Vol.2015-ARC-216 No.5
2015/8/4

履歴を利用したトランザクショナルメモリの競合回避手法

間下 恵介[1]　山田 遼平[1]　三宅 翔[1]　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）が期待を集めている．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（HTM）では，共有変数に対する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．この HTM では一度競合したトランザクション同士が再度並行実行される場合，競合が再発する可能性が高い．ところが従来の HTM では，このように競合が再発する可能性が高い場合も，トランザクションの実行を開始してしまうことで，一度競合したトランザクション同士で競合が頻発する場合がある．そこで本稿では，スレッドがトランザクションの実行を開始する前に，各トランザクションの競合履歴に基づいて，競合の発生を予測することで，競合を回避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 58.7%，平均 13.9%の性能向上を達成した．

1. はじめに

マルチコア環境の普及に伴い，プログラマが容易にプロセッサ性能を引き出すことのできる，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．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ているが，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，ロック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，共有変数の値が更新される際は，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，同一変数に対するアクセス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）．**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 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に，ほぼ同一の命令列を処理することから，同じ共有変数にアクセスする可能性が高い．そのため，一度競合したトランザクション同士が再度並列に実行される場合，再び競合する可能性が高い．ところが従来の HTM では，このように競合が再発する可能性が高い場合も，トランザクションの実行を開始してしまうことで一度競合したトランザクション同士で競合が頻発し，性能低下を引き起こすことがある．そこで本稿では，スレッドがトランザクションの実行を開始する前に，各トランザクションとの競合履歴に基づいて競合の発生を予測することで競合を回避し，HTM の性能を向上させる手法を提案する．

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 に関する数多く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．

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

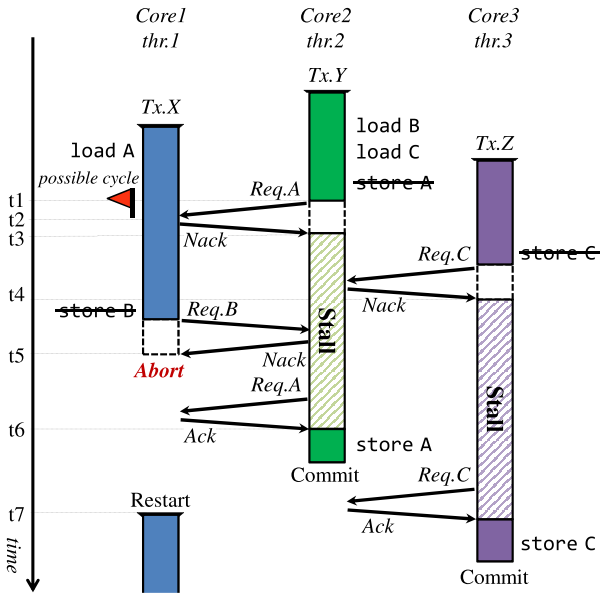


図 1　既存の競合解決と問題

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 [8] 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 [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手法を提案している．また，Akpinar らの研究 [10] では，Eager 方式の HTM 向けに新しい競合解決ポリシーをいくつか提案している．それらのポリシーでは，ストールやアボートしたトランザクションの数やタイムスタンプなど，様々な情報に基づいてトランザクションの実行優先度が決定される．

しかし，以上で述べた手法はいずれもトランザクションの実行中にスレッドの振る舞いを決定しており，スレッドがトランザクション実行前にある程度競合の発生を予測できる場合も，トランザクションの実行を開始することで，競合を引き起こしてしまう可能性がある．そこで本稿では，スレッドがトランザクションの実行を開始する前に，競合履歴を用いて競合の発生を予測し，競合を未然に回避する手法を提案する．

3. 履歴を利用した競合回避手法

本章では，既存の HTM における競合時の動作，およびその問題点を説明したのち，それを解決する手法について述べる．

3.1 HTM における競合解決とその問題

既存の HTM では同一アドレスに対するアクセス競合を検出するため，Read および Write ビットと呼ばれるフィールドが各キャッシュライン上に追加されている．トランザクション内で Read または Write アクセスが発生すると，アクセスされたキャッシュラインに対応する Read または Write ビットがセットされる．各スレッドはメモリアクセスの際に，キャッシュの状態を管理するディレクトリ機構に対し，既に同一アドレスがアクセスされているかを問い合わせるリクエストを送信する．ディレクトリはこのリクエストを受信すると，過去に当該アドレスにアクセスしたスレッドが存在するかをチェックする．その結果，そのようなスレッドが存在した場合，ディレクトリは当該スレッドに対しリクエストを転送し，これを受信したスレッドは当該ラインの Read および Write ビットを参照することで競合を検査する．

ここで，HTM の研究で広く用いられている競合検出方式であり，本研究の対象でもある **Eager Conflict Detection** 方式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．なお，本稿の説明ではディレクトリと各コア間の通信を，便宜上省略する．この図の例において，3 つのスレッド *thr.1*〜*3* がそれぞれ異なるトランザクション *Tx.X*，*Tx.Y*，*Tx.Z* を実行しており，*thr.1* が `load A` を実行し，*thr.2* が `load B` と `load C` を実行済みである場合を考える．この状態で，*thr.2* は `store A` の実行を試みて，*thr.1* に対しリクエストを送信し（時刻 t1），これを受信した *thr.1* は競合を検査する（t2）．その結果，*thr.1* は `load A` を実行済みであることから競合を検出し，*thr.2* に対し *Nack* を返信する．このとき，*thr.1* は *thr.2* に *Nack* を返信したことでその後にデッドロック状態になってしまう場合に備えて，*possible_cycle* と呼ばれるフラグをセットする．一方，*thr.2* はこの *Nack* を受信すると，`store A` を実行せず，*Tx.Y* をストールさせる（t3）．同様に，*thr.3* が `store C` の実行を試みて，*thr.2* に対しリクエストを送信すると，*thr.2* は `load C` を実行済みであることから競合を検出する．そのため *thr.2* は *thr.3* に *Nack* を返信し，これを受信した *thr.3* は *Tx.Z* をストールさせる（t4）．その後，*thr.1* が `store B` の実行を試みて，*thr.2* に対しリクエストを送信すると，*thr.2* は `load B` を実行済みであることから競合を検出する．そのため，*thr.2* は *thr.1* に *Nack* を返信する．これにより，*thr.1* は自身が *Nack* を送信した *thr.2* 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，*possible_cycle* をセットしている *thr.1* が *Tx.X* をアボートすることでデッドロックを回避する（t5）．これにより，*thr.2* はアドレス A にアクセス可能となる（t6）．一方，*Tx.X* をアボートした *thr.1* は一定時間待機した後，*Tx.X* を再実行する（t7）．

既存の HTM では以上で述べたように競合を解決するが，図 1 において *thr.2* と *thr.3* が競合してしまったように，トランザクションをストール中のスレッドによって新たな

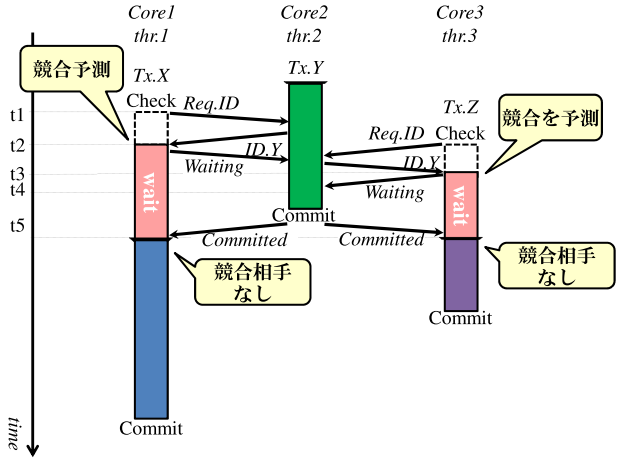
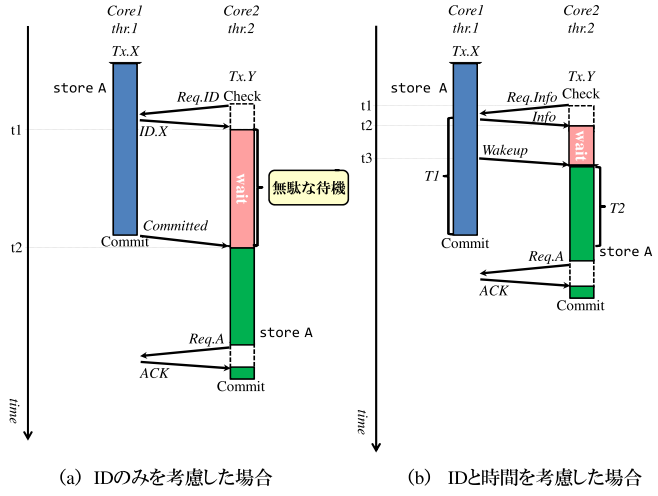


図 2　競合を回避する様子



(a) IDのみを考慮した場合　　(b) IDと時間を考慮した場合

図 3　無駄な待機が発生する様子

競合が引き起こされる可能性がある．このような競合の発生は，ストール中のスレッドが共有変数にアクセス済みであることと，ストールによりトランザクションの終了が遅れてしまうことに原因がある．

3.2　競合歴のあるトランザクションとの再競合の回避

前節で述べたように，スレッドが競合を回避するためにトランザクションをストールさせることで，新たな競合を引き起こしてしまう可能性がある．また，スレッドが同一のトランザクションを実行するたびに，同じ共有変数にアクセスする可能性が高いため，一度競合したトランザクション同士で競合が再発してしまうという特徴がある．そこで本稿ではこの特徴を考慮し，スレッドが自身の実行するトランザクションの ID と，競合相手トランザクションの ID とを履歴として記憶し，この履歴を利用してトランザクションの実行開始前に競合の発生を予測する手法を提案する．この競合予測により，競合が発生すると予測した場合は実行を開始せず，競合相手となるトランザクションを実行中のスレッドがコミットするまで待機することで，競合を未然に回避する．競合が発生すると予測したスレッドは，いずれの共有変数にもアクセスせずに待機するため，ストールとは異なり，待機中のスレッドによって新たな競合が引き起こされない．

ここで，図 1 と同じ例において，スレッドがトランザクションの実行を開始する前に，提案手法の競合予測を適用した場合の動作例を**図 2** に示す．この例では，*Tx.Y* が *Tx.X* および *Tx.Z* と過去に競合しており，それぞれのスレッドが競合の履歴を保持しているものとする．まず，*thr.2* が *Tx.Y* を実行中に，*thr.1* が *Tx.X* の実行を開始したとする（t1）．このとき，*thr.1* は競合予測のために，他の全てのスレッドで実行中のトランザクションの ID を問い合わせるリクエストを送信する．このリクエストを受信した *thr.2* は，自身の実行するトランザクションの ID である *Y* を *thr.1* に対して送信する．一方，*thr.1* はこの受信した ID が履歴の中に含まれているか否かを確認することで競合の発生を予測する（t2）．その結果，*thr.2* が実行中の *Tx.Y* と，*Tx.X* とが過去に競合していることが分かるため，*thr.1* はこのまま *Tx.X* を実行してしまうと再度競合する可能性が高いと判断し，*Tx.X* の実行を待機する．このとき，*thr.1* は自身が待機していることを伝えるため，*Waiting* メッセージを *thr.2* に対して送信する．その後，*thr.3* が *Tx.Z* の実行を開始する前も，*Tx.Z* と過去に競合した *Tx.Y* が *thr.2* で実行中であるため，競合が発生すると予測し，*thr.3* は *Tx.Z* の実行を待機し *Waiting* メッセージを *thr.2* に送信する（t3）．これに対し，*thr.2* は自身が *thr.1* および *thr.3* を待機させているため，*Tx.Y* をコミットする際に自身のコミットを伝える必要がある．そのため，*thr.2* はコミット時に，*Committed* メッセージを *thr.1* および *thr.3* に対して送信する（t4）．これを受信した *thr.1* と *thr.3* は，実行を開始したとしても競合が発生しないと判断し，待機から復帰し *Tx.X* および *Tx.Z* の実行を開始する（t5）．以上のように動作することで，これら 3 つのスレッドは競合することなく，トランザクションを実行することが可能となる．なお，トランザクション ID を問い合わせるリクエスト，*Waiting* メッセージ，および *Committed* メッセージはコヒーレンスプロトコルを拡張して新たに定義する．

3.3　無駄な待機の発生とその抑制

前節で述べた手法によって，スレッドは競合することなくトランザクションを実行することが可能となる．しかし，過去に競合相手となったトランザクションが実行中であるか否かを調べるだけでは，無駄な待機が発生してしまう可能性がある．ここで，前節で述べた手法において無駄な待

機が発生する場合を，**図 3**（a）を用いて説明する．なおここでは，*thr.2* が *Tx.Y* の過去の競合相手のトランザクション ID として *X* を記憶しているものとする．まず，*thr.1* が *Tx.X* を実行中に，*thr.2* が *Tx.Y* の実行を開始しようとする際，*thr.1* との間で競合が発生することを予測し実行を待機する（t1）．その後 *thr.2* は *thr.1* からの *Committed* メッセージを受信したことで競合が発生しないと判断し，*Tx.Y* の実行を開始する（t2）．以上で述べたように動作することで，*thr.1* および *thr.2* は競合を回避する．しかし，*thr.2* の `load A` の実行が *thr.1* のコミット後に行われさえすれば，*thr.1* のコミットを待たずに *thr.2* が *Tx.Y* の実行を開始したとしても競合は発生しないため，図 3（a）の例では待機のほとんどが無駄となってしまっている．

このような無駄な待機は，本来競合が発生しない場合も競合が発生すると予測を誤ることで発生する．そこで，競合予測時に，過去の競合相手のトランザクションを実行中のスレッドがコミットに至るまでの時間と，実行を開始しようとしているスレッドが競合に至るまでの時間とを比較することで，競合の発生をより正確に予測し無駄な待機を抑制する手法を，併せて提案する．この提案手法では，過去の競合相手トランザクションの ID に加え，各トランザクションの実行時間，および，各トランザクションが実行を開始してから競合するまでの時間が，履歴情報として必要となる．

ここで，競合を回避しつつ無駄な待機を抑制する提案手法の動作について図 3（b）を用いて説明する．なお，この例では *thr.1* および *thr.2* がそれぞれ競合予測に必要な履歴を保持しているものとする．まず，*thr.1* が *Tx.X* を実行中に，*thr.2* が *Tx.Y* の実行を開始しようとする際，競合の発生を予測する（t1）．このとき *thr.2* は，他の全てのスレッドで実行中のトランザクションの ID と，そのスレッドがコミットするまでの残り時間を問い合わせるリクエストを送信する．これを受信した *thr.1* は，自身の実行するトランザクションの ID である *X* を送信するとともに，履歴に基づいて予測した，コミットまでの残り時間 *T1* を *thr.2* に返信する．一方，これを受信した *thr.2* は，履歴として保持されている，*Tx.Y* が実行を開始してから *Tx.X* と競合するまでの予測時間 *T2* と，受信した *T1* とを比較する（t2）．その結果，*T2* の方が短い場合，*thr.2* は競合が発生すると予測し，実行を待機する．この時，図 3（a）の場合と同様に，*Waiting* メッセージを *thr.1* に送信する．その後，*thr.1* の実行が進み，*T2* よりも *T1* の方が短くなると，*thr.2* が実行を開始したとしても，競合が発生しないと判断し，実行の開始を許可する *Wakeup* メッセージを *thr.2* に送信する（t3）．これを受信した *thr.2* は *Tx.Y* の実行を開始する．以上で述べたように動作することで，*thr.1* および *thr.2* は競合を回避しつつ，無駄な待機時間の発生を抑制することが可能となる．なお，*Wakeup* メッセージはコヒーレンスプロトコルを拡張して新たに定義する．

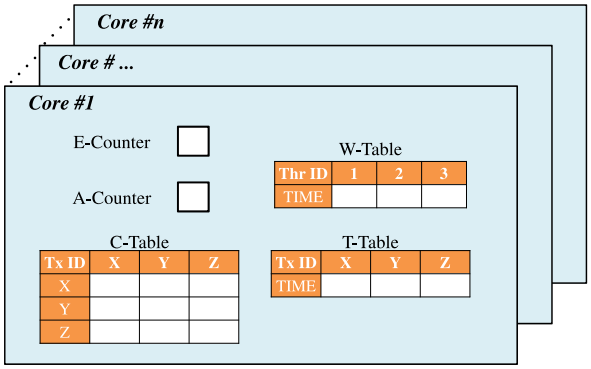


図 4 拡張したハードウェア

4. 実装

本章では 3 章で述べた提案手法の実装方法と，その動作モデルについて述べる．

4.1 拡張したハードウェア構成

提案手法を実現するために既存の HTM を拡張し，**図 4** に示す 5 つのハードウェアを各コアに追加する．なおこの提案手法では，3.3 節で述べたように．無駄な待機を抑制するために，各トランザクションの実行時間，および，各トランザクションが実行を開始してから競合するまでの時間を記憶する必要がある．しかし，このような時間を正確に求めることは困難である上，ストールやキャッシュミスなどが原因で実行時間が大幅に変化してしまう場合がある．そこで提案手法では，メモリアクセス回数を時間の近似値として用いることとする．追加したハードウェアについての詳細を以下に示す．

Enemy-Counter（E-Counter）： いくつのスレッドが，過去に競合相手となったトランザクションを実行中であるかを記憶するカウンタ．

Access-Counter（A-Counter）： トランザクションの実行中に発生したメモリアクセスの回数を記憶するカウンタ．スレッドがトランザクションをコミットもしくはアボートする際にクリアされる．

Conflict-Table（C-Table）： トランザクション ID ごとに，競合相手トランザクションとの競合に至るまでのメモリアクセス回数を管理するテーブル．

Wait-Table（W-Table）： 自身が待機させているスレッドに対して，*Wakeup* メッセージを送信するまでに残り何回メモリアクセスするかを記憶するテーブル．スレッドがメモリアクセスするごとに，値をデクリメントする．

Time-Table（T-Table）： トランザクション ID ごとに，トランザクションが実行を開始してからコミットに至るまでに発生するメモリアクセス回数を管理する

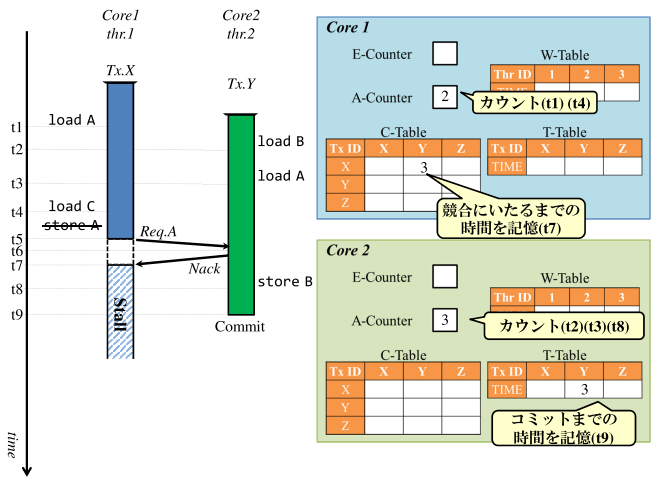


図 5　初回競合時の動作と t9 における追加ハードウェアの様子

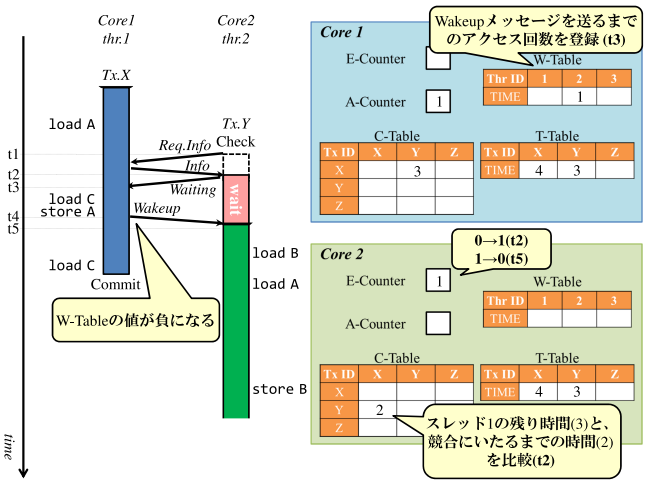


図 6　無駄な待機を抑制する競合予測と t3 における追加ハードウェアの様子

テーブル．

4.2　履歴の登録

前節で示した追加ハードウェアに情報を登録する流れについて図 5 を用いて説明する．この例では *thr.1* および *thr.2* が *Tx.X* と *Tx.Y* を並列実行している．まず，*thr.1* が `load A` を実行すると，自身の A-Counter の値をインクリメントする（t1）．同様に，*thr.2* が `load B` および `load A` を実行すると，自身の A-Counter の値をインクリメントする（t2，t3）．その後，*thr.1* が `load C` を実行し，自身の A-Counter の値をインクリメントした後に（t4）`store A` を試みて，*thr.2* に対してリクエストを送信する（t5）．このリクエストを受信した *thr.2* は，アドレス `A` に既にアクセス済みであるため競合を検出し，*Nack* を返信する（t6）．*Nack* を受信した *thr.1* は既存の HTM と同様にトランザクションをストールさせる（t7）．さらにこのとき *thr.1* は，*Tx.X* が *Tx.Y* と競合に至るまでの時間を示す値として，この時点における A-Counter の値をインクリメントした 3 を C-Table に登録する．これにより，C-Table には *Tx.X* と *Tx.Y* が競合したことと，*Tx.X* が *Tx.Y* と競合に至るまでの時間とが、履歴情報として保持される．その後，*thr.2* は `store B` を実行すると A-Counter の値をさらにインクリメントする（t8）．この状態で *thr.2* が *Tx.Y* のコミットに至ったとすると，*Tx.Y* の開始からコミットまでの時間を示す値として，この時点における A-Counter の値 3 を T-Table に記憶する（t9）．以上で述べたように動作させることで，各スレッドは競合予測に必要な情報を記憶する．

4.3　無駄な待機を抑制する競合予測

本節では，前節で述べた動作により記憶された履歴情報を用いた，競合予測の動作について，図 6 を用いて説明する．この例では，*thr.1* と *thr.2* が前節で述べた動作により，競合予測に必要な履歴を保持しているものとする．まず，*thr.2* は競合予測のために，他の全てのスレッドが実行しているトランザクションの ID と，そのトランザクションのコミットまでの残り時間を問い合わせるリクエストを送信する（t1）．このリクエストを受信した *thr.1* は，自身の T-Table に登録されている *Tx.X* の総アクセス回数 4 から，A-Counter に登録されている現在のアクセス回数である 1 を引いた値 3 を，コミットまでの残り時間を示す値として算出する．そして，*thr.1* は自身が実行しているトランザクションの ID である *X* と，この算出した値 3 とを *thr.2* に返信する．これらを受信した *thr.2* は，*thr.1* から取得した情報と，自身が保持している履歴を用いて競合の発生を予測する（t2）．この際，過去の競合相手トランザクションが実行中であるか否かを調べることに加え，過去の競合相手トランザクションを実行中のスレッドがコミットするまでの残り時間と，C-Table に登録されている，自身がトランザクションの実行を開始してから競合を引き起こすアクセスまでの時間とを比較することで競合の発生を予測する．その結果，コミットまでの残り時間の方が短い場合，トランザクションの実行を開始しても競合が発生しないと判断し，実行を開始する．これに対し，コミットまでの残り時間の方が長いか等しい場合は，トランザクションの実行を開始することで競合が発生すると判断し，実行を待機する．図 6 の例では，*thr.2* が *Tx.Y* の実行を開始してから競合を引き起こすアクセスまでのアクセス回数は C-Table より 2 であり，*thr.1* のコミットまでの残り時間の方が長い．そのため，*thr.2* はこのまま *Tx.Y* の実行を開始すると *thr.1* と競合する可能性があると判断し，実行を待機する（t3）．同時に，*thr.2* は競合相手数である 1 を E-Counter に登録し，*thr.1* に対して自身の待機を伝える *Waiting* メッセージと，*thr.2* が *Tx.Y* の実行を開始してから *Tx.X* と

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

競合を引き起こすアクセスまでの時間を示す値である 2 を *thr.1* に送信する．そして，*thr.1* は，*Tx.X* のコミットまでの残り時間を示す値 3 から，受信した 2 を引いた値である 1 を，*thr.2* に実行開始を許可するまでの残り時間を示す値として W-Table に記憶する．その後，*thr.1* が `load C` および `store A` を実行したとき，*thr.1* は W-Table の値をデクリメントする（t4）．これにより W-Table の値が負になるため，*thr.1* は *thr.2* に対して *Tx.Y* の実行を許可する *Wakeup* メッセージを送信し，これを受信した *thr.2* は E-Counter の値をデクリメントする（t5）．*Wakeup* メッセージを受信したことにより，*thr.2* の E-Counter の値が 0 になると，実行を開始したとしても競合が発生しないと判断し，*thr.2* は *Tx.Y* の実行を開始する．以上で述べたように動作することで競合を回避しつつ，3.3 節で述べた無駄な待機を抑制することが可能となる．

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，その結果について考察する．

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[11] に実装し，シミュレーションにより評価した．評価には Simics[12] 3.0.31 と GEMS[13] 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，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[14]，および STAMP[15] 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．

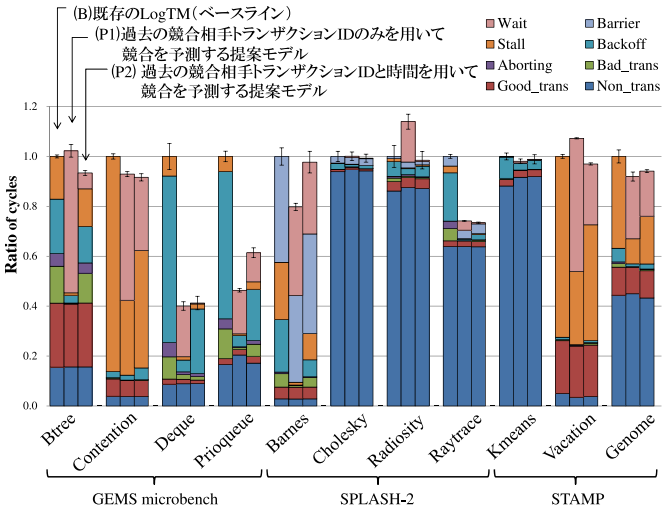


図 7　各プログラムにおけるサイクル数比

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
(P1)	平均	29.6%	8.0%	0.9%	13.9%
	最大	59.9%	25.9%	8.0%	59.9%
(P2)	平均	28.1%	7.8%	3.3%	13.9%
	最大	58.7%	26.5%	5.9%	58.7%

5.2 評価結果

評価結果を**図 7**，および**表 2** に示す．図 7 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（ベースライン）

(P1) 過去の競合相手トランザクションの ID のみを用いて競合を予測する提案モデル

(P2) 過去の競合相手トランザクションの ID と実行時間を用いて競合を予測する提案モデル

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 [16]．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，Barrier はバリア同期に要したサイクル数，Wait は提案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．

評価結果より，(P1) は既存モデル (B) に対して最大 59.9%，平均 13.9%の性能向上を達成できており，(P2) は既存モデル (B) に対して最大 58.7%，平均 13.9%の性能向上を達成できた．

5.3 考察

2 つの提案モデル (P1) および (P2) の性能は，既存モデル (B) と比較して同等か，または向上していることがわかる．特に，提案モデル (P2) では全てのベンチマークで既存モデル (B) と比較して性能が向上している．しかし，2 つの提案モデル (P1) および (P2) を比較すると，提案モデル (P2) の方が提案モデル (P1) よりも Wait を抑制できているものの，性能が低下しているものがある．これは，トランザクションが実行される際に，条件分岐等による影響で，トランザクション内におけるメモリアクセス回数が変化することで，トランザクション実行時間が正しく見積もれず，競合が発生しないと予測した場合も競合が発生してしまうことに原因がある．これはどのプログラムにおいても発生しており，その結果 Stall，Aborting，Bad_trans，および Backoff に関して (P1) よりも (P2) の方が増加している．しかし，Contention，Radiosity，および Raytrace には，メモリアクセス回数が一切変わらないトランザクションが含まれている．そのため，提案モデル (P2) では，スレッドがそれらのトランザクションを実行する際に，競合の発生を比較的正確に予測でき，提案モデル (P1) に比べて Wait を抑制しつつ性能向上したと考えられる．

プログラム別に見るとまず，Deque，Prioqueue では，2 つの提案モデル (P1) および (P2) の両方で性能が大きく向上している．これら 2 つのプログラムでは，実行されるトランザクションが 1 種類のみであるため，実行されるたびに同一の共有変数にアクセスしてしまう．そのため，既存モデル (B) ではそれらが並列実行されることで競合が頻発していた．その結果，ストールだけでなく，アボートが繰り返され，Aborting，Bad_trans，および Backoff が増加してしまったと考えられる．これに対し，2 つの提案モデル (P1) および (P2) では競合予測により，競合の発生回数を大幅に削減することができた．これにより，アボートの発生回数も大幅に削減し，Aborting，Bad_trans，および Backoff も削減できたと考えられる．

また Vacation では，提案手法 (P1) において Stall を大幅に削減したが Wait が増大し，結果として提案モデル (B) に対して性能が低下した．Vacation に含まれるトランザクションは非常に大きいため，一度競合すると Stall も大きくなってしまう場合が多い．これに対し提案モデル (P1) では，競合を抑制することで Stall を削減できているものの，Wait が大幅に発生してしまった．これは，Vacation には条件分岐を含むトランザクションが含まれており，実行されるたびにアクセスする共有変数が異なることで，本来競合が発生しない場合も，競合が発生すると誤って予測してしまう場合が多く，性能が低下したと考えられる．一方，提案モデル (P2) では，Stall を提案モデル (P1) ほど削減できていないものの，性能が若干向上している．これは競合が発生しないと誤って予測することが多いものの，無駄な待機の発生回数も大幅に削減できたためであると考えられる．

Btree でも同様に，提案モデル (P1) において実行時間に占める Wait の割合が非常に大きく，性能が低下している．このプログラムには，2 種類のトランザクション（それぞれを *Tx.Lookup*，*Tx.Insert* とする）が存在している．*Tx.Lookup* には Load 命令のみが含まれる一方，*Tx.Insert* には Load および Store 命令の両方が含まれており，*Tx.Insert* 同士が並列実行される際には競合が頻発してしまうが，*Tx.Lookup* と *Tx.Insert* は並列実行されてもあまり競合しない．ところが，提案モデル (P1) では *Tx.Lookup* と *Tx.Insert* とが一度でも競合すると，お互いが競合相手として記憶されてしまう．その結果，本来競合しない場合も競合が発生すると予測を誤り，これら 2 つのトランザクションが並列実行されるたびに待機が発生してしまうことで，Wait の割合が非常に大きくなってしまったと考えられる．一方提案モデル (P2) でも，*Tx.Lookup* と *Tx.Insert* とが互いに競合相手として記憶されてしまうが，競合予測に時間を用いたことで，提案モデル (P1) と比較して無駄な待機の発生を抑制できたと考えられる．また，提案モデル (P2) では，競合が発生しないと誤って予測することが多く，提案モデル (P1) と比較して競合が頻発してしまっていた．しかし，Vacation と同様に無駄な待機の発生回数も大幅に削減しつつ，ある程度競合の発生を抑制できたことで，性能が若干向上したと考えられる．

また，Barnes において 2 つの提案モデル (P1) および (P2) を比較すると，Wait の割合があまり変化していない．Barnes には最大で約 18 万回メモリアクセスされるトランザクションが含まれている．そのため，提案モデル (P2) において待機時間を抑制できる余地が少なく，2 つの提案モデル (P1) および (P2) において Wait の割合があまり変わらなかったと考えられる．なお，Choleskey および Kmeans では (P1) および (P2) ともに，競合の発生回数を削減できたものの，実行時間に占める Stall，Aborting，Bad_trans，および Backoff の割合が非常に小さいため，大きな性能向上は得られなかった．

5.4 ハードウェアコスト

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について述べる．まず，C-Table には，トランザクション ID ごとに競合までのメモリアクセス回数を全て記憶できるだけのビット幅が必要である．そこで，今回評価に用いた全てのベンチマークを調査したところ，

実行されるトランザクションの最大数は 17 であり，実行されるトランザクションの中でメモリアクセス回数の最大値は 180520 であった．したがって，18bits のエントリが 17 個必要となり，テーブルは 17 行の深さが必要となる．つまり，C-Table は幅 18bits × 17 = 306bits，深さ 17 行の RAM で構成できる．また，A-Counter で記憶すべきメモリアクセス回数の最大値は 180520 であるため，18bits のビット幅が必要である．同様に，T-Table は実行されるトランザクションの最大数である 17 個分だけ 18bits のエントリが必要となる．したがって，T-Table は幅 18bits × 17 = 306bits の RAM で構成される．また E-Counter は，最大で自身を除く全スレッド数をカウントする必要があるため，16 スレッドを並列実行する場合の必要ビット幅は 4bits となる．さらに，自身が待機させているスレッドが存在する場合，そのスレッドにトランザクションの実行開始を許可するまでのメモリアクセス回数を，W-Table に記憶する必要がある．したがって，16 スレッドを並列実行する場合，1 つのスレッドは最大で 15 個のスレッドを待機させるため，18bits × 15 = 270bits 必要となる．これらを総合すると，提案手法を実現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 11.6KBytes となり，1 コアあたり約 725Bytes となる．この 725Bytes 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 と比較しても十分に小さいものである．

6. おわりに

本稿では，ストール中のトランザクションによって新たな競合が引き起こされてしまう問題に対して，トランザクションの実行を開始する前に競合の発生を予測することで，競合を未然に回避する手法を提案した．提案手法では，過去に競合したトランザクションの ID や，競合に至るまでの時間などの履歴を用いて競合の発生を予測する．競合が発生すると予測したスレッドは，実行を開始しても競合が発生しなくなるタイミングまで自身のトランザクションの実行を待機することで，競合を回避する．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した結果，既存の HTM と比較して 16 スレッドで最大 58.7%，平均 13.9%の実行サイクル数が削減されることを確認した．また，既存の HTM に対して提案手法を適用する上で必要となるハードウェアのコストを概算したところ，16 コア構成のプロセッサの場合約 11.6KBytes となり，小容量の追加ハードウェアで実現できることを確認した．

しかし，分岐命令などにより，処理される実行命令列が変化するようなトランザクションにおいて，アクセスする共有変数が変化することで競合予測を誤る場合が見られた．その結果，競合が発生しないと予測したとしても競合してしまう場合や，待機時間が無駄となってしまう場合があった．したがって，今後はより適切な競合予測の方法について検討する必要がある．

謝辞 本研究の一部は，立松財団一般研究助成による．

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 (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., 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 Gonzlez, 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] 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).

[11] 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).

[12] 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).

[13] 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).

[14] 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).

[15] 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).

[16] 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).