Vol.2010-ARC-190 No.4
2010/8/3

ログエントリ数を考慮した LogTM のアボート対象選択手法とその評価

浅井宏樹[†1] 津邑公暁[†1] 松尾啓志[†1]

マルチコア環境における並列プログラミングでは一般的にロックを用いて同期する．しかしロックの問題点として，デッドロックの発生や並列性の低下がある．そこでロックを用いない同期制御機構として LogTM が提案されている．しかし LogTM の問題点としてアボートコストが高いことが挙げられる．アボートコストとはログに保存された値をメモリに書き戻すコストである．LogTM はアボートの対象をトランザクションの開始時刻のみで決定しているため，アボートコストが高いトランザクションをアボートしてしまう可能性がある．そこで本稿ではログエントリ数を考慮してアボート対象を動的に選択する手法を提案する．これによりアボートコストの高いトランザクションのアボートを防ぐことができ，結果としてプログラム全体の実行時間を削減することができる．提案手法の有効性を検証するため，既存の LogTM を拡張し，SPLASH-2 ベンチマークを用いてシミュレーション評価を行った．その結果，既存の LogTM に比べて最大で約 1.3%，平均で約 1.1%の実行サイクル数が削減できた．

Selection of Transaction to be Aborted based on Log Data Size in LogTM

Hiroki Asai,[†1] Tomoaki Tsumura[†1]
and Hiroshi Matsuo[†1]

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. 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 method for selecting which transaction should be aborted taking account of data size in each logs. The result of the experiment with SPLASH-2 benchmark suite programs shows that the proposed methods improve the performance 1.3% in maximum and 1.1% in average.

1. はじめに

今日まで，プログラムの実行を高速化する手法として，スーパスケーラなどの命令レベル並列性 (Instruction-Level Parallelism: ILP) に着目したものが研究されてきた．しかしながら ILP には限界があり，命令レベルの並列化を行うだけではプロセッサの性能向上が頭打ちになりつつある．また，半導体技術の向上によって集積回路の微細化が進み，単一コアの性能向上が図られてきた．しかしながら，消費電力の増加や配線遅延の相対的な増大の問題から，単一コアの性能向上による高速化は難しくなってきている．

この流れを受け，単一チップ上に複数のプロセッサコアを集積したマルチコア・プロセッサが広く普及してきている． マルチコア・プロセッサでは，今までひとつのコアが担っていた仕事を複数のプロセッサ・コアで分担することで，単一コアでの実行よりもスループットを向上させることができる．例えば，スレッドレベル並列性 (Thread-Level Parallelism: TLP) を利用して並列に実行することで，プログラム全体の実行時間の短縮が期待できる．

このようなマルチコア環境における並列プログラミングでは複数のプロセッサ・コア間で単一アドレス空間を共有した共有メモリ型並列プログラミングが一般的である．そのような共有メモリ型並列プログラミングでは，共有リソースに対して同期をとる必要があり，同期制御機構として一般的にロックが多く用いられている．

しかし，ロックを用いた場合の問題点として，デッドロックの発生やロックのオーバヘッド増大による並列性の低下がある．さらにプログラマはロックの粒度を考慮したプログラムを構築する必要があり，ロックの利用が困難である一因となっている．そこで，ロックを用いない同期制御機構として LogTM[1)] が提案されている．

LogTM はトランザクショナル・メモリの一種である．LogTM ではトランザクションを投機的に実行することで，トランザクションのシリアライザビリティとアトミシティを維持する．LogTM の問題点としてアボートコストが高いことが挙げられる．LogTM ではアボート時にログにバックアップされた値を全てキャッシュまたは主記憶に書き戻すことが必要である．つまりログに退避されたエントリの数に比例してアボート時の書き戻しコストが大きくなる．しかし， LogTM はアボートの対象をトランザクションの開始時刻のみに

†1 名古屋工業大学
Nagoya Institute of Technology

よって決定しているため，アボートコストが高いトランザクションをアボートしてしまう場合があり，競合が頻繁に起きるようなプログラムでは性能が低下する可能性がある．

そこで本稿では，各ログが持つエントリ数を考慮してアボート対象となるトランザクションを動的に選択する手法を提案する．これによりアボートコストの高いトランザクションのアボートを防ぐことができ，結果としてプログラム全体の実行サイクル数が削減できる．

以降，2 章では本研究の背景としてトランザクショナル・メモリ及び LogTM の概要とその実装を説明する．3 章では LogTM の問題点及び本稿で提案するモデルについて説明し，4 章でその実装方法を説明する．5 章では本手法を評価し，6 章で結論を述べる．

2. 研究背景

2.1 トランザクショナル・メモリ

マルチコア・プロセッサにおける並列プログラミングでは，共有メモリ型の手法が多く用いられる．共有メモリ型のプログラムでは複数のプロセッサ・コアが単一アドレス空間を共有するため，同一のアドレスに対してアクセスするには同期をとる必要がある．しかし，ロックを用いた同期ではデッドロックが発生する可能性がある．また，並列に実行するスレッド数が増加すると，ロックの獲得・解放に伴うオーバヘッドも増加し，性能が低下する可能性もある．さらに，プログラムごとに最適なロックの粒度を設定するのは難しい．例えば粗粒度ロックを用いる場合，プログラムの構築は容易であるが並列性は損なわれる．対して細粒度ロックを用いる場合，並列性は向上するがプログラムの設計が難しい．

一方でロックを用いない同期制御機構として**トランザクショナル・メモリ (Transactional Memory: TM)**[2] が提案されている．TM はデータベースの一貫性を保つために用いられるトランザクション処理をメモリアクセスに適用した手法である．TM では，クリティカルセクションを含む一連の機械語命令列をトランザクションと呼ぶ．このときトランザクションは以下の性質を満たす．

シリアライザビリティ (直列可能性):
並行して実行されたトランザクションの実行結果は，当該トランザクションを直列に実行した場合と同じである．

アトミシティ (原子性):
トランザクションの操作は完全に実行されるか全く実行されないかのいずれかである．

以上の性質を保証するために，TM はあるトランザクションが他のトランザクションと同じメモリアドレスにアクセスするかどうかを検査する．つまり，他のトランザクションからアクセスされたメモリアドレスと，自身のトランザクション内でアクセスしたメモリアドレスが同一であった場合，そのことを**競合**として検出する．競合が発生した場合，一方のトランザクションの実行を停止し，それまでの結果を全て破棄する**アボート**を行う．アボートされたトランザクションはそのトランザクションの開始時点まで戻り，再実行する．一方でトランザクションの終了までに競合が検出されなかった場合，トランザクション内で実行された結果を全てメモリに反映させる．これを**コミット**という．

このように TM を用いることで競合が発生しない限りクリティカルセクションを並列に実行することができる．これによりロックよりもプログラムの並列性が上がるため，速度性能が向上する．また，細粒度ロックを用いる場合に比べて，プログラマはロックが必要な共有リソースを見極めたり，デッドロックを避けるような複雑なプログラムを設計する必要がないため，容易に並列プログラムを構築することができる．

2.2 LogTM

LogTM はハードウェア TM の一種である．LogTM のハードウェア構成を**図 1** に示す．LogTM ではプロセッサ・コアごとに 1 次データキャッシュ，2 次データキャッシュ，キャッシュコントローラを 1 つずつ持つ．また，主記憶は全てのコアによって共有されている．

2.2.1 データのバージョン管理

トランザクションの投機的実行では実行結果が破棄される可能性があるので，トランザクションはアクセスするデータの古いバージョンを保持し管理する必要がある．そこで LogTM では**ログ**と呼ばれる仮想メモリ領域をスレッドごとに割り当て，トランザクション内のストア命令によって上書きされる前の値とそのアドレスをログにバックアップする．一方でストア命令の結果はストア対象のメモリアドレスに書き込まれる．

投機的実行が成功した場合はコミット操作を行うが，全ての更新は既にメモリに反映されているため，メモリアクセスを行う必要がない．したがって LogTM はログの内容を破棄する操作のみを行う．一方で投機的実行が失敗した場合はアボート操作を行う．このときメモリ状態を開始時点まで戻す必要があるため，ログに保存された全ての値を元のメモリアドレスに書き戻す．

このようにコミットではメモリアクセスが全く行われない．反対にアボートではログに保存された値を全てキャッシュまたは主記憶に書き戻すため，ログエントリ数に比例してメモリアクセスコストが増大する．したがってアボートよりもコミットの方が高速に処理できる．コミットはトランザクション終了時に必ず行われる操作であるため，LogTM ではアボートよりもコミットを高速化することでプログラム全体の実行速度を向上させている．

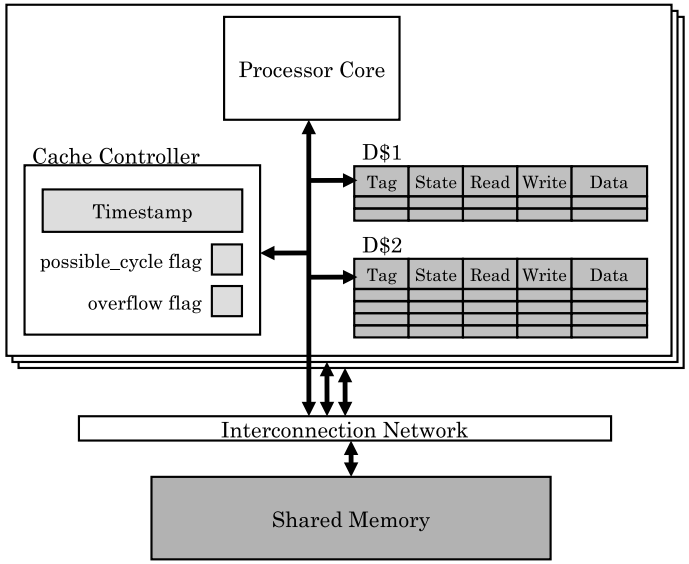

図 1 既存 LogTM の構成

2.2.2 競合検出

競合の検出を行うために，メモリアドレスがトランザクション内部で投機的にアクセスされたかどうかを管理する必要がある．そのため LogTM ではキャッシュライン上に新しく **read ビット**及び **write ビット**を追加している．read ビットと wtite ビットはトランザクション内でそれぞれ該当キャッシュラインに対するリードアクセスまたはライトアクセスが発生した場合にセットされ，トランザクションのコミット及びアボート時にリセットされる．

LogTM では一貫性モデルにディレクトリベース[3] の Illinois プロトコル[4] が採用されており，他のトランザクションとの競合を検査するために，LogTM はこれらのキャッシュ・コヒーレンス・プロトコルを拡張している．通常のプロトコルでは，一貫性を保つためのコヒーレンス・リクエストを送信する．このとき，LogTM はリクエストによってキャッシュラインの状態を変更する前に read ビット及び write ビットを参照する．これによりトランザクション内でアクセスしようとしたメモリアドレスが他のトランザクションによって既にアクセスされているかどうかを検出できる．具体的には以下の 3 パターンのアクセスが行われた場合に競合として検出する．

read after write: write ビットがセットされているアドレスに対するリードアクセス
write after read: read ビットがセットされているアドレスに対するライトアクセス
write after write: write ビットがセットされているアドレスに対するライトアクセス

競合が発生しなかった場合，リクエストを受信したトランザクションは送信者に対して **ack** を返信する．一方で競合が検出された場合は **nack** が返信される．nack を受信したトランザクションは競合が発生したことを知り，競合したトランザクションが終了するまで一時的に実行を停止する．これを**ストール**という．ストールしたトランザクションは同じアドレスに対するリクエストを送信し続ける．競合したトランザクションが終了した場合，そのトランザクションから ack が返信されるため，ストールしたトランザクションは相手の終了を検知できる．

しかし，競合によりストールしたトランザクションが複数発生するとそれらがデッドロック状態に陥る危険性がある． **図 2** では，実際に複数のストールによってデッドロックが発生してしまう動作例を示している．Thread1 と Thread2 はそれぞれスレッドを示し，それら 2 つのスレッドはそれぞれトランザクション trans1 と trans2 を投機的に実行しているとする．

まず，trans1 が実行を開始した後に trans2 が実行を開始する．次に trans1 で `ST 0x100` を実行し，その後に trans2 で `ST 0x200` が実行される．さらにその後 trans1 で 0x200 番地に対するロードが実行されると，trans1 はリクエストを送信する (t1)．リクエストを受信した trans2 は競合したことを検知するため nack1 を送信し，受信した trans1 はストールする (t3)．

その後，trans2 で `ST 0x300`，`ST 0x400` の実行を経て 0x100 番地に対するロードが実行されると (t4)，trans2 は trans1 と競合してストールする (t5)．このようにお互いにストールしてしまうとデッドロックに陥ってしまう可能性がある．LogTM は TLR's distributed timestamp method[5] で用いられる **possible_cycle flag** を利用してこのようなデッドロックを検知する．図 1 にあるように，各コアは possible_cycle flag をひとつ持つため，それぞれのスレッドごとで固有の値を管理することができる．possible_cycle flag はより早く開始したトランザクションに nack を送信したときにセットされる．そして，possible_cycle flag がセットされているトランザクションが，自身よりも早く開始したトランザクションから nack を受信した場合，デッドロックが発生したとみなしてアボートする．

この結果開始時刻のより遅いトランザクションがアボートの対象として選択される．これは，開始時刻が早いトランザクションはより多くのメモリアクセスを行っている可能性が高

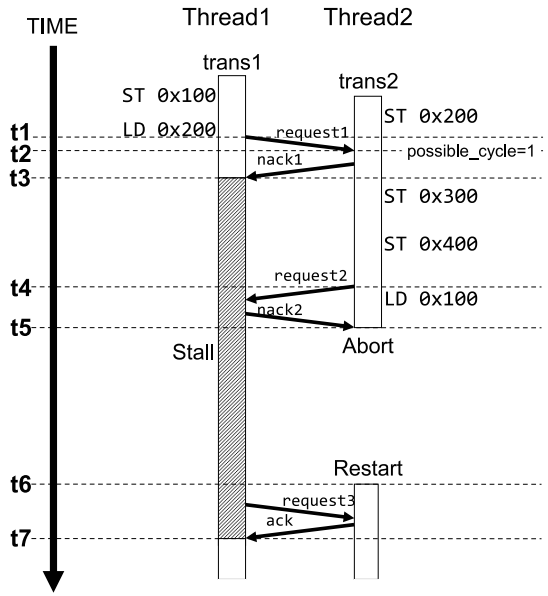

図 2 既存の LogTM でのアボート対象の選択

く，競合の頻発を防ぐために早くコミットすることが望ましいからである．

図 2 では，t2 で trans2 が trans1 へ nack1 を送信したため，trans2 の possible_cycle flag がセットされる．その後，t5 の時点で trans2 は nack2 を受信するため，デッドロックが発生したことを検知してアボートする．アボートした trans2 は開始時点まで戻り，トランザクションを再実行する (t6)．また，trans2 がアボートしたことにより trans1 は 0x200 番地にアクセスできるようになるため，trans1 のストール状態が解消される (t7)．

また，トランザクションの実行中にキャッシュがオーバフローした場合，そのトランザクションを実行しているスレッドは **overflow flag** をセットする．図 1 にあるように，各コアは overflow flag をひとつ持つため，それぞれのスレッドで管理することができる．

3. 提　　案

本章では，LogTM の問題点と，それを解決する提案手法について説明する．

3.1 LogTM の問題点

LogTM の問題点としてアボートコストが高いことが挙げられる．アボートコストとは，アボート時にメモリ状態をトランザクション開始時点の状態まで戻すために，ログに保存された値をキャッシュまたはメモリに書き戻すコストである．このコストはログに保存されたエントリ数に比例する．ここでログに保存されたエントリの総数を以降では**ログエントリ数**と呼ぶ．ログエントリ数はトランザクション内でストア命令が発行されるたびに増加する．つまりログエントリ数が多ければ，それに応じたアクセスコストがアボート時に必要となる．また，競合が頻繁に発生するようなプログラムではアボートが頻繁に発生する．したがってそのようなプログラムが実行されるとアボートコストが増大し，結果として速度性能の低下につながる．

しかし，既存の LogTM はアボートコストがどれだけかかるかを全く考慮せずにトランザクションをアボートさせるため，アボートコストの大きいトランザクションがアボートされる可能性がある．これは 2.2.2 項で説明したようにアボート対象の選択にタイムスタンプが用いられるためである．

2.2.2 項の図 2 で説明した例では，トランザクション trans2 がアボートされている．このとき，trans2 は 3 箇所のメモリアドレスに対する書き戻しが必要である．一方で trans1 で実行されたストア命令は 1 回であるため，trans1 がアボートする場合はメモリへの書き戻しが 1 箇所で済む．したがって，このような場合は trans2 よりも trans1 をアボートした方がアボートコストは少ない．

3.2 ログエントリ数を考慮したアボート対象の選択

前節で述べたように，既存の LogTM ではログからの書き戻しコストが大きなトランザクションをアボートさせてしまう可能性がある．したがって書き戻しコストの小さいトランザクションをアボートさせることで，アボートコスト自体は削減できると考えられる．しかしながら，ログエントリ数の小さなトランザクションをアボート対象として選択することが必ずしも正しいとは限らない．なぜなら，2.2.2 項で述べたように早く開始したトランザクション，すなわち長い間実行しているトランザクションは早くコミットされることが望ましいからである．

そこで，アボートによってどれだけのサイクル数がオーバヘッドとして発生するかを考慮してアボート対象を選択する手法を提案する．このコストを $C(tr)$ と定義する．$C(tr)$ は，アボート時にログから値を書き戻すコストと，トランザクションがロールバックしてから再びアボートの発生時点まで戻るまでにかかるサイクル数の和として定義する．このコストを

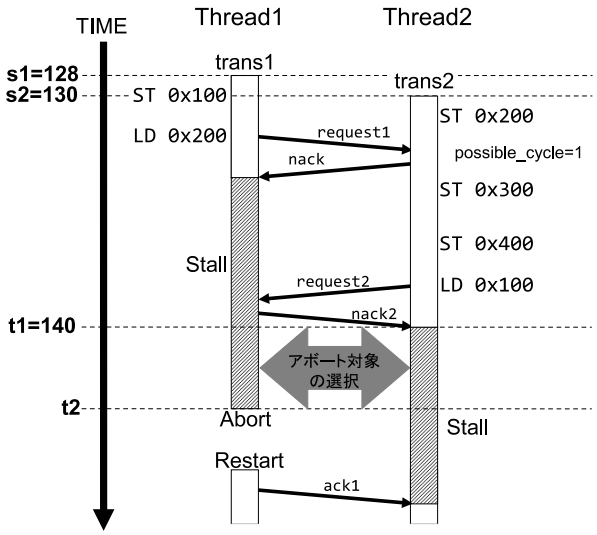

図 3 既存の LogTM でのアボート対象の選択

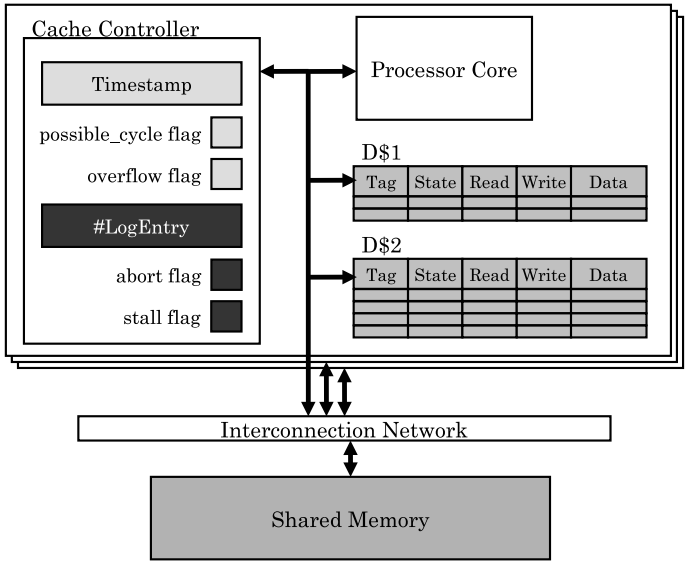

図 4 拡張した LogTM の構成

比較することで，アボートコストとトランザクション実行時間の両方を考慮することができる．

あるトランザクション tr に対し，そのトランザクションが持つログエントリ数を $L(tr)$，アボート発生時点までの実行時間を $T(tr)$ とする．このとき $C(tr)$ は以下のように定義できる．

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

なお k はひとつのログエントリを書き戻すときに必要なオーバヘッドである．また，k と $T(tr)$ の単位はサイクル数であるため，$C(tr)$ もサイクル数単位で表される．

提案手法を用いてアボート対象を選択する動作モデルを**図 3** に示す．図 3 では，t1 の時点で trans2 が trans1 とデッドロックに陥ったことを検知する．ここで trans1 と trans2 のそれぞれの $C(tr)$ を比較することで，アボートの対象を選択することができる (t2)．

ここで k の値を 20 cycles と仮定する．trans1 が持つログエントリ数は $L(1) = 1$ であり，アボート発生時点までの実行時間は $T(1) = t1 - s1 = 12$ であるので，$C(1) = 20 \cdot 1 + 12 = 32$ となる．同様に trans2 は $L(2) = 3$，$T(2) = 10$ であるため，$C(2) = 70$ となる．

t1 で trans2 は $C(1)$ と $C(2)$ を比較する．このとき $C(1) < C(2)$ であるため，trans1 がアボート対象として選択される．この例では，アボート発生時点までの実行時間の差が僅かであるため，全体の性能に対する影響は少ないと考えられる．しかし，ログエントリ数の差は比較的大きく，その影響は大きい．つまり，アボート発生時点までの実行時間に比べてログエントリ数が優先されたといえる．以上のように，$C(tr)$ はアボート時のオーバヘッドとトランザクションの開始時刻の両方をあわせた効果を考慮することができる．

4. 実　　装

本章では前章で提案した手法の実装方法について説明する．

4.1 ログエントリ数の通信

提案手法ではアボート対象の選択時にログエントリ数を比較する．そのため，まず実行中のトランザクションが保有するログの総エントリ数を把握する必要がある．そこで LogTM にログエントリ数のカウンタ (**#LogEntry**) を設ける．拡張した LogTM のハードウェア構成を**図 4** に示す．#LogEntry はそれぞれのキャッシュコントローラ内に 1 つずつ設ける．

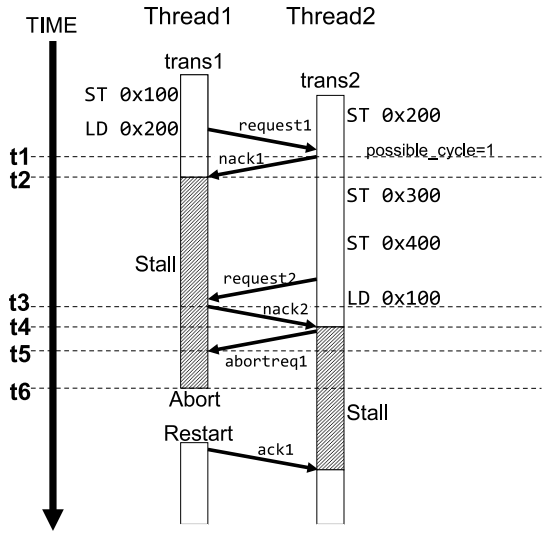

図 5 拡張した LogTM の動作例

そのため，トランザクションを実行する各スレッドはそれぞれ固有のログエントリ数を管理することができる．

次に，デッドロックした相手へ自身のログエントリ数を通信する必要がある．そこで，プロセッサ・コア間の通信メッセージを拡張する．既存の LogTM では，通信はメッセージキューによって構成されており，送受信されるデータはすべてメッセージキューにバッファリングされる．また，トランザクションの開始時刻はメッセージに付加されているため，トランザクションはお互いの開始時刻を知ることができる．そこで，トランザクションの開始時刻と同様にメッセージにログエントリ数を付加する．このメッセージを受信したトランザクションは送信元のログエントリ数を知ることができる．

ここで提案手法の動作モデルについて説明する．アボート対象を選択する前に具体的にどのようなメッセージ交換を行うかを**図 5** に示す．

t1 では，trans2 が trans1 へ nack1 を送信する．このとき trans2 は自身のログエントリ数を nack1 に付加する．しかし，t2 の時点では trans1 と trans2 の間にデッドロックが発生しないため，trans2 のログエントリ数は trans1 で利用されない．

その後，t3 で trans1 は自身のログエントリ数を付加した nack2 を trans2 へ送信する．trans2 には possible_cycle flag がセットされており，かつ trans2 は自身よりも開始時刻の早い trans1 から nack を受信するため，trans2 はデッドロックを検知する (t4)．このとき trans2 は nack2 から trans1 のログエントリ数 を知ることができる．したがって trans1 と trans2 の $C(tr)$ を計算することができる．

4.2 アボート対象の選択

$C(tr)$ の比較により nack の送信者がアボート対象として選択された場合，その旨を送信者に伝える必要がある．そこで，新たに **abortreq** という通信メッセージを定義する．abortreq は nack 送信者がアボート対象として選択された場合に送信され，abortreq を受信したトランザクションはアボートされる．

図 5 より，t4 の時点で trans2 は trans1 と trans2 の $C(tr)$ を計算するため，それぞれの $C(tr)$ 比較することによりアボート対象を選択することができる．このとき得られる結果はそれぞれ $C(1) = 32$，$C(2) = 70$ となり，それらを比較すると $C(1) < C(2)$ であるため，trans1 がアボート対象として選択される．

ここで trans1 がアボートされるためには，trans1 がアボート対象として選択されたことを trans2 から trans1 に対して通知する必要がある．したがって，trans2 は trans1 に対して abortreq1 を送信する (t4)．これにより abortreq1 を受信した trans1 はアボートすることができる (t5)．

しかし，トランザクションは abortreq を受信してもすぐにアボートを開始することはできない．なぜなら，既存の LogTM ではトランザクションは nack を受信した瞬間にしかアボートすることができないからである．したがって nack を受信する前に abortreq を受信した場合は nack を受信するまでアボートの開始を待つ必要がある．そこで，トランザクションが abortreq を受信したことを管理するために，**abort flag** を設ける．図 4 にあるように，abort flag を各キャッシュコントローラに設けることで，各トランザクションは固有の abort flag を管理することができる．abort flag はトランザクションが abortreq を受信したときにセットされる．t5 では，trans1 が abortreq1 を受信したときに abort flag がセットされる．そして， abort flag がセットされたトランザクションが過去に送信したリクエストに対するレスポンスメッセージを受信した時，そのトランザクションはアボートを開始する．したがって，trans1 は t6 で実際にアボートされる．

また，LogTM では nack を受信した，すなわちストールしたトランザクションのみアボートできるが，提案手法ではストールしていないトランザクションを abortreq によってア

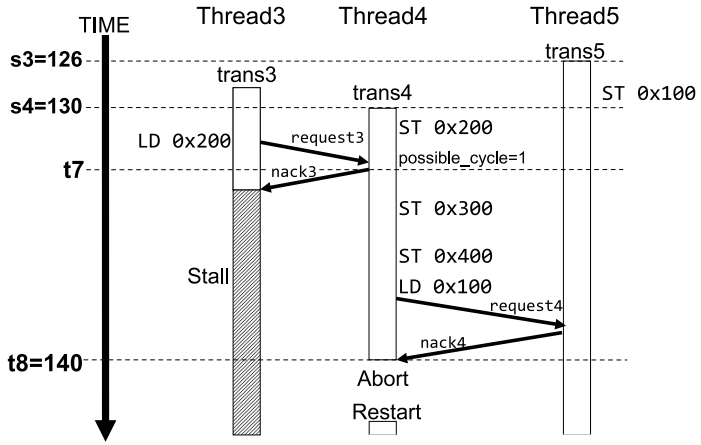

図 6 abortreq を送信しない場合の動作例

ボートしてしまう可能性がある．例えば**図 6** のような例を考える．t7 の時点で trans4 は trans3 へ nack3 を送信する．このとき trans3 の開始時刻は trans4 よりも早いため，trans4 の possible_cycle flag がセットされる．その後，nack4 は自身よりも早く開始した trans5 から nack を受信するため，デッドロック状態に陥ったと検知してしまう．このとき k の値を 20 cycles と仮定すると，trans4 は $L(4) = 3$，$T(4) = t8 - s4 = 10$ であるので，$C(4) = 20 \cdot 3 + 10 = 70$ となる．同様に trans5 は $L(5) = 1$，$T(5) = 14$ であるため，$C(5) = 34$ となる．それらを比較すると $C(4) > C(5)$ であるため，trans5 がアボート対象として選択される．しかし，trans5 はストールしていないため，アボート対象として選択されてはならない．このような場合は trans4 がアボートされる必要がある．

このようなストールしていないトランザクションに対して abortreq を送信する状況を避けるために，競合した相手のストール状態を知る必要がある．そこで，各トランザクション内でそれぞれのストール状態を管理するために，possible_cycle flag や abort flag と同様に **stall flag** を各キャッシュコントローラ内に設ける (図 5)．stall flag はトランザクションがストール状態になったときにセットされ，ストールから開放された時にクリアされる．また，各トランザクションはそれぞれの stall flag の状態を nack に付加して送信する．もし送信元の stall flag がセットされていないならば，受信者は abortreq を返信することができない．図 5 では，trans1 は nack1 を受信した時点でストールするため，stall flag がセットされる (t1)．その後，trans1 はログエントリ数と同様に stall flag を nack2 に付加し，trans2 に送信する (t3)．このとき trans2 は trans1 がストールしていることを知るため，trans1 へ abortreq1 を送信することができる (t4)．

表 1 シミュレーションパラメータ

Processor	32 cores
周波数	1 GHz
	single-issue
	in-order
	non-memoryIPC=1
D1 cache	16 KBytes
ways	4 ways
latency	1 cycle
D2 cache	4 MBytes
ways	4 ways
latency	12 cycles
Memory	4 GBytes
latency	80 cycles
Interconnect Network	Hierarchical switching topology
link latency	14 cycles
LogTM	
Write back latency per log entry	20 cycles

5. 評　　価

5.1 評価環境

前章までで述べた拡張を既存の LogTM に実装し，シミュレーションによる評価を行った．評価にはフルシステムシミュレータである Virtutech Simics[6] と GEMS[7] を用いた．想定するシステムの環境を**表 1** に示す．シミュレーションにおいて Simics は機能シミュレーションを担当する．Simics ではシミュレーションのターゲットモデルとして SPARC V9 ISA を選択し，OS には Solaris10 を用いた．さらに各プロセッサ・コアは単命令発行のインオーダ実行を行う．また GEMS ではメモリシステムの詳細なタイミングシミュレーションを行う．

評価対象のプログラムには共有メモリ型並列計算用のベンチマークプログラムである SPLASH-2[8] ベンチマークプログラムの内 Barnes，Raytrace 及び Cholesky を用いた．それぞれの入力を**表 2** に示す．また，各プログラムは 31 個のスレッドを用いて並列化した．これは想定するシステムモデルは 32 個のコアを持つが，その内のひとつはデフォルトコア

表 2 SPLASH-2 ベンチマークプログラムとその入力

Barnes	512 bodies
Raytrace	small image (teapot)
Cholesky	14

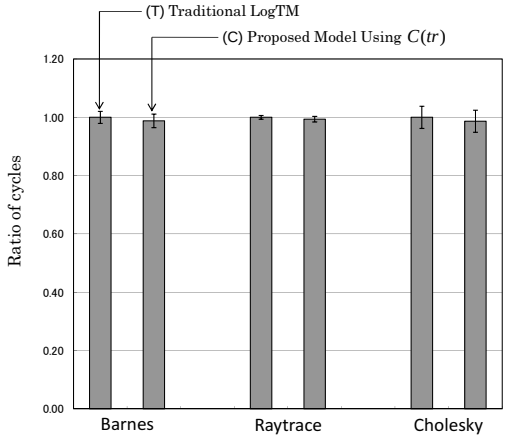

図 7 評価結果

であり，ユーザプログラムに用いることができないためである．

また，フルシステムシミュレータ上でマルチスレッドを用いた動作のシミュレーションを行うには性能のばらつきを考慮しなければならない[9]．したがって各評価対象につき試行を 10 回繰り返し，得られた結果から 95%の信頼区間を求めた．

5.2 評価結果

評価結果を**図 7** に示す．グラフは，左から順に

(T) 既存の LogTM

(C) 3.2 節で提案した $C(tr)$ を用いるモデル

が要した実行サイクル数を表している．なお，各サイクル数は既存モデルを 1 として正規化した．また，信頼区間をエラーバーで表した．

図 7 より，モデル (C) は (T) に比べて平均で約 1.1%の実行サイクル数が削減でき，Barnes では最大で約 1.3%の実行サイクル数が削減できた．**表 3** は各ベンチマークをモデル (T) を用いて実行したとき，実行中にデッドロックに陥ったふたつのトランザクションが持つログエントリ数の差分を計測し，それぞれの最大値，最小値及び平均値を算出した結果を示している．表 3 より，Raytrace と Cholesky ではログエントリ数の差がほとんどないため，ログエントリ数を考慮してもあまり効果が得られない．一方で，Barnes ではログエントリ数の差分が大きい．したがってログエントリ数によってアボート対象を選択する手法の効果が得られたと考えられる．

表 3 既存の LogTM で計測された差分

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

表 4 ログエントリの平均書き戻し数

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

また，**表 4** はモデル (T) と (C) でアボート時に書き戻されたログエントリ数の平均を表している．表 4 より，Raytrace と Cholesky では (C) を用いても書き戻されたエントリ数が全く削減されていないことがわかる．これは表 3 でも示されているように，ログエントリ数 の差分がほとんどないためである．しかし，Barnes では約 13%の書き戻しエントリ数を削減することができた．これにより $C(tr)$ を用いてアボート対象を選択する手法が有効に働いたことがわかる．

6. おわりに

本稿では，既存のハードウェア TM である LogTM を拡張し，ログエントリ数を考慮したアボート対象の選択手法を提案した．拡張した LogTM では，アボート対象の選択の条件として新しくログエントリ数を追加した．これにより，アボート時のメモリへの書き戻しコストを考慮することができ，アボートによる影響が少ないトランザクションを選択することができる．

提案手法の有効性を確認するため，SPLASH-2 ベンチマークのうち，Barnes，Raytrace 及び Cholesky を用いた評価を行った．その結果，既存の LogTM に比べて実行サイクル数を最大で約 1.3%，平均で約 1.1%削減することができた．

今後の課題として，評価の対象を増やすことで提案手法がどのようなプログラムに有効であるかを調査することや，より精密な評価を行うことで実行中のトランザクションの挙動がどのように他のトランザクションに影響を及ぼすかを評価することが挙げられる．

Vol.2010-ARC-190 No.4
2010/8/3

参 考 文 献

1) Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D. and Wood, D.A.: LogTM: Log-based Transactional Memory, *Proc. of 12th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, pp.254–265 (2006).
2) Herlihy, M. and Moss, J. E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. of 20th Annual International Symposium on Computer Architecture*, ACM, pp.289–300 (1993).
3) Sweazey, P. and Smith, A.J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. of 13th Annual International Symposium on Computer Architecture (ISCA'86)*, IEEE Computer Society Press, pp.414–423 (1986).
4) Censier, L.M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers*, pp.1112–1118 (1978).
5) Rajwar, R. and Goodman, J.R.: Transactional Lock-Free Execution of Lock-Based Programs, *Proc of 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM, pp.5–17 (2002).
6) Magnusson, P.S. and et al: Simics: A Full System Simulation Platform, *Computer*, Vol.35, pp.50–58 (2002).
7) Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., E.Moore, K., Hill, M.D. and Wood., D.A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, *Computer Architecture News*, ACM, pp.254–265 (2005).
8) Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A.: The SPLASH-2 programs: characterization and methodological considerations, *Proc of 22nd Annual Int'l Symposium on Computer Architecture (ISCA'95)*, ACM, pp.24–36 (1995).
9) Alameldeen, A.R. and Wood, D.A.: Variability in Architectural Simulations of Multi-Threaded Workloads, *Proc. of 9th Int'l Symposium on High-Performance Computer Architecture (HPCA'03)*, IEEE Computer Society, pp.7–18 (2003).