Vol.2015-ARC-215 No.13
Vol.2015-OS-133 No.13
2015/5/27

トランザクショナルメモリにおける排他的および投機的並行性制御

間下 恵介[1]　三宅 翔[1]　山田 遼平[1]　津邑 公暁[1,a)]

概要：マルチコア環境では，一般的にロックを用いて共有変数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，これを補完する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．この機構をハードウェア上で実現したハードウェアトランザクショナルメモリ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的に実行される．しかし，共有変数に対する複合操作が行われるようなトランザクションが並行実行された場合，その際に発生するストールが無駄となる場合がある．そこで本稿では，このような複合操作を検出し，それに関与するトランザクションを排他実行する手法に加え，同一の共有変数に対してそれ以降変更が行われないと判断した時点で，他スレッドによる投機的アクセスを許可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 16 スレッド実行時において最大 67.2%，平均 13.9%の性能向上を達成した．

1. はじめに

マルチコア環境において一般的となっている，共有メモリ型並列プログラミングでは，共有変数へのアクセスを調停する機構として，広く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．TM 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**トランザクション**として定義することで，共有変数に対するアクセスにおいて競合が発生しない限り，投機的に実行を進めることができ，ロックを用いる場合よりも並列性が向上する．なお，TM ではトランザクションが投機的に実行されるため，共有変数に対して更新がなされる際は，更新前の値を保持しておく必要がある（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，同一変数に対する競合が発生していないかを常に検査する必要がある（競合検出）．トランザクショナルメモリ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およ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，トランザクション操作のためのオーバヘッドを軽減している．

さて，上述の HTM では競合が発生しない限りトランザクションが投機的に実行される．しかし，Read→Write の順序で共有変数にアクセスをする複合操作が行われるようなトランザクションが並行実行された場合，その際に発生するストールが完全に無駄となる場合がある．そこで本稿では，このような同一の共有変数に対する複合操作，すなわち Read→Write の順序でのアクセスを検出し，それに関与するトランザクションを排他実行する手法に加え，同一の共有変数に対してそれ以降変更が行われないと判断した時点で，他スレッドによる投機的アクセスを許可することで HTM の全体性能を向上させる手法を提案する．

2. 提案

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．

2.1 共有変数に対する複合操作に起因する問題

一般に，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．具体的には，複合演算子および複合代入式を用いる複合操作を実現する場合などがこれにあたる．この複合操作を含むトランザクションが複数のスレッドによって並行実行されると，それ

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

Vol.2015-ARC-215 No.13
Vol.2015-OS-133 No.13
2015/5/27

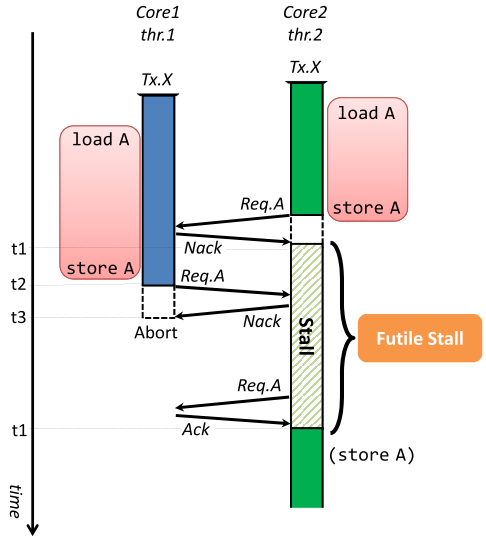

図 1　共有変数への複合操作に起因する Futile Stall

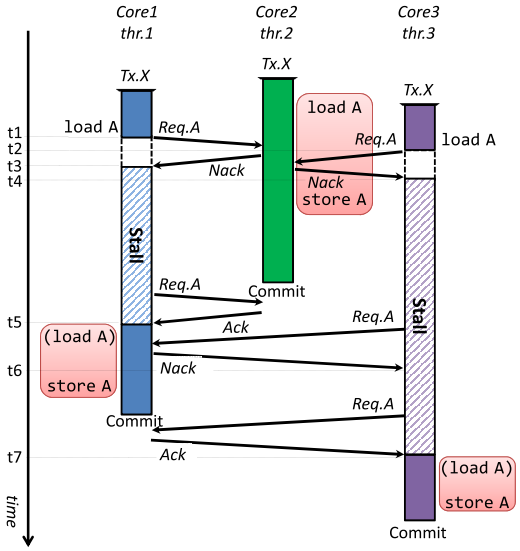

図 2　共有変数に対する複合操作の排他実行

らのスレッドの Read アクセスが許可されたとしても，その後に実行される Write アクセスにより結局競合が発生し，これが HTM の性能低下を引き起こす可能性がある．

図 1 は，上述した共有変数に対する複合操作を含むトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並行実行する様子を示している．まず，双方のスレッドが共有変数のアドレスである `A` に対して `load A` を実行した後，*thr.2* が `store A` を実行しようとした際に，競合が検出される．ここで LogTM[2] に代表される，**Eager Conflict Detection** 方式を採用する HTM では一般に，競合を発生させた *thr.2* が *Nack* の受信にともない，自身の実行する *Tx.X* をストールする（時刻 t1）．その後，*thr.1* が `store A` を実行しようとする際（t2），*thr.2* は既に当該アドレスにアクセス済であるため競合を検出し，*thr.1* へ *Nack* を返信する．この時，*thr.1* は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* を受信するため，*Tx.X* をアボートすることになる（t3）．このアボートにより，*thr.2* は *Tx.X* を再開できるが，この間に *thr.1* の実行は一切進行しておらず，*thr.2* のストールは完全に無駄であったことになる．このように，結果的にアボートされてしまうようなトランザクションとの競合により発生する無駄なストールは **Futile Stall**[3] と呼ばれ，HTM のスループットを低下させる大きな要因となっている．

2.2　複合操作の排他実行手法と提案手法の着眼点

前節で述べた Futile Stall が発生する要因として，ある共有変数に対して Read→Write の順でアクセスするスレッドが複数存在する場合に，それらのスレッドが共に Read のみを完了した状態となってしまうことが挙げられる．そこで他スレッドが当該アドレスに Read アクセス済みであった場合，この Read アクセスを待機させることで，共有変数に対する複合操作を排他実行する手法を我々は提案してきた [4][5]．この手法ではまず，Read→Write の順序でアクセスされる共有変数に対する Read を実行する際に，他のスレッドが当該アドレスに Read アクセス済みであるか否かをチェックする．そして，他スレッドが当該アドレスに Read アクセス済みであった場合，この Read アクセスを即座には許可せず *Nack* を返信して待機させる．

ここで**図 2** に，共有変数に対する複合操作の排他実行手法を用いた場合の動作を示す．この例では，3 つのスレッド *thr.1*〜*3* がそれぞれ，共有変数に対する複合操作，つまり共有変数に対する Read→Write の順序でのアクセスを含む同一のトランザクション *Tx.X* を投機実行している．まず，*thr.2* が共有変数のアドレスである `A` に対して `load A` を実行した後，*thr.1* と *thr.3* が同様に `load A` の実行を試みたとする（時刻 t1，t2）．この際に，*thr.1* と *thr.3* は Read アクセスのためのリクエストを送信するが，この時点で *thr.2* が既にアドレス `A` に Read アクセス済みであるため，*thr.1* と *thr.3* のそれぞれに対して *Nack* が返信される．この *Nack* の受信により（t3，t4），*thr.1* と *thr.3* のアドレス `A` に対する Read アクセスの実行が待機させられる．そのため，*thr.2* はアドレス `A` に Write アクセスを試みたとしても，これらのスレッドと競合することなく *Tx.X* の実行を進めることが可能となる．これにより，図 1 で示した Futile Stall による無駄な待機時間が削減される．その後，*thr.2* が *Tx.X* をコミットすると，待機中であった *thr.1* のアドレス `A` に対する Read アクセスの実行が許可される（t5）．一方 *thr.3* は，*thr.1* からの *Nack* により先程と同様に実行を待機することとなる．続けて，*thr.1* が *Tx.X* をコミットすると，*thr.3* の Read アクセスが許可される（t7）．この手法では，このように Read→Write の順序でのアクセスにおける Read リクエストの時点で *Nack*

Vol.2015-ARC-215 No.13
Vol.2015-OS-133 No.13
2015/5/27

図 3　投機的な実行による高速化

を送信することで，共有変数に対する複合操作の排他実行を実現する．

しかし，この手法では，同一の共有変数に対する Read→Write の順序でのアクセスをしたスレッドが当該トランザクションをコミットするまで，他のスレッドによる当該変数へのアクセスは許可されない．ここで，一般的なプログラムで実行されるトランザクションには，トランザクション内での最後の write アクセスからコミットまでに，一定の処理を含むものがある．そのため，あるスレッドが同一の共有変数に対してそれ以降変更を行わない場合，他スレッドが当該変数に投機的にアクセスしたとしても，結果として一貫性が保たれる可能性がある．この点から，この排他実行手法にはさらなる高速化の余地があると考えられる．

2.3　投機的アクセスの許可による高速化

本稿では，2.2 節で述べた排他実行手法のさらなる高速化を実現するために，トランザクション内で同一の共有変数に対する Read→Write の順序でのアクセスが完了した時点で，当該トランザクションのコミットに先立って他スレッドによるアクセスを投機的に許可する手法を提案する．

ここで**図 3** に提案手法を用いた場合の動作を示す．この例では，提案手法を適用し 2 つのスレッド *thr.1* および *thr.2* が図 2 と同様のトランザクションを同様のタイミングで並列実行している．まず，*thr.2* が `load A` を実行し，`A` に対して Read アクセス済みとなった後，*thr.1* が `load A` の実行を試みたとする（時刻 t1）．この時，*thr.1* は *thr.2* へ `A` に対するアクセスリクエストを送信するが，この時点で `A` は *thr.2* により Read アクセス済みであるため，*thr.2* は *thr.1* に対して *Wait* リクエストを送信し，*thr.1* のアクセスを待機させる（t2）．その後，*thr.2* が `store A` を実行したとすると，*thr.2* はこれ以降に *Tx.X* 内で `A` にアクセスしないため，`A` に対する投機的アクセスを許可することが可能となり，*thr.2* は *thr.1* に対して *Ack* を送信し，*thr.1* のアクセスを許可する（t3）．これにより，*thr.2* のコミットに先行して，*thr.1* が `A` に投機的にアクセスできる．なお，このように *thr.2* が *thr.1* の投機的アクセスを許可したことで，*thr.2* の実行する *Tx.X* 内で更新された値を用いて *thr.1* が *Tx.X* の処理を進めることになる．そのため，*thr.2* は *thr.1* よりも先に *Tx.X* をコミットする必要がある．したがって，*thr.2* は *Tx.X* をコミットした際，投機的アクセスを許可した *thr.1* に対して *Committed* メッセージを送信し，自身が *Tx.X* をコミットしたことを伝える（t4）．以上のように動作することで，投機的アクセスの許可により 2.2 章で述べた手法のさらなる高速化を図る．なお，*Wait* リクエストおよび *Committed* メッセージはコヒーレンスプロトコルを拡張することで新たに定義する．

3.　実装

本章では 2 章で述べた提案手法を実現するための実装方法と，その動作モデルについて述べる．

3.1　追加ハードウェア

提案手法を実現するために，各コアの L1 キャッシュ内のキャッシュラインに以下のフィールドを追加する．

Combined-Control bit（C ビット）:
当該キャッシュラインが Read→Write の順序でアクセスされたか否かを示すビット．排他実行手法 [5] においても用いている．

Lock bit（L ビット）:
当該ライン上のアドレスに対応する共有変数に対する，Read→Write の順序でのアクセスを含むトランザクションにおいて，Read アクセスから Write アクセスまでの処理区間を排他的に実行中であるか否かを示すビット．

なおこの手法では，トランザクション内で Read→Write の順序でアクセスされる共有変数に対して，各スレッドは自身が Write アクセスを完了したか否かを判断する必要がある．この判断のために，Read→Write の順序でアクセスされる変数のアドレス，そのようなアクセスを含むトランザクションの ID，そして Write アクセスが実行される時点におけるプログラムカウンタの値を記憶する機構が必要となる．さらに，待機スレッドに対するアクセス許可やトランザクションのコミット順序の制御を行うために，各スレッドが動作するコアの番号を記憶する機構が必要となる．そこで，それらの情報を記憶するための 2 つの表を各コアに追加する．この 2 つの表をそれぞれ **Address List(A-List)**，**Dependence Table(D-Table)** と呼ぶ．Address List は Read→Write の順序でアクセスされる共有変数のアドレスを記憶する．また，Dependence Table は以下の 4 つのフィールドから構成される．

Prev-Core(Prev):
Address List に記憶したアドレスに対する投機的アク

Vol.2015-ARC-215 No.13
Vol.2015-OS-133 No.13
2015/5/27

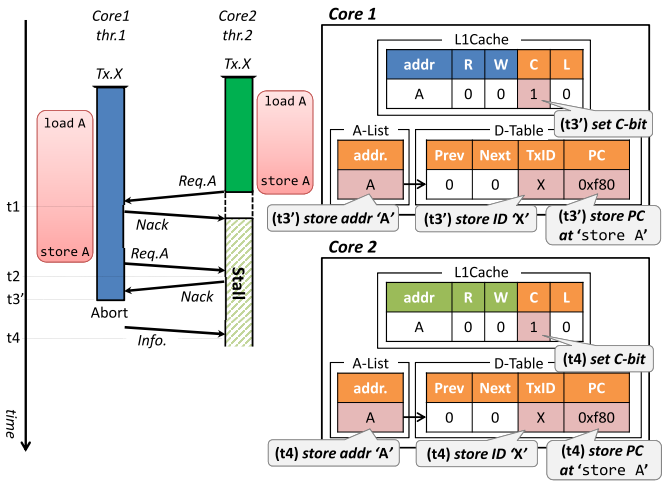

図 4　Read→Write の順序でアクセスされるアドレスを記憶する動作

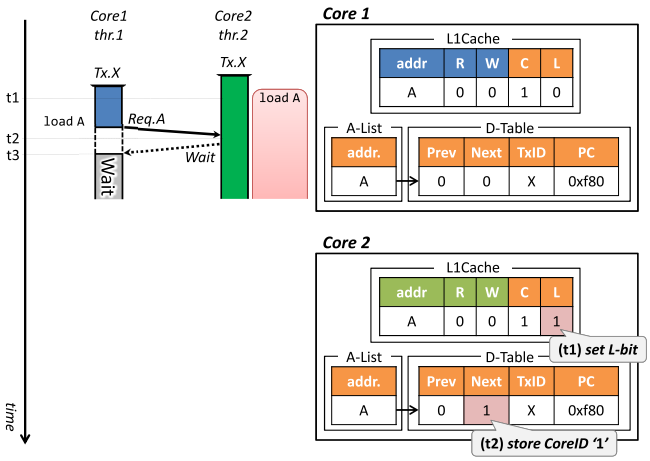

図 5　*Wait* リクエストによってアクセスを待機させる動作

セスを自身に許可したコアの番号

Next-Core(Next):

Address List に記憶したアドレスに対する投機的アクセスを自身が許可すべきコアの番号

Target-TxID(TxID):

Address List に記憶したアドレスに対する Read→Write の順序でのアクセスを含むトランザクションの ID

Target-PC(PC):

Address List に記憶したアドレスに対する Read→Write の順序でのアクセスを含むトランザクション内で，Write アクセスが実行される時点における PC の値

なお実行するプログラムによっては，ある共有変数が複数のトランザクション内で Read→Write の順序でアクセスされる可能性がある．そのため，Dependence Table はそれぞれの共有変数アドレスに対して複数の Target-TxID および Target-PC を記憶できるように構成する．

3.2　動作モデル

本節では，追加ハードウェアを用いた提案手法の具体的な動作モデルについて述べる．

3.2.1　Read→Write の順序でアクセスされるアドレスの記憶

本項では，Read→Write の順序でアクセスされるアドレスに対応する C ビットがセットされるとともに，そのアドレスが Address List に記憶されるまでの動作について，**図 4** の動作例を用いて説明する．この例では，共有変数のアドレス `A` に対する Read→Write の順序でのアクセスを含むトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並行実行している．また，Dependence Table に記憶される Target-TxID および Target-PC は 1 組であると仮定している．

この例において，まず各スレッドが `load A` を実行した後，*thr.2* が `store A` の実行を試みたとすると，*thr.1* はこの時点で既に `A` に対して Read アクセス済みであることから競合を検出する（時刻 t1）．この競合により，*thr.1* から *Nack* が返信されるため，*thr.2* は *Tx.X* をストールさせる．続いて，*thr.1* が `store A` の実行を試み，`A` に対するアクセスリクエスト *Req.A* を送信したとすると，*thr.2* は競合を検出し，*thr.1* に *Nack* を返信する（t2）．ここで，これらのスレッド間でデッドロックが発生してしまうため，*thr.1* が *Tx.X* をアボートする（t3）．この時，`A` が Read→Write の順序でアクセスされたか否かをチェックするために，*thr.1* は `A` に対応する R ビットを参照する．図 4 の例では，*Core.1* における L1 キャッシュ上の `A` に対応する R ビットが既にセット済みであることから，*thr.1* は `A` が Read→Write の順序でアクセスされるアドレスであることが分かる．そのため，*thr.1* は `A` に対応する C ビットをセットするとともに，アドレス `A` を Address List に格納する（t3'）．さらに，この時 *thr.1* は自身が試みた `store A` の実行地点を，`A` に対する Read→Write の順序でのアクセスが完了する地点であると判断し，この時点におけるプログラムカウンタの値および現在実行しているトランザクションの ID である 'X' を Dependence Table に格納する．これにより，以後 *thr.1* は再度 *Tx.X* を実行した際に，`A` に対する Write アクセスの実行を完了したか否かを判断できる．その後，*thr.1* は `A` に関する C-bit，A-List および Dependence Table の値を，*Info.* メッセージによって *thr.2* に伝える（t4）．なお，この *Info.* メッセージはコヒーレンスプロトコルを拡張することで新たに定義する．

3.2.2　依存関係情報の利用による投機的な実行の実現

3.2.1 項で述べた動作により記憶した情報を利用し，Read→Write の順序でアクセスされるアドレスに対する投機的なアクセスを許可する動作例を**図 5**，**図 6** および**図 7**

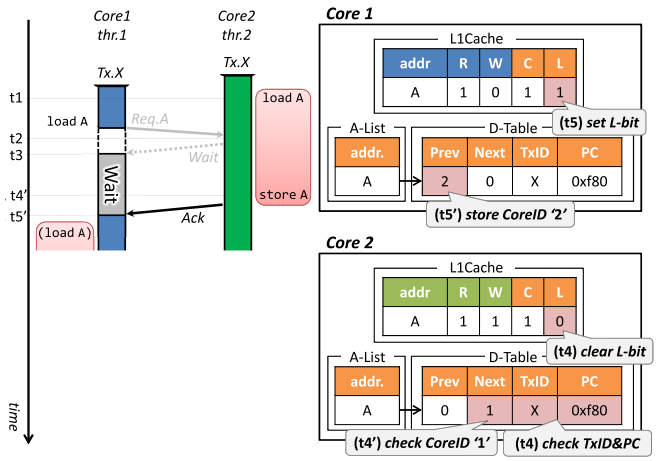

図 6　待機スレッドの投機的アクセスを許可する動作

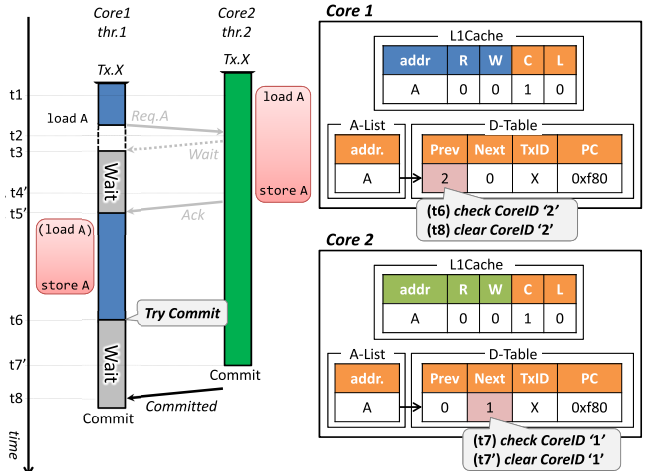

図 7　トランザクションのコミット順序を制御する動作

に示す．これらの例では，共有変数のアドレス A に対する Read→Write の順序でのアクセスを含むトランザクション *Tx.X* を，2 つのスレッド *thr.1* および *thr.2* が並行実行している．また，各コアの L1 キャッシュ上の A に対応する C ビットが既にセットされており，さらに 2 つの表にアドレス 'A'，トランザクション ID 'X' およびプログラムカウンタの値 '0xf80' が格納されているとする．

この状態でまず，図 5 に示すように，*thr.2* が `load A` の実行を試みたとする．この時，C ビットのセットされている A に Read アクセスした *thr.2* は，自身が A に対する Read→Write の順序でのアクセスを含む処理のうち，その Read アクセスから Write アクセスまでの処理区間を排他的に実行中であると判断し，A に対応する L ビットをセットする（時刻 t1）．その後，*thr.1* が同様に `load A` の実行を試み，A に対するアクセスリクエスト *Req.A* を *thr.2* へ送信したとする．この *Req.A* を受信した *thr.2* は，A に対応する C ビットと L ビットを参照する．この時，これらのビットがセット済みであることから，*thr.2* は *thr.1* のアクセスを待機させる必要があると判断する．そのため，*thr.2* は *thr.1* に *Wait* リクエストを送信し，*thr.1* の A に対する Read アクセスを待機させる（t2）．また，同時に *thr.2* は，*thr.1* のアクセスを後に許可できるようにするために，*thr.1* が動作しているコア番号を取得し，この値である 1 を Dependence Table の Next-Core に格納する．

ここで，この例において，C ビットおよび L ビットが共にセット済みである A に *thr.2* が `store A` を試みたとする（図 6，時刻 t4）．この時，*thr.2* は Read→Write の順序でアクセスされる A に対する Write アクセスを完了したか否かを確認するために，Dependence Table を参照する．そして，*thr.2* は自身が実行しているトランザクションの ID およびこの時点におけるプログラムカウンタの値を，Dependence Table の Target-TxID および Target-PC に格納されている値と比較する．この時，これらの値が一致したとすると，*thr.2* は A に対する Write アクセスを完了したと判断し，A に対応する L ビットをクリアする．これにより，*thr.1* が待機している A に対する投機的アクセスを，*thr.2* が許可できるようになるため，*thr.2* は Dependence Table の Next-Core に格納されているコア番号を参照する（t4'）．その結果，*thr.2* はコア番号 1 を取得することになるため，*Core.1* において Read アクセスを待機している *thr.1* に *Ack* を送信する．この *Ack* を受信した *thr.1* は，A に対する Read アクセスが *thr.2* により許可されたと判断し，待機していた Read アクセスを投機的に実行する（t5）．なお，このように *thr.2* が *thr.1* の投機的アクセスを許可したことで，*thr.2* の実行する *Tx.X* 内で更新された値を用いて *thr.1* が *Tx.X* の処理を進めることになる．そのため，*thr.2* は *thr.1* よりも先に *Tx.X* をコミットしなければならない，というトランザクションのコミット順序に関する制約が発生する．このコミット順序を制御するために，*thr.1* は *thr.2* が動作しているコア番号を取得し，この値である 2 を Dependence Table の Prev-Core に格納する（t5'）．

その後，各スレッドの処理が進み，*thr.1* が *Tx.X* のコミットに到達した際（図 7，時刻 t6），*thr.1* は Dependence Table の Prev-Core に格納されているコア番号を参照する．その結果，コア番号 2 が取得されるため，*Core.2* 上で動作する *thr.2* が実行トランザクションをコミットするまで，*thr.1* は *Tx.X* のコミットを待機する．その後，*thr.2* が *Tx.X* のコミットした際（t7），*thr.2* は Dependence Table の Next-Core に格納されているコア番号を参照する．その結果，*thr.2* はコア番号 1 を取得するため，*Tx.X* をコミットしたことを *Core.1* に伝える必要があると判断し，*Core.1* に *Committed* メッセージを送信するとともに，Dependence Table の Next-Core に格納されているコア番号をクリアする（t7'）．この *Committed* メッセージを受信した *thr.1* は，*thr.2* の実行する *Tx.X* がコミットされたと判断し，Dependence Table の Prev-Core に格納されているコア番号をクリアするとともに，自身の *Tx.X* をコミット

する（t8）.

4. 関連研究

競合の発生を抑制するという観点から行われた研究として，Yoo ら [6] は HTM に Adaptive Transaction Scheduling と呼ばれる方式を適用することで，競合の頻発によって並列性が著しく低下するアプリケーションの実行を高速化する手法を提案している．一方で，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手法を提案している．また，Akpinar ら [8] は HTM の性能を低下させるような競合パターンに対する，様々な競合解決手法を提案している．

また Bobba ら [3] は，本研究と同様に共有変数に対するアクセス順序に着目し，Store Predictor という機構を用いたスケジューリング手法を提案している．この Store Predictor とは，実行プログラム中で一度でも Read→Write の順序でアクセスされたアドレスを記憶しておくための機構である．Bobba らの手法では，各スレッドがこの機構に記憶されたアドレスに Read アクセスを試みる際に，他のスレッドに対して Read アクセスリクエストではなく，Write アクセスリクエストを送信する．これにより，既に当該アドレスに Read アクセス済みである他のスレッドは Write after Read 競合を検出して *Nack* を返信するため，複数のスレッドが Read アクセスのみを完了した状態となってしまうことを防ぎ，Futile Stall を抑制できる．しかし，この手法で用いる Store Predictor には，実行プログラム中で一度でも Read→Write の順序でアクセスされた変数のアドレスが全て記憶される．そのため，条件分岐などにより必ずしも Read→Write の順序でアクセスされるとは限らないアドレスに対しても，この手法の動作が適用されてしまい，実行するプログラムによっては大幅な性能低下に繋がってしまう可能性がある．

このように，Bobba らの手法は本研究と着眼点が共通しているため，後述する 5 章で提案手法との比較評価を行う．

5. 性能評価

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，考察を行う．

5.1 評価環境

これまで述べた提案手法を，HTM の研究で広く用いられている LogTM[2] 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 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環境を示す．評価対象のプログラムとしては GEMS microbench, SPLASH-2[11], および STAMP[12] から計 12 個を使用し，各ベンチマークプログラムを 16 スレッドで実行した．

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

5.2 評価結果

評価結果を**図 8**，および**表 2** に示す．図 8 では，各ベンチマークプログラムの評価結果をそれぞれ 4 本のバーで表しており，左から順に，

(B) 既存の LogTM（ベースライン）
(R) Store Predictor を用いる既存手法
(E) 同一の共有変数に対する Read→Write の順序でのアクセスを含む処理を排他実行するモデル
(P) 同一の共有変数に対する Read→Write の順序でのアクセスが完了した時点で，投機的アクセスを許可する提案モデル

の実行サイクル数の平均を表しており，モデル (B) の実行サイクル数を 1 として正規化している．なお，フルシステムシミュレータ上でマルチスレッドによる動作シミュレーションを行う際には，性能のばらつきを考慮する必要がある [13]．したがって，各対象につき試行を 10 回繰り返し，得られた結果から 95%の信頼区間を求めた．信頼区間は図中にエラーバーで示す．なお，4 章でも述べたように，参考モデル (R) で用いる Store Predictor には Read→Write の順序でアクセスされるアドレスが記憶される．本評価では，この参考モデル (R) の理想的な性能を評価するために，ベンチマークプログラム中で出現する，Read→Write の順序でアクセスされる全てのアドレスを Store Predictor に記憶できる状況における，参考モデル (R) の性能を評価した．

図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，

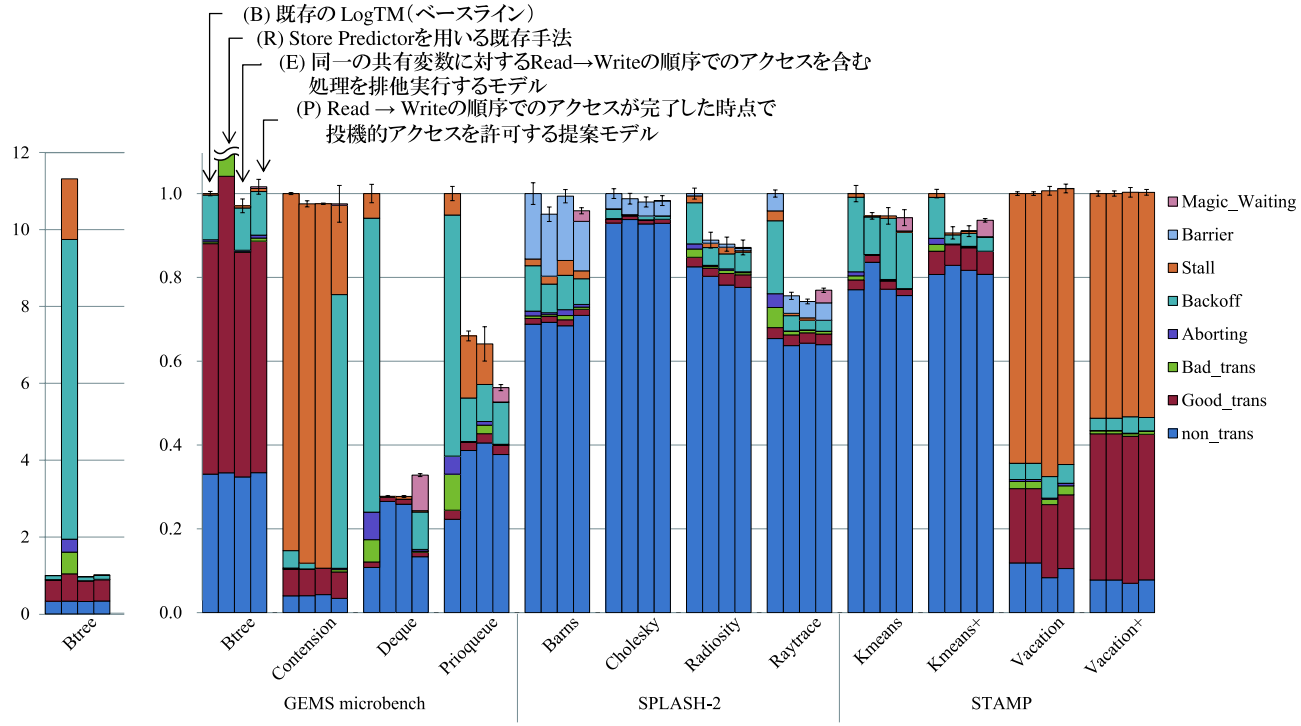

図 8　各プログラムにおけるサイクル数比

表 2　各ベンチマークプログラムにおけるサイクル削減率

		GEMS	SPLASH-2	STAMP	All
(E)	平均	28.4%	9.4%	3.0%	13.6%
	最大	72.3%	25.7%	7.5%	72.3%
(P)	平均	28.6%	10.4%	2.7%	13.9%
	最大	67.2%	23.1%	6.4%	67.2%

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 は参考モデル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．

まず，図 8 に示す評価結果のグラフを見てみると，Btree において参考モデル (R) の性能が大幅に低下していることが見てとれる．さらに，ほぼ全てのプログラムで (E) および (P) は既存モデル (B) と比較して，大幅に性能向上していることが分かる．このことから，これらのモデルでは Futile Stall とそれに起因するアボートを十分に抑制できることが確認できた．ここで，(E) および (P) の性能向上率をまとめると，(E) は既存モデル (B) に対して最大 72.3%，平均 13.6%の性能向上を達成できており，提案モデル (P) は既存モデル (B) に対して最大 67.2%，平均 13.9%の性能向上を達成できた．

5.3　考察

まず (E) と提案モデル (P) を比較すると，Prioque および Radiosity において提案モデル (P) の方が (E) よりも性能向上している．この理由として，これらのプログラムでは (E) においてデッドロックの誤検出によるアボートが頻繁に発生していたことが挙げられる．ここで，そのような問題が発生してしまう例を，図 9(a) に示す．この例では，トランザクション *Tx.P*，*Tx.Q* および *Tx.R* を，3 つのスレッド *thr.1*〜*thr.3* がそれぞれ実行している．はじめに，*thr.2* が自身よりもトランザクション開始時刻の早い *thr.3* に対して *Nack* を返信した際に，possible_cycle フラグがセットされる（時刻 t1）．この possible_cycle フラグは各コアが他のスレッドで実行中のトランザクションをストールさせている場合にセットするフラグであり，デッドロックの検出に用いられる．その後，*thr.2* は `load A` の実行を試みるが（t2），この時点では既に *thr.1* が `load A` を実行しているため，*thr.1* は (E) の動作に従って *thr.2* に *Nack* を

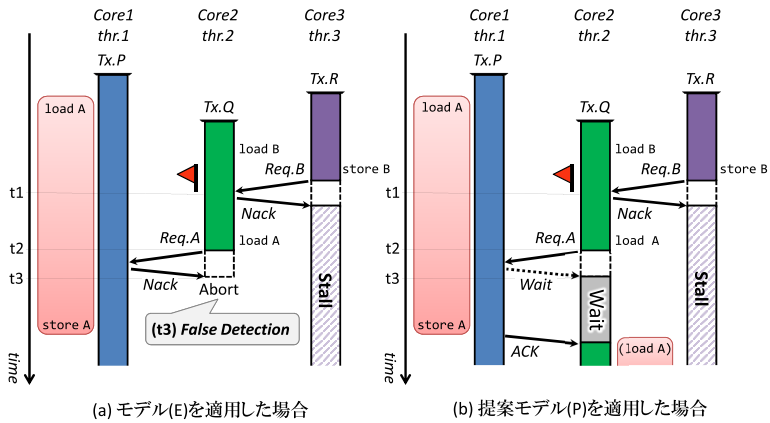

図 9　デッドロックの誤検出によるアボート

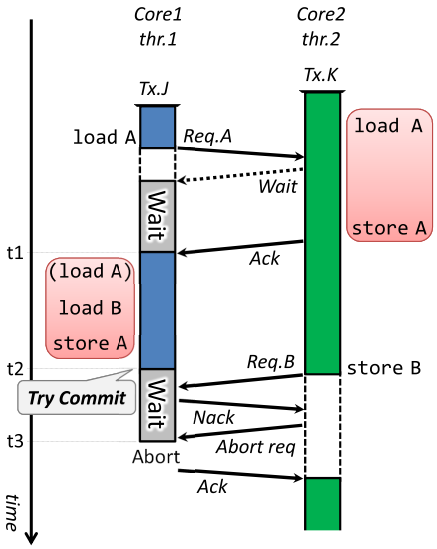

図 10　デッドロック状態を回避するための例外処理に起因するアボート

返信する．この時，*thr.2* は possible_cycle フラグ をセットした状態で，自身よりも早くトランザクションを開始した *thr.1* から *Nack* を受信することでデッドロックが発生したと誤検出し，*Tx.Q* をアボートしてしまう（t3）．これに対し，同様の状況で提案モデル (P) の動作を適用した例を図 9(b) に示す．この例では，時刻 t3 において *thr.2* が *thr.1* から *Nack* ではなく *Wait* リクエストを受信することでデッドロックの誤検出による *thr.2* のアボートを防ぐことができ，さらに *thr.1* が *thr.2* の投機的アクセスを許可できる．Prioque と Radiosity では，上述したような動作が頻繁に発生したため，提案モデル (P) が (E) よりも高い性能を達成できたと考えられる．

続いて，Btree，Contention，Deque，Raytrace および Kmeans+の 4 つのプログラムでは，(E) の方が提案モデル (P) よりも性能向上している．この理由として，これらのプログラムでは提案モデル (P) の例外処理である，デッドロック状態を回避するためのアボートが頻繁に発生していたことが挙げられる．ここで，そのような問題が発生してしまう例を**図 10** に示す．この例では，トランザクション *Tx.J* および *Tx.K* を，2 つのスレッド *thr.1* および *thr.2* がそれぞれ実行しており，*thr.1* が *thr.2* によって `load A` の実行を投機的に許可された後に（時刻 t1），*thr.1* が *thr.2* よりも早くトランザクションのコミットに到達することで *thr.2* のコミットを待ち続けている（t2）．このような状況で，*thr.2* が `store B` の実行を試みたとすると，この時点では既に *thr.1* が `load B` を実行済みであることから，*thr.1* は *thr.2* に対して *Nack* を返信する．この *Nack* により，これらのスレッド間でデッドロック状態が発生するため，*thr.2* は例外処理として *thr.1* に対してアボートリクエストを送信し，*thr.1* の *Tx.J* を結果としてアボートさせてしまう（t3）．提案モデル (P) では，上記のような動作に起因するアボートの繰り返しによって，(E) よりも性能が低下してしまったと考えられる．したがって，今後はプログラムごとにアクセスパターンを詳細に調査し，投機的アクセスの許可対象とする変数をより適切に決定できる枠組みを検討する必要がある．

5.4　ハードウェアコストとアクセスオーバヘッド

本節では，提案手法を実現するために追加したハードウェアの実装コストとそのアクセスオーバヘッドについて述べる．

5.4.1　ハードウェアコスト

提案手法では C ビット，L ビットと Address List および Dependence Table の 2 つの表が追加される．このうち，Address List と Dependence Table には，各トランザクションで Futile Stall とそれに起因するアボートが引き起こされたアドレスを全て記憶できるだけのエントリ数が必要となる．さらに，Dependence Table には，ある共有変数に対する複合操作が含まれているトランザクションの最大数だけ，Target-TxID と Target-PC を記憶するフィールドが必要となる．そこで，各プログラムに提案手法の動作を適用した際に，上述したエントリ数とトランザクション数がどの程度になるのかを調査した．その結果，Address List は 10 行のエントリ，そして Dependence Table は 10 行のエントリおよび 3 組の Target-TxID と Target-PC を記憶するフィールドがあれば，今回評価に用いたプログラムでは情報を全て記憶できることが分かった．ここで，Address List の 1 つのエントリで必要となる記憶容量は，1 つの Target-Address に対して 64bits である．一方，Dependence Table の 1 つのエントリで必要となる記憶容量は Prev-Core，Next-Core，Target-TxID，Target-PC に対してそれぞれ 4bits，4bits，4bits，64bits である．なお，Address List は格納されているアドレスを高速に検索する必要があるため CAM で構成する．これに対し，Dependence Table の各エントリは Address List の各エントリと一対一に対応しており，Address List のインデクスを用いて高速に検索が行うことができるため RAM で構成する．つまり，Address List は 1 行あたり 64bits の幅を持つエントリが 10 行ある CAM で構成でき，Dependence Table は 1 行あたり $4\text{bits} + 4\text{bits} + (4\text{bits} + 64\text{bits}) \times 3 = 212\text{bits}$ の幅を持つエントリが 10 行ある RAM で構成できる．また，この提案手法では L1 キャッシュラインに対して C ビットと L ビットを追加するため，1 ラインあたり 2bits のフィールドが必要となる．したがって，提案手法を実装するために必要となるハードウェアコストは，16 スレッドを実行可能な 16 コア構成のプロセッサにおいて約 6KBytes となり，1 コアあたり約 350Bytes となる．この 350Bytes という数値は 1 コアあたりの L1 キャッシュサイズである 32KBytes

の約 1%と十分に小さいものである．

5.4.2 アクセスオーバヘッド

本項では，提案モデル (P) で追加した 2 つの表のアクセスオーバヘッドが性能に及ぼす影響について述べる．まず，2 つの表のアクセスオーバヘッドはそれぞれの表を参照した総回数 C，そして参照時のレイテンシ T を用いて，$C \times T$ として概算する．なお，5.4.1 項でも述べたとおり，Address List は 10 行のエントリを持つ CAM で構成されるため，この表を一般的な TLB と同じ 1cycle のレイテンシで参照できると仮定する．一方，Dependence Table は 212bits の幅を持つエントリが 10 行ある RAM で構成される．この Dependence Table の各エントリを参照する際には，Address List のインデクスから対象のエントリを検索するのに 1cycle が，そして Dependence Table の各フィールドに対するマスク操作と比較操作にそれぞれ 1cycle が必要となると仮定する．また，Dependence Table には 3 組のトランザクション ID とプログラムカウンタの値が格納されていることから，各フィールドに対するマスク操作と比較操作が最大で 3 回行われることになる．したがって Dependence Table は，最大で $1\text{cycle} + (1\text{cycle} + 1\text{cycle}) \times 3 = 7\text{cycles}$ のレイテンシで参照できる．この概算したレイテンシとアクセス回数から，2 つの表のアクセスオーバヘッドが各ベンチマークプログラムの総実行サイクル数に占める割合を算出したところ，その割合が最大となる Prioque でも 0.89%となり，非常に小さなものであることが確認できた．なお，このオーバヘッドは 2 つの表の構成や動作アルゴリズム次第でさらに小さなものにできると考えられる．

6. おわりに

本稿では，共有変数に対してそれ以降変更が行われないと判断した時点で他スレッドによる投機的アクセスを許可する手法を提案した．提案手法では，各スレッドが実行トランザクション内で同一変数に対する Read→Write の順序でのアクセスを完了した時点で，その変数への変更が完了したと判断し，当該トランザクションのコミットに先立って他のスレッドによる当該変数への投機的なアクセスを許可することで，HTM のさらなる高速化を目指した．

提案手法の有効性を確認するために，既存の HTM を拡張し，GEMS microbench，SPLASH-2 および STAMP を用いてシミュレーションによる評価を行った．評価の結果，提案手法は既存の HTM と比較して，16 スレッド実行時で最大 67.2%，平均 13.9%の高速化を達成でき，複合操作の排他実行を行う手法と比較しても高速化を達成できるプログラムがあることを確認した．また，提案手法を実現するために必要な追加ハードウェアのコストを概算したところ約 6KBytes となり，少量であることを確認した．

しかし提案手法では，同一変数に対する Read→Write の順序でのアクセスの後にさらに当該変数に対するアクセスを含むようなトランザクションが実行された場合，投機的アクセスの許可に起因するメモリ一貫性の欠如した状態の発生を防ぐための例外処理が行われる．これにより，トランザクションがコミットされるまで他のスレッドが当該変数にアクセスできず，待機処理が増大してしまう可能性がある．したがって，今後はベンチマークプログラムごとにアクセスパターンを詳細に調査し，同一の共有変数に対する Read→Write の順序でのアクセスが完了する地点をより厳密に記憶できる枠組みを検討する必要がある．

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] 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).

[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 (ACS44)， Vol. 6, No. 4, pp. 58–71 (2013).

[5] 橋本高志良，井出源基，山田遼平，堀場匠一朗，津邑公暁：共有変数に対する複合操作を排他実行するハードウェアトランザクショナルメモリの高速化，情処研報 (ARC200)，Vol. 2014-ARC-208, No. 22, pp. 1–8 (2014).

[6] Yoo, R. M. and Lee, H.-H. S.: Adaptive Transaction Scheduling for Transactional Memory Systems, *Proc. 20th Annual Symp. on Parallelism in Algorithms and Architectures (SPAA'08)*, pp. 169–178 (2008).

[7] Blake, G., Dreslinski, R. G. and Mudge, T.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th International Conference on High-Performance Computer Architecture (HPCA-17 2011)*, pp. 75–86 (2011).

[8] Akpinar, E., Tomić, S., Cristal, A., Unsal, O. and Valero, M.: A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory, *Proc. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)* (2011).

[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] 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).