Vol.2014-ARC-208 No.23
2014/1/24

投機的実行により並列度を向上させるハードウェアトランザクショナルメモリ

山田 遼平[1]　堀場 匠一朗[1]　井出 源基[1]　橋本 高志良[1]　津邑 公暁[1,a)]

概要：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共有リソースへのアクセスを調停する．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．これをハードウェアで実現する HTM では，一般的にアクセス競合が発生した場合にトランザクションの実行を停止する必要があるため，一時的に並列度が低下してしまう．そこで本稿では，競合が発生したとしてもトランザクションの実行を停止させず，競合相手がコミットまで到達すると仮定して投機的に実行を継続することで並列度を増大させる手法を提案する．評価の結果，既存手法に比べて，最大 9.63%，16 スレッドで平均 1.74%の実行サイクル数の削減を確認した．

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を調停する機構として，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，プログラムごとに適切なロック粒度を設定するのは困難であるため，この機構は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（Transactional Memory: TM）**[1] が提案されている．このトランザクショナルメモリをハードウェアで実現する**HTM**（Hardware Transactionl Memory）では，各キャッシュラインに対して read および write ビットという，トランザクション内で発生した Read および Write アクセスの有無を記憶するフィールドが追加されている．そして，キャッシュコヒーレンスリクエストを受け取った際に，これらのビットを参照することで競合検出を実現する．しかし，一般的にメモリアクセス競合が発生した場合，これを検出したスレッドは実行中のトランザクションを停止させる必要があるため，一時的に並列度が低下してしまう．特に，多くの命令を含むトランザクションが競合に関わる場合には，他スレッドによる当該アドレスへのアクセスを長期間停止することなるため，並列性が著しく損なわれる可能性がある．そこで本稿では，競合が発生したとしてもトランザクションの実行を停止させず，競合相手がコミットまで到達すると仮定して投機的に実行を継続することで並列度を増大させる手法を提案する．

[1] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi, 466-8555, Japan
[a)] tsumura@computer.org

2. ハードウェアトランザクショナルメモリ

本章では，まず研究対象である HTM の概要について述べる．

2.1 基本概念

マルチコア・プロセッサ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制御にロックが用いられてきたが，これにはデッドロックの発生や並列性の低下等の問題がある．

そこで，ロックを用いない並行性制御機構である**トランザクショナルメモリ（TM）**が提案されている．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対して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**トランザクション**として定義される．

これら 2 つの性質を保証するために，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．ここで，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性質を満足しない場合，**競合**として判定される．この操作

ⓒ 2014 Information Processing Society of Japan

を**競合検出**という．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**という．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**という．そして，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．一方で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すべての結果をメモリに反映させる．これを**コミット**という．

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 で行われる競合の検出，コミット，およびアボート等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．これらのうち，ハードウェア上に実装された TM はハードウェアトランザクショナルメモリ（HTM）と呼ばれる，

2.2 競合検出とデータのバージョン管理

2.2.1 競合の検出と解決

競合を検出するためには，どのアドレスがトランザクション内でアクセスされたかをトランザクションごとに記憶する必要がある．そのため，HTM では一般的に，各キャッシュラインに対して **read** ビットおよび **write** ビットと呼ばれるフィールドが追加されている．各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 Read アクセスおよび Write アクセスが発生した場合にそれぞれセットされ，コミットおよびアボート時にクリアされる．これらのビットを操作するために，HTM ではキャッシュコヒーレンスプロトコルを拡張している．このプロトコルでは，あるスレッドがメモリアドレスにアクセスする場合，キャッシュラインの状態を変更させるリクエストが他のスレッドに送信される．このとき，リクエストを受信したスレッドはキャッシュラインの状態を変更する前に，キャッシュに追加された read および write ビットを参照する．これにより，他のスレッドとの間で発生する競合を監視する．なお，以下の 3 パターンのアクセスが競合として判定される．

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 Read 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 Write 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 Write アクセス．

以上のような競合パターンが検出されると，競合を検出したスレッドからリクエストを送信したスレッドに対して **NACK** が返信される．NACK を受信したスレッドは自身のアクセスで競合が発生したことを知るが，すぐにはアボートせず，トランザクションをストールする．なお，この競合検出方式は，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ないか検査する．

Lazy Conflict Detection: トランザクションがコミットしようとした時点で，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．

実際に競合が発生してからそれを検出するまでの時間が長くなる lazy 方式では，無駄な実行時間が増大してしまい効率が悪い．

2.2.2 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，アボートにより実行結果が破棄される可能性があるため，トランザクション内で更新した値と更新前の値とを併存させる必要がある．そこで HTM では，トランザクション内で発生した Write アクセスにより更新した値，あるいは更新される前の古い値を，そのアドレスとともにメモリ上の別領域に保持する．このようなデータの管理はバージョン管理と呼ばれ，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領域にバックアップし，新しい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だけであ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書き戻す必要がある．

Lazy Version Management: 書き換え前の古い値をメモリに残し，新しい値を別領域に登録する．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コピーが必要となる．

ここで，eager 方式は，必ず実行されるコミットを高速に行い，必ずしも発生するとは限らないアボートにコストを払う考え方である．アボートが繰り返し発生してしまうようなプログラムでは不利となる場合もあるが，lazy 方式ではコミットのためのオーバヘッドは削減の余地がほぼないのに対し，eager 方式では競合や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．

よって本稿では，競合検出方式とバージョン管理方式について，eager 方式を採用した（eager/eager）HTM の実装の一つである **Log-based Transactional Memory**（**LogTM**）[2] に対し，提案する手法を実装し，評価する．

3. 関連研究

アボートしたトランザクションを途中から再実行することで，その再実行コストを抑える部分ロールバック [3], [4] の研究や，バージョン管理や競合検出の方式を動的に変更

ⓒ 2014 Information Processing Society of Japan

する研究 [5], [6] など HTM に関する数多くの研究が行われてきた．特にスレッドスケジューリングに関しては，実行並列度に着目した改良手法が提案されてきた．

Geoffrey ら [7] は複数のトランザクション内でアクセスされるアドレスの局所性を similarity と定義し，これが一定の閾値を超えた場合に，当該トランザクションを逐次実行することで競合を抑制する手法を提案している．しかし，この手法は性能評価において関連手法のみを比較対象としているため，既存の HTM に対してどの程度性能が向上したのかが明確に示されていない．

一方で，Titos ら [8] は，eager，および lazy な競合検出方式を組み合わせることで競合を効果的に解決し，トランザクションの実行並列度を向上させる手法を提案している．しかし，この手法はあるアドレスに対する WaR アクセスを検出した際にのみ適用可能であるため，その効果は極めて限定的であり，既存の競合検出方式に対して十分な性能向上は得られていない．さらに，一般に多く見られる，ある共有変数に対し Write アクセスに先立って Read アクセスが行われるようなトランザクション処理を継続して実行することができず，HTM の並列性を低下させ得る競合パターンが根本的に解決されていない．

そこで本稿では，RaW アクセスを投機的に実行可能とし，さらにその後，同一アドレスに対して続けて行われる WaR，および WaW アクセスを継続して実行させることで，HTM における実行並列度のさらなる向上を図る．

4. 投機的実行による並列度の向上

本章では，既存の HTM における問題点と，それを解決する提案手法について述べる．

4.1 競合時におけるメモリアクセスの継続

アクセス競合が発生する場合，これを検出したスレッドは，メモリ一貫性を保証するため実行中のトランザクションをストールさせる必要がある．ここで，2 つのスレッド *thr.1, 2* 上でそれぞれトランザクション *Tx.X, Y* が実行される**図 1**（a）の例を用いて，競合が発生する様子を示す．まず，*thr.1* が `load A` を実行した後（時刻 t1），続けて `store A` を実行する（t2）．その後，*thr.2* が `load A` の実行を試みるため当該アドレスへのアクセスリクエストを送信するが（t3），*thr.1* は既にアドレス A へ Write アクセス済みであるため RaW 競合を検出し，*thr.2* へ *NACK* を返信する（t4）．一方，*NACK* を受信した *thr.2* は自身の実行する *Tx.Y* をストールさせ，*Tx.X* の完了を待機する．その後 *Tx.X* がコミットされると，*thr.2* はトランザクションの実行を再開し (t5)， 続いてアドレス A 上の値を書き換える．このように，競合時には実行中のトランザクションを停止させる必要があるため，並列度が一時的に低下してしまう．特に，多くの命令を含むトランザクションが競合に関わる場合には，他スレッドによる当該アドレスへのアクセスを長期間待機させることになるため，並列性が著しく損なわれる可能性がある．しかしながらこの例のように，*thr.1* が時刻 t1，t2 においてアドレス A にアクセスしてから，同一アドレスに対して再度アクセスせずにコミットに至るのであれば，他スレッドがアドレス A 上の値を書き換えたとしても，一貫性は保たれる．

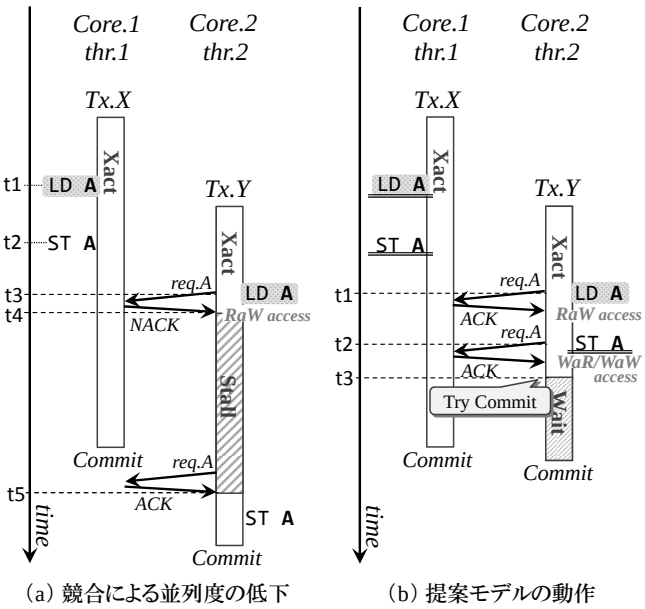

図 1　競合による並列度の低下とその解決

そこで，競合が発生したとしてもトランザクションの実行を停止させず，競合相手がコミットまで到達すると仮定して投機的に実行を継続することで並列度を増大させる手法を提案する．なお，一般に Test-and-Set のような操作を実現する場合など，共有変数への Read アクセスは，その後に Write アクセスをともなう場合が多く見られる．本稿ではこのようなメモリアクセスに対応するため，RaW アクセスおよび，その後同一アドレスに対して続けて行われる WaR/WaW アクセスも合わせて投機的実行を継続可能とする．

図 1（a）と同じ例に提案モデルを適用した場合の動作を図 1（b）に示す．*thr.1* はまず `load A` を実行した後，続けて `store A` を実行する．その後，同様に *thr.2* が `load A` の実行を試みる（t1）．このとき，Read アクセスのためのリクエストを受信した *thr.1* は RaW 競合を検出するが，*Tx.X* 内でこれ以降アドレス A 上の値は変更されないものと仮定し，*thr.2* に対して ACK を返信する．これにより，*thr.2* は Dirty な状態であるアドレス A に配置された値を利用した投機的実行が可能となる．続いて，*thr.2* は時刻 t2 において同一アドレスへの書き込みを試みる．このとき，アクセスリクエストを受信した *thr.1* はアドレス A に対して WaR，および WaW 競合を検出するが，*thr.2* に対して再び ACK を返信することで，*Tx.X* のコミットに先

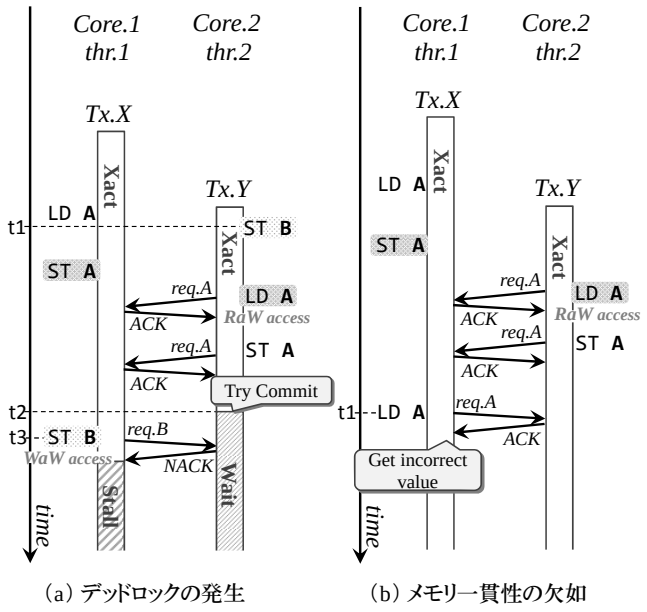

図 2　デッドロックの発生と一貫性の欠如

行して *Tx.Y* の処理を進行させる．その後，時刻 t3 において *thr.2* が *Tx.Y* の処理を終了したとする．ところが前述したとおり，*thr.2* は未コミットの値を使用して実行を継続したため，*Tx.X* の実行結果が確定されるまではトランザクションをコミットできない．したがって *thr.2* は *Tx.X* がコミットされるのを待機した後に *Tx.Y* をコミットする必要がある．また，競合の発生により *Tx.X* がコミットに到達することなくアボートされた場合にはアドレス A 上の値は破棄されてしまうため，この値を読み出した *Tx.Y* も共にアボートする．なおアボートすることで，それまでにトランザクション内で実行された命令は結果的に無駄となる．このため，提案モデルの適用による投機的実行が開始された時点で，既に多くの命令を実行済みである場合，ストールにより競合を解決する既存モデルと比較して，アボート処理および再実行のオーバヘッドが増大する可能性がある．

4.2　デッドロックの回避と一貫性の保証

競合時における投機的実行の継続によりスレッド間に依存関係が発生すると，トランザクション処理の実行順によってはデッドロック状態に陥る可能性がある．また，複数のトランザクションが互いに未コミットの値を読み出し合う場合，メモリ一貫性が失われてしまう．

ここで，デッドロックの発生とメモリ一貫性が失われる様子をそれぞれ**図 2** の（a），（b）に示す．まず，図 2（a）の例では *thr.1* がアドレス A に対して値の書き換えを行う一方で，*thr.2* が *Tx.Y* 内で `store B` を実行する（t1）．続いて，*thr.2* は *Tx.X* 内で変更されたアドレス A 上の値を用いてトランザクションの実行を継続し，まもなく *Tx.X* のコミットを待機する（t2）．この後，*thr.1* は `store B` の実行を試みるためアクセスリクエストを送信するが（t3），*thr.2* は当該アドレスにおいて WaW 競合を検出し，*NACK* を返信する．ここで *thr.1* は *Tx.X* をストールさせるため，結果的にそれぞれのスレッドが互いに実行するトランザクションのコミットを待機する状態となってしまう．また，図 2（b）では（a）の例と同様に，*thr.2* が *Tx.X* 内で変更された値を用いて *Tx.Y* の実行を継続している．この後，*thr.1* はアドレス A に対して再度 Read アクセスを試みる（t1）．ところが，アドレス A 上の値は *thr.2* の投機的実行により既に変更済みであるため，*thr.1* は *Tx.X* の実行中に不正な値を読み出してしまう．

そこで本稿では，複数のスレッド間における特定の依存関係の発生を検出し，トランザクションをアボートすることでこのような問題の発生を回避する．なお，図 2 の例では（a），（b）どちらの場合も，*thr.2* の実行する *Tx.Y* をアボートすることで正常な動作へと復帰させることができる．また，禁止される全ての事象とその対策については 5.4 節でより詳細に述べる．

5.　実装

本章では提案手法を実現するために追加するハードウェア，およびそれらの動作について説明する．

5.1　拡張したハードウェア構成

4 章で述べたとおり，投機的実行により複数のスレッド間に依存関係が発生している場合には，これらのスレッドが実行するトランザクションのコミットおよびアボートの実行順序を制御する必要がある．そこで，各スレッドは依存関係を持つ相手スレッド，および投機的実行が行われたアドレスを記憶し，これらを利用することでメモリの一貫性を保証する．

この操作を実現するため，既存の HTM を拡張し，小容量のハードウェアを各コアに追加する．これを **Speculative Table** と呼ぶ．なお，この表は以下の 3 つのユニットにより構成される．また，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．

- **Speculative Address（Sp-addr.）**：各スレッドにおいて RaW アクセスにより投機的実行が行われたアドレスを記憶する領域．
- **Speculative Parent bits（SpP bits）**：相手トランザクション内で変更された未コミットの値を使用して投機的実行を継続する際，どのスレッドによって RaW アクセスを許可されたかを記憶する，n bit のビットマップ．
- **Speculative Child bits（SpC bits）**：自トランザクション内で変更した未コミットの値を使用して投機的実行を継続させる際，どのスレッドに対して RaW アクセスを許可したかを記憶する，n bit のビットマップ．

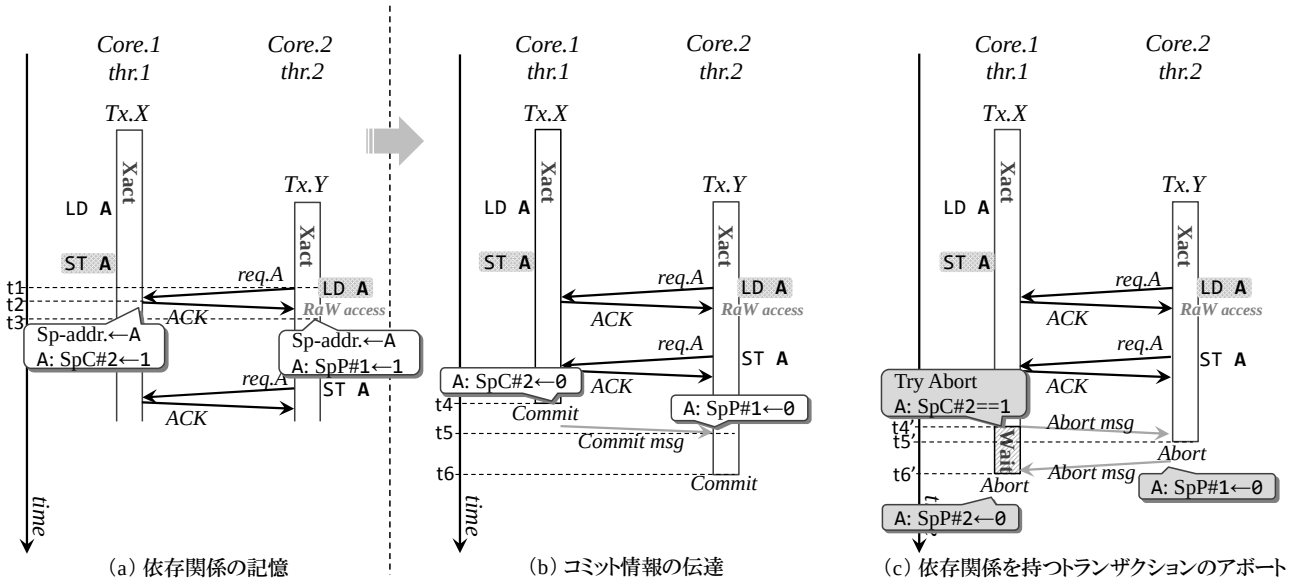

図 3　Speculative Table を利用した提案モデルの動作

5.2　依存関係の記憶

2 つのスレッド *thr.1*, *2* 上でそれぞれトランザクション *Tx.X*, *Y* が実行される**図 3**（a）の例を用いて，スレッド間の依存関係を記憶する様子を示す．*thr.1* はまず `load A` を実行した後，続けて `store A` を実行する．その後，*thr.2* が `load A` の実行を試みる（t1）．このとき，アクセスリクエストを受信した *thr.1* は *thr.2* による投機的実行を許可すると同時に，アドレス A を Sp-addr. に記憶し，さらにこれに対応する SpC bits の内，*Core.2* に該当するビットをセットする（t2）．一方，この RaW アクセスに対して ACK を受信した *thr.2* も同様に，当該アドレスを Sp-addr. に記憶し，SpP bits の *Core.1* に対応するビットをセットすることで互いに依存関係が生成されたことを記録する（t3）．

5.3　コミット/アボート時の操作

本節では，Speculative Table に記憶された情報を用いてどのように動作制御を実現するかを説明する．

5.3.1　コミット情報の伝達

トランザクションのコミット時，各スレッドは記憶された全ての Sp-addr. に対応する SpP bits および SpC bits を参照する．ここでまず，SpC bits がセットされたコアに対してコミットの実行を伝えるメッセージを送信する．一方で，メッセージを受信したコアは SpP bits の内，送信者に対応するビットをクリアする．これは，投機的実行に利用された値をコミットすることで，スレッド間の依存関係が解消されるためである．またコミットを試みる際，任意の Sp-addr. について，SpP bits の内いずれかのコアに対応するビットがセットされているならば，当該コアの実行結果が確定するまではトランザクションをコミットできないため，これを待機する必要がある．一方，記憶したすべての Sp-addr. に対応する SpP bits がクリアされた状態であれば，コミットを試みたトランザクション内で投機的実行が行われていないか，もしくは自身が投機的に使用した値は既に確定済みであることが分かるため，トランザクション処理を完了できる．

ここで，図 3（a）の例の後，各スレッドがそれぞれトランザクションの実行を進行させた様子を図 3（b）に示す．この例では *thr.1* が *thr.2* に先行して *Tx.X* のコミットに到達するため，この時点で両スレッド間の依存関係は解消される（t4）．したがって，*thr.1* はアドレス A に対応する SpC bits の *Core.2* に該当するビットをクリアすると同時に，*thr.2* へコミットの完了を伝達する．一方 *thr.2* は SpP bits を参照し，同様の手順で *thr.1* との依存関係を解消する（t5）．その後，*thr.2* は SpP bits がクリアされた状態でトランザクション処理を完了するため，継続して *Tx.Y* のコミットを実行する（t6）．

5.3.2　依存関係を持つトランザクションのアボート

競合の発生によりトランザクションをアボートする際も同様に，各スレッドは記憶された全ての Sp-addr. に対応する SpP bits および SpC bits を参照する．ここで，LogTM に代表される eager 方式のバージョン管理を採用した HTM ではトランザクションをアボートする際，コミット以前に書き換えた値はすべて破棄されてしまう．つまり，この値を読み出して投機的に実行を継続したスレッドも同様に，その時点における実行結果を破棄する必要がある．したがって，SpC bits のセットされたコアへ，実行するトランザクションをアボートするよう伝達する．なおこのとき，メモリ一貫性を保証するため，依存関係の存在するアドレスに対して最初に Write アクセスしたスレッドは，ロールバック処理を最後に行う必要がある．また，アボート時に

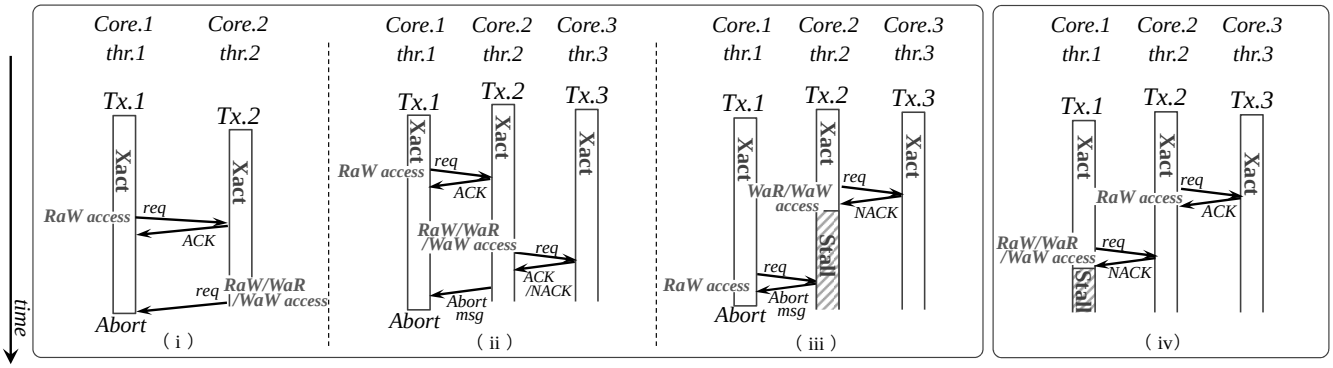

図 4　禁止される事象の検出

はスレッド間の依存関係が解消されるため，SpP bits がセットされているコアに対し，送信者自身のコア番号に対応する SpC bits をクリアするようメッセージを送信する．

ここで，図 3（a）の例の後，依存関係を持つ 2 つのスレッドがそれぞれトランザクションをアボートする様子を図 3（c）に示す．この例では時刻 t4’ において *thr.1* が他スレッドとの競合により *Tx.X* のアボートを試みるとする．このとき，アドレス A に対応する SpC bits の内，Core.2 に該当するビットがセットされていることが分かるため，このコアに対しトランザクションのアボートを要求するメッセージを送信する．この間，*thr.1* は *Tx.Y* のロールバックにより投機実行前の値が書き戻されるのを待機する．その後，*thr.2* はアボート処理を完了すると（t5’），SpP bits の *Core.1* に該当するビットをクリアすると同時に，*thr.1* へアボート処理の再開を要求する．一方，これを受けた *thr.1* は同様に依存関係を解消し，アボート処理を再開，完了する（t6’）．

5.4　禁止される事象の検出

4.2 節で述べたとおり，競合時における投機的実行の継続によりスレッド間に依存関係が発生すると，デッドロックや一貫性が保たれない状態に陥る可能性がある．そこで．複数のスレッド間における以下の事象の発生を禁止することでこの問題を回避する．また，禁止される各事象それぞれを検出する例を**図 4** に示す．

（i）任意のアドレスについて，SpP bits の内いずれかのコアに対応するビットがセットされた状態で，RaW，WaR または WaW アクセスリクエストを受信する．
（ii）任意のアドレスについて，SpC bits の内いずれかのコアに対応するビットがセットされた状態で，第三者のコアへ RaW，WaR または WaW アクセスリクエストを行う．
（iii）任意のアドレスについて，実行トランザクションをストール中に RaW アクセスリクエストを受信する．
（iv）任意のアドレスについて，SpP bits の内いずれかのコアに対応するビットがセットされた状態で，第三者のコアから RaW，WaR または WaW アクセスリクエストを受信する．

まず，事象（i）が検出される場合，2 者のスレッド間でデッドロック，あるいは一貫性が欠如した状態に陥ることになるため，一方のトランザクションをアボートすることで正常な動作へと復帰させる．また，事象（ii，iii）に該当するアクセスが発生すると，3 者以上のスレッド間でもデッドロックに陥る可能性がある．そこで，3 スレッド以上に渡って依存関係が生成された時点でいずれかのトランザクションをアボートすることで，デッドロックの発生を未然に防止する．なお本稿では，すべての事象について，未コミットの値を用いて投機的に実行，あるいはこれを試みるトランザクションをアボートの対象とするが，事象 (iv) に限り，対象トランザクションをストールさせる．これは，即座にアボートせずとも，今後デッドロックに陥る場合には事象 (iii) に該当するアクセスが発生することになり，結果的にトランザクションがアボートされるためである．

6.　評価

本章では，提案手法の速度性能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．

6.1　評価環境

これまで述べた拡張を HTM の一実装である 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タを示す．

ⓒ 2014 Information Processing Society of Japan

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

評価対象のプログラムは GEMS 付属の microbench に含まれる Btree，Contention，Prioque に加えて，SPLASH-2[11] から Cholesky を，STAMP[12] から Genome，Kmeans，Vacation を用いた．

6.2　評価結果

評価結果を**図 5** に示す．図 5 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，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，Barrier はバリア同期に要したサイクル数，MagicWaiting 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．

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左から順に

(B)　既存モデル（ベースライン）

(P)　投機的実行により並列度を向上させる提案モデル

の実行に要した総サイクル数を表しており，各サイクル数は既存モデル (B) を 1 として正規化している．

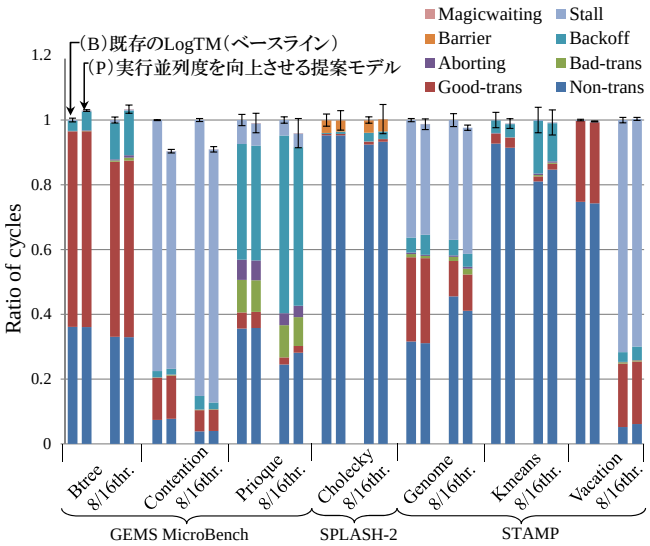

図 5　各プログラムにおけるサイクル数比

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場合は性能のばらつきを考慮しなければならない [13]．したがって，各評価対象につき試行を 10 回繰り返し，得られたサイクル数から平均値と 95%の信頼区間を求めた．平均値をグラフの縦軸に，信頼区間をグラフ中のエラーバーで示している．

評価の結果，既存モデル (B) と比較して最大 9.63%，16 スレッドで平均 1.74%の実行サイクル数が削減された．

6.3　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．このことから，多くのプログラム中で RaW アクセスの継続による投機的実行が成功しており，既存の HTM よりさらに並列度を向上させる余地があることを確認できた．また，コミットおよびアボート順序を制御するための待機処理である MagicWaiting サイクルは全てのプログラムにおいてほぼ見られず，このオーバヘッドが総実行サイクル数に占める割合は，16 スレッドで実行した場合平均 0.04%と，ごく僅かなものであることが分かった．以下，各プログラムについて詳細な検証を行う．

まず Contention に注目すると，8，16 スレッドで実行したどちらの場合においても，Stall サイクルが大幅に削減されていることが分かる．ここで，Contention 内でストールを引き起こす処理を簡略化したコードを**図 6** に示す．図中のトランザクションは 3 つの do_phase 関数から成り，その実行フローは各関数の実引数として使用されるコマンドライン引数の値に依存する．このとき，デフォルトで設定された入力値を用いる場合，これらにはそれぞれ 0，1，1 が与えられる（図中 2〜4 行目）．一方，do_phase 関数内では仮引数 `phase` の値に従って処理を分岐させ，共有メモリ配列 `gm.array1[index]` および

```
1  BEGIN_TRANSACTION;
2   do_phase(argv[0]);  /*argv[0]=0*/
3   do_phase(argv[1]);  /*argv[1]=1*/
4   do_phase(argv[2]);  /*argv[2]=1*/
5  COMMIT_TRANSACTION;
6
7  void do_phase(int phase){
8    switch(phase){
9    case 0:
10     gm.array1[index]++;
11     break;
12   case 1:
13     gm.array2[index]++;
14     break;
15   }
16 }
```

図 6　Contention プログラムの擬似コード

gm.array2[index] に対してアクセスを行う（10，13 行目）．つまり，トランザクション中の 1 つめの do_phase 関数内で gm.array1[index] へのアクセスが行われて以降，次に続く処理はすべて gm.array2[index] へのアクセスとなることが分かる．これより，この処理を含む複数のトランザクションが並列に実行される場合，gm.array1[index] へのアクセスが完了した時点で，そのコミットに先立って他のスレッドによる当該配列要素へのアクセスが投機的に実行可能となったため，ストールが削減されたと考えられる．

また，Prioque でも僅かながら Stall サイクル，Backoff サイクルが減少している．この原因を調査したところ，Prioque で実行されるトランザクションには共有変数への Read アクセスの後，同一変数に対し Write アクセスをともなう操作が多く含まれることが分かった．提案モデル (P) では，これらの処理を並列実行する際に発生する RaW アクセスおよび，続けて行われる WaR，WaW アクセスの投機的実行を可能としたことで並列度を増大させることができ，これが性能の向上に寄与したと考えられる．さらに，既存モデル (B) と比較してトランザクションを早期にコミット可能としたことで，競合の発生する可能性を低減させ，結果的に Backoff サイクルを削減することができた．

次に，Genome/16thr では，Non-trans の削減が性能向上に寄与していることが確認できる．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．2.2.1 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．さらに，通常は共有変数にアクセスするこ

表 2　Btree/16thr における禁止した各事象の検出回数

	(i)	(ii)	(iii)	(iv)	Total
検出回数	1483	275	211	0	1969

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．なお，これまでに我々はキャッシュラインを複数のサブブロックに分割することで競合検出単位を細粒度化し，false sharing による競合の誤検出を防止する手法を提案している [14]．これに対し，提案モデル (P) 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．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．

なお，Cholesky，Kmeans および Vacation は RaW アクセスの発生が少ないプログラムであり，投機的に実行を継続できる機会がほとんど存在せず，目立った性能向上は得られなかった．

一方，Btree では，Aborting サイクル，Bad-trans サイクル，Backoff サイクルの増加により性能が悪化した．これは，提案手法の適用により 5.4 節で述べた 4 パターンの事象が検出され，投機的実行を行うトランザクションが繰り返しアボートされたことが大きな原因であった．ここで，Btree/16thr における各事象の検出回数を**表 2** に示す．この結果から，事象（i）の発生率が圧倒的に高く，2 者のスレッド間で投機的実行が頻繁に失敗していることが分かる．提案モデル (P) では，5.4 節で示した事象が検出されない限りすべての RaW アクセスを継続実行可能としており，これが本来並列に実行できないトランザクションを数多く含むプログラムに適用される場合，投機的実行失敗によるペナルティを受け続けてしまうと考えられる．したがって，本提案モデルが有効となるアクセスパターンを詳細に調査し，投機的実行を適用するか否かを動的に判定できる機構を今後検討する必要がある．

6.4　ハードウェアコストとアクセスレイテンシ

提案手法を実現するため，Speculative Table には，RaW アクセスが発生したキャッシュラインの数だけのエントリが必要となる．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，Sp-addr. ひとつ当たりに使用されたエントリ数を調査した．その結果，最大で 14 エントリあれば，全てのプログラムにおいて Speculative Table が溢れることなくスレッド間の依存関係を記憶できることが分

ⓒ 2014 Information Processing Society of Japan

表 3 (P) における Speculative Table エントリの総参照回数

Btree	103,464	Genome	7,723
Contention	1,844	Kmeans	4,320
Prioque	26,946	Vacation	4,417
Cholesky	6,790		

かった．ここで，32 スレッドを実行可能な 32 コア構成のプロセッサの場合では Speculative Table のひとつのエントリ当たりに必要となる Sp-addr. は 64 bit であり，また SpP bits および SpC bits はそれぞれ 32 bit である．したがって，1 つの Speculative Table は幅 128 bit 深さ 14 行の RAM で構成でき，Speculative Table サイズの総和は $32 \times 128 \times 14 =$ 約 7.2KBytes とごく少量である．

次に，これら追加ハードウェアに対するアクセスレイテンシによるアクセスオーバヘッドが性能に及ぼす影響について考察する．まず，Speculative Table エントリを参照した総回数を C，Speculative Table エントリを 1 回参照するためのレイテンシを T とすると，その参照コストは $C \times T$ として概算できる．ここで，提案モデル (P) において 16 スレッドで実行した場合のエントリの総参照回数を**表 3** に示す．また，前述したとおり，Speculative Table は約 7.2KBytes の RAM で構成できる．そこで，このエントリを L1 キャッシュと同レイテンシで参照できると仮定すると，本稿で用いたシミュレーション環境では $T =$ 1cycle とおくことができる．これらより，総参照回数の最も多い Btree についてそのコストを求めると，$103{,}464 \times 1 =$ 約 10 万 cycles となる．一方，Btree/16thrs の総実行サイクル数は約 700 万 cycles 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 1.4%と僅かなものであることが分かった．

7. おわりに

本稿では，競合発生時にもトランザクションの実行を停止させず，競合相手がコミットまで到達すると仮定して投機的に実行を継続することで並列度を増大させる手法を提案した．また，Speculative Table というハードウェアを追加することでコミットおよびアボートの実行順序を制御し，メモリ一貫性の保証された動作を実現した．

GEMS 付属の microbench，SPLASH-2，および STAMP ベンチマークを用いてシミュレーションにより評価した結果，提案モデルにより実行並列度が向上し，ストールが削減されたことを確認した．また，既存の HTM に比べて，最大 9.63% ，16 スレッドにおいて平均 1.74%の実行サイクル数の削減が確認できた．

なお本稿で提案したモデルでは，5.4 節で示した事象が検出されない限りすべての RaW アクセスを投機的に継続可能としており，これが一部のプログラムで性能の低下を引き起こしていた．このため，本提案モデルが有効となるメモリアクセスパターンを詳細に調査し，投機的実行を適用するか否かを動的に判定できる機構の実現を検討していく予定である．

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] J.Moravan, M. et al.: 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).
[4] Moss, J. E. B. and Hosking, A. L.: Nested Transactional Memory: Model and Preliminary Architecture Sketches, *Science of Computer Programming*, Vol. 63, No. 2, pp. 186–201 (2006).
[5] Lupon, M., Magklis, G. and González, A.: A Dynamically Adaptable Hardware Transactional Memory, *Proc. 43rd Annual IEEE/ACM Microarchitecture (MICRO)*, pp. 27–38 (2010).
[6] Shriraman, A., Dwarkadas, S. and Scott, M. L.: Flexible Decoupled Transactional Memory Support, *Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA)*, pp. 139–150 (2008).
[7] Blake, G., Dreslinski, R. G. and Mudge, T.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th Int'l Conf. on High-Performance Computer Architecture (HPCA-17)*, pp. 75–86 (2011).
[8] Titos, R., Acacio, M. E. and García, J. M.: Speculation-Based Conflict Resolution in Hardware Transactional Memory, *Proc. Int'l. Symp. on Parallel Distributed Processing (IPDPS 2009)*, pp. 1–12 (2009).
[9] Magnusson, P. S. et al.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).
[10] 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).
[11] Woo, S. C. et al.: The SPLASH-2 Programs: Characterization and Methodological Considerations, *Proc. 22nd Int'l. Symp. on Computer Architecture (ISCA'95)*, pp. 24–36 (1995).
[12] Minh, C. C., Chung, J., Kozyrakis, C. and Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
[13] 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).
[14] Horiba, S., Asai, H., Eto, M., Tsumura, T. and Matsuo, H.: Fine-Grain Conflict Management for Hardware Transactional Memory Systems Employing Eager Version Management, *Proc. 4th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA2013), held in conjunction with HiPEAC'13* (2013).

ⓒ 2014 Information Processing Society of Japan