競合の再発抑制による LogTM の高速化手法

江藤正通[†1] 堀場匠一朗[†1] 浅井宏樹[†1,*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メモリアクセスの調停がとられている．しかしロックを使用する場合，デッドロックの発生や並列性の低下などの問題がある．そこでロックを用いない並行性制御機構として LogTM が提案されている．LogTM では possible_cycle というフラグを用いて競合を解決する．しかし，この競合解決手法では starving writer が発生し，長期に渡るストールや競合の繰り返しにより性能が大きく低下してしまう．そこで本稿では，starving writer の解決手法を提案する．提案手法の有効性を検証するためにシミュレーションによる評価を行った結果，既存の LogTM に比べて最大で 18.7%，平均で 6.6%の性能向上が得られた．

A Speed-Up Technique for LogTM by Preventing Recurrence of Conflicts

Masamichi Eto,[†1] Shoichiro Horiba,[†1] Hiroki Asai,[†1]
Tomoaki Tsumura[†1] and Hiroshi Matsuo[†1]

Lock-based thread 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. To solve conflicts in LogTM, a flag called 'possible_cycle' is used. However, the performance can be decrease because of some conflict patterns. This paper proposes a method for dynamically changing the priority of threads to solve the conflict patterns. Our model reduces the number of aborts and reccurence of aborts. The result of the experiment shows that proposing method improve the performance 18.7% in maximum and 6.6% in average.

1. はじめに

現在一般的となったマルチコア環境では，複数のプロセッサ・コア間で単一アドレス空間が共有されるプログラミングモデルが多く用いられる．このようなプログラミングモデルでは，共有リソースに対する競合を解決する必要があり，その排他制御機構としてロックが用いられてきた．しかしロックを用いた場合，デッドロックの発生やロック操作のオーバヘッド増大に伴う並列性の低下などの問題が起こりうる．さらに，各プログラムに最適なロックの粒度を設定することは困難であるため，プログラマにとって必ずしも利用しやすいものではない．そこで，ロックを用いない並行性制御機構としてトランザクショナル・メモリ[1] が提案されている．

トランザクショナル・メモリのハードウェアによる一実装である LogTM[2] では，クリティカルセクションを含む一連の命令列として定義されるトランザクションが投機的に実行される．そして，処理のアトミシティを保つために，あるトランザクションで発生したメモリアクセスが他のトランザクションで発生したメモリアクセスと競合するか検査される．競合が検出された場合トランザクションをストールさせるが，複数のトランザクションにおいてストールが発生するとデッドロックとなる可能性があるため，トランザクションをアボートさせる．

この際 LogTM では，possible_cycle と呼ばれるフラグを用いてアボート対象を選択しているが，この方法では starving writer と呼ばれるトランザクションが発生するような競合パターンにおいて性能が大きく低下してしまう．したがって，本稿では starving writer の発生に着目し，これを抑制する手法を提案する．また，starving writer を解決しつつ，競合の繰り返しを抑制する手法も提案する．

2. 背景

本章では，**トランザクショナル・メモリ**（**Transactional Memory**，以下 **TM**）の基本概念および TM のハードウェア実装（HTM）の 1 つである LogTM について説明する．

2.1 TM の基本概念

TM における**トランザクション**は，クリティカルセ

†1 名古屋工業大学
Nagoya Institute of Technology
*1 現在，株式会社デンソー
Presently with DENSO Corporation

© 2012 Information Processing Society of Japan

クションを含む一連の命令列として定義され，以下の2つの性質を満たす．

シリアライザビリティ（直列可能性）： 並行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り，全てのスレッドにおいて同一の順序で観測される．

アトミシティ（不可分性）： トランザクションはその操作が完全に実行されるか，もしくは全く実行されないかのいずれかでなければならず，各トランザクション内における操作はトランザクションの終了と同時に観測される．そのため，操作の途中経過が他のスレッドから観測されることはない．

以上の性質を保証するために，TM はトランザクション内のメモリアクセスを監視する．そして，あるトランザクション内でアクセスされたメモリアドレスと他のトランザクション内でアクセスされたメモリアドレスが同一であった場合，競合として検出する．競合を検出した場合は，片方のトランザクションの実行を中断する．これを**ストール**と言う．さらに，複数のトランザクションがストールした場合で，デッドロックが発生したと判断された場合，片方のトランザクションの実行結果を破棄する**アボート**を行う．そしてトランザクション開始時点の状態を復元し，トランザクションを再実行する．一方でトランザクションの終了まで競合が発生しない場合は，トランザクション内で実行された結果をメモリに反映させる**コミット**を行う．

TM はこのように動作することで，競合が発生しない限りトランザクションを並列に実行することができ，ロックを用いる場合よりもプログラムの並行性が向上する．また，プログラマはロックの粒度を考慮する必要がなくなり，容易に並列プログラムを構築できる．

2.2 LogTM

本節では，HTM の一種であり本研究のターゲットとなる LogTM について述べる．

2.2.1 データのバージョン管理

TM におけるトランザクションの投機実行では，実行結果が破棄される可能性があるため，アクセスするデータをバージョン管理する必要がある．本稿がターゲットとする LogTM は，仮想メモリ領域を用いることでこのバージョン管理を実現している．LogTM 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このログに退避する．一方でストア命令の結果はメモリに書き込まれる．

ここで投機実行が失敗した場合はアボートを行い，ログに保存されている全ての値をメモリに書き戻すことでトランザクション開始時点の状態を復元する．一方で，投機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，ログの走査や退避した値の書き戻し等のメモリアクセスは必要なく，ログの内容を破棄するだけでよい．

2.2.2 競合検出

トランザクションのアトミシティを保つために，トランザクション内で実行される命令における競合の有無を監視する必要がある．そこで LogTM は，キャッシュライン上に新しく read ビットおよび write ビットを追加している．read ビットと write ビットは，トランザクション内で当該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それぞれセットされ，トランザクションのコミットおよびアボート時にクリアされる．

LogTM は一貫性モデルにディレクトリベースの Illinois プロトコルを採用し，これを拡張することでトランザクションを実行する他のスレッドとの競合を監視している．競合として検出されるのは以下の 3 パターンのアクセスが行われた場合である．

Read after Write (RaW): 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 (WaR): 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 (WaW): 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

例えば，あるスレッドがリード/ライトアクセスを行う場合，トランザクション内の一貫性を保つために，アクセス対象となるラインに他のスレッドが既にアクセス済であるかどうかをディレクトリに対して問い合わせる．既にアクセスされていた場合，コヒーレンスリクエストを当該スレッドに送信する．このリクエストを受信したスレッドは，どのメモリアドレスへのアクセスが行われようとしているのか知ることができ，当該キャッシュライン上の read ビットおよび write ビットを参照することで競合を検出することができる．競合が検出されなかった場合は，リクエスト送信者に対して ACK が返信される．一方で競合が検出された場合は NACK が返信される．NACK を受信したスレッドは競合の発生を知り，競合相手の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**ストール**を行う．ストールしている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．競合相手のトランザクションが終了した場合，そのスレッドから ACK が返信されるため，ストールしているトランザクションは相手の終了を検知して実行を再開できる．

しかし**図 1** で示すように，複数のアドレスで競合が発生（時刻 t3 および t5）するとデッドロック状態に陥る場合がある．この例では，2 つのスレッド *thr.1* と *thr.2* がそれぞれトランザクション *Tx.X* と *Tx.Y* を投機的に実行している．まず，*thr.1* が *Tx.X* の実行を開始した後に *thr.2* が *Tx.Y* の実行を開始する．そして先に *thr.1* が `ST A` を実行し，その後に *thr.2* が `ST B` を実行済みである場合を考える．その後 *thr.1* が `LD B` を実行しようとする際，*thr.1* は他のスレッドに対してアクセスリクエストを送信する（t1）．これを受信した *thr.2* は競合の発生を検知するため NACK を返信し，NACK を受信した *thr.1* はストールする（t3）．なお図中では省略しているが，*thr.1* はアクセスの許可を受けるまで定期的にリクエストを送信している．この後，*thr.2* が `LD A` を実行しようとする（t4）と，

© 2012 Information Processing Society of Japan

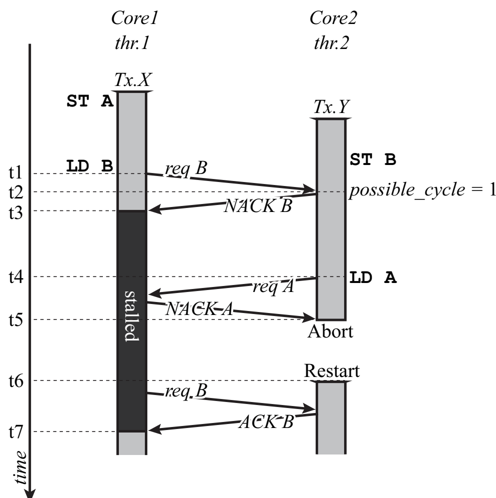

図 1 LogTM におけるトランザクションの競合解決

thr.2 は *thr.1* と競合してお互いの実行を制止し合う結果となり，デッドロック状態に陥ってしまう．

そこで LogTM では，Transactional Lock Removal[3] の分散タイムスタンプに倣った方法を採用している．具体的には，まず図 1 の時刻 t2 で示すように，自身より早くトランザクションを開始したスレッドに NACK を返信すると **possible_cycle** と呼ばれるフラグをセットする．そして，このフラグがセットされている状態で，自身よりも早くトランザクションを開始したスレッドから NACK を受信すると，デッドロックを回避するためにアボートする（t5）．こうして，開始時刻の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．*Tx.Y* をアボートした *thr.2* はトランザクション開始時の状態を復元し，トランザクションを再実行する（t6）．また，*thr.2* が *Tx.Y* をアボートしたため，*thr.1* は `B` にアクセスできるようになり，*Tx.X* のストール状態が解消される（t7）．

2.2.3 競合の抑制

LogTM では，競合の発生を抑制するために exponential backoff および magic waiting という手法が実装されている．Exponential backoff は，トランザクションをアボートした後，再実行開始までの間，一定期間待機する手法である．アボートが発生するたびに，待機期間を指数関数的に増大させることで競合の再発を抑制している．なお，この待機時間はコミット時に初期化される．

一方 magic waiting は，アボートした後，その競合相手がコミットするまで実行を再開せず待機し続けることで，完全に競合を防ぐ手法である．複数のスレッドと競合した場合には，相手から受信したリクエストまたは NACK から相手のトランザクション開始時刻を取得し，その時刻が一番遅いスレッドがトランザクションをコミットするまで待機する．なお，これらの手法を用いた場合，待機しているスレッドが遊休状態となるため，並列度が低下するという問題がある．

3. 関連研究

トランザクションの途中から再実行することにより，その地点までの再実行を省略する部分ロールバック[4] に関する研究や，適切なスレッド数を動的に設定する研究[5] など数多くの LogTM に関する研究が行われている．前者の手法を改良した伊藤らの研究[6] では，競合を起こした命令のプログラムカウンタの値を記憶し，再度そのプログラムカウンタに該当する命令を実行する際，その位置を再実行開始位置 (チェックポイント, CP) として設定することで，再実行命令数の削減を可能にしている．また，既存の LogTM では CP の作成数に制限があったが，その制限を無くす手法も提案している．しかし，LogTM で標準的な possible_cycle flag を用いる方法ではなく，競合発生時に即座にトランザクションがアボートする，よりアボートが発生しやすいベースラインモデルに対する高速化で評価している．また，評価に対する考察が少なく，改良モデルのベースとしている Williullah らによる手法[7] との比較評価もなされていないため，提案手法による効果の範囲が明らかになっていない．

一方後者のスレッド数の動的制御に関する研究では，競合とトランザクション数に相関関係があることに着目し，動的にスレッド数を調整することでアボート回数を削減し，高速化を実現している．しかし，提案手法をによって発生するオーバーヘッドや，実装に必要なハードウェアコストについて評価していない．また，評価に用いたベンチマークプログラムが 1 つのみであり，効果の汎用性も明らかではない．

なお，部分ロールバックを適用した場合，LogTM のスケジューリングでは競合がより再発しやすくなる．また，後者の研究のようにスレッド数を減少させなくともスケジューリングの改良次第では並列度を高く保ち続けることができる可能性もある．したがって，本稿では，まずはスレッドのスケジューリングに着目することで，従来の問題の解決を目指す．

4. 競合抑制手法の提案

本章では，既存手法の問題点とそれを解決する 3 つの提案手法について説明する．

4.1 既存手法の問題点

LogTM では，ある特定の競合パターンが発生すると著しく性能が悪化する場合がある．その競合パターンの 1 つに大きく関わっているのが **starving writer** と呼ばれるトランザクションの発生である．この競合パターンは，ストア（ST）の実行が複数のロード（LD）の実行により妨げられ続けることにより発生する．

いま**図 2** のように，3 つのスレッド (*thr.1*〜*3*) が，それぞれトランザクションを実行する例を考える．なお，*thr.1* および *thr.3* は同じトランザクション *Tx.X* を実行し，*thr.2* は *Tx.X* で LD されるアドレス `A` に対する ST を含む *Tx.Y* を実行するとする．まず *thr.1* が `LD A` を実行済みの状態で，*thr.2* が `ST A` を実行し

ⓒ 2012 Information Processing Society of Japan

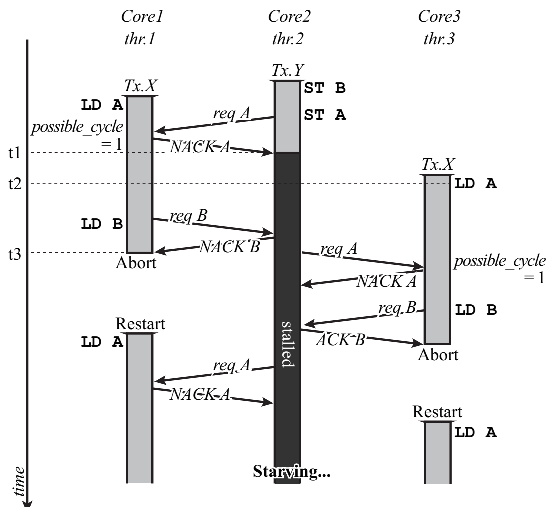

図 2　starving writer の発生

ようとして競合が発生し，*Tx.Y* はストールする（時刻 t1）．この場合，*thr.1* が *Tx.X* をコミットもしくはアボートしない限り *thr.2* は `ST A` を実行できない．次に，*thr.3* が `LD A` を実行しようとするが（t2），これは 2.2.2 項で述べたいずれのアクセスパターンにも該当せず，競合は検出されない．これにより，*thr.1* および *thr.3* が実行中の 2 つの *Tx.X* が，共にアボートもしくはコミットしない限り *thr.2* は再開することができない状態となる．この後，*thr.1* が *thr.2* と競合して *thr.1* の *Tx.X* がアボートされた場合（t3）でも，*thr.3* が既にアドレス `A` にアクセスしているため，*thr.2* は実行を再開できない．そして *thr.1* は *Tx.X* の再実行後，再度 `A` にアクセスしてしまう．このように，同一アドレスに対する LD を実行するスレッドが複数存在することで，当該アドレスに対する ST を実行しようとしているスレッドが飢餓状態（starving writer）となる．実際には，更に多くのスレッドが LD を実行している場合が多く，それら全てのスレッドがトランザクションをアボートあるいはコミットしてリソースを解放しない限り，ST を実行しようとしているスレッドは再開することができない．

この競合パターンは 2.2.3 項で述べた exponential backoff または magic waiting によって対処できる．しかし，前者ではアボートしたトランザクションが一時的にしか待機しないため，starving writer が解決されるまでに何度もアボートを繰り返してしまう．一方後者では，アボートを繰り返していない場合にも待機し続けてしまい並列度が低下してしまう．

そこで本稿では，starving writer によってアボートが繰り返される場合に，アボートしたトランザクションを一時停止させ，starving writer を解決する手法を提案する．

4.2　提案モデルとその動作

前節で述べた starving writer の発生を抑制するために，LD を実行するトランザクション（reader）が，ST を実行しようとするトランザクション（writer）を競合相手としてある条件を満たすようなアボートを繰り返す傾向を見せた場合，その reader の実行に magic waiting を適用し，相手 writer を優先的にコミットさせる手法を提案する．

さて，starving writer は WaR 競合パターンが存在する場合に発生する．また，2.2.2 項で述べたように，アボート処理までに少なくとも 2 つのキャッシュラインで競合が発生する．これをふまえ本節では，starving writer が発生したと判断する条件セットを 3 種提案し，それぞれを用いた場合の動作モデルを説明する．

モデル 1：同一 writer との競合アドレスの組が一致

あるトランザクションが以下に示す 2 つの条件を共に満たす場合，競合相手を starving writer であると判定し，自身に magic waiting を適用することでその相手 writer を優先的にコミットさせる．

- 条件 I：自身が LD 済のアドレスに対して他スレッドが ST 要求を発行することにより WaR 競合が発生
- 条件 II：同一の writer との間の競合によって発生した，直近の過去 2 回のアボートに関与したアドレスの組が一致

なお，アボートが発生するためには，2.2.2 項で見たように possible_cycle フラグをセットする原因になった競合と，アボートの直接的な引き金となった競合の 2 つのアドレス競合が必要であるが，条件 II の「アボートに関与したアドレスの組」とは，これら 2 つの競合それぞれにおける対象アドレスの組を指す．

これら 2 つの条件について**図 3** の starving writer が発生する場合の例を用いて説明する．例ではまず *thr.2*（reader）が `LD B` を実行し，次に *thr.1*（writer）が `ST B` を実行しようとして競合が発生する（時刻 t1）．これは WaR 競合であるため，条件 I を満たす．その後 *thr.2* が実行する *Tx.2* は，*thr.1* によって 2 度アボート（t2 および t3）させられており，アボートに関与したアドレスの組み合わせが共に {`B`, `A`} となっている．したがって条件 II を満たす．このように両方の条件を満たした場合，ST を実行しようとしていたトランザクションを starving writer であるとみなし，LD を実行していたスレッドに対して 2.2.3 項で述べた magic waiting を有効にすることで，starving writer となっていたトランザクションを優先的に実行させる．

モデル 2：同一 writer との競合アドレスが一致

ある writer トランザクション *Tx.W* が，ある reader トランザクション *Tx.R* にストールさせられて starving 状態にあるとき，*Tx.W* は終始ストールし続けているとは限らず，他の第 3 者のトランザクションとの競合によりアボートさせられてしまう場合もある．この場合 *Tx.W* は再実行されるが，制御フローの変化等により，*Tx.R* との再競合の際に，possible_cycle フラグをセットする原因となる競合アドレスが前回とは異なる場合もあり得る．このような場合も解決するために 2 つめのモデルとして，モデル 1 の条件 II を以

© 2012 Information Processing Society of Japan

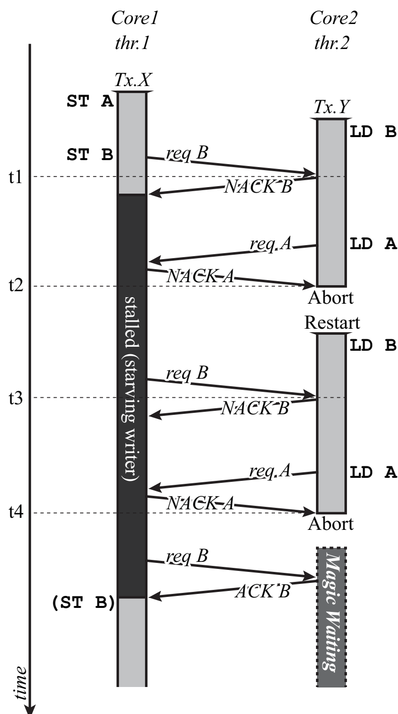

図 3 Starving Writer 発生時の動作モデル
Fig. 3 Proposed model with a starving writer.

下のように緩和したものを考える．

- 条件 II′: 同一の相手による直近の過去 2 回のアボートにおいて，そのアボートに直接関与するアクセス対象アドレスが同一

すなわち，possible_cycle フラグをセットする原因となったアクセス対象アドレスの一致を必要としないよう，条件 II を変更する．これと，モデル 1 の条件 I を併用することで，starving writer を解消する．

図 3 の例では，*Tx.Y* のアボートは共にアドレス A へのアクセス（t2 および t3）を直接的な原因として発生しているため，条件 II′ に該当し，*thr.2* に magic waiting が適用される．

モデル 3：任意 writer との競合アドレスが一致

同じアドレスに対する WaR 競合は，異なる複数の writer との間で発生する可能性は少ないと考えられる．したがってモデル 2 に対し，競合相手を考慮しないように条件を簡略化した拡張モデルを考える．これを実現するため，モデル 2 の条件 II′ において，競合相手に関する部分を次のように緩和する．

- 条件 II″: 競合相手を問わず，直近の過去 2 回のアボートにおいて，そのアボートに直接関与するアクセス対象アドレスが同一

なおモデル 2 と同様に，モデル 1 の条件 I を併用する．

5. 競合の再発検知機構

本章では，3 つの提案手法を実現するにあたり必要なハードウェアおよびその動作モデルについて述べる．

5.1 ハードウェア拡張

4.2 節で述べた提案手法を実現するために，既存の LogTM を拡張して以下に示す 3 つの機構を各コアに追加する．

WaR flags: 競合パターン WaR 発生の有無を示す．自身が LD を実行済みであるアドレスに対して，他スレッドが ST を実行しようとした際の競合発生時にセットする．総スレッド数 n に対して n bit 必要であり，アボートおよびコミット時にクリアされる．

Conflict Table (C-Tbl): スレッド番号をインデクスとし，当該スレッドとの間に起こった直近の競合において対象となったアドレスを記憶する表．競合発生時に参照され，現競合の対象アドレスと比較される．アドレスが一致した場合は後述の M-W flags の状態を更新し，一致しない場合は現競合アドレスでエントリを上書きする．提案モデル 1 は直近の 2 つのアドレスを条件判定に利用するため，このテーブルを 2 つ用意し，競合したアドレスに対して先にアクセスしたのが自分である場合は C-Tbl1，競合相手である場合は C-Tbl2 を用いる．なお，テーブル内の情報はコミット時のみクリアされる．提案モデル 2 では C-Tbl は 1 つでよく，提案モデル 3 ではスレッドを区別しないため C-Tbl は 1 つかつ深さ 1 でよい．

Magic Waiting flags (M-W flags): Magic waiting を有効にするかどうかを示す $2n$ bit からなるビット列で，各スレッドに対して 2 ビットずつ使用する．C-Tbl1 で比較したアドレスが一致した場合は 1 ビット目，C-Tbl2 では 2 ビット目のビットをセットし，アボート時にこれら 2 つのビットが両方セットされている場合，magic waiting を有効にする．コミットおよびアボート時にクリアされる．

n スレッドを実行可能な n コア構成のプロセッサの場合，1 コアあたりに必要となる WaR flags は n bit，また M-W flags は $2n$ bit であり，あわせて $3n$ bit と少量である．また C-Tbl については，幅 64 bit 深さ n 行の RAM で構成でき，例えば $n = 32$ では C-Tbl サイズの総和は 16kB と少量である．

また，提案手法 2 の場合は 4.2 節で述べたように，1 つのアドレスのみを条件に利用するため，C-Tbl は 1 つ，M-W flags は 1 bit となる．したがってハードウェアコストは提案手法 1 の約半分となる．そして提案手法 3 の場合は，4.2 節で述べたように，競合相手ごとにアドレスを管理しないため，C-Tbl サイズの総和は 256B と，ごく小さいものとなる．

5.2 動作モデル

図 4 の starving writer が発生する場合の例を用いて，*thr.2* における提案モデル 1 のハードウェア動作について説明する．まず，*thr.2* が `LD B` を実行し，その後に *thr.1* が `ST B` を実行しようとした場合，WaR パターンの競合が検出される（時刻 t1）．したがって *thr.2* では，競合相手のスレッド *thr.1* に対応する WaR flags がセットされ，スレッド番号 1 をインデクスとして C-Tbl が参照される．なお，今回競合が発

© 2012 Information Processing Society of Japan

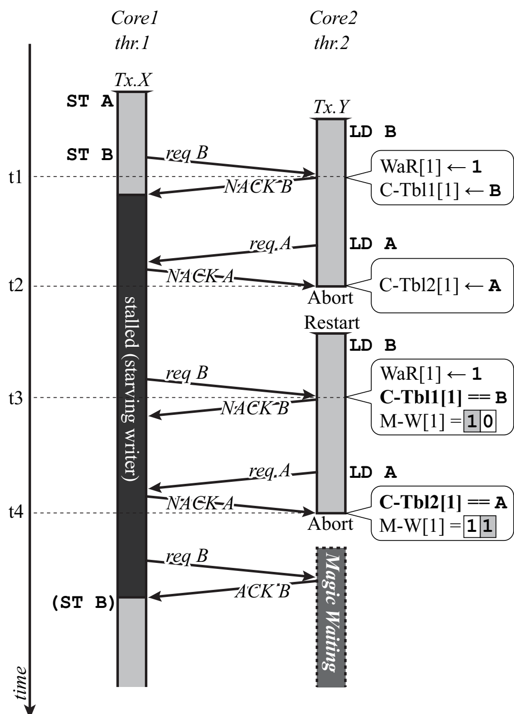

図 4 追加したハードウェアの状態変移

生したアドレス B に先にアクセスしたのは *thr.2* であるため，C-Tbl1 が参照される．ここでは，C-Tbl1[1] にはアドレスが未登録であるため，B が登録される．

次に，*thr.2* が LD A を実行し，競合が発生すると（t2），アドレス A へ先にアクセスしたのは *thr.1* であるため，先ほどとは別のテーブルである C-Tbl2 が参照される．ここで *thr.1* に対応するアドレスは C-Tbl2 にはまだ登録されていないため，A が登録される．そしてデッドロックの発生を検知したことにより，*thr.2* は *Tx.Y* をアボートする．なお，アボート後は全ての競合が解決されるため，WaR flags はリセットされる．

続いて *thr.2* が *Tx.Y* を再実行し，アドレス B で競合すると（t3），時刻 t1 と同様に WaR flags がセットされ，B がテーブルに登録されているか参照される．今回は既に同一のアドレスが登録されているため，M-W flags の左ビットをセットし，結果 10 となる．次に *thr.2* が LD A を実行し，競合が発生すると（t4），時刻 t3 の時と同様に C-Tbl2 が参照される．そして，登録済みのアドレスと今回競合したアドレス A とが一致するため，M-W flags の右ビットをセットする．この結果 M-W flags は 11 となり両ビットがセットされている状態になるため，magic waiting が有効となる．

一方提案モデル 2 では，提案モデル 1 と比べ C-Tbl が 1 つ少なく済み，C-Tbl1 への B の登録および参照の処理が省略される．また，M-W flags も 1 ビットとなっており，時刻 t3 における M-W flags に対する処理も省略される．最後に，提案モデル 3 は提案モデル 2 と比べて C-Tbl の構造が異なるが，2 者間によるアボートの繰り返し時には提案モデル 2 と同じ動作となる．

表 1 シミュレータ諸元

Processor	SPARC V9
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

6. 評　　価

6.1 評価環境

前章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 TM の研究で広く用いられている Simics[8] 3.0.31 と GEMS[9] 2.1.1 の組合せを用いた．Simics は機能シミュレーションを行うフルシステムシミュレータであり，GEMS 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担う．プロセッサは 32 コアの SPARC V9 とし，OS は Solaris10 とした．**表 1** に詳細なシミュレーション環境を示す．評価対象のプログラムとしては GEMS 付属 microbench，SPLASH2,[10] および STAMP[11] から計 12 個を使用した．

なお，各コアが 1 スレッドを実行し，プロセッサ全体で 32 スレッドを実行するが，OS 用に 1 コアを使用するとし，31 スレッドによる評価を行った．ただし，STAMP は 2 の冪乗数のスレッドでしか動作しないため，STAMP に限り 16 スレッドで評価した．また，3 章で述べたように，本稿ではまずスレッドのスケジューリングに着目するため，部分ロールバック手法を用いていない．したがって，GEMS 付属の部分ロールバック用テストプログラムである partial rollback は評価対象から除外した．

6.2 評価結果

図 5 および**表 2** に実行サイクル数比，**表 3** にアボート回数の削減率を示す．図 5 のグラフは左から順に

(B) 既存の LogTM
(S_1) 4.2 節の提案モデル 1
(S_2) 4.2 節の提案モデル 2
(S_3) 4.2 節の提案モデル 3

の実行サイクル数比を表しており，既存手法 (B) の実行サイクル数を 1 として正規化している．また，凡例は内訳を示しており，Non_trans はトランザクション外，Good_trans，Bad_trans はそれぞれ結果的にコミット/アボートされたトランザクション内，Abort-

ⓒ 2012 Information Processing Society of Japan

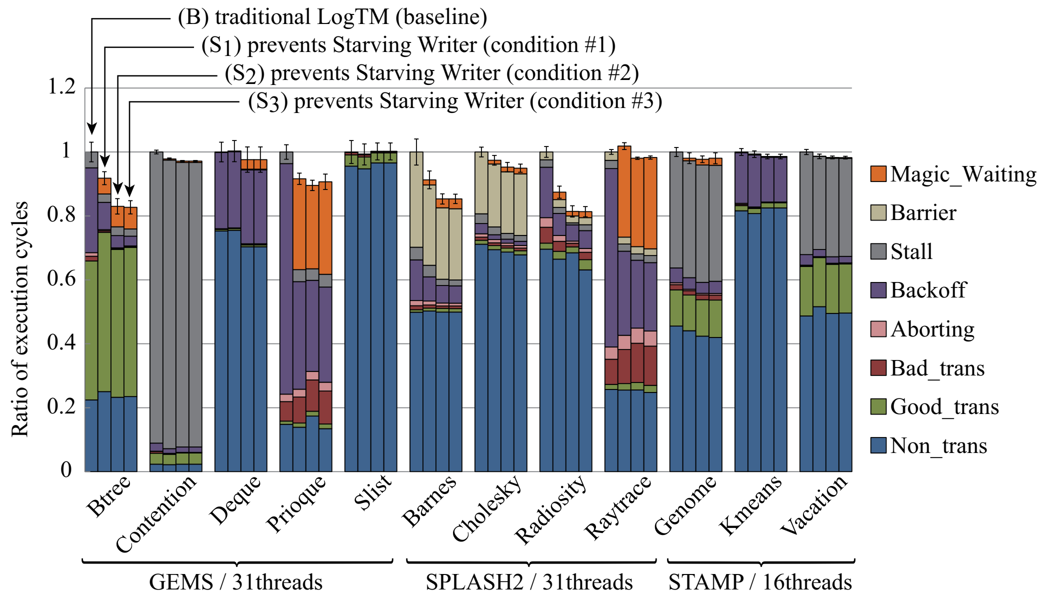

図 5 実行サイクル数比（GEMS，SPLASH2，STAMP ベンチマーク）

表 2 実行サイクル数の削減率

		GEMS	SPLASH2	STAMP	all
(S_1)	平均	3.9%	5.7%	1.3%	3.9%
	最大	8.4%	12.6%	1.9%	12.6%
(S_2)	平均	6.7%	10.2%	1.8%	6.7%
	最大	17.0%	18.6%	2.3%	18.6%
(S_3)	平均	6.6%	10.3%	1.7%	6.6%
	最大	17.3%	18.7%	1.9%	18.7%

表 3 アボート発生回数の削減率

		GEMS	SPLASH2	STAMP	all
(S_1)	平均	37.1%	25.5%	40.0%	34.2%
	最大	76.2%	45.7%	67.7%	76.2%
(S_2)	平均	46.6%	44.7%	47.9%	46.3%
	最大	86.8%	67.1%	72.9%	86.8%
(S_3)	平均	46.1%	45.4%	47.6%	46.3%
	最大	86.6%	67.4%	72.9%	86.6%

ing，Stall，MagicWaiting，Barrier，Backoff はそれぞれ，アボート，ストール，magic waiting，バリア同期，exponential backoff に要したサイクル数である．

なお，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．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%の信頼区間も求めた．信頼区間はグラフ中にエラーバーで表している．

結果から，実行サイクル数に関しては，評価に使用したベンチマークプログラムの多くは，本提案手法が解決すべき対象とした競合の再発，およびそれに伴うアボートの頻発を含んでいたため，提案手法によりこれを解決することで性能が向上した．ただし，Slist に関しては，競合の繰り返しがほとんど発生しないプログラムであるため，既存モデルとほぼ同等の結果となっている．一方アボートの発生回数に関しては，表 3 から分かるように大きく削減できており，提案手法が非常に有効に働いていることが分かる．

プログラムを個別に見ると，まず Contention, Deque, Genome, Kmeans, Vacation では，ほぼ全ての手法で提案手法によりわずかに高速化している．これは主にアボートの抑制によるもので，アボート回数は既存手法 (B) に対し最大 72.9%（Kmeans），最低でも 15.1%（Deque）削減されている．また，全実行サイクルに占める magic waiting の割合は，例えば Kmeans では 0.1%以下となっており，本提案によって新たに加えられた待機処理が短時間で済んでいることが分かる．しかしこれらのプログラムでは，元来アボートが実行サイクルに与える影響は小さかったため，高速化率は小さくなっている．

次に，Btree, Prioque, Barnes, Radiosity については，アボート回数の削減による Bad_trans や Aborting サイクルの減少，競合自体の削減による Stall サイクルの減少，アボートの繰り返しを抑制したことによる Backoff サイクルの減少などにより大きく高速化しており，提案手法の有効性が確認できた．

中でも Btree および Prioque は最も starving writer の影響を受けるプログラムであり，starving writer 発生時のアボート抑制および Backoff の削減が高速化に寄与している．その効果が最も顕著である Btree において，手法 (S_2) および (S_3) のアボート回数を調査したところ，既存手法に対してそれぞれ 86.8%，86.7%も削減できていることが確認できた．また，最長のアボート繰返し回数についても，それぞれ約 1/4 程度に削減されていた．

また，Barnes の場合，提案モデル (S_2)，(S_3) は，既存モデル (B) に対して Barrier を約 25%削減して

© 2012 Information Processing Society of Japan

いる．これは，特定のトランザクションがアボートの繰り返しにより遅延することで，バリア同期において他のトランザクションを長期間待機させてしまう状況を解決したことによると考えられる．

提案手法による効果が最も大きかった Radiosity については，提案手法により半数以上のスレッドが一時的に magic waiting を有効にする状況が見られた．これは即ち，非常に競合を起こしやすい特徴を持つスレッドが存在し，提案手法によりそのスレッドをコミットまで優先的に進行させることで，既存モデルで発生していたアボートの頻発を抑制したと考えられる．

一方，Prioque や Raytrace に見られる傾向として，アボート回数は減少しているものの Bad_trans サイクルが増加してしまっていることが挙げられる．既存手法ではトランザクション開始直後にアボートする状況が頻発しており，個々のアボートで計上される Bad_trans も小さいものであったのに対し，提案手法では，トランザクション中のより多くの命令を実行した後にアボートする場合があり，アボート回数は少ないものの個々のアボートで計上される Bad_trans サイクルが大きくなったためであると考えられる．

結果を総合すると，手法 (S_2)，(S_3) は (S_1) よりも高い性能を示しており，possible_cycle フラグをセットする原因となった競合アドレスを考慮せず，同アドレス競合によるアボートの繰り返しを抑制することが重要であることが分かった．また，(S_2) と (S_3) には有意な差はなく，直近のアボートに関係した単一アドレスのみを判定に利用することで十分であると考えられることから，ハードウェアコストも軽量である手法 (S_3) が最も優れていると言える．

7. おわりに

本稿では LogTM を拡張し，starving writer に対処するための競合抑制手法および競合の再発抑制手法を提案した．

シミュレーションにより GEMS 付属 microbench, SPLASH-2，および STAMP ベンチマークを用いて評価した結果，提案手法は競合再発に起因するアボートの繰り返しを抑制することで，結果的にアボートによって破棄されてしまう実行サイクルや，再実行までの backoff サイクルを削減することを確認した．その結果，既存の LogTM に比べてアボート発生回数を最大 86.6%削減することに成功し，実行サイクル数でも最大で 18.7%，平均で 6.6%の高速化を実現した．

なお，本稿では競合パターンの 1 つである starving writer の影響に着目したが，LogTM には著しく性能を低下させてしまう競合パターンが他にも複数存在する．特に，今回の評価結果からストールサイクルや backoff サイクルの占める割合が多いプログラムが存在していることから，今後これらの要因を調査し，対処方法を検討していきたい．また，magic waiting やストール時における遊休状態コアを有効活用する手法を検討することも今後の課題である．

参考文献

1) Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. of 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp.289–300 (1993).
2) Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D. and Wood, D. A.: LogTM: Log-based Transactional Memory, *Proc. of 12th Int'l Symp. on High-Performance Computer Architecture*, pp.254–265 (2006).
3) Rajwar, R. and Goodman, J. R.: Transactional Lock-Free Execution of Lock-Based Programs, *Proc of 10th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.5–17 (2002).
4) J.Moravan, M. et al.: Supporting Nested Transactional Memory in LogTM, *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp.1–12 (2006).
5) 武田　進，島崎慶太，井上弘士，村上和彰：トランザクショナルメモリにおける並列実行トランザクション数動的制御法の提案とその評価，信学技報，Vol.108, No.ICD-28, pp.81–86 (2008).
6) 伊藤悠二，塩谷亮太，五島正裕，坂井修一：最適なロールバック・ポイントを選択するトランザクショナル・メモリ，先進的計算基盤システムシンポジウム SACSIS2011 論文集，pp.324–331 (2011).
7) Waliullah, M.M. and Stenstrom, P.: Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems, *Proc. of Int'l Symp. on Parallel and Distributed Processing (IPDPS)*, pp.1–11 (2008).
8) Magnusson, P.S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A. and Werner, B.: Simics: A Full System Simulation Platform, *Computer*, Vol.35, No.2, pp.50–58 (2002).
9) Martin, M. M. K. et al.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, *ACM SIGARCH Computer Architecture News*, Vol.33, No.4, pp.92–99 (2005).
10) Woo, S.C. et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc of 22nd Int'l. Symp. on Computer Architecture (ISCA'95)*, pp.24–36 (1995).
11) Minh, C. C., Chung, J., Kozyrakis, C. and Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. of IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).

ⓒ 2012 Information Processing Society of Japan