LogTMにおける依存関係情報を用いたアボート抑制手法

堀場 匠一朗[†1] 江藤 正通[†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において，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し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を検討する．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，最大 31.5% の高速化を確認した．

Reducing Aborts on LogTM by Considering Deadlock Loops

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

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.

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伴う並列性の低下や，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のは難しく，プログラマにとって必ずしも利用し易いものではない．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．

このトランザクショナル・メモリのハードウェア実装のひとつである 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，本研究の対象となるトランザクショナル・メモリ及び LogTM の基本的概念とその関連研究について述べる．

†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

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，部分的に実行されてはいけない．

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，それまでの結果を全て破棄する．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及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全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一つに，LogTM 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及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，そのアドレスをこのログに退避する．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，既に全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 ビット及び 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及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，キャッシュコヒーレンス要求を他の各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ている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 (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．

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得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 の作成数制限を無くす手法も提案している．しかし，LogTM で標準的である 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点．評価結果に対する考察が少なく，改良手法のベースモデルである Williullah らの手法[7] による効果も含めて評価しているため，提案手法による性能向上への寄与が明らかでない点など，評価手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一つのみであり，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．

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，当該アドレスは既に *Tx.2* がアクセス済であるため，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）．

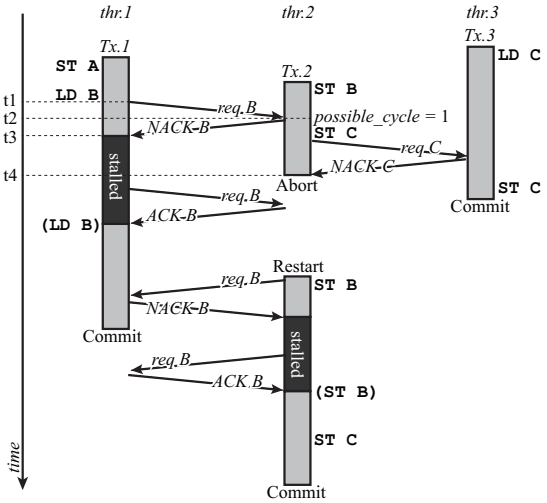

図 1 possible_cycle フラグにおける問題点

次に *Tx.2* が `ST C` を実行しようとするが，既に *Tx.3* によりアクセス済であるため，NACK を受信する．この時，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 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

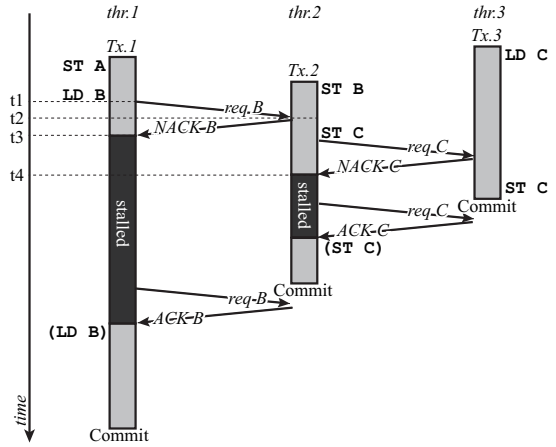

図 2 アボート条件を厳格化した動作モデル

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列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. 依存情報の伝搬とデッドロック検出

本章では提案手法の具体的な実装方法を述べる．

4.1 ハードウェア拡張

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，コア数および最大同時実行スレッド数は n であるとする．

stall bits 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番号を記憶する，n bit のビットマップ．

clear bits 依存関係の解消されたスレッド番号を記憶する，n bit のビットマップ．

提案手法では，複数のトランザクション間に依存関係の環が発生したときにデッドロックとして検出する．このために各スレッドは，自身のトランザクションを直接・間接的にストールさせているスレッド番号を stall bits として記憶する．そして新たな競合が発生した際，この stall bits を NACK に付加して送信することで，依存情報を伝搬する．

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，伝搬すべきクリアビット情報を clear bits として一時的に記憶しておき，次回の NACK 送信時にその情報を付加することで，解決された依存関係情報を伝搬する．

次節以降で，競合発生時，およびコミット/アボート時の操作を，順に具体例を用いて説明する．

4.2 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* は既に C にアクセス済であるため，*Tx.2* に 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立てたものである 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 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 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立てた列 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立っているときに，デッドロックの発生を検出する．

4.3 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，

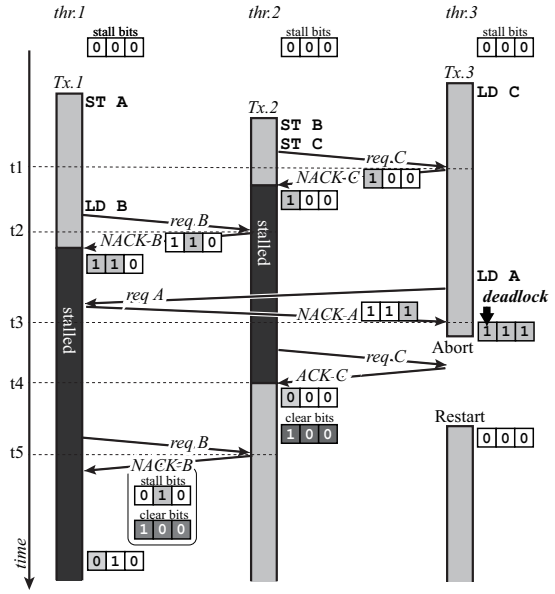

図 3 ビット列の伝搬によるデッドロックの検出

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一つを選択し，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 は全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 は全てクリアする（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．これにより，依存の解消情報が伝搬される．

4.4 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一周するまで待つようにすることで可能である．

しかしこれは，特に 3 者以上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と考えられるため，アボート対象の適切な選択によって大き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評価，考察する．

表 1 シミュレータ諸元

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

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[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，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[10] 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 及び 31 スレッドで実行した．

5.2 評価結果

評価結果を**図 4** および**表 2** 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 はトランザクション外，good-trans は結果的にコミットされたトランザクション内，bad-trans は結果的にアボー

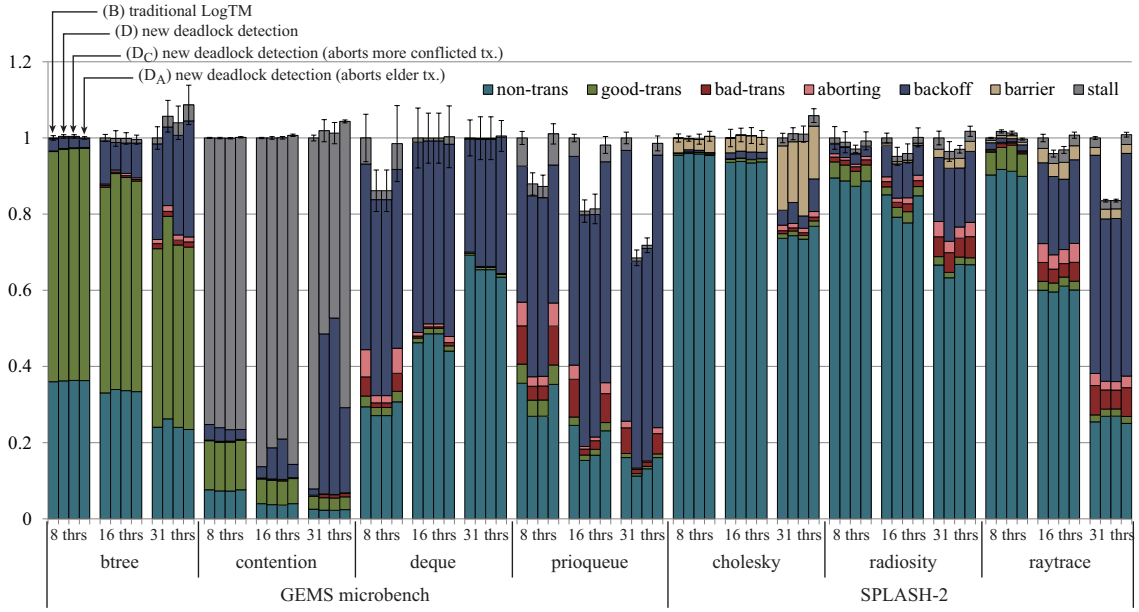

図 4 各プログラムの実行に要したサイクル数比

表 2 各ベンチマークにおける削減サイクル数

		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%

トされたトランザクション内，aborting はアボート処理，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に設定されており，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．

図中では，各ベンチマークプログラムと前述の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

(D_C) 競合数の最も多いものをアボートする参考モデル

(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．

なお，マルチスレッドの動作シミュレーションでは性能ばらつきを考慮する必要があるため[11]，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．

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，概ね性能が向上しており，特にデッドロック検出者自身がアボートする提案モデル (D) では，ほとんどの場合において性能が向上した．一方，競合者数を比較する参考モデル (D_C) は既存モデル (B) に対しては多くの場合速度向上しているものの，(D) に対しては目立った速度向上が得られておらず，情報収集のためのコストも考慮すると，(D) が優れていると言える．また，開始時刻の遅いトランザクションをアボートするという，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留まった．

以下，各ベンチマーク別に詳細な検証を行う．

GEMS microbench

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，アボートの発生

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

btree/31thrs.	命令数	(B)	(D)
$Tx(0)$	多い	2199.6	2305.5
$Tx(1)$	少ない	1037.3	85.3

表 4 contention/31 スレッドにおける平均アボート回数及び最大アボート繰り返し回数

contention/31thrs.	(B)	(D)
アボート回数	549.2	361.1
最大アボート繰り返し回数	13.8	21.0

が抑制されたことが分かる．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．

しかし，btree/31 スレッド ではどのモデルにおいても性能が低下してしまっている．btree は，大きい（命令を多く含む）トランザクションと，小さいトランザクションのふた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．しかし，アボート対象が命令数の多いトランザクション $Tx(0)$ に偏ることで，アボート時にメモリ及びレジスタの状態をトランザクション開始時の状態へ復元する，ログの書き戻しコストが増大したことにより，速度低下を引き起こしたと考えられる．

また，contention 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，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 の平均アボート回数及び平均最大アボート繰り返し回数を**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に，backoff サイクルが増加したと考えられる．

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 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，並列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 及び 31 スレッドで実行した場合には，提案モデル (D) と同等の性能に留まった．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 では，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ひとつである．

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 では有意に高速化しているが，既存モデルにおいては read アクセス許可を待つ多数のトランザクションが

ひとつの write 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．

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 とともに伝搬させることで，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再実行サイクル数の削減が性能に寄与していた．一方で，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も確認された．これは同一トランザクションによるアボートの繰り返しに起因する 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．

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挙げられる．これは，本稿で示したふたつの方法が，対象選択のための情報収集コストに見合う性能向上が得られるものではなかったためである．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する必要があると考えている．

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