of concurrent transactions using runtime feedbacks.

Titos et al.[12] have proposed a novel conflict resolution method. This is a hybrid method of the pessimistic approach which detects and resolves conflicts as soon as possible and the optimistic approach which detects and resolves conflicts when the transaction is committed.

VII. CONCLUSIONS

This paper proposed two criteria for selecting which transaction should be aborted in LogTM. The one considers the log data size of each transaction, and the other considers both the log data size and the conflict chains.

Through an evaluation with three SPLASH-2 benchmark programs, it is found that the new LogTM models with the proposed criteria improve the performance 2.7% in maximum.

Our future work is developping an algorithm of how to decide two weights w_C and w_D for $P(tr)$ criterion appropriately and dynamically. The appropriate values for these weights should be different between programs. Hence, these weights should be defined dynamically by profiling the performance and some characteristics of running programs.

REFERENCES

[1] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *Proc. of 12th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Feb. 2006, pp. 254–265.

[2] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proc. of 20th Annual International Symposium on Computer Architecture*. ACM, May. 1993, pp. 289–300.

[3] P. Sweazey and A. J. Smith, "A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus," in *Proc. of 13th Annual Int'l. Symp. on Computer Architecture (ISCA'86)*, 1986, pp. 414–423.

[4] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, vol. c-27, no. 12, pp. 1112–1118, Dec. 1978.

[5] R. Rajwar and J. R. Goodman, "Transactional Lock-Free Execution of Lock-Based Programs," in *Proc of 10th Symposium on Architectural Support for Programming Languages and Operating Systems*. ACM, Oct. 2002, pp. 5–17.

[6] 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.

[7] M. M. 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," *AMC SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[8] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc of 22nd Annual Int'l. Symp. on Computer Architecture (ISCA'95)*, 1995, pp. 24–36.

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

[10] M. Lupon, G. Magklis, and A. González, "FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery," in *Proc. of 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, Sep. 2009, pp. 293–302.

[11] R. M. Yoo and H.-H. S. Lee, "Adaptive Transaction Scheduling for Transactional Memory Systems," in *Proc. of 20th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, Jun. 2008, pp. 169–178.

[12] R. Titos, M. E. Acacio, and J. M. García, "Speculation-based conflict resolution in hardware transactional memory," in *Proc. Int'l. Symp. on Parallel Distributed Processing (IPDPS 2009)*, May. 2009, pp. 1–12.