情報処理学会論文誌　コンピューティングシステム　Vol.5　No.5　43–54 (Oct. 2012)

デッドロック検出の厳密化による LogTMのアボート削減手法

堀場 匠一朗[1]　江藤 正通[1]　浅井 宏樹[1,†1]　津邑 公暁[1,a)]　松尾 啓志[1]

受付日 2012年3月28日, 採録日 2012年5月1日

概要：マルチコア環境における並列プログラミングでは，共有メモリへのアクセス制御にロックが広く用いられてきた．しかし，ロックには並列性の低下やデッドロックの発生などの問題がある．そこで，ロックを用いない並行性制御機構として，トランザクショナル・メモリが提案されている．このハードウェアによる一実装である LogTM においては，possible_cycle と呼ばれるフラグを用いてデッドロックの発生を検出する．しかしこの手法では，デッドロックの判定に偽陽性が存在し，アボートが過剰に発生する可能性がある．そこで本稿では，3 者以上のトランザクション間の依存関係を考慮することでデッドロックを検出可能とする手法を提案し，さらに適切なアボート対象を選択する手法も検討する．シミュレーションによる評価の結果，提案手法によりアボートの発生が抑制され，ログの書き戻しコストなどが削減されることで，最大 31.5% の高速化を確認した．

キーワード：ハードウェア・トランザクショナル・メモリ，マルチスレッド，デッドロック，メモリアクセス競合

Reducing Aborts on LogTM by Strictly Detecting Deadlocks

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

Received: March 28, 2012, Accepted: May 1, 2012

Abstract: Lock-based synchronization techniques have been commonly used for parallel programming on multi-core processors. However, lock can cause deadlock and poor scalability. Hence, hardware transactional memory has been proposed and studied for lock-free synchronization. In LogTM, an implementation of hardware transactional memory, a flag called 'possible cycle' is used for detecting dead-locks. However, a false positive can occur and it leads to useless aborts. In this paper, we propose a deadlock detection method for LogTM by considering dependencies between several transactions, and also discuss how to decide which transaction should be aborted. The result of the experiment shows that the proposed method improves the performance 31.5% in maximum.

Keywords: hardware transactional memory, multithreading, deadlock, memory access conflicts

1. はじめに

マルチコア環境において一般的な共有メモリ型並列プログラミングでは，共有リソースへのアクセスに対する排他制御機構として一般にロックが用いられてきた．しかしロックを用いた場合，ロック操作のオーバヘッド増大にともなう並列性の低下や，デッドロックの発生などの問題が起こりうる．さらに，プログラムごとに最適なロック粒度を設定することは容易ではないため，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．

[1] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi 466–8555, Japan
[†1] 現在，株式会社デンソー
Presently with DENSO CORPORATION
[a)] tsumura@nitech.ac.jp

このトランザクショナル・メモリのハードウェア実装の1つであるLogTM [2] 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を投機的に実行する．投機実行するトランザクションのアトミシティを保つために，LogTMは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しているかどうかを検査する．そしてデッドロックを検出した場合，片方のトランザクションをアボートし，それまでの実行結果を破棄する．

LogTMでは，possible_cycleと呼ばれるフラグを用いてデッドロックの発生を検出する．しかしこの手法では，デッドロックの判定に偽陽性が存在し，アボートが過剰に発生する可能性がある．そこで本稿では，トランザクション間の依存関係を考慮して正しくデッドロックを検出することで，無駄なアボートの発生を抑制しLogTMの高速化を図る．また，デッドロック検出後に必要となる，アボート対象を選択するための指針についても検討する．

2. 研究背景

本章では，本稿が対象とするトランザクショナル・メモリおよびLogTMの基本的概念とその関連研究について述べる．

2.1 トランザクショナル・メモリ

トランザクショナル・メモリ（Transactional Memory，以下TM）では，クリティカルセクションを含む一連の命令列を，以下の2つの性質を満たす**トランザクション**として定義する．

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，すべ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，もしくはまった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．

これらを保証するため，TMはトランザクション内のメモリアクセスを監視する．このとき，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，これを競合として検出する．トランザクション間で複数の競合が検出された場合，片方のトランザクションの実行を中断し，それまでの結果をすべて破棄する．これを**アボート**と呼ぶ．トランザクションをアボートしたスレッドはメモリおよびレジスタの状態をトランザクション開始時の状態に戻し，トランザクションを再実行する．この，状態復元の処理を**ロールバック**と呼ぶ．一方で，トランザクションの終了まで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すべてメモリに反映させる．これを**コミット**と呼ぶ．TMはこのようにして，ロックによる排他制御と同等のセマンティクスを維持しつつ，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る．

2.2 LogTM

TMをハードウェア上に実現したものの1つに，LogTM [2] がある．本節ではLogTMにおけるデータのバージョン管理方法および競合検出方法について述べる．

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的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の古いバージョンを保持し管理する必要がある．LogTMは，これを仮想メモリ領域を用いることで実現している．具体的には，**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，そのアドレスのセットをこのログに退避させる．一方，ストア命令の結果はメモリに書き込まれる．なお，トランザクション内で同じアドレスに対して複数回ストア命令が実行された場合，ログに保存するのは最初の1回だけでよい．なぜなら，ロールバックにはトランザクション開始時点のメモリ状態だけが必要だからである．

その後，当該トランザクションがアボートした際は，ログに退避した値を書き戻すことで状態を復元する．一方コミットは，すでにすべての値がメモリ上に反映済であるため，単にログを破棄することで実現できる．

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は，あるトランザクション内のメモリアクセスと他のトランザクションのメモリアクセスとの間に競合が発生しているかどうかを検査する必要がある．そのため，トランザクション内でどのメモリアドレスがアクセスされたかを記憶しておかなければならない．

これを実現するために，LogTMではキャッシュライン上に新たにreadビットおよびwriteビットと呼ばれるフィールドを追加している．トランザクション内でリード/ライトアクセスが発生すると，アクセスのあったラインのread/writeビットがセットされる．そして，各ビットは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．

また，LogTMでは競合が発生したことをトランザクションに通知するため，キャッシュコヒーレンスプロトコルを拡張している．あるスレッドがメモリにアクセスする際，アクセス対象となるラインにすでに他のスレッドがアクセス済みであるかどうかをディレクトリに対して問い合わせ

る．すでにアクセスされていた場合，キャッシュコヒーレンス要求を当該スレッドに送信する．拡張したプロトコルにおいて，各スレッドは要求を受信すると，キャッシュラインの状態を変更する前に，read ビットおよび write ビットを参照し，競合の有無を検査する．以下の 3 パターンが競合として検出される．

Read after Write： write ビットがセットされているキャッシュライン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キャッシュライン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キャッシュラインに対するライトアクセス．

以上のような競合パターンが検出されると，要求を送信したスレッドに対して NACK が返信される．一方で競合が検出されなかった場合は ACK が返信される．なお，実際に NACK および ACK を送信するのはキャッシュディレクトリであるが，本稿では便宜的に，先行して当該アドレスにアクセスしたスレッドが送信するとして説明する．

NACK を受信したスレッドは競合の発生を知り，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**と呼ぶ．ストールしたスレッドは同じアドレスに対するリクエストを送信し続ける．競合スレッドがそのトランザクションを終了すると，ACK が返信されるため，ストールしていたスレッドは実行を再開できる．

しかし，複数のスレッドがストールするとデッドロックに陥る可能性がある．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，デッドロックを起こしうると考えられるトランザクションが競合相手のトランザクションよりも遅い開始時刻を持つ場合，そのトランザクションをアボートする．これは各プロセッサコアに **possible_cycle** と呼ばれるフラグを保持させることで実現されている．あるコアが持つフラグは当該コアで実行中のトランザクションよりも開始時刻の早いトランザクションに NACK を送信した際にセットされ，セットされた状態で同一または他の開始時刻の早いトランザクションから NACK を受信した場合にデッドロックが発生したと判定される．

2.3　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，必要な命令数を削減する部分ロールバックに関する研究 [4] 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をチェックポイント（CP）として記憶し，その位置から再実行可能とすることで，実行命令数の削減を図っている．さらに，CP の作成数制限をなくす手法も提案している．しかし，LogTM の標準である possible_cycle フラグを用いるモデルではなく，アクセス競合発生時に即座にトランザクションをアボートさせる，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点や，評価結果に対する考察が少なく，提案手法がどのように性能に影響を与えたのかが不明である点，提案手法のベースモデルである Waliullah らの手法 [7] による効果も含めた性能向上幅で評価しているため，提案手法が性能向上に直接与えた影響が明らかとなっていないない点など，評価方法にさまざまな問題がある．

一方，後者のスレッド数の動的制御に関する研究 [5] 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数を削減し，高速化を実現している．しかし，提案手法によって発生するオーバヘッドや，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．

3.　アボート条件の厳格化モデル

本章では，既存手法である LogTM の問題点と，それを解決する提案手法について述べる．

3.1　既存手法の問題点

2.2.2 項で述べたとおり，LogTM では競合検出に possible_cycle フラグを用いる．しかし，フラグをセットすべきか否か，また，フラグに基づきトランザクションをアボートすべきか否かを，現在競合が発生した 2 つのトランザクション間のみで判断しているため，過剰にアボートが発生してしまう可能性がある．LogTM は，コミットよりアボートの際にかかるオーバヘッドが大きくなる eager version management モデルを採用しており，過剰なアボートの発生は性能を大きく低下させる原因となりうる．

ここで，3 スレッド（*thr.1*〜*3*）上でそれぞれトランザクション（*Tx.1*〜*3*）が実行される**図 1** の例を用いて，不要なアボートが発生する場合を示す．*Tx.1* は時刻 t1 において `LD B` を実行しようとするが，当該アドレスはすでに *Tx.2* がアクセス済みであるため，NACK を受信する．このとき *thr.2* は，自身が実行中の *Tx.2* よりも早く開始された *Tx.1* へと NACK を送信することになるため，possible_cycle フラグをセットする（t2）．一方 *Tx.1* はこれによりストールする（t3）．

次に *Tx.2* が `ST C` を実行しようとするが，すでに *Tx.3* によりアクセス済みであるため，NACK を受信する．このとき，実際にはデッドロックは発生していないが，possible_cycle フラグがセットされている状態で自身よりも早く開始されたトランザクションから NACK を受信したため，*Tx.2* はアボートすることとなる（t4）．

この例のように，possible_cycle フラグをセットする原因

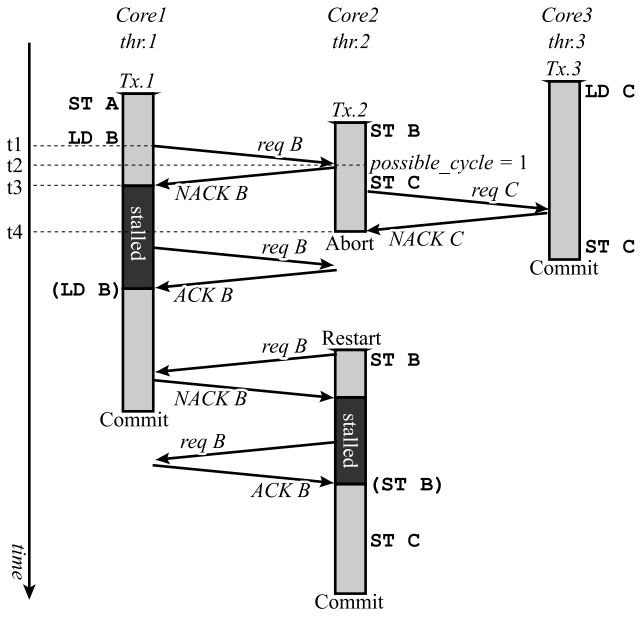

図 1　possible_cycle フラグにおける問題点
Fig. 1　A problem with possible_cycle flag.

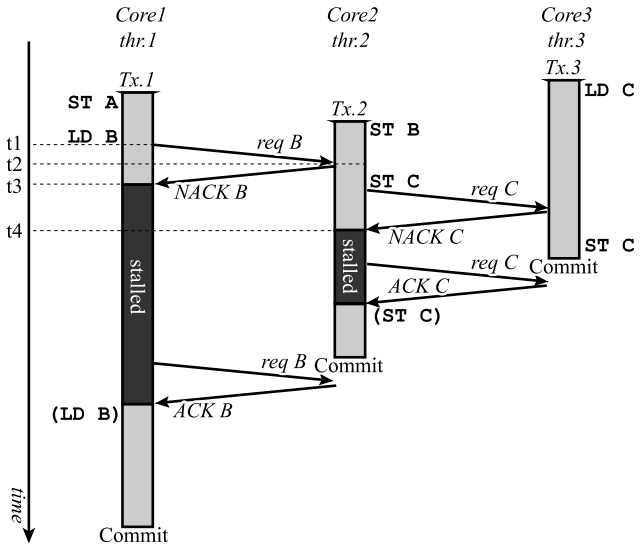

図 2　アボート条件を厳格化した動作モデル
Fig. 2　New model with a strict abort condition.

となった相手以外のトランザクションに影響を受け，実際にはデッドロックを起こしていない場合にもアボートが発生する．また，アボートによりトランザクションを再実行することで再度競合が発生し，同一トランザクションがアボートを繰り返し発生させてしまう可能性もある．

3.2　提案手法の動作モデル

多数のスレッドが並行実行される環境においては，トランザクション間に多くの競合が発生するため，possible_cycle フラグを用いる方法では多くの無駄なアボートが発生する可能性がある．そこで本稿では，デッドロックの検出条件をより厳格化することでアボートの発生を抑制し，ログの書き戻し処理に要するサイクル数を削減することで高速化を図る手法を提案する．

図 1 と同じ例に提案モデルを適用した場合の動作を**図 2** に示す．提案モデルでは既存モデルの場合と同様，*thr.2* は時刻 t4 において，自身の *Tx.2* より早期に開始された *Tx.3* を実行中の *thr.3* から NACK を受信するが，t2 において先に NACK を送信した相手 *thr.1* とは異なる相手からの NACK であるため，この時点ではデッドロックは発生していない．よって，アボートせず単にストールする．

これにより，既存モデルでは *Tx.2* のアボートによって発生していたログの書き戻しおよび実行再開に要するオーバヘッドが削減できる．一方，*Tx.1* のように一部のトランザクションでストールサイクルが増大することで並行実行スレッド数が減少し，性能に悪影響を及ぼす可能性もある．

3.3　アボート対象トランザクションの選択

本提案手法では possible_cycle フラグを用いる代わりに，3 つ以上のトランザクション間にまたがる依存関係の環の発生をデッドロックとして検出する．この検出方法の詳細については次の 4 章で述べる．

さて，デッドロックが検出された場合，そのデッドロックの発生に関与しているトランザクションのうちいずれかを選択してアボートさせる必要がある．本稿では，デッドロック検出後即座にアボート対象を決定可能な，以下の方針を提案手法に採用する．

検出者をアボート　デッドロック検出者自身がアボートする．検出後に情報収集などの操作が必要なく，即座にアボート可能である．

ただし，より適切にアボート対象を選択できれば，さらなる性能向上が得られる可能性もある．よって，以下の 2 つの方針についても，その効果をあわせて検討する．

競合相手数の最も多いものをアボート　多くの他トランザクションをストールさせているものをアボートする．より多くのトランザクションが復帰可能となることで，並列性の向上が期待できる．

実行開始時刻の最も遅いものをアボート　既存モデル同様，実行時間が長いトランザクションを優先し，スタベーションを防ぐ．また，実行時間が長いトランザクションは多くのメモリアクセスを行っている可能性も高く，これをいち早くコミットさせることで競合の頻発が抑制されると期待できる．

なお，1 つめの検出者自身をアボートさせる方法を除く 2 つの方法では，デッドロックに関与している全トランザクションから，それらの競合相手数もしくは実行時間の情報をそれぞれ収集する必要がある．このため，デッドロック検出後に複数のスレッド間通信が必要となり，アボート対象を選択するためのオーバヘッドが発生する．

4. 依存情報の伝播とデッドロック検出

本章では提案モデルの具体的な実装方法について述べる．

4.1 デッドロック検出アルゴリズムの概要

分散システムやデータベース管理におけるデッドロック検出方式として，資源の解放待ちによる関係を表した待ち状態グラフ（TWFG: Transaction Wait-For Graph）を常時管理する方式がよく用いられる [8]．この TWFG に閉路が存在することは，デッドロック状態にあることの必要十分条件であるため，定期的に閉路探索アルゴリズムを実行することでデッドロックを正確に検出できる．一方，ある程度の時間が経過しても終了しないトランザクション処理はデッドロックに陥っていると推定する，タイムアウト方式なども存在する．しかし，マルチコア環境で実行されるトランザクション間においてデッドロックを検出する場合，これらの手法を適用するには大きなオーバヘッドが発生すると考えられる．

まず前者では，単一のコアに TWFG を管理させる場合，そのグラフの更新のたびにメッセージ通信が必要となるため通信コストが大きくなる．また，グラフを管理するコアに負荷が集中するという問題もある．一方，各コアがそれぞれグラフの一部を分割して保持する場合，新たにトークンを定義することで閉路探索アルゴリズムを実行することが可能である．しかし，トランザクション間の依存関係が複雑になるほど，探索によるトークンのコア間のホップ数は増大すると考えられ，大きなオーバヘッドが発生する．また，競合発生時にグラフを更新し，閉路検出アルゴリズムを規定の間隔を指定することで定期的に実行することになるため，その時間間隔が小さすぎた場合にはデッドロック検出のオーバヘッドが増大し，大きすぎた場合にはデッドロックの検出が遅れてしまうという問題もある．

一方後者のタイムアウト方式の場合，デッドロック自体を正確に検出することはできないため，無駄なタイムアウト待ちによるオーバヘッドが発生しうる．また，デッドロックを検出するタイミングはトランザクションの規模に依存することになるため，タイムアウト時間の設定も困難である．さらに，単に処理時間がかかっているトランザクションを，誤検出によりアボートさせてしまう可能性もある．

したがって，できるだけ余分な処理や通信を追加することなく，デッドロックをある程度正確に検出したうえで，それを解消しうるトランザクションを適切にアボートし，ロールバックさせることが望ましい．そこで本稿では，競合発生時にキャッシュコヒーレンスプロトコルに基づき必ず返信される NACK メッセージに，ストールによるトランザクション間の依存関係情報を付加することで，スレッド間における情報の共有を実現する手法を提案する．これにより，新たなメッセージを定義する必要がなく，かつ情報共有のオーバヘッドを最小限に抑えた，リアルタイムなデッドロック検出を可能にする．

4.2 拡張ハードウェアとその操作アルゴリズム

前節で述べたアルゴリズムを実現するため，既存の LogTM を拡張し，以下の 2 つの記憶ユニットを各プロセッサコアに追加する．なお，以下ではコア総数を n とする．また LogTM に準じ，各コアは単一のスレッドを同時実行可能であるとする．したがってプロセッサ全体で同時実行可能な最大スレッド数も n である．以下，簡単のため，i 番目のコア $Core_i$（$i = 1, ..., n$）で実行中のスレッドおよびトランザクションをそれぞれ $thr.i$，$Tx.i$ と呼ぶ．

stall bits（S_i）　自身を直接的および間接的にストールさせているスレッドを実行中のコア番号を記憶する，n bit のビットマップ．以下，$Core_i$ の持つ stall bits を $S_i = s_i^n ... s_i^2 s_i^1$ と表記する．$thr.i$ が $thr.j$ に直接または間接的にストールさせられているか否かは，s_i^j がセットされているか否かにより表現され，$s_i^i = 1$ がデッドロックの存在を表す．各コアが n bit を保持するため，プロセッサ全体で必要となる記憶容量は n^2 bit である．

clear bits（C_i）　依存関係の解消されたスレッドを実行中のコア番号を記憶する，n bit のビットマップ．以下，$Core_i$ の持つ clear bits を $C_i = c_i^n ... c_i^2 c_i^1$ と表記する．stall bits と同じく，プロセッサ全体で必要となる記憶容量は n^2 bit である．

各コアは，自身のスレッドを直接・間接的にストールさせている他スレッドを実行中のコア番号を，stall bits S_i に記憶する．たとえば $Core_i$ が実行中である $thr.i$ が $thr.j$ によってストールさせられると，$Core_i$ は自身の S_i 内の s_i^j を 1 にセットする．そして新たな競合が発生した際，この S_i を NACK とともに送信することで，依存情報を伝播させる．

また，トランザクションのコミットおよびアボート時には部分的に依存関係が解決されるため，stall bits の対応ビットをクリアする必要がある．そこで，解消された依存関係情報を clear bits として一時的に記憶しておき，次回の NACK 時に clear bits をあわせて送信することで，解消された依存関係情報を伝播させる．

ここで，これらのビット列を用いてデッドロックを検出するための具体的な操作を，以下の 4 つの状況に分けて説明する．

ACK 受信時：　$Core_i$ が他スレッドから ACK を受信すると，自身の持つ S_i をすべてクリアし初期化する．あるトランザクションのストールに直接的に関わっているのは最大 1 つのアドレスに対するアクセスであり，そのキャッシュラインにおける競合が解消されると，

それにともなう間接的な依存関係もすべて解消されるためである．なお，このとき，クリアする直前の S_i の内容を C_i にコピーし記憶する．

NACK 受信時： $Core_i$ が $thr.j$ から NACK を受信すると，その NACK に付加されていた S_j と自身の持つ S_i の論理和をとり，新たな S_i として記憶する．さらに，NACK に C_j が付加されていた場合，C_j 内でセットされているビットに対応するビットを S_i からクリアする．この操作の結果新しく得られた S_i に含まれる，自身の番号に対応するビット s_i^i がセットされていた場合，デッドロックを検出する．

NACK 送信時： $Core_i$ が現在 S_i に保持しているビット列に対し，i 番目のビット，すなわち自身を表す s_i^i に相当する位置のビットをセットしたものを NACK に付加することで依存関係情報を伝播させる．ただし自身の持つ S_i にはこの変更は加えない．なお，ACK 受信直後の NACK 送信時には，通常の S_i に加え C_i も NACK に付加して送信する．

コミット/アボート時： $Core_i$ が実行中であった $Tx.i$ は実行を終了するため，そのトランザクションに関わる依存関係の記憶はすべて破棄される必要がある．よって自身の持つ S_i，C_i をすべてクリアし初期化する．$Tx.i$ のためにストールさせられていたスレッドが他コアに存在する場合，そのスレッドは近々 ACK を受信することになるため，$Tx.i$ のコミット/アボートにより解消された依存関係の情報は，先に述べた ACK 受信時の操作によって順次伝播される．

次節以降では，競合発生時，およびコミット/アボート時のそれぞれの状況において，本節で述べた各操作が適用される様子を具体例を用いて順に説明する．

4.3　競合発生時の操作

図 3 の例（$n = 3$）を用いて，まず競合発生時における動作を説明する．最初，各スレッドが持つ stall bits は初期化されている．時刻 t1 において，*Tx.3* は *Tx.2* から C へのアクセス要求を受信するが，すでに C にアクセス済みであるため NACK を返信する．この際，*thr.3* の持つ stall bits のうち，自身に対応するビットをセットしたものである 100 が NACK に付加される．これを受信した *thr.2* は，自身の持つ stall bits 000 と，NACK に付加されていたビット列との論理和をとることで，新しい stall bits 100 を得る．

次に時刻 t2 において，*Tx.2* は B に対する *Tx.1* のアクセス要求に NACK を返信するが，この際にも同様に，自身の持つ stall bits に対して自身に対応するビットをセットした列 110 を付加する．その後 *Tx.3* が A にアクセスしようとした際，*thr.1* からビット列 111 が付加された NACK を受信する（t3）．これを使用して自身の stall bits を更新すると，111 となり，自身を表す左端のビットが立っていることが確認できる．このように，自身の stall bits 内の自身に相当するビットがセットされているときに，デッドロックの発生が検出できる．

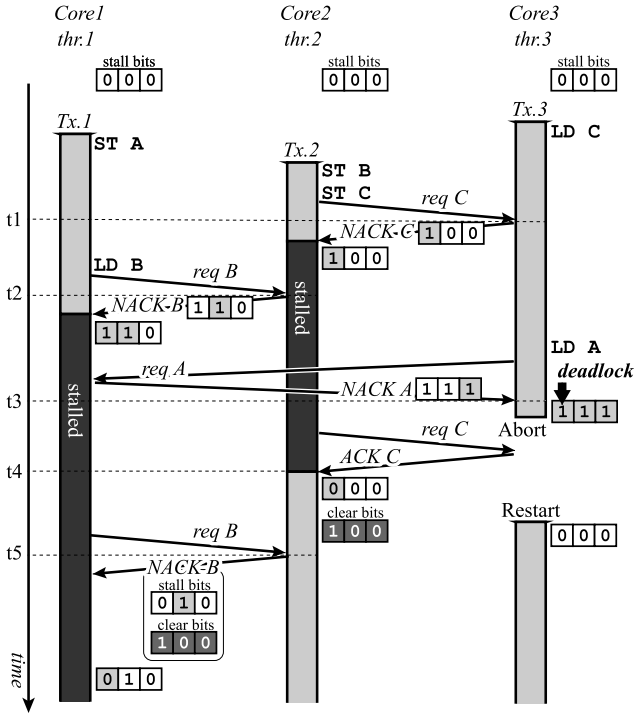

図 3　ビット列の伝播によるデッドロックの検出

Fig. 3　Detecting a deadlock by transferring dependency bitmaps.

4.4　コミット/アボート時の操作

トランザクションがコミット/アボートされた場合，競合が解決し，一部依存関係が解消されるため，各スレッドが保持する stall bits 内の対応するビットを正しく修正する必要がある．図 3 の例において，このビット修正がどのように行われるか説明する．

時刻 t3 においてデッドロックを検出した *thr.3* は，そのデッドロックに関与しているトランザクションのうち 1 つを選択し，アボートさせる．いま，デッドロック検出者自身が実行中のトランザクションをアボートさせると仮定すると．*Tx.3* がアボートされ，*thr.3* の持つ stall bits はすべてクリアされ初期化される．

一方，*thr.2* はアドレス C にアクセス可能となり，ACK を受信する．これにより *Tx.2* は *Tx.3* との競合が解決されたことを知り，自身の持つ stall bits を clear bits として記憶し，また stall bits はすべてクリアする（t4）．この結果，*Tx.2* をストールさせているトランザクションはなくなったため，*Tx.2* はストールから復帰する．

しかし *thr.1* の持つ stall bits には依然，*thr.3* への依存が記憶されている．これは *thr.2* を介して間接的に存在した依存であり，*Tx.3* のアボートにより解消されているため，クリアする必要がある．さて，*thr.1* はストール中は

定期的にアドレス B に対する要求を *thr.2* に送信し続けている．そこで *thr.2* は clear bits の定義後，その要求に対する NACK に，通常の stall bits に加え clear bits も付加して送信する（t5）．これを受信した *thr.1* は，自身の stall bits と受信した stall bits の論理和をとる通常の操作に加え，受信した clear bits に対応するビットを降ろす操作を行う．このようにして依存の解消情報が伝播され，*thr.1* は *thr.3* との依存が解消されたことを知ることができる．

4.5　アボート対象の選択

デッドロック検出者が自身のトランザクションをアボートする場合を除き，3.3 節で述べたような指針でアボート対象を選択するためには，デッドロックに関与しているトランザクションの情報を収集する必要がある．これは，デッドロック検出後に，各トランザクションから送信される NACK に自身の競合数や実行開始時刻の情報を付加し，このやりとりが，デッドロックの環を 1 周するまで待つようにすることで可能である．

なお競合数は，自コアが実行するスレッドによってストールさせられている，他スレッドを実行中のコア数として定義できる．これは，n bit のビットマップにより自身が NACK を返信した相手を記憶しておくことで，そのビットマップ中でセットされているビット数として知ることができる．したがって，競合数を記憶するためにプロセッサ全体で必要となるビットマップ容量は，stall bits および clear bits と同じく n^2 bit となる．

ここで，たとえばトランザクションの開始時刻の比較によりアボート対象を選択する場合を考える．図 3 の時刻 t3 において *thr.3* がデッドロックを検出するが，まだアボートを行わないでいると，*thr.2* から再び C へのアクセス要求が届く．これに NACK を返信する際，自身の開始時刻情報を付与する．これを受信した *thr.2* はデッドロックの発生を知る．しばらくすると *thr.1* から *thr.2* に対し B に対するアクセス要求が届くため，*thr.2* は NACK を返信する際に同様に自身の開始時刻情報をこれに付与する．これを繰り返すことで，開始時刻情報が付与された NACK がデッドロック検出者である *thr.3* に最終的に到達し，*thr.3* は蓄積された情報を用いてアボート対象を選択のうえ，対象スレッドにアボート要求を送信すればよい．

しかしこの NACK の循環待ちは，特に多くのスレッドがデッドロックに関与している場合，大きなオーバヘッドになる可能性があるため，アボート対象の適切な選択によって有意な性能向上が見込めない限り，採用することは困難であると考えられる．本稿では，アボート対象の選択が性能に与える影響についても次章で評価，考察する．

表 1　シミュレータ諸元
Table 1　Simulation parameters.

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

表 2　各ベンチマークにおける削減サイクル数
Table 2　Reduced execution cycles.

		提案手法 (D)		参考モデル (D_C)		(D_A)	
bench	/thrs	最大	平均	最大	平均	最大	平均
GEMS	/8	13.9%	6.6%	13.9%	6.8%	1.5%	0.7%
	/16	19.2%	5.2%	18.6%	5.0%	1.8%	0.3%
	/31	31.5%	7.3%	28.2%	6.8%	1.4%	−3.0%
SPLASH-2	/8	1.1%	−0.1%	2.9%	0.6%	0.8%	0.2%
	/16	4.8%	2.7%	4.1%	2.2%	−0.1%	−0.3%
	/31	16.5%	6.6%	16.5%	6.4%	−0.9%	−2.8%

5.　評価

5.1　評価環境

これまで述べた拡張を LogTM に実装し，シミュレーションによる評価を行った．評価にはトランザクショナルメモリの研究で広く用いられている Simics [9] 3.0.31 と GEMS [10] 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，また 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．プロセッサ構成は 32 コアの SPARC V9 とし，OS は Solaris 10 とした．**表 1** に詳細なシミュレーションパラメータを示す．

評価対象のプログラムとしては，GEMS 付属の microbench から btree，contention，deque，prioqueue，SPLASH-2 [11] から cholesky，radiosity，raytrace の計 7 種のベンチマークプログラムを用い，それぞれのプログラムを 8，16 および 31 スレッドで実行した．

5.2　評価結果

評価結果を**表 2** および**図 4** に示す．図 4 中の凡例はサイクル数の内訳を示しており，non-trans はトランザクショ

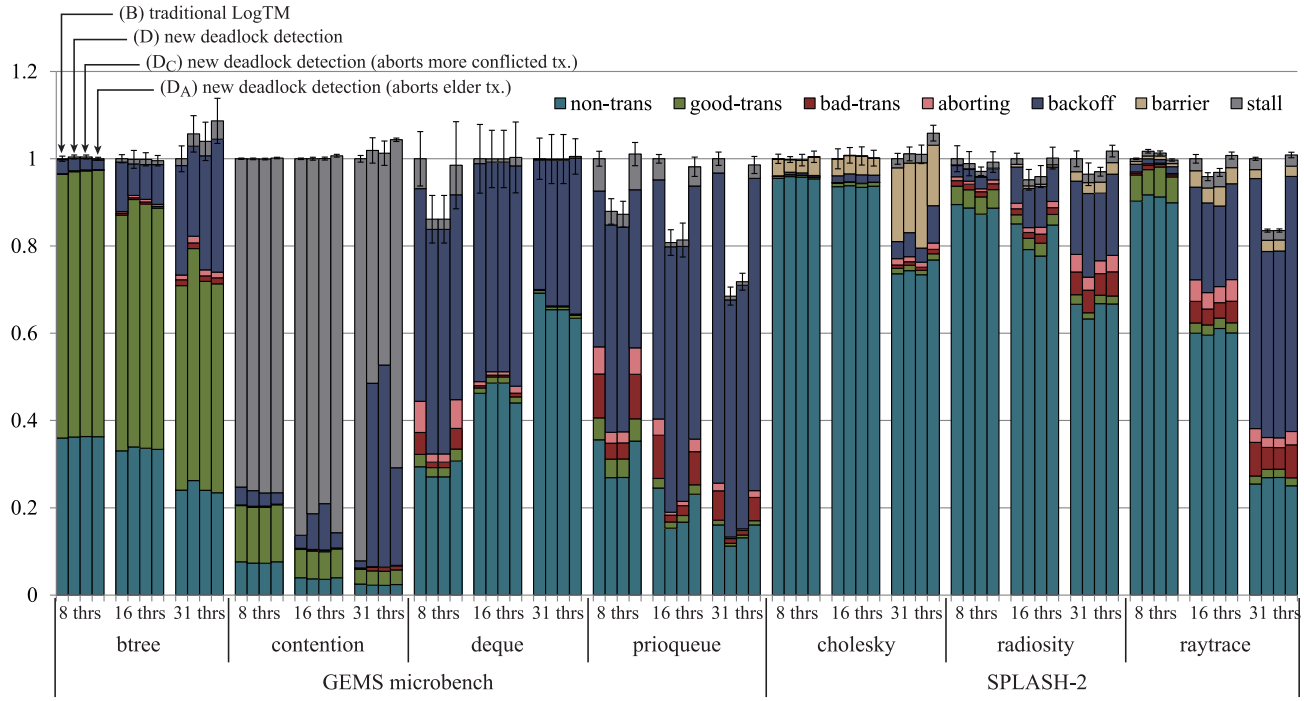

図 4　各プログラムの実行に要したサイクル数比
Fig. 4　Ratio of execution cycles.

ン外の実行サイクル数，good-trans は結果的にコミットされたトランザクション内の実行サイクル数，bad-trans は結果的にアボートされたトランザクション内の実行サイクル数，aborting はアボート処理に要したサイクル数，barrier はバリア同期，stall はストールに要したサイクル数をそれぞれ示している．

なお，アボート直後にトランザクションを再開してしまうと，同じ競合の再発により他トランザクションの実行を妨げる可能性がある．このため LogTM では，アボート後から再実行開始までランダム時間待機する機能を備えている．この待機時間はアボートが繰り返されるごとに指数関数的に増大するよう設定されており，この機能を exponential backoff と呼ぶ．凡例の backoff はこの待機時間の総和を表している．この backoff を設けない場合，アボート直後の再実行時に同様の競合パターンが発生し続けることでトランザクションが繰り返しアボートされ，ライブロック状態に陥る場合があることを確認している．

これらの各内訳に対する効果としては，まず本手法によりアボートの発生を抑制することで，アボート処理自体に要するオーバヘッドである aborting はもちろんのこと，bad-trans も削減されることが期待できる．一方 stall に関しては，3.2 節で述べたように局所的には増大する可能性があるが，スケジューリングが効率化することで全体では削減される可能性もある．また，ある単一のトランザクションがコミットに到達するまでに直面するアボートの機会が減少することで，アボートの繰り返しが発生する機会も減少し，結果として backoff が削減される可能性もある．

図 4 中では，各ベンチマークプログラムと前述した実行スレッド数との組合せによる結果が，各 4 本のグラフで表されている．4 本は左から順に，それぞれ

(B)　既存モデル（ベースライン）
(D)　デッドロック検出後，検出者自身がアボートする提案モデル
($\mathbf{D_C}$)　競合数の最も多いものをアボートする参考モデル
($\mathbf{D_A}$)　実行開始時刻の最も遅いものをアボートする参考モデル

の実行に要した総サイクル数を表しており，各サイクル数は (B) を 1 として正規化している．ただし (D_C) および (D_A) は，アボート対象選択アルゴリズムを変更することの効果を (D) との比較から検証することを目的とし，デッドロック検出後に各トランザクションの情報を収集するために要するコストがゼロであると仮定して計測した参考モデルである．これらの参考モデルが情報収集のために要するオーバヘッドについては，5.3 節で概算し議論する．

なお，マルチスレッドの動作シミュレーションでは性能のばらつきを考慮する必要があるため [12]，それぞれの評価結果は 10 回の試行による平均値で示しており，さらに 95%の信頼区間をエラーバーで示している．

結果を見ると，31 スレッド実行では一部性能が低下しているものもあるが，おおむね性能が向上しており，デッドロック検出者自身をアボートさせる提案モデル (D) では，ほとんどの場合において性能が向上した．一方，競合者数の多いトランザクションをアボートさせる参考モデル (D_C) も既存モデル (B) に対しては多くの場合速度向上し

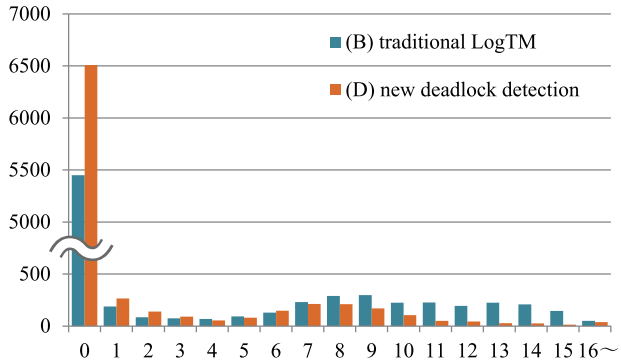

図 5　prioqueue/31 スレッドにおけるアボート繰返し回数別トランザクション実行数

Fig. 5　Distribution of abort repetition with prioqueue/31thrs.

ているものの，(D) に対しては目立った速度向上が得られておらず，情報収集のためのオーバヘッドも考慮すると，提案モデル (D) が優れているといえる．また，開始時刻の遅いトランザクションをアボートする，既存モデルと同じ方針による参考モデル (D_A) では，速度低下する場合が多く見られ，速度向上する場合においても僅かな向上幅にとどまることが分かった．以下，各ベンチマーク別に詳細な検証と考察を行う．

GEMS microbench

図 4 より，まず GEMS microbench の結果を見ると，提案モデル (D) において deque，prioqueue では aborting，bad-trans，stall が削減されており，期待どおりアボート発生の抑制が性能向上に寄与していることが分かる．結果として，31 スレッドでは最大 31.5%，平均 7.3%の実行サイクル数が削減された．

特に，prioqueue を 31 スレッドで実行した場合（以下，prioqueue/31thrs と表記）では，backoff サイクルが大きく削減された．この原因を探るため，prioqueue/31thrs において，各トランザクションがコミットまでにアボートをそれぞれ何回繰り返したかを調査した．**図 5** はトランザクション実行をアボート繰返し回数で分類したヒストグラムであり，横軸がアボートの繰返し回数，縦軸はそのトランザクション実行数となっている．なお LogTM に実装されている exponential backoff では，アボート繰返し回数に応じて待機時間が増大してゆくが，アボート繰返し回数が 16 回を超えるとほぼ定数時間の待機時間が設定されるため，16 回以上のものについてはまとめて掲載している．

この結果より，提案手法によりアボートの発生が抑制されたことで，提案モデル (D) では既存モデル (B) に比べ，コミットまでに要するアボート繰返し回数の分布がより小さい値に偏っていることが分かる．特にアボートを 10 回以上繰り返してしまう場面の発生を大きく抑制できていることが見てとれ，これが backoff サイクルの削減に寄与したと考えられる．

一方で，prioqueue/31thrs における両手法のアボート繰返し回数の最大値を調査したところ，既存モデル (B) では 17 回にとどまっていたのに対し，提案モデル (D) では 21 回まで増加してしまっていた．提案手法ではアボートの発生を抑制するため，ストールにより待機中のトランザクションが増加する可能性があることはすでに述べた．ここで，あるトランザクションが競合の発生しやすいアドレスにアクセス済みの状態でストールするような特殊な状況において，当該アドレスへアクセスしようとする他トランザクションを繰り返しアボートさせる状況が発生する．これが最大アボート繰返し回数を増大させた原因であると考えられる．これが性能低下を引きおこす可能性もあるため，今後は backoff アルゴリズムの改良も検討していく必要がある．

表 3　btree/31 スレッドにおけるトランザクション別アボート回数
Table 3　The number of aborts of each transaction with btree/31thrs.

btree/31thrs	命令数	(B)	(D)
$Tx(0)$	多い	2,199.6	2,305.5
$Tx(1)$	少ない	1,037.3	85.3

次に btree/31thrs の結果を見ると，提案モデル (D) では数%ではあるが性能が低下してしまっている．btree は，大きい（構成命令数の多い）トランザクションと，小さいトランザクションの 2 つを含んでいるプログラムであるが，これらそれぞれで発生した平均アボート回数を調査した結果を**表 3** に示す．

この結果から，規模の小さい $Tx(1)$ のアボート回数は大きく削減されていることが分かる．一方で，大きいトランザクション $Tx(0)$ のアボート回数が増加してしまっている．btree/31thrs における (D) の速度低下は good-trans の増大によるところが大きいが，これはトランザクション内におけるキャッシュミスの増大が原因である．トランザクション開始直後にアクセスされたアドレスは，トランザクションが早期にアボートされると再実行時にもキャッシュヒットする場合が多いが，アボートが抑制され遅らせられることで，そのようなアドレスに対し再実行時のキャッシュミスが増えてしまう．これは命令数を多く含むトランザクションでより顕著となる．このように再実行されたトランザクションが結果的にコミットされても，そのトランザクションに要したサイクル数が増大してしまう場合があることが分かった．

また，contention/31thrs では stall サイクルが大きく削減されたが，同時に backoff サイクルが増加したため，既存モデル (B) に対する性能向上は得られなかった．提案モデル (D) における contention/31thrs の平均アボート回数および平均最大アボート繰返し回数を**表 4** に示す．この結果から，平均アボート回数が減少する一方で同一トランザクションのアボートを繰り返す回数が増えてしまったため

表 4　contention/31 スレッドにおける平均アボート回数および最大アボート繰返し回数

Table 4　The average/maximum abort repetition with contention/31thrs.

contention/31thrs	(B)	(D)
平均アボート回数	549.2	361.1
最大アボート繰返し回数	13.8	21.0

に，backoff サイクルが増加したことが分かる．

次に参考モデルについては，全体的に (D_A) では既存モデル (B) に比べて性能が向上したものの，提案モデル (D) に比べてアボート回数が増加したために，サイクル削減率の悪化が目立った．この原因を調査したところ．主に starving writer [13] と呼ばれる競合パターンに陥ってしまうことによるものであることが分かった．これは，あるトランザクションのライトリクエストが，複数のトランザクションによる同アドレスへのリードリクエストの交互発生により飢餓状態となってしまう競合パターンである．よってアクセスのリード/ライト種別を考慮したアボート対象選択手法も今後検討していく必要がある．

また，参考モデル (D_C) に関しては 31 スレッドで最大 28.2%，平均 6.8%サイクル削減率が向上し，並行実行スレッドの増加によりサイクル数が削減されることを多くのプログラムで確認した．しかし，prioqueue を 16 および 31 スレッドで実行した場合には，提案モデル (D) と同等の性能にとどまった．これは，競合しやすいトランザクションがアボートされることで，実行再開後も再度他のトランザクションとの間で競合が発生してしまったためである．したがって，条件次第では競合相手数の多いトランザクションを優先し，早くコミットさせる手法をとる必要があると考えられる．

SPLASH-2

まず raytrace/31thrs は，GEMS ベンチマークの prioqueue/31thrs と同様の傾向を見せていることが分かる．期待どおり提案モデル (D) において，bad-trans，aborting に加えて backoff が大きく削減できており，16.5%のサイクル数削減率を達成した．

次に radiosity では，提案モデル (D) においてわずかではあるが aborting，bad-trans が削減されたことから，こちらも期待どおりアボートが抑制されたことが分かる．また，non-trans の削減が性能向上に寄与していることも確認できる．これは，false sharing による競合の誤検出を低減できたことが原因であると考えられる．2.2.2 項でも述べたように，HTM では一般に，競合の検査をキャッシュライン単位で行う．このため，複数トランザクションがそれぞれ異なるアドレスにアクセスした場合でも，それらが同一キャッシュライン上に存在していた場合，競合として検出されてしまい，不必要なストールが発生する．なお，通常は共有変数にアクセスすることのない，トランザクション外の処理を実行中のスレッドであっても，トランザクション内を実行中である他スレッドとの false sharing により，NACK を受信しストールする場合がある．これが non-trans が増大してしまう主な原因の 1 つである．

提案手法ではアボートの発生を抑制することでトランザクションの再実行やロールバック回数が削減されるが，これがトランザクションの早期コミットに寄与した場合，トランザクション外を実行中のスレッドが，トランザクション内を実行中のスレッドとの false sharing によりストールさせられる機会が減少することで，結果として non-trans が削減されたと考えられる．

一方，cholesky では既存モデル (B) に対する性能向上は得られなかった．これは，プログラム実行中に発生したデッドロックのほとんどが 2 者間におけるものであったため，結果的に提案モデル，参考モデルの両者において，possible_cycle flag を用いた既存の競合解決手法に近い動作となったためであると考えられる．

参考モデルについては，まず競合相手数を比較するモデル (D_C) の結果を見ると，cholesky は競合の少ないプログラムであるため競合相手数に差が生まれにくく，目立った効果は得られていない．一方 radiosity では有意に高速化されているが，既存モデルにおいてはリードアクセス許可を待つ多数のトランザクションが 1 つのライトアクセス済トランザクションにストールさせられている状況が存在しており，これを解決できたことによる効果が大きい．

一方，トランザクション開始時刻を比較する参考モデル (D_A) では，ほとんどの場合において性能が悪化してしまった．この原因を調査したところ，cholesky，radiosity の両方において，GEMS 同様 starving writer の発生によるところが大きいことが分かった．これらのプログラムは規模の小さいトランザクションを多く含んでいるが，starving writer の発生により，本来であればすぐにコミットに至るこれらのトランザクションがコミットまでに長いサイクルを要するようになっていた．さらに non-trans も増加していることから，このトランザクションの処理の遅れが，トランザクション外におけるキャッシュアクセスとの false sharing を増大させていると考えられるため，今後これらを詳細に調査し，対策を検討する必要がある．

5.3　アボート対象選択コスト

本節では，4.5 節で述べたアボート対象を選択する際に発生するオーバヘッドがどれだけ発生し，それが全体の性能にどの程度影響するかについて考察する．

デッドロックに関与しているスレッド数を N，リクエスト送信から NACK 受信までに要するサイクル数を T，NACK 受信からリクエスト再送までの待ち時間を Δ とすると，情報収集のためにかかる最大サイクル数は $N \times (T + \Delta)$

として概算できる．

ここで，参考モデル (D_C) が提案モデル (D) よりも有意に性能向上している radiosity/8thrs において，これらの値がどの程度になるか調査した．まず N については，発生した全デッドロックの 99.5%が 2 者間によるものであり，残りの 0.5%もすべて 3 者間によるものであった．このことから，$N \simeq 2$ であり，N が大きな値をとることはごく稀であることが分かった．また計測の結果，T，Δ の最大値はそれぞれ約 80 cycle，約 10 cycle であった．よって，デッドロック発生後に情報収集にかかるコストの最大値は，$2 \times (80 + 10) =$ 約 180 cycle となる．また radiosity/8thrs では，最も実行時間の長いスレッドで発生するデッドロック回数は約 900 回であったため，アボート対象の選択に要するオーバヘッドの総計は $180 \times 900 =$ 約 16 万 cycle となる．

一方 radiosity/8thrs の総実行サイクル数は約 3,300 万 cycle であるため，このオーバヘッドが総実行サイクル数に占める割合は約 0.5%と比較的小さく，これを上回るサイクル数削減が見込めるアボート選択アルゴリズムが採用できれば，全体性能を向上させることができる余地があることが分かった．よって今後は，より適切なアボート対象選択アルゴリズムを検討することで，本手法をさらに改良していく必要があると考える．

6. おわりに

本稿では，既存のハードウェアトランザクショナルメモリである LogTM を拡張し，デッドロックの検出を厳格化することでアボートの過剰発生を防ぎ，高速化する手法を提案した．拡張した LogTM では，各スレッドが自身をストールさせているスレッドを表現したビット列を保持し，これを競合時に送信される NACK とともに伝播させることで，デッドロックを検出できるようにした．

提案手法の有効性を確認するため，GEMS microbench および SPLASH-2 ベンチマークプログラムを用いて評価した結果，possible_cycle フラグを用いた既存モデルに比べて最大 31.5%の実行サイクル数が削減されることを確認した．特に，アボート発生の抑制による，ロールバック時の書き戻しコストや，アボートによって無駄となってしまう実行サイクル数，アボートから再実行までの間に設定される backoff 時間などの削減が性能に寄与していた．

また，デッドロック検出時にアボート対象を選択する基準が性能に与える影響の評価もあわせて行ったところ，アボート対象の選択方針によっては性能が大きく悪化する場合があることを確認した．これは同一トランザクションによるアボートの繰返しに起因する backoff サイクルの増大や，starving writer と呼ばれる競合パターンの影響によるものであった．一方でアボート対象を選択するために必要となるオーバヘッドはわずかであることも確認した．

今後の課題としては，まず，より適切なアボート対象選択方針の検討があげられる．これは，本稿で検討した 2 つの方法が，大きな性能向上が得られるものではなかったためである．また，本稿中でも述べた false sharing に起因する不必要な競合については，対策手法を検討し，現在評価中である．さらに，starving writer を含むさまざまな競合パターンの発生を調査，分類し，それらに適切に対応可能なトランザクションスケジューリング手法についても考察していく必要があると考えている．

参考文献

[1] Herlihy, M. and Moss, J.E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Annual Int'l Symp. on Computer Architecture*, 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. 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. 10th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp.5–17 (2002).

[4] Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M. and Wood, D.A.: Supporting Nested Transactional Memory in LogTM, *Proc. 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. Int'l Symp. on Parallel and Distributed Processing* (*IPDPS*), pp.1–11 (2008).

[8] Makki, K. and Pissinou, N.: Detection and Resolution of Deadlocks in Distributed Database Systems, *Proc. 4th Int'l Conf. on Information and Knowledge Management* (*CIKM'95*), pp.411–416 (1995).

[9] 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).

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

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

[12] Alameldeen, A.R. and Wood, D.A.: Variability in Architectural Simulations of Multi-Threaded Workloads, *Proc. 9th Int'l Symp. on High-Performance Computer Architecture* (*HPCA'03*), pp.7–18 (2003).
[13] Bobba, J., Moore, K.E., Volos, H., Yen, L., Hill, M.D., Swift, M.M. and Wood, D.A.: Performance Pathologies in Hardware Transactional Memory, *Proc. 34th Annual Int'l Symp. on Computer Architecture* (*ISCA'07*), pp.81–91 (2007).

堀場 匠一朗　（学生会員）

1989 年生．2012 年名古屋工業大学工学部情報工学科卒業．現在，同大学大学院工学研究科創成シミュレーション工学専攻博士前期課程在籍．計算機アーキテクチャ，マルチコア・プロセッサ等に興味を持つ．

江藤 正通　（学生会員）

1987 年生．2011 年名古屋工業大学工学部情報工学科卒業．現在，同大学大学院工学研究科創成シミュレーション工学専攻博士前期課程在籍．計算機アーキテクチャ，並列処理等に興味を持つ．

浅井 宏樹

1988 年生．2010 年名古屋工業大学工学部情報工学科卒業．2012 年同大学大学院工学研究科創成シミュレーション工学専攻博士前期課程修了．同年（株）デンソー入社．計算機アーキテクチャ，並列処理等に興味を持つ．

津邑 公暁　（正会員）

1973 年生．1996 年京都大学工学部情報工学科卒業．1998 年同大学大学院工学研究科情報工学専攻修士課程修了．2001 年同大学院情報学研究科博士後期課程学修認定退学．同年同大学院経済学研究科助手．2004 年豊橋技術科学大学工学部助手．2006 年名古屋工業大学大学院工学研究科助教授．2007 年同准教授．博士（情報学）．プロセッサアーキテクチャ，並列処理，脳型情報処理等に関する研究に従事．ACM，IEEE-CS 各会員．

松尾 啓志　（正会員）

1960 年生．1985 年名古屋工業大学大学院修士課程修了．1989 年同大学院博士後期課程修了．同年同大学工学部電気情報工学科助手．1993 年同講師．1995 年同助教授．2003 年同教授．2004 年同大学工学部情報工学科教授．工学博士．計算機工学，分散協調システムに関する研究に従事．IEEE，人工知能学会，電子情報通信学会各会員．