

競合アクセスを投機的に許可する トランザクショナルメモリの検討

多治見 知紀¹ 林 昌樹¹ 二間瀬 悠希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 (Tx) 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が頻発すると性能が低下してしまうため、アクセス競合を極力回避する必要がある。一般に Tx には、ある共有変数に対する read・write アクセスが完了した後も、コミットまで長時間処理が継続するものがある。このような場合、当該 Tx においてその変数に再度アクセスしないにも関わらず、当該変数に対する他スレッドによるアクセスは競合として検出され、並列性が損なわれてしまう。本稿では、Tx 内でアクセス済みの共有変数に対し、Tx をコミットする前であっても他のスレッドによる read および write アクセスを投機的に許可するスケジューリング手法を提案し、それに伴うコヒーレンシ制御について議論する。提案手法を LogTM に実装し評価を行った結果、平均 63.6%、最大 38.8%の性能向上を達成した。

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ー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 (Transaction : Tx) 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列実行することで、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ーヘッドを軽減している。この HTM では、同一共有変数に対するアクセス競合が頻発すると性能が低下する可能性があるため、アクセス競合を極力回避する必要がある。

さて、Tx には一般に、ある共有変数に対する read・write アクセスが完了した後も、コミットまで長時間処理が継続するものが存在する。このような場合、当該 Tx においてその変数に再度アクセスしないにも関わらず、この変数に対する他スレッドからのアクセスは競合として検出され、並列性が損なわれてしまう。本稿では、Tx 内でアクセスした共有変数に対し、Tx をコミットする前であっても、他スレッドからの read および write アクセスを投機的に許可するスケジューリング手法を提案し、それに伴うコヒーレンシ制御について議論する。

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

これまで HTM に関して、実行中の Tx をアボートした後、その Tx 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、アプリケーションの振る舞いに応じて競合検出やバージョン管理の方式を動的に変更する研究 [5], [6], [7] など、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 Tx 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 Tx 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 Tx 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。我々も、競合の発生を予測し、競合を引き起こすと予測されるメモリアクセスのタイミングを競合 Tx のコミット後になるように待機させることで競合を回避する手法 [10] を提案している。

しかし、以上で述べたいずれの手法においても、競合に起因するオーバーヘッドを削減できていないプログラムがいくつか存在する。そこで我々はさらなる性能向上のため、Tx が持つ特徴的なメモリアクセスパターンについて調査した。その結果、Tx 内である共有変数に対する read・write アクセスが完了した後も、コミットまで長時間処理が継続するものがあることが分かった。このような場合、Tx 内におけるある共有変数に対する最終アクセスからコミットまでの区間で、当該変数に対する他のスレッドによる競合アクセスを投機的に許可することで、並列性を向上させられると考えた。そこで先に我々は、Tx 内の処理が、ある共有変数に繰り返しアクセスするフェーズと、その後コミットまで当該変数に一切アクセスしないフェーズとに分離できると仮定し、後者のフェーズにおいて当該変数に対する競合アクセスを投機的に許可することで、性能が向上することを確認した [11], [12]。しかしこの方法には、プログラム自身がフェーズを定義する必要があること、Tx 内に多くの共有変数が混在するプログラムではフェーズを適切に定義することが困難であることなどの問題があった。本稿ではこれらの問題を改善し、Tx 内における共有変数毎の最終アクセスのタイミングを検出することで、競合アクセスを投機的に許可するスケジューリング手法を提案する。

3. HTM における競合解決とその問題点

本章ではまず、既存の HTM における競合解決方法とそ

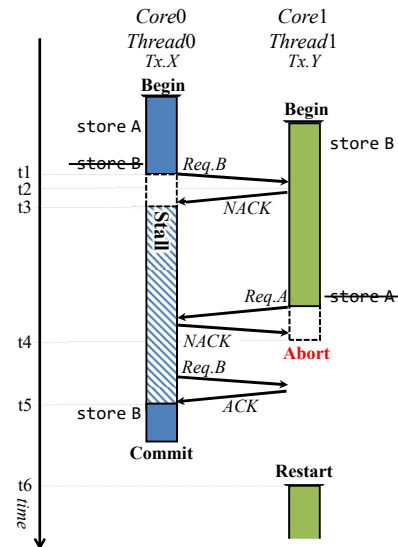


図 1 HTM における競合解決

の問題点について述べる。次に、既存の HTM で競合として検出されるアクセスリクエストの中でも、コミット前に許可したとしてもプログラムの実行結果に影響しないものについて述べる。

3.1 HTM における競合解決とその問題点

HTM は、Tx に対して以下の性質を保証する。

Atomicity (不可分性) Tx 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ない。

Isolation (独立性) 各 Tx 内における処理結果は、Tx のコミットと同時に観測される。また、並列実行された Tx の処理結果は、これら Tx を逐次実行した場合と同一でなければならない。

既存の HTM では、Tx が以上の性質を満たさなくなる場合に、共有変数へのアクセスを競合として検出する。

ここで、既存の HTM における競合検出と競合解決の動作について、図 1 を用いて説明する。この図は下向きに時間軸をとっており、各バーは Thread0, Thread1 がそれぞれ Tx.X, Tx.Y を実行中であることを表している。なお、各 Tx は固有の ID を持ち、Tx.X, Tx.Y はそれぞれ X, Y を ID として持つ Tx である。まず、Thread0 が Tx.X で store A を、Thread1 が Tx.Y で store B をそれぞれ実行済みであるとする。その後、Thread0 が store B の実行を試みて、Thread1 に対しアドレス B へのアクセスリクエストを送信する (時刻 t1)。これを受信した Thread1 は store B を実行済みであり、Tx.Y のコミット前であるため、Thread0 がアドレス B にアクセスすると Isolation が満たされなくなってしまう。したがって、Thread1 はこれを競合として検出し、Thread0 に対し NACK を返信する (t2)。Thread0 は NACK を受信すると store B を実

行せず, $Tx.X$ をストールさせる (t3). その後, $Thread1$ が **store A** の実行を試みて, $Thread0$ に対しアクセスリクエストを送信すると, $Thread0$ は **store A** を実行済みであるため, これを競合として検出し, $Thread1$ に対し **NACK** を返信する. これにより, $Thread1$ はストール中の $Thread0$ から **NACK** を受信するため, このまま $Thread1$ がストールしてしまうとデッドロック状態に陥ってしまう. これを回避するため, $Thread1$ は自身が行った **store B** の実行結果を破棄する. ただし, Atomicity を保証するため, $Thread1$ は **store B** の実行結果だけでなく, $Tx.Y$ で行った全ての実行結果を破棄しなければならない. そこで, $Thread1$ は $Tx.Y$ をアボートし, $Tx.Y$ を実行開始する前のメモリ状態を復元する (t4). これにより, $Thread0$ 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, $Thread1$ は $Thread0$ との再競合を防ぐために一定時間待機した後, $Tx.Y$ を再実行する (t6).

このようなメモリ状態の復元を可能にするため, HTM にはバージョン管理機構が必要となる. HTM では, Tx 内で変数の値を更新する前に, 更新前の値と, その変数のアドレスとを Log という領域に保持する. その後, メモリに値を書き込む. Tx をコミットする場合は, Log の値を破棄するだけでよいが, Tx をアボートする場合は, メモリに Log の値を書き戻すことで, Tx 開始前と同様のメモリ状態にロールバックする. Tx をアボートする際にはこのようなロールバック処理および Tx の再実行による性能低下が問題となる.

3.2 実行結果に影響しない競合アクセス

既存の HTM では Isolation を保証するため, あるスレッドが Tx 内である変数を更新してからコミットするまでの区間では, 当該変数に対する他のスレッドによるアクセスは競合として検出される. しかし, コミットまでの間に当該スレッドが当該変数に再度アクセスしなければ, 他のスレッドが当該変数にアクセスしても, スレッド間で共有変数の一貫性が損なわれない. このようなアクセスについて, GEMS microbench [13] に含まれるプログラムの一つである Deque を例に説明する. 図 2 に Deque の Tx を簡略化したコードを示す. 1 行目の **BEGIN_TRANSACTION()** および 17 行目の **COMMIT_TRANSACTION()** はそれぞれ, Tx の開始とコミットを表している. この Tx では, 2 行目で共有変数 **count** に対するインクリメント操作を行った後, 4 行目の **switch** 文で deque の右端または左端のいずれかに対して, エンキューまたはデキューのいずれかを実行する. あるスレッドがこの Tx 内で **count** にアクセスしてからコミットするまでの間は, Isolation を保証するため, 他のスレッドによる **count** に対するアクセスは競合として検出される. しかし, この Tx 内での共有変数 **count** に対する処理は 2 行目で完了しており, その後コミットするまでこ

```

1 BEGIN_TRANSACTION();
2 count++;
3
4 switch(op){
5   case 0:
6     enqueue_right();
7     break;
8   case 1:
9     dequeue_right();
10    break;
11  case 2:
12    enqueue_left();
13    break;
14  case 3:
15    dequeue_left();
16    break;
17 }
18 COMMIT_TRANSACTION();

```

図 2 Deque のトランザクションを簡略化したコード

の共有変数にアクセスすることはないため, 3 行目からコミットするまでの区間ではこのアクセスを許可したとしても一貫性が損なわれることはない. そこで, このような競合アクセスを投機的に許可することで, 性能向上が期待できる.

4. 競合アクセスを投機的に許可するスケジューリング手法

本章では, Tx 内における各変数に対する最終アクセスからコミットするまでの区間で, 既存手法では競合と検出されていたメモリアクセスを投機的に許可するスケジューリング手法を提案する.

3.2 節で述べたように, あるスレッドが Tx 内である変数にアクセスしてから Tx をコミットするまでの区間で, 当該スレッドが再度当該変数にアクセスしなければ, コミット前に他のスレッドが当該変数にアクセスしたとしても, スレッド間で共有変数の一貫性は損なわれない. よって, スレッドがある変数へのアクセスリクエストを受信した時点で, 当該スレッドがコミットするまで当該変数に再度アクセスしないと予測できれば, Tx のコミット前であっても, このアクセスを投機的に許可することで, 性能向上が期待できる.

Tx 内で再度変数にアクセスするか否かを判断するためには, 変数毎に Tx 内での最終アクセスのタイミングを予め知っておく必要がある. そのために提案手法では, 各変数の Tx 内での最終アクセスまでに発生するメモリアクセス回数を記憶する. 具体的には, Tx を実行中のスレッドは, 実行中の Tx の ID とアクセスした変数のアドレスとを, Tx 開始から当該変数の最終アクセスまでに行ったメモリアクセス回数に紐づけて記憶する. この動作について, 図 3 を例に説明する. まず, $Thread0$ が $Tx.X$ を開始し, **store A** を実行したとする (t1). このとき, 実行中の Tx の ID 「X」とアクセス先アドレス「A」, Tx 開始後にメ

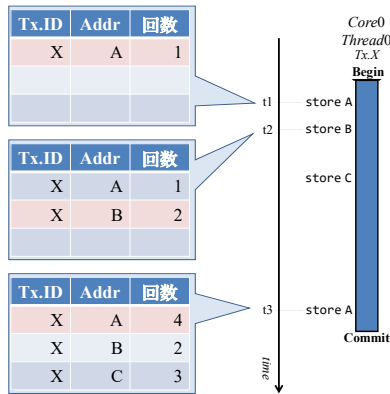


図 3 最終アクセスまでの時間を記憶する動作

モリアクセスした回数「1」を関連付けて記憶する。次に、Thread0 が store B を実行すると (t2)、同様に Tx の ID 「X」、アクセス先アドレス「B」、メモリアクセス回数「2」を関連付けて記憶する。その後、実行が進み、Thread0 が store A を再度実行したとする (t3)。このとき、アドレス A に対する最終アクセスまでのメモリアクセス回数が「1」として既に記憶済みであるが、この時点におけるメモリアクセス回数は 4 であるため、これを「4」に更新する。

以上のように変数ごとにメモリアクセス回数を記憶したスレッドは、他スレッドからある変数に対するアクセスリクエストを受信し競合を検出すると同時に、このアクセスを投機的に許可して良いか否か进行检查する。この動作について、図 4 を例に説明する。この図において、各アドレスに対する最終アクセスに関する情報は図 3 で示した動作により既に記憶済みであるとする。まず、Tx.Y を実行中の Thread1 が、store B を試みて、Thread0 に対しアドレス B へのアクセスリクエストを送信したとする。これを受信した Thread0 は、アドレス B にアクセス済みであるため、これを競合として検出する。ここで、Thread0 はコミットまでアドレス B に再度アクセスする可能性があるか否か进行检查する。そのために、Tx.X を開始してからその時点までのメモリアクセス回数「3」と、アドレス B に対する最終アクセスまでに行うメモリアクセス回数「2」とを比較する (t1)。その結果、Tx.X を開始してからメモリアクセス回数の方が、最終アクセスまでに行うメモリアクセス回数より多いため、Thread0 はコミットまでアドレス B に再度アクセスすることはないと判断し、Thread1 によるアドレス B に対するアクセスを許可する。本稿では以降、アクセスを投機的に許可した側のスレッドを *permitter*、許可された側のスレッドを *requester* と呼ぶこととする。

なお、Tx の実行パスによって、ある変数へのアクセスを投機的に許可した *permitter* が、当該変数に再度アクセスしてしまう場合がある。このような場合には *permitter* と *requester* との間で一貫性を維持するための制御が必要となる。そこでまず、*requester* が Tx をアボートおよびロールバックすることで、*permitter* がアクセスを投機的に許

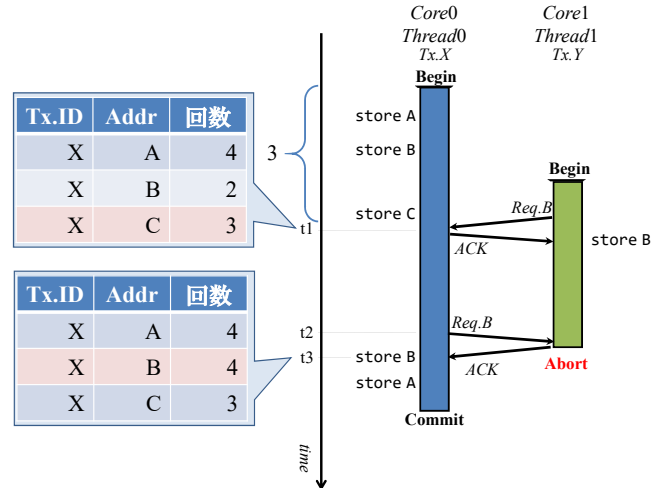


図 4 投機的に競合アクセスを許可したアドレスに対し一貫性を保持する動作

可する前のメモリ状態を復元する。その後、*permitter* は当該変数にアクセスし、最終アクセスまでに行うメモリアクセス回数を更新する。以上のようにアクセスの投機的許可に失敗した場合の動作について、図 4 を用いて引き続き説明する。実行パスが図 3 における実行時のものから変化し、*permitter* である Thread0 が再度 store B を試みて、*requester* である Thread1 に対しアドレス B へのアクセスリクエストを送信したとする (t2)。これを受信した Thread1 は、Thread0 によってアドレス B へのアクセスを投機的に許可されているため、実行中の Tx.Y をアボートおよびロールバックする。これにより、Thread0 はアドレス B にアクセス可能となるため、アドレス B にアクセスし、最終アクセスまでに行うメモリアクセス回数を更新する (t3)。

以上の動作を保証するため、*requester* はアクセスを許可された時点で、*permitter* のスレッド ID とアクセスを許可されたアドレスとを関連付けて記憶する。この例では、時刻 t1 の時点で *requester* である Thread1 が、*permitter* である Thread0 からアドレス B へのアクセスを投機的に許可されたことを記憶する。以上のように動作することで、Thread1 が Thread0 からアドレス B に対するアクセスリクエストを受信した時点で、投機アクセスに失敗したことを検出できる。ただし、後述の動作によりアドレス B を記憶する必要はなくなるため、実際には *requester* は *permitter* のスレッド ID のみ記憶する。この理由については、5.2 節で述べる。

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 [14] 上に実装する。提案手法では競合アクセスを投機的に許可するため、*permitter* が Tx をアボートする際に Isolation が保証されなくなる。したがって、提案

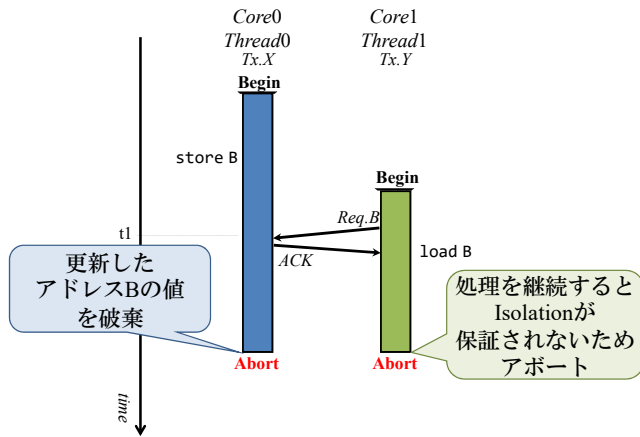


図 5 Isolation を保証するために行うアボート

手法ではコヒーレンシ制御の動作を変更して Isolation を保証する必要がある。本章ではまず、提案手法においてどのようにアボートすることで Isolation を保証するかについて述べ、次にこのアボートに伴い必要となるロールバック処理について説明する。

5.1 Isolation 保証のための制御

Permitter が Tx をアボートすると、Isolation が保証されなくなる。この問題とこれを解決するための動作について、図 5 を用いて説明する。この図において、まず、Thread1 によるアドレス B に対する競合アクセスを、Thread0 が投機的に許可したとする (t1)。その後、permitter である Thread0 が他のスレッドとの競合を検出し、Tx.X をアボートする必要があるとする。このとき、Thread0 は Tx.X 内で行ったアドレス B に対する変更を破棄しなければならない。ここで、Thread0 が Tx.X をアボートしたとしても、そのままでは Thread0 が変更した後のアドレス B の値を使用したまま Thread1 が実行を継続してしまうため、Isolation が保証されなくなってしまう。そこで、提案手法ではコヒーレンシ制御の動作を変更してこれを保証する。この例では、permitter である Thread0 が Tx をアボートする際に、連鎖的に requester である Thread1 にも Tx をアボートさせることで、アドレス B の値を破棄し、Isolation を保証する。

この動作を実現するために、他スレッドで実行中の Tx をアボートさせるメッセージ *Req.Abort* を新たに実装する。permitter が Tx をアボートさせる必要がある場合、この *Req.Abort* を requester に対して送信する。そして、requester も Tx をアボートさせることで、Isolation を保証できる。なお、permitter は requester をアボートさせる場合に備えて、競合アクセスを投機的に許可する時点で、予め requester のスレッド ID を記憶しておく。

一方 requester 側は、permitter がコミットするまでの間、permitter によって Tx をアボートさせられる可能性がある。そこで提案手法ではこのようなアボートに備え

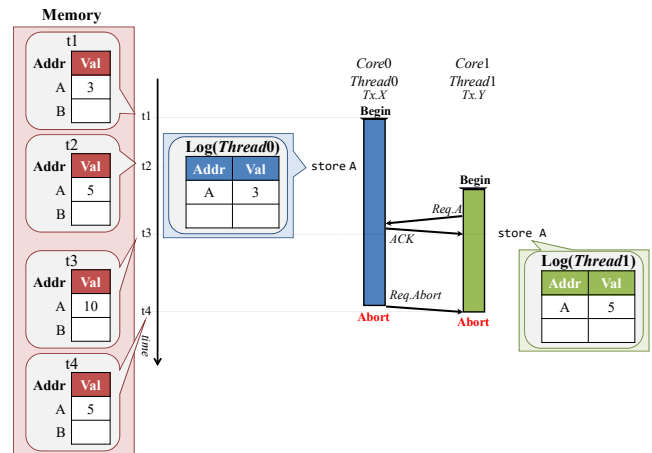


図 6 連鎖的なアボートで正しくロールバックできない例

て、permitter が requester より先にコミットするよう制御する。これを実現するために、コミットしたことを他のスレッドに通知するメッセージ *Committed* を新たに実装する。Tx をコミットした permittter は、requester に対し *Committed* メッセージを送信する。requester は Tx 処理の終了に至ったとしても、この *Committed* メッセージを受信するまで自身のコミットを待機する。以上のようにコミット順序を制御することで、requester は permittter によって Tx をアボートさせられる場合に備えることができる。

5.2 連鎖的なアボートに伴うロールバック

提案手法で発生する連鎖的なアボートにより、それに関与する複数の Tx でロールバック処理が必要となるが、この処理順序によっては、Tx 実行開始前のメモリ状態を正しく復元できない。この問題について、図 6 を例に説明する。この図は、各時刻 t1~t4 におけるメモリ状態およびスレッドが保持している Log の状態を表している。まず、時刻 t1 において、メモリ上のアドレス A に値 3 が格納されているとする。次に、時刻 t2 において Thread0 がアドレス A に値を書き込むために、変更前のアドレス A の値 3 とそのアドレスとを、Thread0 が保持している Log に記憶し、値 5 をアドレス A に書き込む。その後、時刻 t3 において、Thread1 がアドレス A に対するアクセスを投機的に許可され、Thread0 の時と同様に変更前のアドレス A の値 5 とそのアドレスとを、Thread1 が保持している Log に記憶した上で、アドレス A に値 10 を書き込む。その後、permitter である Thread0 に Tx.X をアボートさせる必要があるとする。このとき、Thread0、Thread1 の順に Tx をアボートさせると、アドレス A には Thread0 が Log に保持している値 3 が書き戻された後、Thread1 が Log に保持している値 5 が書き戻される。このようにしてロールバックが完了した時刻 t4 には、アドレス A に値 5 が格納されており、t1 のメモリ状態を復元できていない。

以上のように、Tx をアボートさせる順序によっては、

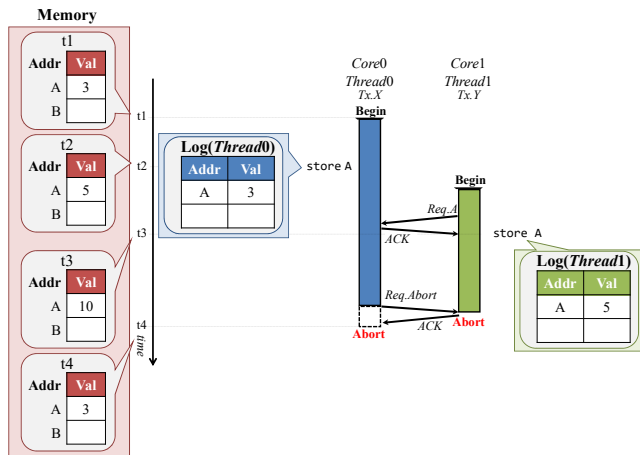


図 7 ロールバック順序を制御してメモリ状態を復元する例

permitter と *requester* が共に Tx を実行開始する直前のメモリ状態に復元することができないという問題が生じる。この問題に対し、*requester* が *permitter* より先にロールバックするように制御すれば、メモリ状態を正しく復元できる。このようにして正しいメモリ状態を復元する動作について、図 7 に示す例を用いて説明する。この図は時刻 t_3 まで、図 6 と同様に処理が進行した状態を表している。その後、Thread0 が Tx.X をアボートする必要があるとする。ここで、Thread0 は *permitter* であるため、その *requester* である Thread1 が先にロールバックするよう、アボート順序を制御する。そのために、Thread0 は Thread1 に対し Req.Abort を送信する。これを送信してから Thread1 がロールバックを完了するまで、Thread0 はアボートを待機する。一方、Thread1 は Log に保持している値 5 をメモリ上のアドレス A に書き戻し、ロールバックを完了する。その後、Thread1 は Thread0 に ACK を返信する。これを受信した Thread0 は、先ほどの Thread0 と同様に Log に保持している値 3 をメモリ上のアドレス A に書き戻すことでロールバックする。以上のように動作することで、時刻 t_4 には時刻 t_1 と同様のメモリ状態を復元できる。

ここで、投機アクセスの許可-被許可の関係が、複数のスレッドにまたがって閉路構造を成すと、その閉路内のいずれかにおいて、*requester* を *permitter* より先にロールバックすることができなくなってしまう。したがって、アクセスを投機的に許可する前に、そのような閉路が構成されることを検出し、回避する必要がある。

投機的アクセス許可が、メモリ状態を正しく復元可能な範囲内で行われることを保証するため、提案手法では各スレッドのスレッド ID を利用する。具体的には、*permitter* は投機アクセスを許可する際、*requester* に対して自身のスレッド ID を通知し、*requester* はこれを記憶する。その後、この *requester* が他スレッドに対する *permitter* となった場合、当該スレッドが保持しているこれまで受信した全

ての ID と併せて、自身のスレッド ID を新たな *requester* に対して送信する。こうすることで、ある *requester* が受信した ID のリストに、自身のスレッド ID が含まれていた場合、そのリクエストを許可すると許可-被許可の関係が一巡してしまうと判断できる。よってその場合、*permitter* に Req.Abort を送信してアボートさせることで、これを回避する。

4 章の図 4 では、あるアドレスに対して投機的アクセスを許可した *permitter* が当該アドレスに再度アクセスしてしまう場合に、問題が発生するとして説明した。これは言い換えると、同一アドレスに対してお互いが投機的アクセスを許可し合う場合に当たる。しかし本節で説明してきたように、ロールバック順の制約を考慮すると、対象が同一アドレスであるか否かに関わらず、投機的アクセスをお互いに許可し合う状況は回避する必要がある。よって本節で述べた手順により、スレッド ID を用いて許可-被許可の関係を管理するだけで十分であり、対象のアドレスを併せて記憶する必要はない。

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。また、提案手法の実装に必要となるハードウェアコストについても概算する。

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 [14] 上に実装し、シミュレーションにより評価した。評価には Simics 3.0.31 [15] 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としては GEMS microbench から Btree, Contention, Deque, SPLASH-2 [16] から Raytrace, STAMP [17] から Kmeans の計 5 個を使用し、それぞれ 16 スレッドで実行した。

6.2 評価結果

評価結果を図 8 に示す。図中では、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
- (R1) 競合を未然に予測し回避する参考モデル [10]
- (R2) フェーズを考慮して競合アクセスを投機的に許可する参考モデル [11], [12]
- (P) 共有変数に対する最終アクセスを検出し競合アクセスを投機的に許可する提案モデル

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
L1 cache	32 KBytes
ways	4 ways
latency	3 cycles
L2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

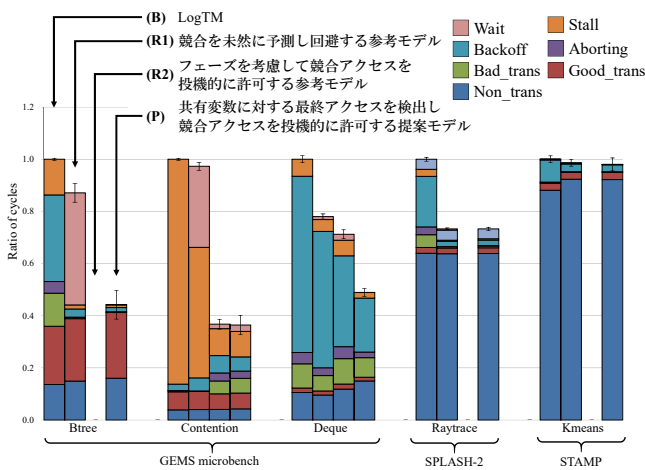


図 8 評価結果

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。ただし (R2) の評価結果のうち、適切にフェーズを定義することが困難なプログラムについては記載していない。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 [18]。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 Tx 外の実行サイクル数、Good_trans はコミットされた Tx の実行サイクル数、Bad_trans はアボートされた Tx 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 (R1) 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スレッドが Tx の実行開始を待機したサイクル数を示し、(R2) および (P) における Wait は、*permitter* がコミット

```

1 int A[1];
2 int B[1024];
3
4 BEGIN_TRANSACTION();
5 // A[0]にアクセスする
6 for( i = 0; i < 10; i++ ){
7     if( access_type[i] == READ )
8         var = A[0];
9     else
10        A[0] = 0;
11 }
12
13 // A[0]にアクセスしない
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();

```

図 9 Contention のトランザクションを簡略化したコード

を完了するまで、対応する *requester* が Tx のコミットを待機したサイクル数を示す。

評価の結果、全てのベンチマークプログラムで提案モデル (P) は既存モデル (B) より高い性能を示し、最大 63.6%、平均 38.8%、実行サイクル数が低減した。また、参考モデル (R1) または (R2) と比較しても、Btree、Contention および Deque については特に高い性能を示している。

6.3 考察

まず Btree について、提案モデル (P) は、参考モデル (R1) と比較して性能が大きく改善しており、スケジューリングオーバーヘッドにあたる Wait の大幅な抑制にその要因があることが分かる。参考モデル (R1) では、競合を引き起こすと予測されるアクセスが、競合相手の Tx がコミットした後に行われるよう、Tx の実行開始を待機する。一方、提案モデル (P) ではコミットまでアクセスされない共有変数に対する競合アクセスを投機的に許可する。Btree は B 木に対する検索と挿入を行うプログラムである。検索する際に根からノードを辿っていくが、一度参照したノードを再参照することはない。したがって、Btree で競合が検出されるアドレスは、コミットまで再度アクセスされることが少ない。以上の特徴から、参考モデル (R1) が競合アクセスが競合相手の Tx のコミット後になるよう Tx の実行開始を待機する区間で、提案モデル (P) はアクセスを投機的に許可でき、より高い並列度が得られたと考えられる。

次に Contention では、提案モデル (P) は既存モデル (B) および参考モデル (R1) と比較して大幅に性能が優れているものの、参考モデル (R2) との性能差は僅かである。ここで、Contention が持つ Tx を簡略化したコードを図 9 に示す。このコードにおいて、6 行目から 11 行目の for 文では変数 A[0] に対し繰り返しアクセスする。その後、14 行

目から 19 行目の for 文では、配列 B のいずれかの要素に対し、ランダムな順序でアクセスする。したがって参考モデル (R1) では、A[0] に最後にアクセスしてからコミットするまでの区間で、競合相手の Tx が実行開始を待機する。提案モデル (P) では Btree と同様にこのような区間で競合アクセスを投機的に許可できたため、既存モデル (B) および参考モデル (R1) と比較して Wait または Stall のサイクル数が少なく抑えられている。参考モデル (R2) については、フェーズをプログラム中に明記する必要があるが、本評価では 2 つの for 文をそれぞれフェーズとして定義した。この場合、前半のフェーズでは変数 A[0] に対する競合が発生しやすいが、後半のフェーズではアクセス先がランダムなため、競合が発生しにくい。このように定義した場合、A[0] に対する競合アクセスに関しては、後半のフェーズで投機的に許可されることで、提案モデル (P) に近い振る舞いをし、競合が抑制される。一方で、配列 B の各要素に対する競合アクセスは、これを許可するためのフェーズがそれより後に定義されていないため、投機的に許可されることはないが、元来配列 B の要素に対するアクセスでは競合が発生しにくいため、その影響は僅かである。これらのことから、参考モデル (R2) の性能が提案モデル (P) と大きく変わらない結果となったと考えられる。

また Deque では、提案モデル (P) が既存モデル (B) および提案モデル (R1) だけでなく、参考モデル (R2) と比較しても高い性能を示している。既存モデル (B) および提案モデル (R1) に対しては Contention と同様に、3.2 節で述べたような共有変数 `count` に対する競合アクセスを提案モデル (P) で投機的に許可した結果、より高い並列度が得られたと考えられる。さらに、参考モデル (R2) に対して高い性能が得られた理由は、参考モデル (R2) ではフェーズを定義しても投機的に許可できない競合アクセスがあり、提案モデル (P) でそのようなアクセスを許可できたためだと考えられる。図 2 に示したように、この Tx ではまず共有変数 `count` に対するインクリメントを実行してからキュー操作を行う。キュー操作の際には、キューの先頭または末尾にアクセスが集中し、競合が発生しやすい。参考モデル (R2) の評価ではこの Tx 内において、`count` に対するインクリメント操作と、その後の処理とをそれぞれフェーズとして定義した。後者のフェーズでは、キューの両端にアクセスが集中するため競合が発生しやすいが、やはりそれより後にフェーズが定義されていないため、競合アクセスが投機的に許可されることはない。フェーズをさらに分割できれば、キュー操作におけるアクセスに対しても投機的許可が行えるが、switch 文により処理フローが分岐していること、処理内容が別関数に分離されていること、キューの両端にあたるアクセス先アドレスはキュー操作に伴い変化すること、などの理由から、それも困難である。一方提案モデル (P) では、各アドレスに対する最終アクセスまでの

メモリアクセス回数を記憶することで、アドレス単位で競合アクセスを許可することができたため、より高い性能が得られたと考えられる。

次に、Raytrace では既存モデル (B) に対し、Bad_trans, Aborting, Backoff, Stall が削減できており、競合の回避が性能に寄与していることが分かる。Raytrace には実行時間が非常に長い Tx が存在する。このような Tx では、ある変数に最後にアクセスしてからコミットするまでの区間も長くなりやすい。したがって、この Tx における競合アクセスを投機的に許可した場合には、並列度が大きく向上しやすいと考えられる。このような理由から、長い区間で多くの競合アクセスを許可でき、高い性能が得られたと考えられる。

最後に Kmeans では、既存モデル (B) および参考モデル (R1) と比較して提案モデル (P) が高い性能を示していることから、提案手法の有効性は確認できるものの、実行サイクル数全体に占める Non_trans の割合が大きいため、これらの性能差は小さかった。

6.4 ハードウェアコスト

本節では、提案手法を実装するために必要なハードウェアのコストについて述べる。提案手法では、Tx の ID と Tx 内でアクセスしたアドレスとを、Tx 開始から当該変数の最終アクセスまでに発生するメモリアクセス回数と関連付けて記憶する。これらの情報を記憶するハードウェアのコストを算出するために、実行される Tx の最大数と最終アクセスまでに要したメモリアクセス回数の最大値とを調査した。まず、今回評価に用いたプログラムの中で、実行される Tx の最大数は 8 であったため、3bit で Tx の ID を記憶できる。また、最終アクセスまでに要したメモリアクセス回数の最大値は 4,807 であったため、この回数を記憶するためには 13bit あればよい。

次に、Tx 内でアクセスしたアドレスを記憶するために必要なコストについて考える。LogTM ではキャッシュライン単位で競合を検出するため、実際にはラインアドレスを記憶する。今回使用したシミュレータではメモリアドレスの上位 26bit がラインアドレスを表すため、これを記憶する。以上から、あるアドレスに対する最終アクセスに関する情報は、 $3 + 13 + 26 = 42\text{bit}$ で記憶できる。この情報は Tx 内でアクセスされる各アドレスに対して記憶する。そこで、何アドレス分の情報を記憶できれば十分かについて調査するために、プログラムの実行中に Tx 内でアクセスされる総アドレス数を計測した。調査の結果、今回使用したプログラムの中で最もアクセス先アドレスの数が多かったのは Btree であり、その数は 15,018 であった。したがって、1 アドレスの最終アクセスに関する情報 42bit が 15,018 アドレス分記憶できればよいことが分かり、合計 $42 \times 15,018 = 630,756\text{bit}$ で、プログラム内でアクセス

される全てのアドレスについて情報を記憶できる。

さらに、提案手法では各スレッドは *permitter* と *requester* のスレッド ID を記憶する。このために、スレッド数と同じビット数のビット列を用意する。このビット列の中で、各スレッド ID に対応するビットを立てることで、*permitter* と *requester* の ID をそれぞれ記憶できる。よって、今回のように 16 スレッドで実行する場合は、 $16 + 16 = 32\text{bit}$ でスレッド ID を記憶できる。

以上から、提案手法を実装するために必要なハードウェアコストは、1 スレッドあたり $630,756 + 32 = 630,788\text{bit}$ $\equiv 77.1\text{KByte}$ と算出でき、16 スレッドを実行可能な 16 コア構成のプロセッサ全体においては、 $77.1 \times 16 = 1,233.6\text{KByte}$ $\equiv 1.2\text{MByte}$ 必要となる。今回使用したシミュレータの L1 キャッシュが 32KByte であることから、1 スレッドあたり 77.1KByte というハードウェアコストは非常に大きい。本稿では、Tx 内でアクセスされるアドレス全てに対して情報を記憶できると仮定して、提案手法の評価を行ったが、記憶容量と性能との関係の調査や、合理的なハードウェア量を想定した場合の評価を、今後行っていく必要がある。

7. おわりに

本稿では、Tx 内でアクセスした共有変数に対し、Tx をコミットする前であっても、他スレッドからの競合アクセスを投機的に許可するスケジューリング手法を提案した。ただし競合アクセスを単に許可するだけでは、アクセスを許可した側のスレッドがアボートすることにより、Isolation が保証されなくなってしまう。そこで Isolation を保証するために、提案手法ではロールバック処理を適切に制御する。提案手法の有効性を確認するために、GEMS microbench, SPLASH-2 および STAMP ベンチマークプログラムを用いて評価を行った結果、既存の HTM と比較して 16 スレッドで最大 63.6%、平均 38.8% の実行サイクル数を削減できることを確認した。しかし、本稿の評価で想定した実装方式ではハードウェアコストが大きいので、今後は合理的なハードウェア量で十分な性能向上を達成できる実装方法について検討していきたい。

謝辞 本研究の一部は、JSPS 科研費 JP17H01711, JP17H01764, JP17K19971 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.: Flexible Decoupled Transactional Memory Support, *Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA'08)*, pp. 139–150 (2008).
- [6] Tomic, S., Perfumo, C., Kulkarni, C., Armejach, A., Cristal, A., Unsal, O., Harris, T. and Valero, M.: Eazy-htm, 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] 多治見知紀, 廣田杏珠, 塩谷亮太, 五島正裕, 津呂公曉: 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, 情報研報 (SWoPP2017), Vol. 2017-ARC-227, No. 40, pp. 1–9 (2017).
- [12] Tajimi, T., Hirota, A., Shioya, R., Goshima, M. and Tsumura, T.: Initial Study of a Phase-Aware Scheduling for Hardware Transactional Memory, *Proc. IEEE Pacific Rim Conf. on Communications, Computers and Signal Processing (PacRim 2017)*, pp. 1–6 (2017).
- [13] 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).
- [14] 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 (HPCA'06)*, pp. 254–265 (2006).
- [15] Magnusson, P. S. et al.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).
- [16] 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).
- [17] Minh, C. C. et al.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
- [18] Alameldeen, A. R. et al.: Variability in Architectural Simulations of Multi-Threaded Workloads, *Proc. 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03)*, pp. 7–18 (2003).