Vol.2017-ARC-227 No.40
2017/7/28

実行フェーズを考慮した トランザクショナルメモリのスケジューリング手法

多治見 知紀[1]　廣田 杏珠[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

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，ロックにはデッドロックの発生や並列度の低下などの問題があるため，ロックを使用しない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．この機構をハードウェア上に実装したハードウェアトランザクショナルメモリ（HTM）では，トランザクションを投機的に並列実行することで，ロックに比べ並列度が向上する．しかし，HTM では同一共有変数へのアクセス競合が頻発することによる性能低下が問題となる．この問題に対し，トランザクションのスケジューリングを改良することにより競合の発生を抑制する研究が多く行われてきたが，そのいずれの手法を用いても，十分な性能向上が得られていないプログラムが存在する．そこで本稿ではまず，そのようなプログラムが持つメモリアクセスパターンを調査し，性能向上が妨げられている原因を調査した．その結果，複数の実行フェーズを持ち，あるフェーズでアクセスした共有変数に対し，以降のフェーズで再度アクセスしないようなトランザクションが存在することが分かった．そこで本稿では，トランザクションが持つ実行フェーズを考慮して競合検出を行うことで性能を向上させる手法を提案する．Contention および Deque を用いて評価を行った結果，Contention で平均 63.2%，Deque で平均 6.3%の性能向上を確認した．

1. はじめに

マルチコア環境の普及に伴い，プログラマが容易に並列処理を記述できる，共有メモリ型並列プログラミングの重要性が増している．この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一般的にロックが用いられることが多いが，ロック操作のオーバヘッドに伴う並列度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の粒度を設定することは困難であるため，ロック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，ロックを使用し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 では従来ロックとして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**として定義し，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，トランザクションを投機的に並行実行することで，ロックを用いる場合に比べて並列度が向上する．なお，TM ではトランザクションの実行が投機的であるため，共有変数の値が更新される際は，更新前と更新後の両方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で同一リソースに対するアクセス競合が発生していないかを監視する必要がある（競合検出）．**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オーバヘッドを軽減している．

さて，この HTM ではスレッドが同一のトランザクションを実行するたび，ほぼ同一の命令列を実行し，同じ共有変数にアクセスする可能性が高い．したがって，一度競合したトランザクション同士で競合が再発しやすく，これによる性能低下が問題となる．この問題に対し，トランザクションのスケジューリングを改良することにより競合の発生を抑制する手法が数多く提案されてきたが，いずれの手法を用いても十分な性能向上を達成できていないプログラムが存在する．本稿ではまず，そのようなプログラムの持つメモリアクセスパターンの特徴を調査する．次に，その特徴を持つプログラムに対し，性能向上を達成するスケジューリング手法を提案する．

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 名古屋大学
Nagoya University
[3] 国立情報学研究所
National Institute of Informatics

2. 関連研究

実行トランザクションをアボートした後に，そのトランザクションを途中から再実行することで，再実行に要するコストを抑える部分ロールバックに関する研究 [2] [3] [4] や，トランザクションの様々な情報に基づいて競合を抑制する研究 [5] [6] [7] など，HTM に関する数々の研究がなされてきた．特に，複数のスレッド間で実行順序などを制御するスレッドスケジューリングに関して様々な改良手法が提案されてきた．

Yoo ら [8] は HTM に Adaptive Transaction Scheduling と呼ばれるスケジューリング機構を実装し，並列に実行するトランザクションの数を制御することで，競合の頻発によって並列度が著しく低下するようなアプリケーションの実行を高速化するスケジューリング手法を提案している．一方で，Blake ら [9] 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．さらに，Hirota ら [10] は一度競合したトランザクション同士は再度競合を引き起こしやすいという特徴から，トランザクションの実行開始前に競合の発生を予測し，トランザクションを実行開始しても競合が発生しないタイミングまでトランザクションの実行開始を待機することで，競合を未然に回避する競合予測手法を提案している．また，Bobba ら [11] は共有変数に対するアクセス順序に着目し，Read, Write の順にアクセスされる共有変数に対する競合検出を改良することで，無駄なストールを削減している．

しかし，以上で述べたいずれのスケジューリング手法を用いても，一部のプログラムでは競合に起因するオーバヘッドが削減できていない．そこで本稿では，そのようなプログラムが持つメモリアクセスパターンの特徴を調査し，性能向上を達成するスケジューリング手法を提案する．

3. 性能向上を妨げるメモリアクセスパターン

本章ではまず，既存の HTM の競合解決方法と問題点について述べる．次に，これまで提案されてきた様々なスケジューリング手法を用いても，十分な性能向上が得られていないプログラムについて，そのメモリアクセスパターンを調査し，性能向上を妨げている原因を特定する．

3.1 HTM における競合解決と問題点

本節では既存の HTM における競合解決の動作について，**図 1** を用いて説明する．この図の例において，2 つのスレッド *Thread*0, *Thread*1 がそれぞれ異なるトランザクション *Tx*.0, *Tx*.1 を実行しており，*Thread*0 が `store A` を，*Thread*1 が `store B` を実行済みの場合について考える．なお，本稿では *Tx*.0 は ID が 0，*Tx*.1 は ID が 1 のトランザクションを表す．この状態で，*Thread*0 は `store B` の実行を試みて，*Thread*1 に対しアドレス B へのアクセス許可を求めるリクエストを送信する（時刻 t1）．これを受信した *Thread*1 は `store B` を実行済みであるため競合を検出し，*Thread*0 に対し *NACK* を返信する（t2）．*Thread*0 は *NACK* を受信すると `store B` を実行せず，*Tx*.0 をストールさせる（t3）．その後，*Thread*1 が `store A` の実行を試みて，*Thread*0 に対しリクエストを送信すると，それを受け取った *Thread*0 は `store A` を実行済みであるため，競合を検出し *Thread*1 に対し *NACK* を返信する．これにより，*Thread*0 は自身が *NACK* を送信した先である *Thread*1 から *NACK* を受信するため，このままではデッドロック状態に陥ってしまう．そこで，*Thread*1 は自身が行った `store B` の結果を破棄するため，*Tx*.1 をアボートする（t4）．これにより，*Thread*0 はアドレス B にアクセス可能となり，`store B` を実行する（t5）．一方，*Thread*1 は一定時間待機した後，*Tx*.0 を再実行する（t6）．

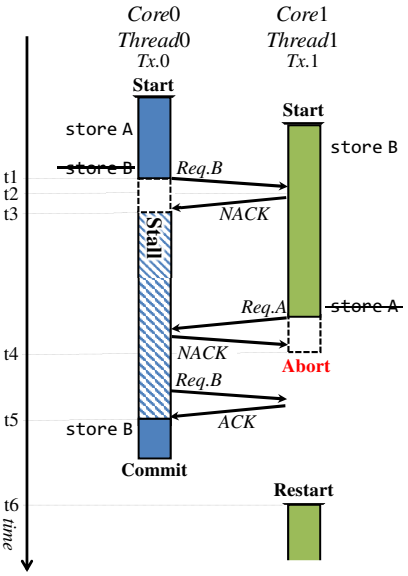


図 1 既存 HTM における競合解決

以上のように，HTM では競合が発生すると，ストールによりトランザクションの進行が停止し，並列度が低下する．このストール中のトランザクションは実行が進行していないにも関わらず，共有変数にアクセス済みであるため，新たな競合を引き起こす原因となってしまう．また，トランザクションをアボートした場合，そのトランザクションを再実行するなどのペナルティも発生し，これらによる性能低下が問題となる．

3.2 既存のスケジューリング手法の問題点

以上のように，HTM では競合により性能が低下するため，スレッド間でトランザクションの実行順序などを制御することで競合を抑制する様々なスケジューリング手法が

```
1  int A[1];
2  int B[1024];
3
4  BEGIN_TRANSACTION(0);
5  // phase1
6  for( i = 0; i < 10; i++ ){
7    if(access_type[i] == READ )
8      var = A[1];
9    else
10     A[1] = 0;
11 }
12
13 // phase2
14 for( i = 10; i < 100; i++ ){
15   if(access_type[i] == READ )
16     var = B[index[i]];
17   else
18     B[index[i]] = 0;
19 }
20 COMMIT_TRANSACTION(0);
```

図 2　Contention のトランザクションを簡略化したコード

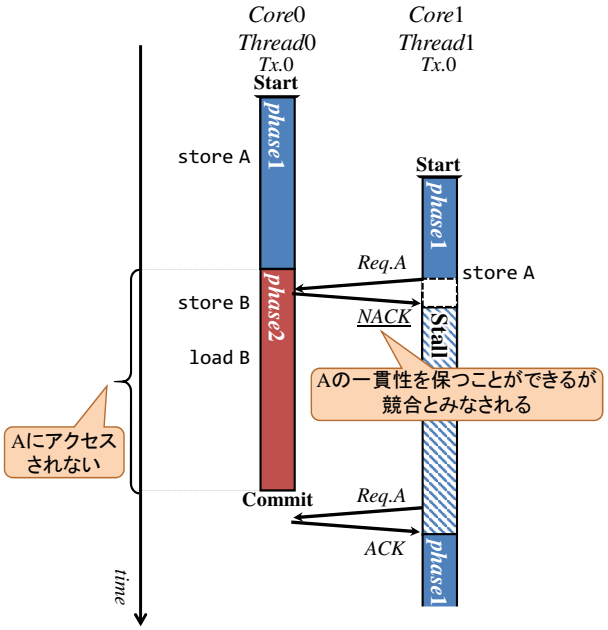


図 3　一貫性を損なわないにも関わらず競合として検出されるメモリアクセス

提案されてきた．我々も，共有変数へのアクセスパターンによってトランザクションを排他実行する手法 [12] や，優先度という値を設定し，競合を引き起こしやすいトランザクションを優先的に実行しコミットさせる手法 [13]，トランザクションの実行開始前に競合の発生を予測し，競合を回避する手法 [10] など，様々なスケジューリング手法を提案している．

しかし，そのいずれの手法についても，GEMS microbench [14] のベンチマークプログラムの 1 つである Contention では，実行サイクル数の多くを占めるストールを削減できておらず，競合に起因するオーバヘッドを削減できていない．そこで本稿ではまず，この Contention のプログラムが持つメモリアクセスパターンに着目し，性能向上が妨げられている原因を調査する．

3.3 Contention

Contention はトランザクション内で異なる 2 つの配列に対し，ロードまたはストアのいずれかを繰り返し行うプログラムである．Contention が持つトランザクションを簡略化したコードを図 **2** に示す．なお，4 行目の `BEGIN_TRANSACTION(0)` および 20 行目の `COMMIT_TRANSACTION(0)` はそれぞれトランザクションの開始と終了を表し，その引数はトランザクションの ID を表している．このトランザクションは，2 つの配列のうち配列 A のみにアクセスする *phase*1 と，配列 B のみにアクセスする *phase*2 の 2 つのフェーズからなる．16 行目および 18 行目で参照されている配列 index[i] には乱数が格納されており，配列 B のアクセス先要素はランダムに決定される．

このトランザクションでは *phase*1 で配列 A にアクセスするが，*phase*1 から *phase*2 に移行した後は配列 A へはアクセスしなくなる．このように，トランザクションがあるフェーズでアクセスしたアドレスに対し，以降のフェーズで再度アクセスしないという特徴を持つ場合，従来では競合とみなされる他スレッドからのアクセスリクエストを許可しても一貫性が損なわれない場合がある．

ここで，一貫性が損なわれないにも関わらず競合として検出されてしまうメモリアクセスについて，図 **3** の例を用いて説明する．この図において，*Thread*0 が *Tx*.0 を開始すると *phase*1 内で配列 A にアクセスし，*phase*1 が終了すると，*phase*2 内では *Thread*0 が再度配列 A にアクセスすることはない．したがって，*Thread*0 が *phase*2 を実行している際に *Thread*1 によって配列 A の値が更新されても，2 つのスレッドの間で配列 A の一貫性は損なわれない．しかし，既存の HTM ではこのようなメモリアクセスを競合とみなしてしまうため，Contention の性能向上が妨げられていた．そこで，このような実行フェーズを考慮し，本来競合として検出されるメモリアクセスを投機的に許可することで，競合を抑制するスケジューリング手法を提案する．

4. 実行フェーズを考慮したスケジューリング手法

本章では，実行フェーズを考慮して競合を検出することで，既存手法では競合として検出されていたメモリアクセスを投機的に許可する提案手法について述べる．

4.1 実行フェーズを考慮した競合検出

3.3 節で述べたように，トランザクションが，アクセス先アドレスの異なる複数の実行フェーズを持つ場合，並列に実行しているトランザクションの実行フェーズが異なって

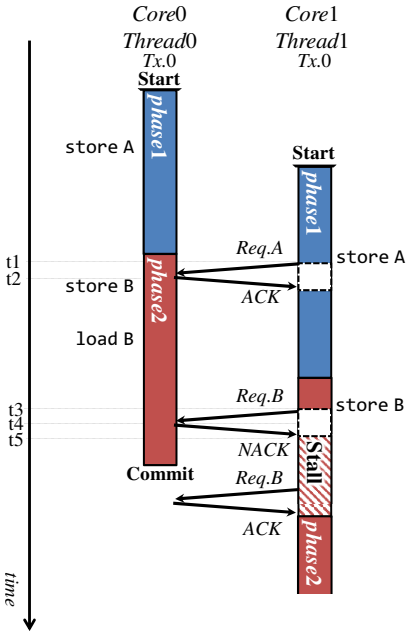


図 4　実行フェーズを考慮し，メモリアクセスを許可する動作

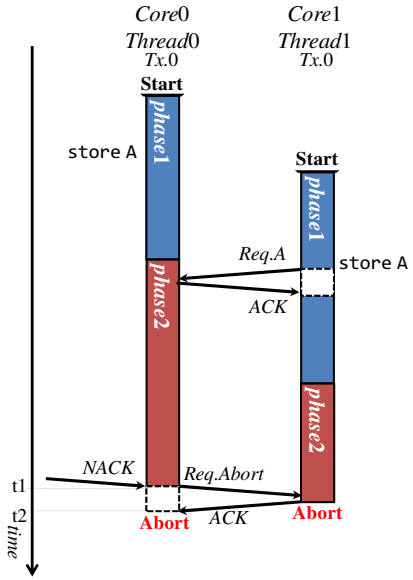


図 5　提案手法で連鎖的に行うアボート

いれば，競合を引き起こすメモリアクセスにより一貫性が損なわれない場合がある．そこで，競合検出の際にまず，それぞれのトランザクションが実行している実行フェーズを検査する．そして，それらのアクセス先アドレスが異なっていた場合には，そのメモリアクセスを許可することで，性能向上が期待できる．

ここで，提案手法の動作について**図 4** を用いて説明する．なお，このトランザクションは図 3 と同様のトランザクションであるとする．また，各スレッドは過去の実行において，トランザクション内の各フェーズにおけるアクセス先アドレスに関する情報を予め収集しているものとする．この例において，まず *Thread*0 が *Tx*.0 を開始し，*phase*1 内で `store A` を実行したとする，その後，*Thread*1 が *Tx*.0 を実行開始し，*phase*1 内で `store A` を試みると，*Thread*1 はアドレス A に対するリクエスト *Req.A* を *Thread*0 に送信する（t1）．すると，この *Req.A* を受信した *Thread*0 はアドレス A にアクセス済みであるため，従来では *Thread*1 によるアドレス A へのアクセスは競合として検出される．しかし，この *Req.A* を受信した時点で *Thread*0 は *phase*2 を実行中であり，これ以降再度アドレス A にアクセスすることはないと予想できる．したがって，*Thread*0 はこのまま *Thread*1 がアドレス A にアクセスしても一貫性が損なわれないだろうと判断し，*Thread*1 に対し *ACK* を返信する（t2）．このように，従来ではストールにより実行を中断しなければならなかった場合でも，投機的にトランザクションの実行を継続できるため，並列度が向上する．その後，*Tx*.0 の進行が進み *Thread*0 が *phase*2 内で `store B` を実行した後に，*Thread*1 も *phase*2 内で `store B` を試みて，*Thread*0 へアドレス B に対するリクエスト *Req.B* を送信したとする（t3）．この時，この *Req.B* を受信した *Thread*0 は *phase*2 を実行中であり，*Thread*1 がアドレス B にアクセスすると一貫性が損なわれてしまう．したがって，*Thread*0 は *Thread*1 に対し *NACK* を返信する（t4）．*NACK* を受信した *Thread*1 は *Thread*0 が *Tx*.0 をコミットするまで自身が実行している *Tx*.0 をストールする（t5）．このように，実行フェーズ内でのアクセス先アドレスを考慮することで，一貫性が損なわれるメモリアクセスであるか否かを判別することができる．

以上で述べた動作を実現するために，各スレッドはトランザクションが現在どのフェーズを実行中であるか知る必要がある．そのために，プログラマはフェーズを入れ子のトランザクションとして定義する．これにより，スレッドは実行しているトランザクションの ID から疑似的に実行中のフェーズを知ることができる．

4.2　提案手法で必要となるアボートの動作

図 4 では，*Thread*0 が *Thread*1 からのアドレス A に対するアクセスリクエストを受けた時点で，*Thread*0 はアドレス A に対する操作を終了しており，再度アドレス A にアクセスすることなく自身のトランザクションをコミットするという前提で *Thread*1 のアクセスリクエストを許可する．しかし，トランザクション内の実行パスが変化し，*Thread*0 が再度アドレス A にアクセスするような場合，この前提が崩れてしまう．このような場合，アクセスリクエストを投機的に許可された *Thread*1 は共有変数の一貫性を保つため，自身のトランザクションをアボートする．

また，投機的にアクセスリクエストを許可した *Thread*0 が，自身のトランザクションをアボートする場合についても考慮する必要がある．この時行う動作について**図 5** を用いて説明する．この例において，*Thread*1 によるアクセスリクエストを許可した *Thread*0 が，ストール中の他のスレッ

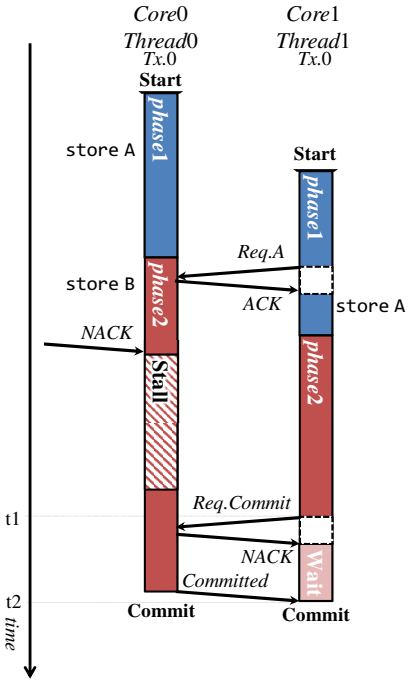


図 6　メモリアクセスを許可されたスレッドがコミットを待機する動作

ドとの間で共有変数に対する競合を検出したとする（t1）．このような場合，*Thread*0 は自身のトランザクションをストールするとデッドロック状態に陥ってしまうためアボートしなければならず，このままでは *Thread*0 がコミットするという前提が崩れてしまう．そこで，*Thread*0 は *Thread*1 によるアクセスを許可した時点で，予め *Thread*1 の ID を記憶しておく．その後，*Thread*0 がトランザクションをアボートする際に，記憶した *Thread*1 に対し，トランザクションをアボートするよう要求するリクエスト *Req.Abort* を送信する．これを受信した *Thread*1 がトランザクションをアボートした後，*Thread*0 も自身のトランザクションをアボートする（t2）．

同様の理由により，*Thread*1 が *Thread*0 より先にトランザクションのコミットに至ったとしても，*Thread*0 がトランザクションをアボートする可能性がある間は *Thread*1 はトランザクションをコミットしてはならない．そこで，**図 6** のように，コミット可能な状態に至った *Thread*1 は *Thread*0 がコミット済みかを確認するリクエスト *Req.Commit* を送信する（t1）．*Thread*0 がコミット前であれば *Thread*1 はコミットを待機することで *Thread*0 のアボートに備える．*Thread*0 がコミットを完了したことを知らせるメッセージ *Commited* を *Thread*1 に送信すると，これを受信した *Thread*1 はコミットしても良いと判断し，自身のトランザクションをコミットする（t2）．

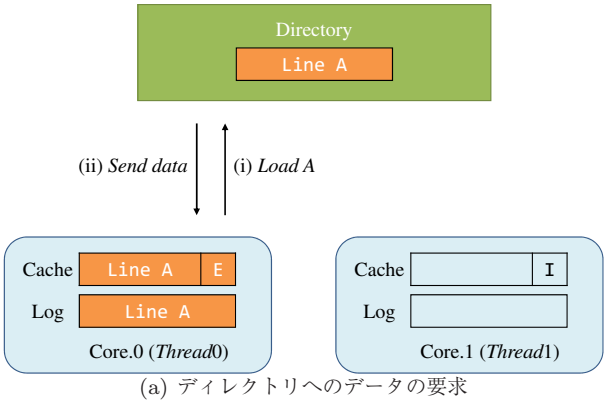
5. 実装

以上で述べた提案手法を．HTM の研究で広く用いられている LogTM-SE [15] 上に実装する．提案手法では，変更済みのキャッシュラインを，スレッドを実行するコア間で共有する必要がある．LogTM-SE は MESI プロトコルをベースに実装されているが，トランザクション内で変更した共有変数を保護するために，更新されたキャッシュラインは，トランザクションがコミットされるまでは，ディレクトリへの書き戻しを伴う共有ができないようになっている．したがって，提案手法ではキャッシュコヒーレンシプロトコルを拡張する必要がある．本章では，既存の LogTM-SE におけるキャッシュコヒーレンス制御の動作について述べた後，提案手法におけるコヒーレンシプロトコルの動作について述べる．

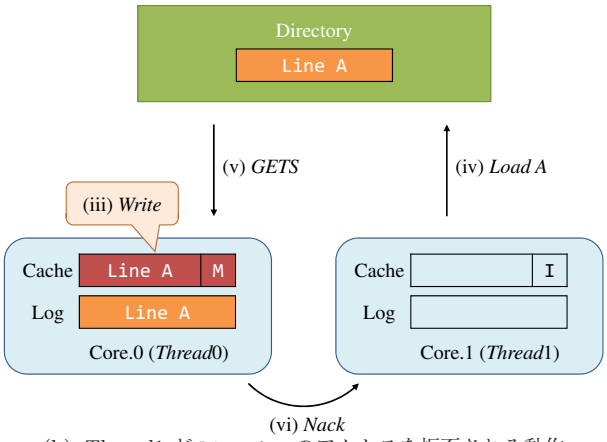
5.1 既存のキャッシュコヒーレンス制御

MESI プロトコルでは，あるプロセッサが保持しているキャッシュラインに対し他プロセッサからのアクセスリクエストを受け取ると，該当するキャッシュラインが変更済みの場合にそれをディレクトリに書き戻す．次に，アクセスリクエストを発行したプロセッサがディレクトリに書き戻されたデータを読み出すことで，プロセッサ間でのデータの受け渡しを実現している．しかし，既存の TM では変更済みかつコミット前の共有変数を，複数のトランザクションで共有することを認めていない．そこで LogTM-SE は，更新済みかつコミット前のキャッシュラインを共有しないよう，MESI プロトコルのキャッシュコヒーレンス制御の機構を拡張している．

ここで，LogTM-SE におけるキャッシュコヒーレンス制御の動作について**図 7** の例を用いて説明する．なお，各スレッドは競合検出を行うために，トランザクション内で実行された，キャッシュラインに対する Read および Write アクセスを記憶している．この図において，Core.0，Core.1 がそれぞれ *Thread*0，*Thread*1 を実行しているとする．図中のキャッシュライン右側に付した `M`，`E`，`S`，`I` はそれぞれ，MESI プロトコルにおける Modified，Exclusive，Shared，Invalid の状態を表している．まず，*Thread*0 がディレクトリにデータを要求する動作を図 7(a) に示す．トランザクションを実行している *Thread*0 がディレクトリに `Line A` の取得を要求するリクエスト *Load A* を送信し（i），これを受信したディレクトリは *Thread*0 に `Line A` のデータを送信する（ii）．このとき，*Thread*0 は `Line A` の変更前の値を Log として保持する．その後，*Thread*0 が書き換えた `Line A` に対し，*Thread*1 がアクセスを試みる動作を図 7(b) に示す．まず *Thread*0 が，取得した `Line A` を書き換えたとする（iii）．次に，同様にトランザクションを実行している *Thread*1 が `Line A` の取得を要求するリクエストをディレクトリに送信すると（iv），ディレクトリは *Thread*0 に `Line A` を共有するよう要求するリクエスト *GETS* を送信する（v）．これを受信した *Thread*0 は `Line A` に Write 済みであるため競合を検出し，*Thread*1 に *Nack* を送信する

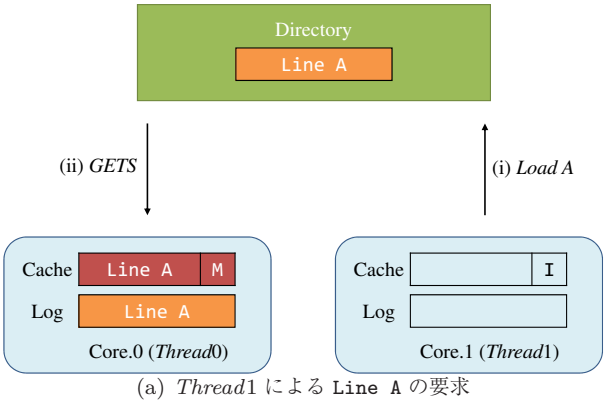


(a) ディレクトリへのデータの要求

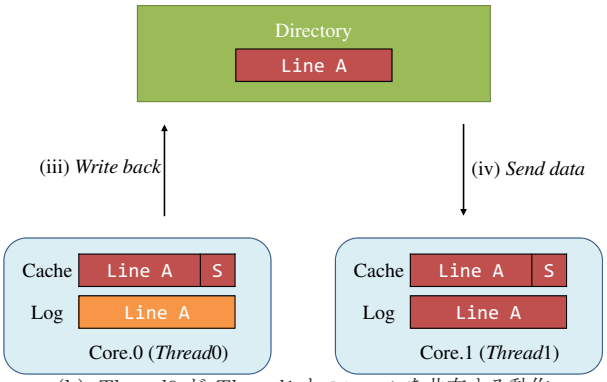


(b) *Thread*1 が `Line A` へのアクセスを拒否される動作

図 7 LogTM-SE におけるキャッシュコヒーレンス制御



(a) *Thread*1 による `Line A` の要求



(b) *Thread*0 が *Thread*1 と `Line A` を共有する動作

図 8 提案手法におけるキャッシュコヒーレンス制御

(vi)．このように LogTM-SE では，変更済みかつコミット前のキャッシュラインを共有しない．なお，トランザクションの実行中にキャッシュがオーバーフローすると，キャッシュラインが変更済みであってもこれをディレクトリに書き戻すが，この場合にもディレクトリに書き戻されたデータに対して他スレッドからのアクセスは許可しない．

以上のように，既存の LogTM-SE ではトランザクションの実行中に変更されたキャッシュラインを複数のプロセッサ間で共有することを許可していないため，提案手法ではこれが可能になるよう動作を拡張する必要がある．

5.2 キャッシュコヒーレンシプロトコルの拡張

提案手法では，共有変数に対するアクセスリクエストが発生した際に，2 つのトランザクションが異なるフェーズを実行中であり，かつそれぞれのフェーズでアクセスするアドレスが異なる場合に，これを競合とみなさず，キャッシュラインを共有するよう動作を拡張する．ここで拡張したキャッシュコヒーレンス制御の動作について，**図 8** を用いて説明する．図 7(b) では *Thread*1 による `Line A` へのアクセスリクエストが競合として検出されるのに対し，提案手法ではこのリクエストに対し，競合が発生したとみなさない．*Thread*1 が `Line A` を取得するリクエストを送信すると（i），ディレクトリは *Thread*0 に `Line A` を共有するよう要求するリクエスト *GETS* を送信する（ii）．次に，これを受信した *Thread*0 が競合を検出せず，キャッシュラインを *Thread*1 と共有する動作を図 8(b) に示す．まず，*Thread*0 が *GETS* リクエストを受信すると，*Thread*0 は自身と *Thread*1 の実行フェーズを検査する．その結果，これらのスレッドがアクセス先アドレスの異なるフェーズを実行中であったとすると，*Thread*0 はこれを競合とみなさず，`Line A` の値をディレクトリに書き戻す（iii）．その後，ディレクトリは *Thread*0 が書き戻したデータを *Thread*1 に送信することで，*Thread*0 と *Thread*1 との間で `Line A` を共有できる（iv）．

提案手法ではこのようにしてプロセッサ間でデータを共有するが，*Thread*0 がトランザクションをアボートした場合，トランザクションを開始する前のメモリ状態を復元する必要がある．既存の LogTM-SE では，アボートしたトランザクションは自身が保持している Log のデータを元のキャッシュラインに書き戻すことで，トランザクション実行前の状態を復元する．提案手法では投機的にアクセスリクエストを許可されたトランザクションも実行開始前の状態を復元する必要があるため，4.2 節で述べたように，*Thread*1 がトランザクションをアボートし，その後 *Thread*0 もトランザクションをアボートする．これにより まず，*Thread*1 は Log に保持しているデータを `Line A` に書き戻す．次に，*Thread*0 も同様に Log のデータを `Line A` に書き戻そうと試みるが，`Line A` は *Thread*1 の書き戻しにより更新済みであるため，*Thread*1 は `Line A` をディ

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

レクトリに書き戻し，*Thread*0 に `Line A` のデータを受け渡す．このとき，*Thread*1 は自身の `Line A` を無効化する．その後，*Thread*0 が `Line A` に Log のデータを書き戻すことで，トランザクション実行開始前のキャッシュライン状態を復元する．なお，*Thread*0 が書き戻しを完了した時点で，*Thread*0 は `Line A` を Modified の状態で保持している．したがって，他のスレッドから `Line A` へのアクセスリクエストが発生した場合は，これをディレクトリに書き戻し，要求したスレッドへデータを渡す．このようにして，*Thread*0 がディレクトリから `Line A` を読みだす前のメモリ状態を復元できる．

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，その結果について考察する．

6.1 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-SE [15] に実装し，シミュレーションにより評価した．評価には Simics 3.0.31 [16] と GEMS 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，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は GEMS microbench の中でもアクセス先アドレスが異なる 2 つの実行フェーズを持つ Contention と Deque を使用し，それぞれ 16 スレッドで実行した．

6.2 評価結果

図 9 では，各ベンチマークプログラムの評価結果をそれぞれ 3 本のバーで表しており，左から順に，

(B)　既存の LogTM-SE

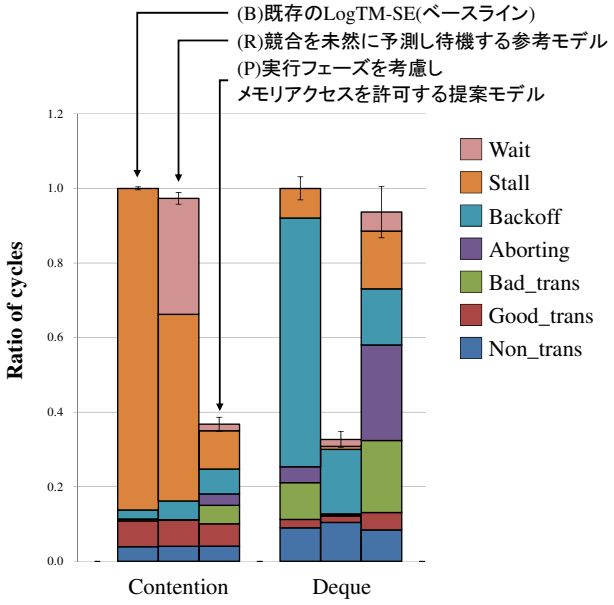


図 9　各プログラムのサイクル数比

(R)　競合を未然に予測し回避する参考モデル [10]

(P)　実行フェーズを考慮しメモリ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，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．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 [17]．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．

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，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 はアボートから再実行までの待機時間である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を示している．また (R) における Wait は，競合を回避するために競合が発生しないタイミングまで自身のトランザクションの実行開始を待機したサイクル数を示し，(P) における Wait はアクセスリクエストを許可したスレッドがコミットを完了するまで，自身のトランザクションのコミットを待機したサイクル数を示す．

評価結果より，Contention では既存モデル (B) に対して 63.2%の性能向上を達成できた．一方，Deque では既存モデルに対して性能向上を達成できたものの，その向上幅は 6.3%に留まり，参考モデルと比較すると性能は大幅に悪化した．

6.3 考察

Contention では，既存モデル (B) および参考モデル (R)

と比較すると，提案モデル (P) では Stall のサイクル数を大きく削減できており，一貫性を損なわないメモリアクセスを許可することで競合を抑制し，並列度が向上していることが分かる．

一方，Deque は既存モデル (B) と比較すると僅かに性能は向上したものの，トランザクションが連鎖的にアボートする動作により，アボートに起因するオーバヘッドが増大し，参考モデル (R) より性能が悪化した．提案モデル（P）では，アクセスリクエストを投機的に許可したスレッドがトランザクションをアボートすると，そのスレッドによりアクセスリクエストを許可されたスレッドもトランザクションをアボートする．そのため，既存モデル（B）および参考モデル（R）では 1 つのトランザクションをアボートするだけで済んだ処理でも，提案モデル（P）では複数のトランザクションをアボートする必要があり，これによりオーバヘッドが増大する．特に，Deque では Contention と比較して既存モデル (B) における Bad_trans および Aborting が大きかったため，提案モデル (P) ではこれらの増加率が大きくなり，十分な性能向上が得られなかったと考えられる．一方で，参考モデル（R）では競合を抑制することで既存モデル（B）に対して約 67.3%の性能向上を達成できていることから，提案モデル（P）はアボートに起因するオーバヘッドを削減することで，並列度を向上できる余地があると考えられる．

以上のような Contention と Deque の性能差は，共有変数へのアクセス頻度に原因があると考えられる． 3.3 節で述べたように，Contention は *phase*1 で配列 A，*phase*2 で配列 B にアクセスする．なお，配列 A の要素数は 1，配列 B の要素数は 1024 である．したがって，前半のフェーズでは配列 A に対しアクセスが集中するが，後半のフェーズで配列 B のどの要素にアクセスするかは乱数によって決定するため，後半のフェーズでは各スレッドのアクセス先アドレスが分散しやすいと考えられる．一方 Deque は，前半のフェーズで 1 つの共有変数にアクセスした後，後半のフェーズでで両端キューを操作する．両端キューとは，両端から出し入れ可能なキューのことである．この両端キューを複数のスレッドが同時に操作しようとすると，キューの右端を指すアドレスと，左端を指すアドレスにアクセスが集中する．そのため，Deque では前半のフェーズで投機的に並列度を向上させたとしても，後半のフェーズで競合が引き起こされる可能性が高く，結果的に並列度が向上しにくいと考えられる．また，Contention と比較すると後半のフェーズでアボートが発生しやすく，アボートに起因するオーバヘッドが増大してしまうと考えられる．したがって今後，投機的にアクセスリクエストを許可した後に発生する競合を抑制する方法について検討する必要がある．

7. おわりに

本稿では，これまでに提案されてきたスケジューリング手法を用いても性能向上が困難だったプログラムについて，そのトランザクションが持つメモリアクセスパターンの特徴を調査した．その結果，複数の実行フェーズを持ち，あるフェーズでアクセスしたアドレスに対しそれ以降のフェーズで再度アクセスしないという特徴を持つトランザクションを持つプログラムでは，本来一貫性を損なうとは限らないにも関わらず競合として検出されてしまうメモリアクセスにより，性能向上が妨げられていることが分かった．そこで，従来手法では競合として検出されるメモリアクセスであっても，それぞれのスレッドで実行しているトランザクションの実行フェーズにおけるアクセス先アドレスが異なる場合には，このメモリアクセスを投機的に許可することで，性能を向上させる手法を提案した．提案手法の有効性を確認するために，GEMS microbench のプログラムのうち，アクセス先アドレスが異なる 2 つの実行フェーズを持つ Contention と Deque を用いて評価を行った結果，既存の LogTM-SE と比較して Contention は平均 63.2%，Deque は平均 6.3%の性能向上を達成した．

しかし提案手法では，投機的にアクセスリクエストを許可したスレッドがトランザクションをアボートすると，そのスレッドによりアクセスリクエストを許可されたスレッドもトランザクションをアボートする必要があり，従来の手法に比べてアボートに起因するオーバヘッドが増大する．このオーバヘッドにより Deque の性能向上幅は小さく，このオーバヘッドを削減する方法について今後検討する必要がある．

謝辞 本研究の一部は，JSPS 科研費 JP17H01711, JP17H01764 の助成を受けたものである．

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] Moss, E. and Hosking., T.: Nested Transactional Memory: Model and Preliminary Architecture Sketches., *Science of Computer Programming*, pp. 186–201 (2006).
[3] 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).
[4] McDonald, A., Chung, J., Caristrom, B. D., Minh, C. C., Chafi, H., Kozyrakis, C. and Olukotun., K.: Architectural Semantics for Practical Transactional Memory, *Proc. 33rd Annual Int'l Symp. on Computer Architecture (ISCA'06)*, pp. 53–65 (2006).
[5] Shriraman, A., Dwarkadas, S. and Scott., M. L.: Flex-

ible Decoupled Transactional Memory Support, *Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA'08)*, pp. 139–150 (2008).

[6] Tomic, S., Perfumo, C., Kulkami, C., Armejach, A., Cristal, A., Unsal, O., Harris, T. and Valero., M.: Eazyhtm, Eager-lazy Hardware Transactional Memory, *Proc. 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-42)*, pp. 145–155 (2009).

[7] Lupon, M., Magklis, G. and González, A.: A Dynamically Adaptable Hardware Transactional Memory, *Proc. 43rd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-43)*, pp. 27–38 (2010).

[8] 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).

[9] 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).

[10] Hirota, A., Mashita, K. and Tsumura, T.: A Concurrency Control in Hardware Transactional Memory Considering Execution Path Variation, *Proc. 4th Int'l Symp. on Computing and Networking (CANDAR'16)*, pp. 77–83 (2016).

[11] 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).

[12] 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).

[13] Yamada, R., Hashimoto, K. and Tsumura, T.: Priority-Based Conflict Reslolution for Hardware Transactional Memory, *Proc. 2nd Int'l Workshop on Computer Systems and Architectures (CSA'14)*, IEEE, pp. 433–439 (online), DOI: 10.1109/CANDAR.2014.47 (2014).

[14] 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).

[15] 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. 13th Annual Int'l Symp. on High Performance Computer Architecture (HPCA-13)*, pp. 261–272 (2007).

[16] 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).

[17] 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).