Vol.2014-ARC-211 No.1
2014/7/28

待機アルゴリズムの改良による トランザクショナルメモリの高速化

山田 遼平[1]　橋本 高志良[1]　津邑 公暁[1]

概要：マルチコア環境では，共有変数へのアクセス調停のためにロックを用いることが一般的である．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ため，これに代わる並行性制御機構としてトランザクショナルメモリ（TM）が提案されている．この機構のハードウェア実装であるハードウェア・トランザクショナルメモリ（HTM）では，アクセス競合が発生しない限りトランザクションが投機実行される．HTM では投機実行が失敗した場合，再び競合が発生することをを防ぐため，トランザクションの再実行までに待機時間を設定するアルゴリズムが採用されている．しかし，既存の待機アルゴリズムでは適切な待機時間を設定できていないため，再び競合が発生して投機実行の失敗が繰り返されることで，HTM の性能が著しく低下してしまう場合がある．本稿では，この待機アルゴリズムを改良し，トランザクションの実行状況に応じた待機処理を行うことで HTM を高速化する手法を提案する．シミュレーションによる評価の結果，提案手法により最大 59.9%，16 スレッドで平均 11.2%の高速化を確認した．

1. はじめに

マルチコア環境における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調停のための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にはロック操作のオーバヘッドに伴う並列性の低下やデッドロックの発生など多くの問題が起こりうる．さらにプログラマにとって適切なロック粒度の設定は困難であり，この機構は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（Transactional Memory: TM）**[1] が提案されている．

TM では，従来ロックで保護されていたクリティカルセクションを含む一連の命令列を，**トランザクション**として定義する．この機構では，共有リソースへのアクセスにおいて競合が発生しない限り，トランザクションを投機的に実行できるため，一般にロックを用いる場合よりも並列性が向上する．

なお，トランザクションの実行中においては，その実行が投機的であるがゆえに，共有リソースに対する更新の際には更新前の値を保持しておく必要がある（**Version Management; バージョン管理**）．また，トランザクションを実行するスレッド間において，共有リソースに対する競合が発生していないかを検査する必要がある（**Conflict Detection; 競合検出**）．TM のハードウェア実装である**ハードウェア・トランザクショナルメモリ（Hardware Transactional Memory: HTM）**では，このバージョン管理及び競合検出のための機構をハードウェアで実現することで，これらの処理を高速化している．

さて，この HTM では一般に，競合が発生した場合，開始時刻の早いトランザクションのみが処理を継続できる．したがって，開始時刻の遅いトランザクションがアボートされてから即座に再実行される場合，開始時刻の早いトランザクションと再び競合することでアボートが繰り返されてしまう可能性がある．そこで，既存の HTM ではこのようなアボートの繰り返しを抑制するため，トランザクションの再実行までに待機時間を設けるアルゴリズムが採用されている．なお，HTM では多くの場合，アボートの繰り返し回数に応じて待機時間を指数関数的に増大させる，Exponential Backoff というアルゴリズムに基づいて待機時間が設定される．そのため，アボートが繰り返されて待機時間が大きく増大した場合に，本来であれば再実行を待機しなくてもよい状況であっても待機を継続してしまう．このような状況が多く発生すると，HTM の性能が著しく低下する可能性がある．そこで，本稿ではこの待機アルゴリズムを改良し，トランザクションの実行状況に応じた待機処理を行うことで HTM を高速化させる手法を提案する．

[1] 名古屋工業大学
Nagoya Institute of Technology

2. 背景

本章では，本研究の対象である TM と，そのハードウェア実装である HTM の基本概念について述べる．

2.1 TM の基本概念

ロックに代わる並行性制御機構である TM はデータベース上で行わ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り，クリティカルセクションを含む一連の命令列を投機的に実行する．この命令列は，シリアライザビリティおよびアトミシティを満たすべき**トランザクション**として定義される．これら 2 つの要件を保証するために，TM は各トランザクション内でアクセスされるメモリアドレスを監視する．具体的には，複数のトランザクション内において同一アドレスへのアクセスが検出されると，これがトランザクションの要件を満足しない場合，**競合**として判定される．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一時的に停止する．これを**ストール**と呼ぶ．さらに，複数のトランザクションがストールした状態で，デッドロックの可能性があると判断された場合，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，トランザクション開始時点から処理を再実行する．一方，トランザクションが終了するまでに競合が発生しなかった場合，トランザクション内で更新された全ての結果をメモリに反映させる．これを**コミット**と呼ぶ．

TM で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投機的に並列実行することができる．そのため，TM は一般的にロックと比較して並列性が向上する．なお，TM で行われる競合検出およびバージョン管理のための操作はハードウェア上またはソフトウェア上に実装されることで実現される．ソフトウェア上に実装されたソフトウェアトランザクショナルメモリ（STM）[2] では，特別なハードウェア拡張は必要ないが，ソフトウェア処理のためのオーバヘッドが大きい．これに対し，ハードウェア上に実装された HTM では，競合を検出および解決する機構をハードウェアによってサポートしているため，STM に比べて速度性能が高い．

2.2 競合の検出

同一アドレスに対する競合を検出するため，既存の HTM ではアクセス対象のアドレスをデコードした**シグネチャ**[3] と呼ばれるビット列を用いる．このシグネチャは各コア上で，Read アドレス用，Write アドレス用のものがそれぞれ保持される．これらのシグネチャは，トランザクション内でアクセスされたアドレスの一部をデコードし，それまで保持していたシグネチャと論理和を取ることで更新される．これにより，更新されたシグネチャからはそれまでにアクセスしたアドレスに加えて，新たにアクセスされたアドレスにもアクセス済みであるという情報が分かるようになる．その後，アクセスを試みたアドレスに対して同様に一部をデコードしたものと，現在保持しているシグネチャとの論理積を取ったものが，アクセスを試みたアドレスのシグネチャと同一のデータとなった場合に競合が検出される．

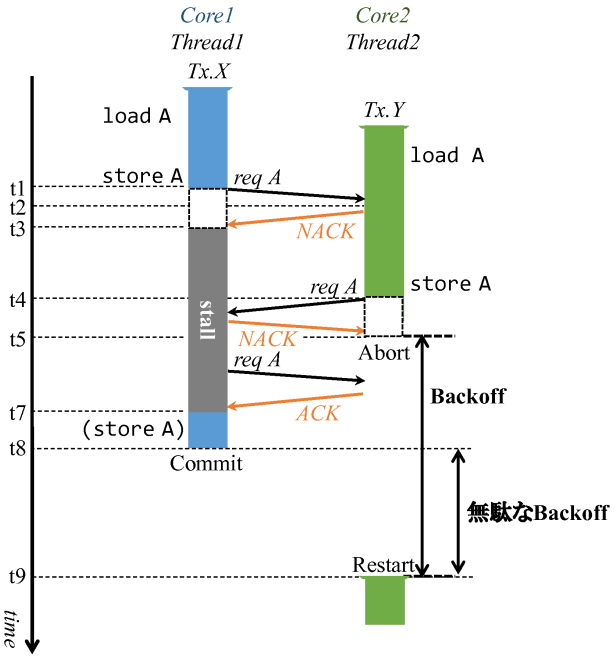

図 1　Eager 方式におけるトランザクションの競合解決

なお，HTM における競合検出方式は，そのタイミングによって以下の 2 つに大別される．

Eager Conflict Detection: トランザクション内でメモリアクセスが発生した時点で，そのアクセスに関する競合が存在していないか検査する．

Lazy Conflict Detection: トランザクションのコミットを試みる時点で，そのトランザクション内で行われた全てのアクセスに関して競合が発生していないか検査する．

これら 2 つの方式のうち，Lazy 方式ではトランザクション内で競合が発生してから検出されるまでの時間が Eager 方式と比べて長くなり，無駄な処理が増大して実行効率が悪くなるため，本研究では Eager 方式を対象とする．ここで，Eager 方式の競合検出および競合解決の動作について**図 1** を用いて説明する．この図において，*Core1* および *Core2* はコアを，*Thread1* および *Thread2* は各コア上で動作するスレッドを示している．ここで，*Thread1* と *Thread2* がそれぞれトランザクション *Tx.X*，*Tx.Y* の実行を開始しており，*Thread1* による `load A` の実行後に，*Thread2* による `load A` の実行が完了した場合を考える．ここで，*Thread1* が `store A` の実行を試みた場合（時刻 t1），*Thread2* はす

でにアドレス A に対してアクセス済みであるため，競合が検出される（t2）．この場合，Eager 方式を採用する HTM では，一般にリクエストを送信した側の *Thread1* が *Tx.X* をストールさせる（t3）．その後，*Thread2* が `store A` を実行しようとすると（t4），再び競合が発生する．この場合はリクエストを送信した *Thread2* の方が開始時刻が遅いため，*Tx.Y* をアボートする（t5）．この時，*Thread2* は *Thread1* と再び競合することを防ぐため，即座にトランザクションを再実行せず，**バックオフ（Backoff）**として設定された時間だけ待機する．そして，*Thread2* はこのバックオフが終了したのち，*Tx.Y* を再実行させる（t6）．一方，*Thread1* は *Thread2* が *Tx.Y* をアボートしたことでアドレス A にアクセス可能となるため，*Tx.X* をストール状態から復帰させる（t7）．なお，HTM では一般に，バックオフは Exponential Backoff と呼ばれるアルゴリズムを用いて決定される．このアルゴリズムでは，同一トランザクションが繰り返しアボートされることで，指数関数的に待機時間が増大する．

2.3 データのバージョン管理

前節で述べたように，HTM におけるトランザクションの投機実行では，各スレッドはデッドロックの回避のために自身の実行トランザクションをアボートし，実行結果を破棄しなければならない場合がある．このような場合に備えて，HTM ではデータのバージョン管理が行われている．これは以下の 2 つの方式に大別される．

Eager Version Management: 書き換え前の古い値を別の領域にバックアップし，更新後の値をメモリに上書きする．コミットはバックアップを破棄するだけで実現できるため高速に行うことができるが，アボート時にはバックアップされた値をメモリにリストアする必要がある．

Lazy Version Management: 書き換え前の古い値をメモリに残し，更新後の値を別領域に書き込む．書き換え前の値がメモリに残っているためアボートは高速に行うことができるが．コミット時にメモリへの値の反映が必要になる．

つまり，Lazy 方式ではアボートが高速に処理されるのに対し，Eager 方式では必ず行われるコミットが高速に処理される．また，Lazy 方式におけるコミットのオーバヘッドは削減の余地がほぼ無いのに対し，Eager 方式ではスレッドスケジューリングの改良によってアボートの発生自体を抑制することで，性能向上できる余地が大きいと考えられる．そこで本稿では，この Eager 方式を採用した HTM の実装の一つである LogTM-SE[3] をベースアーキテクチャとして採用する．

3. 優先度を用いた待機アルゴリズムの改良

本章では既存の HTM における問題点を示し，これを解決する新たな待機アルゴリズムを提案する．

3.1 既存の待機アルゴリズムの問題点

これまでに我々は Starving Writer や Futile Stall などの特定の競合パターンや，共有変数に対する特定のアクセスパターンを解決する改良手法を提案してきた [4][5][6]．しかし，これらの手法ではいずれも特定の競合やアクセスに対する性能低下は防止できるものの，これらのパターンに該当しないものでは大きな性能向上が望めないと考えられる．したがって本稿では，特定の競合パターンやアクセスパターンという局所的な視点ではなく，トランザクションの実行状況に応じた待機時間の設定という大局的な視点の手法を提案し，HTM の高速化を目指す．

さて，2.2 節で述べたように，HTM では競合が発生した際に開始時刻が遅いトランザクションをアボートすることで競合解決を図る．そして，アボートされたトランザクションはバックオフによる待機時間が経過した後，再実行される．しかし，このように待機時間を設定する方法では，再び競合することなく再実行可能となる時刻を超えて待機してしまい，バックオフによる待機時間が無駄になってしまう可能性がある．

ここで，そのような無駄な待機処理が生じる例を図 1 を用いて説明する．この例において，*Thread1* がストール状態から復帰した後，実行が進み *Tx.X* をコミットしたとする（t8）．この時，*Thread1* は既に *Tx.X* をコミット済みであるため，この時点で *Thread2* が *Tx.Y* を再実行したとしても，再び競合することはない．しかし，実際にはバックオフで設定された時間だけ再実行を待機するため，*Thread1* のコミットから大幅に遅れてトランザクションを再実行してしまう可能性がある（t9）．この場合，*Thread1* が *Tx.X* をコミットしてから *Thread2* が *Tx.Y* を再実行するまで無駄に待機していることになり，HTM の性能低下に繋がってしまう．

3.2 優先度による待機処理の改良

前節で述べた無駄な待機時間は，競合の発生するスレッド数の増加，つまり並列実行されるトランザクション数の増加につれて，より多く発生する可能性がある．そこで本稿では，各スレッドが実行するトランザクションに**優先度（Priority）**と呼ぶ値を設定し，この値を用いて待機処理を設定する手法を提案する．この優先度の値は，トランザクションの進行状況に比例して大きくなるように設定する．なお，競合が発生した場合には，各トランザクションの優先度の値を比較し，より大きな値が設定されているトラン

ザクションを優先的に継続させるようにする．これにより，残り実行時間が少ないトランザクションを優先的に実行させることができる．また，既存の exponential backoff を用いず，優先度を考慮した逐次実行を行うことで，無駄な待機時間を生じることなく，トランザクションを実行できる．

ここで，この優先度を設定する基準として，以下のパラメタを用いる．

トランザクション経過時間（T）：各スレッドがトランザクションを開始してからの経過時間．

現在までのロード・ストア回数（L，S）：各スレッドが，現在実行中のトランザクション内でロード・ストアアクセスした回数．

過去のロード・ストア回数（L_0，S_0）：各スレッドが現在実行中のトランザクションに関して，過去に実行した際にロード・ストアアクセスした回数．

これらのパラメタを用いることで優先度の値を

$$priority = \frac{1}{\frac{\alpha}{T} + \beta(S_0 - S) + \gamma(L_0 - L)} \qquad (1)$$

のように定義する．なお，α，β，γ は各パラメタの重みを示している．ここで，$(S_0 - S)$ や $(L_0 - L)$ は実行中のトランザクションが終了するまでに今後アクセスすると予測されるストアやロードの回数を示している．したがって，この値がより小さいトランザクションは大きいトランザクションと比べて早く実行が完了すると考えられるため，優先的に実行させるべきである．また，経過時間の長いトランザクションが経過時間の短いトランザクションに待たされ続けた場合，共有変数にアクセスしたまま待機することとなり，他のトランザクションと競合する可能性が高くなる．そのため，経過時間の長いトランザクションは共有リソースを開放させるため早期にコミットさせるべきである．以上により，優先度は式（1）のように定義する．そして，競合が発生した際には，式（1）を用いて算出された優先度の大小を比較することで，優先的に実行するトランザクションを決定する．

3.3 初回競合時の動作

ここで，3.2 節で定義した優先度を用いた手法を適用した場合の動作を**図 2** に示す．なお，本提案手法ではメモリアクセス時の load/store リクエストに対し算出した優先度を付加させることで，優先度の大小を比較できるようにする．この例では，2 つのスレッド *Thread1*，*Thread2* が，それまでに競合が発生していないトランザクション *Tx.X*，*Tx.Y* を実行しており，両スレッドがともに `load A` を実行済みである場合を想定している．

この状況で，*Thread1* が `store A` の実行を試みるため，優先度を付与したリクエストを送信した場合（t1），アドレス A へはすでに *Thread2* がアクセス済みであるため競合が検出され，*Thread1* は *Tx.X* をストールさせる（t2）．その後，*Thread2* が同様に `store A` の実行を試みたとすると，*Thread1* は既にアドレス A にアクセス済みであるため，競合が発生する（t3）．この時，既存の HTM では開始時刻の遅いトランザクションをアボートすることで競合解決を行うが，本提案手法ではこの時点で優先度を比較し，より値の小さい優先度が設定されているトランザクションをアボートすることで競合を解決する．この例では，*Tx.X* の優先度が 1/2，*Tx.Y* の優先度が 1/5 と算出されたとすると，優先度の値が小さい *Tx.Y* を実行する *Thread2* はトランザクションをアボートさせる（t4）．これにより，*Thread1* は *Tx.X* の実行を継続できる（t5）．また，アボート処理を行った *Thread2* を実行している *Core2* は，アボートの原因となった相手トランザクションのトランザクション ID と競合アドレスを記憶し，後述する方法により，次回以降の競合時にこれらの情報を用いてアボートを抑制する．そして，*Thread2* はバックオフによる待機を行わず，*Thread1* のトランザクションのコミットを待った後に，*Tx.Y* を再実行する．そのために，*Wakeup* メッセージを新たに定義し，*Thread1* は *Tx.X* のコミット後にこれを送信することで *Thread2* のトランザクションを再実行させる（t6）．

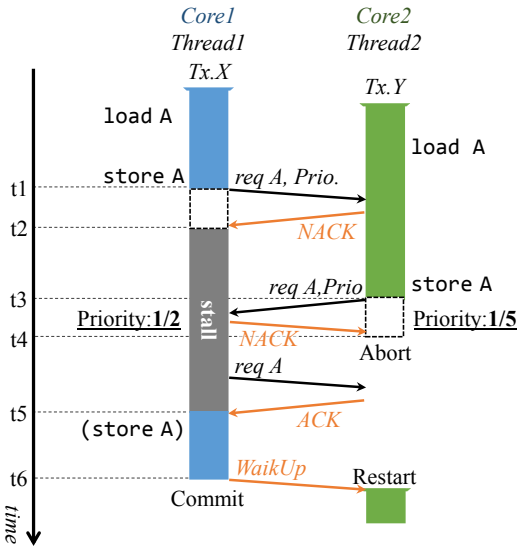

図 2 初回競合時の動作

3.4 同一トランザクションとの再競合時

3.3 節で述べた動作を行うなかで，競合が発生するトランザクションおよびアドレスが記憶される．ここで，図 2 の状況からさらに実行が進み，*Tx.Y* とアドレス A で競合したという情報を *Core1* が記憶している状態で，先ほどと同様に *Thread1*，*Thread2* が *Tx.X*，*Tx.Y* を実行している例を**図 3** に示す．ここで，*Thread1* が `load A` を実行済みである状態で，*Thread2* が `load A` の実行を試みたとする（t1）．この時，このアクセスは本来競合とはならな

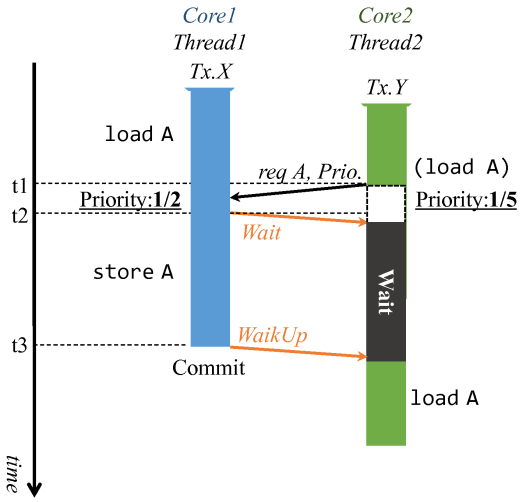

図 3　同一トランザクションとの再競合時の動作

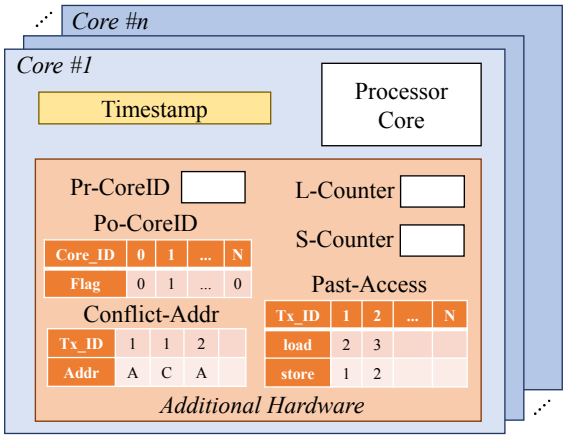

図 4　拡張したハードウェア

いが，記憶した情報から *Tx.Y* と過去にアドレス A で競合していたことが判明し，このまま実行が進んだ場合に再び競合が発生する可能性が高いと予測できる．そこで，この時点で両スレッドの実行するトランザクションの優先度を比較し，競合が発生する前に優先的に実行するトランザクションを決定する．このとき，*Tx.X* の優先度が 1/2，*Tx.Y* の優先度が 1/5 と算出されたとすると，優先度の値の大きい *Tx.X* を実行する *Thread1* の実行を継続させる．それと同時に，*Thread1* は *Thread2* に対して，実行を待機させる通知である *Wait* リクエストを送信する（t2）．なお，*Wait* リクエストはコヒーレンスプロトコルを拡張する形で新たに定義する．この *Wait* リクエストを受信した *Thread2* は *Tx.Y* の実行を待機するため，*Thread1* はその後に `store A` を実行したとしても，*Thread2* と競合することなく *Tx.Y* の実行を継続できる．その後，*Thread1* が *Tx.X* をコミットした際には，*Thread2* の wait 状態を解除する必要がある．そこで，3.3 節で定義した *Wakeup* メッセージを利用し，これを送信することで待機スレッドを再開させる（t3）．

4. 実装

本章では，提案手法を実現するために拡張したハードウェアと具体的な動作モデルについて述べる．

4.1 拡張したアーキテクチャ

既存の HTM に対して本提案手法を実装するために各コアに追加したハードウェアを**図 4** に示す．また，これらの追加ハードウェアの詳細について以下で説明する．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．

Load Counter（L-Counter）：各スレッドが，現在実行中のトランザクション内でロードアクセスした回数を記憶するカウンタ．

Store Counter（S-Counter）：各スレッドが，現在実行中のトランザクション内でストアアクセスした回数を記憶するカウンタ．

Prior Core ID（Pr-CoreID）：待機スレッドが受信した *Wait* リクエストを送信したスレッドが存在するコア ID を記憶するレジスタ．待機スレッドは，このレジスタに記憶されたコア ID のスレッドから *Wakeup* メッセージが送信されるまで実行を待機する．

Posterior Core ID bits（Po-CoreID）：各スレッドが，実行を待機させているスレッドの存在するコア ID を記憶する n-bit のビットマップ．各スレッドは，トランザクションのコミット完了時に，このビットマップの対応するビットが立っているコア ID 上のスレッドに対して *Wakeup* メッセージを送信する．

Past Access Table（Past-Access）：これまでに実行したトランザクション内で行った，ロード/ストア回数をトランザクション ID ごとに記憶する表．

Conflict Address Table（Conflict-Addr）：過去にアボートが発生した競合相手のトランザクション ID および競合アドレスを記憶する表．

なお，トランザクション経過時間については，本研究で対象とする LogTM-SE は各コアに開始時刻を保持する Timestamp と呼ばれるハードウェアがすでに存在するため，実行中の時刻とここに保持されている値の差をとることで計算される．

4.2 各ハードウェアに対する操作

前節で示した追加ハードウェアに対する操作について，3 章で述べた動作と対応させて**図 5** および**図 6** を用いて説明する．まず，図 5 において，2 つのスレッド *Thread1*，*Thread2* がそれぞれ `load A` を実行した際，ロードアクセスの回数を記憶するため L-Counter の値をインクリメントする（t1，t2）．その後，*Thread2* が `store A` の実行を試みた際には，追加ハードウェアに記憶された情報を用いて優先度を計算し，その値をリクエストに付加して送信する

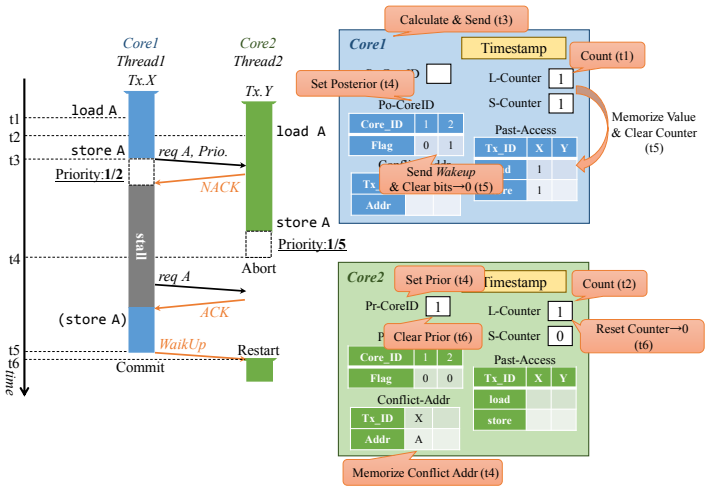

図 5 初回競合時の追加ハードウェアに対する操作

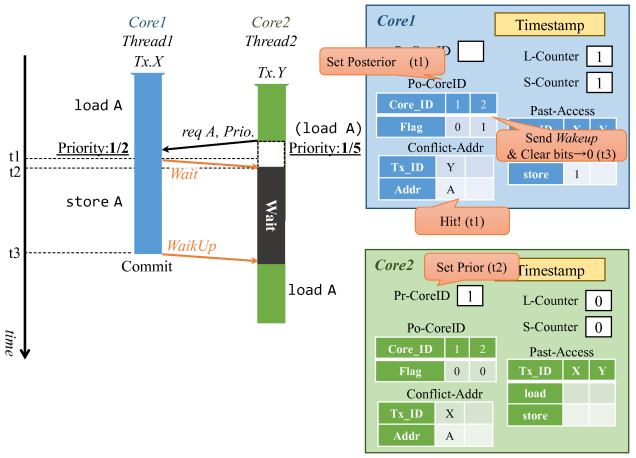

図 6 再競合時の追加ハードウェアに対する操作

(t3)．実行が進み，*Thread2* が *Tx.Y* をアボートする際には，競合相手である *Tx.X* とアドレス A を Conflict-Addr に記憶し，さらにその実行を待機している事を記憶するため，Pr-CoreID に競合した *Tx.X* を実行するコア番号である 1 を記憶する（t4）．一方，*Core1* はコミット時に *Thread2* を待機状態から復帰させる必要があるため，Po-CoreID の *Core2* に対応するビットをセットする．その後，*Thread1* が *Tx.X* をコミットする際には，*Tx.X* 内で行われたアクセス回数を記憶するため，L-Counter と S-Counter の値を Past-Access に記憶する．なお，Past-Access に過去のロード/ストアアクセス回数が既に記憶されている場合には，記憶されている値と L-Counter，S-Counter の値の算術平均を用いて更新する．これは，トランザクション内で分岐命令により実行命令列が変化した事により，各アクセス回数が急激に変化した際，今後のアクセス回数予測の精度が低下することを防ぐためである．また同時に，自身のコミットを待機しているスレッドを復帰させるため，Po-CoreID のビットがセットされているコアに対して *Wakeup* メッセージを送信し，対応するビットをクリアしていく（t5）．

表 1 シミュレータ諸元

Processor	SPARC V9
#cores	32 cores
clock	4 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	3 cycles
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

一方，*Thread2* は *Tx.Y* を再実行し，同時に L-Counter，S-Counter，Pr-CoreID をリセットする（t6）．

次に，図 6 において *Tx.Y* とアドレス A で競合したという情報を *Core1* が Conflict-Addr に記憶している状態を考える．ここで，*Thread1* が `load A` を実行済みである状態で，*Thread2* が `load A` の実行を試みたとする．この時，*Core1* の Conflict-Addr を検索すると，記憶されている競合と同じ状況であることが分かるため，*Thread2* に *Wait* メッセージを送信すると同時に，Po-CoreID の対応するビットをセットする（t1）．一方，*Thread2* は実行を待機すると同時に Pr-CoreID に相手コア番号である 1 を記憶する（t2）．実行が進み，*Thread1* が *Tx.X* をコミットする際は図 5 と同様に，Po-CoreID のビットがセットされているコアに対して *Wakeup* メッセージを送信することで，*Thread2* の実行を再開させる（t3）．

5. 評価

本章では，これまでに述べたトランザクションの実行状況に応じた待機処理の設定手法をシミュレーションにより評価し，それを実現するためのハードウェアコストを概算する．

5.1 評価環境

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[7] 3.0.31 と GEMS[8] 2.1.1 の組み合わせを用いた．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環境を示す．評価対象のプログラムとしては GEMS 付属 microbench，および SPLASH-2[9] から計 6 個を使用し，各ベンチマークはそれぞれ 8，16 スレッドで実行した．なお，定義式（1）の重み

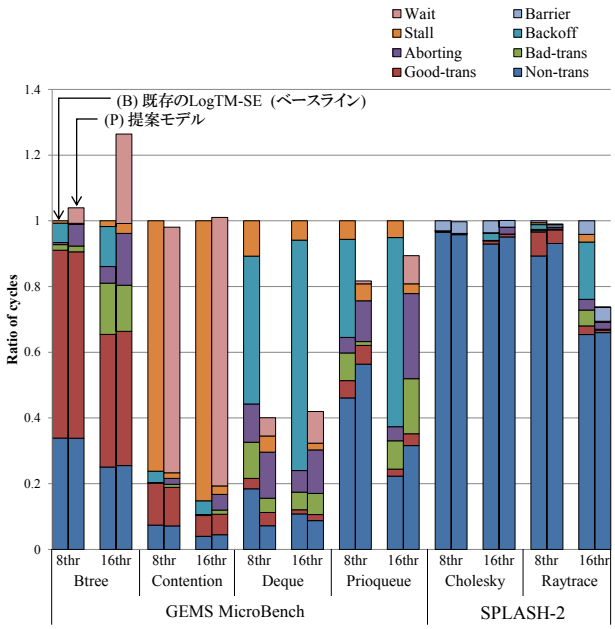

図 7　各プログラムにおけるサイクル数比

α，β，γ には等しい値として 1 を設定した．

5.2　評価結果

各ベンチマークの実行結果を**図 7** に示す．図 7 中の凡例はサイクル数の内訳を表しており，Non-trans はトランザクション外の実行サイクル数，Good-trans はコミットされたトランザクションの実行サイクル数，Bad-trans はアボートされたトランザクションの実行サイクル数，Aborting はアボート処理に要したサイクル数，Barrier はバリア同期に要したサイクル数，Backoff はバックオフ処理に要したサイクル数，Stall はストールに要したサイクル数，Wait は提案手法で追加した待機処理に要したサイクル数をそれぞれ示している．また，図中では各ベンチマークを 8，16 スレッドで実行した結果ごとにまとめて示しており，各ベンチマークプログラムとスレッド数との組み合わせによる結果をそれぞれ 2 本のグラフで表している．2 本のグラフはそれぞれ

(B)　既存モデル（ベースライン）
(P)　提案モデル

の実行サイクル数を表しており，既存モデル (B) の実行サイクル数を 1 として正規化している．

評価の結果，既存モデル (B) と比較して最大 59.9%，16 スレッドで平均 11.2%の実行サイクル数が削減された．

5.3　考察

評価結果から，一部性能が低下しているものもあるが，既存モデル (B) と比較した性能は概ね同等か，または向上していることが分かる．まず Deque および Prioqueue に注目すると，バックオフの使用を停止した事により，特に Deque において大きく性能向上していることが分かる．この原因を調査したところ，これらのプログラム中にはある共有変数へのロードアクセスの後，同一変数に対してストアアクセスするトランザクションが存在することが分かった．このようなトランザクションが並列実行された場合，既存モデル (B) では同じアドレスに複数のスレッドがロードアクセスした後にストアアクセスを実行することで競合が発生し，アボートが引き起こされていた．一方，提案手法では，一度アボートを引き起こしたトランザクション間でのアクセスの並列実行を禁止するため，アボートの発生を抑制できていた．また，提案手法で追加した待機処理は既存モデル (B) のバックオフと比較しても少なく，トランザクションの実行状況に応じた待機処理が行えている事が確認できる．

一方，Btree ではスレッド数の増大にともない，Aborting サイクルの増加と Wait サイクルにより性能が悪化していた．この原因を調査したところ，Btree には 2 つのトランザクション（仮に *Tx.A*，*Tx.B* とする）が存在しており，*Tx.A* にはロードアクセス・ストアアクセスの両方が含まれているが，*Tx.B* には共有変数に対するロードアクセスのみしか含まれていないことが分かった．そのため，複数の *Tx.A* が実行される際には提案モデル (P) が効果的であるが，複数の *Tx.B* が実行される場合にはストアアクセスが発生しないため，ロードアクセスの実行を待機させてしまうと性能低下が引き起こされてしまう．提案モデル (P) では，優先度の値の小さい *Tx.A* が優先度の値の大きい *Tx.B* にアボートさせられると，各コアは競合が発生した相手トランザクションとして *Tx.B* を記憶し続ける．この情報は *Tx.B* を実行している間も記憶し続けているため，複数の *Tx.B* が実行される際，本来競合が発生しないにも関わらず，その実行を待機させてしまうことがあった．また，優先度の値の小さいトランザクションによりストアアクセス済みの共有変数に対して，優先度の値が大きいトランザクションがロードアクセスを試みた際，前者を待機させた後ロードアクセスを実行すると，コミット前の不正な値を読み出すことになり，アトミシティが満たせなくなってしまう．そのため，このような場合にはストアアクセス済みのトランザクションをアボートすることでアトミシティを保つことになる．このプログラムでは，このような状況が多く発生していた．これらにより，Wait サイクルおよび，Aborting サイクルが増加し，性能低下を引き起こしていたと考えられる．したがって今後は，トランザクションの処理内容を考慮し，ロードアクセスのみのトランザクションに関する情報を適宜 Conflict-Addr から削除するような手法を考案する必要がある．

なお，Cholesky および Raytrace (8thr) では大きな性能向上は得られなかった．これらのプログラムでは，トランザクション外の実行サイクル数がプログラムの総実行サイクル数の大半を占めているため，提案モデル (P) によって

バックオフの使用を停止したとしても，性能向上の割合が小さくなってしまったと考えられる．また Contention では，既存モデル (B) に対して Stall サイクルが大幅に削減されたが，同時に提案手法で追加した待機処理のサイクル数が，削減サイクル数とほぼ同量発生していた．これは，既存モデル (B) では，競合が発生した際に実行中のトランザクションをストールすることで実行を待機する場面が存在したが，本提案手法では競合が発生する前にトランザクションの実行を待機させるため，Stall サイクルが削減された．しかし，Contention には非常に多くの命令を含むトランザクションのみが存在するため，競合が発生した際，相手スレッドはその実行を待機する時間が長くなり，結果として Wait サイクルが増大したと考えられる．

5.4 ハードウェアコスト

提案手法を実現するために追加したハードウェアのコストについて考える．提案手法を実現するため L-Counter と S-Counter にはそれぞれ最大のロードアクセス回数，最大のストアアクセス回数をカウントできるだけのビット幅が必要となる．そこで，提案モデル (P) で実行した場合の各プログラムにおいて，それぞれのアクセス回数を 16 スレッドで実行し調査した．その結果，最大で L-Counter は 470 回，S-Counter は 944 回まで記憶できれば，全てのプログラムにおいてオーバーフローすることなくアクセス回数をカウントできることが分かった．したがって，L-Counter と S-Counter に必要となるサイズはそれぞれ 10bit となる．ここで，16 スレッドを実行可能な 16 コア構成のプロセッサの場合では，Pr-CoreID は 4bit，Po-CoreID は 16bit となる．次に，Past-Access に必要な深さを調べるために各プログラムのトランザクション数を調べると，最大で 19 のトランザクションが含まれていることが分かった．また，Past-Access の 1 つのエントリに必要なサイズは 20bit である．したがって，1 つの Past-Access は幅 720bit 深さ 2 行の RAM で構成できる．一方，Conflict-Addr には競合したアドレス分のエントリが必要となる．ここで，提案モデル (P) で実行した結果，最大で 36 エントリまで記憶できれば，すべてのプログラムにおいてテーブルが溢れることなくアクセス回数を記憶できることがわかった．また，Conflict-Addr の 1 つのエントリに必要なサイズは 32bit であり，1 つの Conflict-Addr は幅 1152bit 深さ 2 行の RAM で構成できる．これらを総合すると，追加ハードウェアの総和は $16 \times (2 \times 10 + 4 + 16 + 720 \times 2 + 1152 \times 2) =$ 約 7.6Kbytes とごく少量である．

6. おわりに

本稿では，トランザクションがアボートされた際の待機アルゴリズムを改良し，トランザクションの実行状況に応じた待機時間を設定する手法を提案した．これにより，トランザクションの再実行までの時間を従来と比較して短く設定することができた．提案手法の有効性を確認するために GEMS 付属の 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，既存の HTM と比較して最大 59.9%，16 スレッドで平均 11.2%の実行サイクル数が削減されることを確認した．しかし，提案モデルでは本来トランザクションを並列実行すべき状況でもそれらを待機させてしまう場合があった．また，同一共有変数に対してストアアクセスした後に，他のトランザクションがロードアクセスを行う際にアボートが頻発していた．したがって，今後はトランザクションの処理内容を考慮した待機処理の設定を検討する必要がある．

参考文献

[1] Herlihy, M. et al.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. 20th Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289–300 (1993).
[2] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995).
[3] Yen, L., Bobba, J., Marty, M. R., Moore, K. E., Volos, H., Hill, M. D., Swift, M. M. and Wood, D. A.: LogTM-SE: Decoupling Hardware Transactional Memory from Caches, *Proc. IEEE 13th Int'l Symp. on High Performance Computer Architecture (HPCA'07)*, pp. 261–272 (2007).
[4] 橋本高志良，堀場匠一朗，江藤正通，津邑公暁，松尾啓志：Read-after-Read アクセスの制御によるハードウェアトランザクショナルメモリの高速化，情報処理学会論文誌コンピューティングシステム， Vol. 6, No. 3(ACS44), pp. 58–71 (2013).
[5] 江藤正通，堀場匠一朗，浅井宏樹，津邑公暁，松尾啓志：Starving Writer の解消による LogTM の高速化，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(ACS40), pp. 55–65 (2012).
[6] 堀場匠一朗，江藤正通，浅井宏樹，津邑公暁，松尾啓志：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，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(ACS40), pp. 43–54 (2012).
[7] 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).
[8] 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).
[9] 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).