Vol.2017-ARC-225 No.1
Vol.2017-SLDM-179 No.1
Vol.2017-EMB-44 No.1
2017/3/9

トランザクショナルメモリにおける競合予測手法の精度解析および改良

廣田 杏珠[1]　多治見 知紀[1]　間下 恵介[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リソースへのメモリ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，共有メモリ上でのアクセスが競合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，競合の発生によりトランザクションの投機実行失敗が頻発すると，性能が低下する場合がある．この問題に対し，トランザクション実行開始前に競合の発生を予測し，実行を待機することで競合を回避する手法を我々は提案している．しかし，なお性能向上が達成されていないプログラムが存在する．そこで本稿では，そのようなプログラムに対する性能向上を妨げている原因を調査した．競合予測精度の結果をうけて，待機時間の上限値を設定してシミュレーションによる予備評価を行った結果，Vacation において最大約 4.5%の実行サイクル数が削減できることを確認した．

1. はじめに

マルチコア環境の普及に伴い，プログラマが比較的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，共有リソース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きたが，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ある．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が困難であるなど，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含む処理区間を，**トランザクション**として定義し，共有メモリ上でアクセス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度が向上する．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，共有メモリ上のデータが更新される際は，更新前のデータ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，同一リソースに対するアクセス競合が発生していないかを常に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ためのオーバヘッドを軽減している．

さてこの HTM では，トランザクションはその処理内容が変化しなければ実行のたびに同じ共有変数にアクセスするため，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．この特徴を用いて，我々はトランザクション実行開始前に競合の発生を予測する手法 [2] を提案している．この手法では，トランザクションを開始する際に，過去に競合したことのあるトランザクションが他スレッドで実行中である場合，トランザクションの実行履歴から当該トランザクションの実行時間を予測する．そして，予測した実行時間に基づいて，競合を引き起こすメモリアクセスが競合相手トランザクションのコミット後になるように実行を待機することで競合の発生を未然に回避する．この手法は多くのプログラムに対して性能向上を達成しているが，その一方で性能向上を達成できていないプログラムも存在する．そこで本稿では，それらのプログラムに対する性能向上を妨げている原因を調査し，性能向上を達成するための手法について考察する．

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

Vol.2017-ARC-225 No.1
Vol.2017-SLDM-179 No.1
Vol.2017-EMB-44 No.1
2017/3/9

2. 競合予測手法

本章では HTM の概要と，実行パスを考慮して競合の発生を予測・回避する，既存手法 [2] について述べる．

2.1 HTM における競合解決とその問題点

本節では，標準的な 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 *thr.*1，*thr.*2 がそれぞれ異なるトランザクション *Tx.X*，*Tx.Y* を実行しており，*thr.*1 が `load A` を，*thr.*2 が `load B` 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 に対しアドレス A 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 *thr.*1 は競合を検出し，*thr.*2 に対し *Nack* を返信する（t2）．*thr.*2 は *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）．その後，*thr.*1 が `store B` の実行を試みるが，*thr.*2 は `load B` を実行済みであることから競合を検出し，*thr.*1 に *Nack* を返信する．これにより，*thr.*1 は自身が *Nack* を送信した先である *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，*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（t4）．これにより，*thr.*2 はアドレス A にアクセス可能となり，`store A` を実行する（t5）．一方，*thr.*1 は一定時間待機した後，*Tx.X* を再実行する（t6）．

既存の HTM では以上で述べたように競合を解決するが，ストール中のトランザクションが他の共有変数に既にアクセス済みである場合，自身の実行が進行していないにもかかわらず他のトランザクションとの新たな競合を引き起こす可能性がある．また，アボート発生時には，トランザクション開始時点のメモリ状態を復元するコストがオーバヘッドとなる上，アボートまでの処理が無駄となる．このように，HTM では競合解決時に発生するオーバヘッドによる性能低下が問題となる．

2.2 実行時間を用いた競合予測

HTM では，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いという特徴がある．これは，スレッドが同一のトランザクションを実行するたびに，同じ共有変数にアクセスする可能性が高いためである．そこで我々はこの特徴を考慮し，トランザクション開始前に過去に競合したトランザクションとの競合の発生を予測し，トランザクションの実行時間に基づいて実行を待機することで競合を回避するスケジューリング手法 [3] を提案している．この手法では，トランザクション実行開始時に，過去に競合したことのあるトランザクションが他スレッドで実行中であった場合，そのトランザクションのコミットまでの残り時間 $\tau 1$ と，自身が実行を開始してから競合を引き起こすアクセス

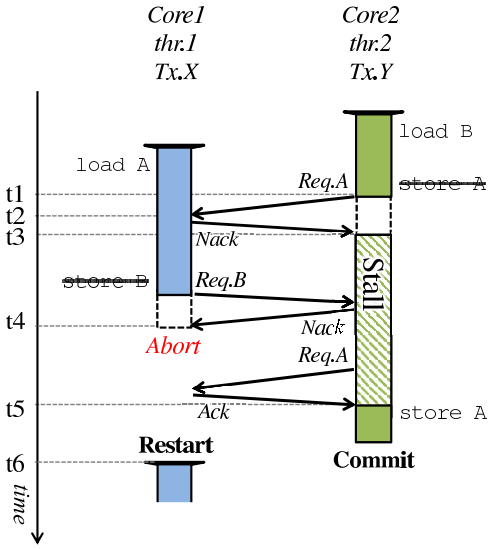


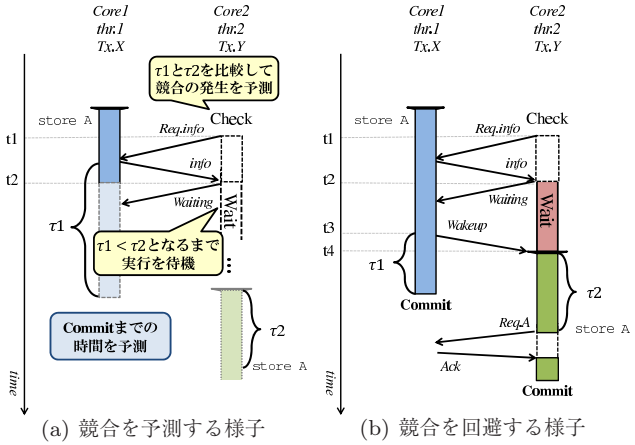
図 1 既存の競合解決と問題

までの時間 $\tau 2$ とを比較することで競合の発生を予測する．競合を引き起こすアクセスより前に競合相手がコミットすれば競合は発生しないため，$\tau 1 < \tau 2$ であれば競合が発生しないと予測し，自身の実行を開始する．一方，$\tau 1 \geqq \tau 2$ であれば競合すると予測し，$\tau 1 < \tau 2$ となるまでトランザクションの実行を待機する．この比較のため，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．そして，これらの情報を用いて競合の発生を予測し，実行を待機することで競合を回避する．

この競合予測手法により競合を回避する動作について，**図 2** を用いて説明する．なお，実行時間表現として実時間やサイクル数を用いると，ストールやキャッシュミスの影響によりその値が大きく変動する可能性があるため，この手法ではメモリアクセス回数を実行時間表現として用いている．図 2(a) はトランザクション実行開始時に競合の発生を予測する様子を表しており，図 2(b) は図 2(a) における予測結果に基づいて実行を待機し，競合を回避する様子を表している．なお，この例では *Tx.X* と *Tx.Y* は過去に競合しており，それぞれのスレッドが競合予測に必要な情報を既に記憶しているものとする．

まず，図 2(a) において *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，競合の予測を行う（t1）．このとき *thr.*2 は，競合予測に必要な情報を得るため，他のスレッドに対して，現在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 *Req.info* 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* と，自身が保持している情報を基に予測したコミットまでの残り時間 $\tau 1$ とを *thr.*2 に返信する．一方，これらを受信した *thr.*2 は，自身が保持している情報である *Tx.Y* が競合を引き起こすメモリアクセスまでの予測時間 $\tau 2$ と，

Vol.2017-ARC-225 No.1
Vol.2017-SLDM-179 No.1
Vol.2017-EMB-44 No.1
2017/3/9



(a) 競合を予測する様子　(b) 競合を回避する様子

図 2　実行時間を用いた競合予測

受信した $\tau 1$ とを比較する（t2）．比較した結果，$\tau 1$ の方が短い場合は競合が発生しないと予測し，*thr.*2 は *Tx.Y* の実行を開始する．一方，この例のように，$\tau 2$ の方が短い場合は *thr.*2 が *Tx.Y* の実行を開始すると競合が発生すると予測し，*Tx.Y* の実行を待機する．このとき *thr.*2 は，*thr.*1 が実行している *Tx.X* によって待機させられていることを伝えるために，*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，図 2(b) の t3 において $\tau 2$ よりも $\tau 1$ の方が短くなると，*thr.*2 が実行を開始しても競合が発生しないと判断し，*thr.*1 は実行の開始を許可する *Wakeup* メッセージを *thr.*2 に送信する．*thr.*2 は *Wakeup* メッセージを受信すると待機状態から復帰し，*Tx.Y* の実行を開始する（t4）．

以上で述べたように動作することで競合を回避する．なお，競合を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り新たな競合が引き起こされることはない．

2.3　実行パスの変化を考慮した実行時間の予測

前節で述べた競合予測の有効性は，トランザクション実行時間の予測精度に大きく影響を受けると考えられる．HTM では，トランザクション内の実行パスが常に同じであれば，トランザクションの実行時間はほぼ一定である可能性が高い．しかし，トランザクション内に分岐命令を含む場合，その影響により，同一のトランザクションでも実行パスが変化し，実行時間が大きく変動する可能性がある．それに伴い，前節で述べた競合予測も失敗する可能性が高くなる．これはトランザクション内の実行パスを予測することで解決可能であると考えられるが，その予測結果を待機時間の設定に用いるためには，予測をトランザクションの実行を開始する前に行う必要がある．そこで我々が提案している競合予測手法 [2] では，広域分岐予測 [4] の考え方を応用している．広域分岐予測とは，全ての分岐命令の履歴を一元的に扱い，直近の実行パスから今後実行されるパスを予測する手法である．競合予測手法ではこれを利用して，トランザクションの実行開始時点に到達するまでに発行された，直近のロード・ストアの出現パターンを直近の実行パス表現とみなし，これを用いてトランザクションの実行時間を予測する．本稿では以降，このロード・ストアの出現パターンを**グローバルロードストア履歴**と呼ぶ．このグローバルロードストア履歴のパターン別に実行時間を記憶することで，トランザクション内の実行パスが変化する場合でも，適切な待機時間を設定することができる．

```
1  int i = 20;
2  int j = 0;
3   :
4  func( i, j ){
5   if( i > 10 ){
6     : /* 実行パスA */
7   }else{
8     : /* 実行パスB */
9   }
10  if( j > 10 ){
11    : /* 実行パスC */
12  }else{
13    : /* 実行パスD */
14  }
15  BEGIN_TRANSACTION();
16    if( i > 10 && j < 10){
17      : /* 実行パスE */
18    }else{
19      : /* 実行パスF */
20    }
21  COMMIT_TRANSACTION();
22 }
```

図 3　トランザクション内に分岐命令を含むプログラム

ここで，実行パスの変化を考慮した実行時間予測の概要を**図 3** に示すプログラム例を用いて説明する．なお，15 行目の `BEGIN_TRANSACTION`，および 21 行目の `COMMIT_TRANSACTION` はそれぞれトランザクションの開始と終了を表している．また，プログラム中に存在する 3 つの if 文それぞれに対応する then 側および else 側の実行パスを，図中に示すように $A \sim F$ と呼ぶこととする．このプログラムでは 1 番目，2 番目の分岐命令（5，10 行目）の分岐結果が 3 番目の分岐命令（16 行目）の分岐方向に影響を与える．

まず，`i` の値が 20，`j` の値が 0 であったとすると，1 番目の分岐命令（5 行目）が成立し，2 番目の分岐命令（10 行目）が不成立となる．この場合，パス *A*，*D* が順番に実行される．その後トランザクションの実行が開始されると，トランザクション内の分岐命令（16 行目）が成立するため，パス *E* が実行される．このとき，*A*，*D* で実行されたロードおよびストアの出現パターンをグローバルロードストア履歴として，*E* が実行された場合のトランザクション実行時間と関連づけて記憶する．その後，再び同じ関数 func が実行され，`i`，`j` の値が先ほどと同じであった場合，パス *A*，*D* が順番に実行される．そして，トランザクショ

Vol.2017-ARC-225 No.1
Vol.2017-SLDM-179 No.1
Vol.2017-EMB-44 No.1
2017/3/9

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

ン開始時にグローバルロードストア履歴を参照すると，過去にパス A，D が順番に実行された後，トランザクション内でパス E が実行された際の実行時間を取得できる．

このように，グローバルロードストア履歴とトランザクションの実行時間を関連づけて記憶し，トランザクション実行開始前にグローバルロードストア履歴を取得した上で，そのパターンと関連づけて記憶されている実行時間を基に競合予測を行うことで，競合を未然に回避する．しかし，グローバルロードストア履歴が同一であっても，実行時間が変化するようなトランザクションが存在する場合，適切な実行時間を予測できない可能性がある．これにより競合予測に失敗し，競合や無駄な待機時間が発生する．

3. 性能および予測精度の評価

本章では，競合予測手法の評価結果を示し，十分な性能向上を達成できていないベンチマークプログラムが存在することを示す．また，それらのプログラムについて，本手法の実行時間予測の精度を評価する．

3.1 評価環境

2 章で述べた競合予測手法を，HTM の研究で広く用いられている LogTM [5] に実装し，シミュレーションにより評価した結果を**図 4** に示す．評価には Simics 3.0.31 [6] と GEMS 2.1.1 [7] 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タ構成を示す．評価対象のプログラムとしては GEMS microbench, SPLASH-2 [8]，および STAMP [9] から計 11 個を使用した．なお，各ベンチマークプログラムはそれぞれ 16 スレッドで実行した．

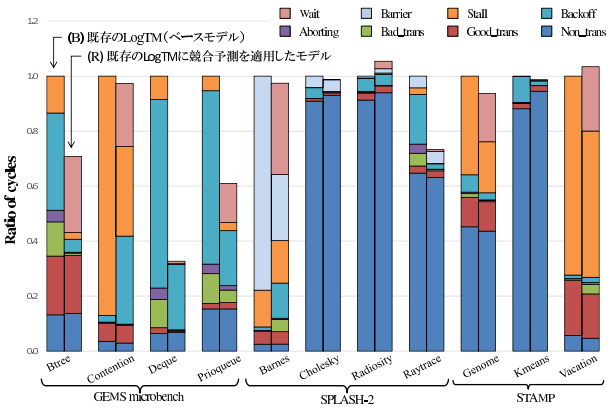


図 4　各プログラムにおける実行サイクル数比

3.2 競合予測手法による性能向上

図中では，各ベンチマークプログラムの評価結果がそれぞれ 2 本のバーで示されている．これらのバーはそれぞれ左から順に，

(B)　既存の LogTM（ベースモデル）
(R)　競合予測を行うモデル

の実行サイクル数を表しており，既存の LogTM (B) の実行サイクル数を 1 として正規化している．なおこの評価では，グローバルロードストア履歴の長さは 8 とし，それと関連づけて記憶する時間として，過去の実行において最短のものを選択することとしている．

凡例はサイクル数の内訳を示しており，Wait は競合予測を行うモデルで待機処理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，Stall はストール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に要したサイクル数，Aborting はアボート処理に要したサイクル数，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，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，Non_trans はトランザクション外の実行サイクル数をそれぞれ示している．

評価結果を見ると，多くのベンチマークプログラムで性能向上を達成できているが，Radiosity と Vacation では，性能向上が得られていない．特に，Vacation では依然として Stall，Backoff，Aborting，および Bad_trans を含む，競合解決に要する処理の占める割合が大きく，これは，実行時間の予測を誤る場合が多いことの表れであると考えられる．結果として，ベンチマーク全体では平均 13.8%の実行サイクル数を削減しているものの，Radiosity では約 5.4%，Vacation では約 3.4%増加してしまっている．

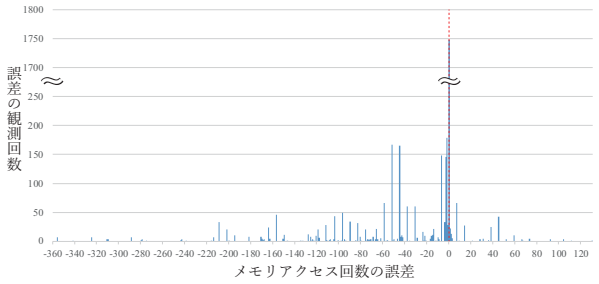
3.3 実行時間予測の精度

本節では，Radiosity および Vacation の性能向上が妨げられている原因を探るため，実行時間予測の精度を調査し，考察する．

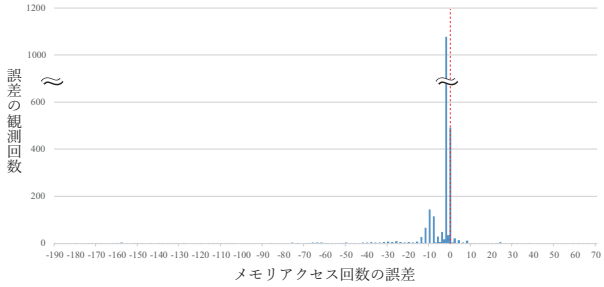
Radiosity

Radiosity に含まれる 10 種類のトランザクションのう

Vol.2017-ARC-225 No.1
Vol.2017-SLDM-179 No.1
Vol.2017-EMB-44 No.1
2017/3/9



(a) *Tx*.3



(b) *Tx*.10

図 5 Radiosity の予測精度

ち，実行される回数が多かったトランザクション *Tx*.3 および *Tx*.10 の実行時間予測の精度を**図 5** に示す．このグラフは，横軸が実行時間（メモリアクセス回数）予測の誤差，縦軸はその誤差を観測した回数を示している．なおここでは，トランザクションの予測実行時間から実際の実行時間を引いた差を誤差として定義している．つまり，誤差が正値の場合，実際の実行時間より予測時間が長く見積もられたことを示しており，競合相手のトランザクションがコミットされた後に自身のトランザクションを実行開始するまでの待機が無駄となる．一方，誤差が負値の場合，実際の実行時間より予測時間が短く見積もられることを示しており，競合相手のトランザクションがコミットされる前に競合を引き起すメモリアクセスが実行されてしまうことで競合を回避できていない可能性が高い．

Tx.3 の予測精度を示した図 5(a) を見ると，−10 ∼ 0 の誤差を生じている頻度が高く，一見その精度は高いように見える．しかし，*Tx*.3 の平均実行時間は約 13.5 であり，実行時間比で考えると比較的大きな誤差が生じてしまっている．加えて，全予測機会のうち誤差が負値となった割合が，約 75.2%と非常に高く，予測を行っても競合回避に失敗している可能性が高いことが分かった．

一方，*Tx*.10 の予測精度を示した図 5(b) を見ると，ほとんどの場合において誤差が 0 となっており，正しく予測ができていること，また *Tx*.10 においては実行パスが同一であれば実行時間が変動する場合が少ないことが伺える．誤差が負値となった割合も，約 12.8%と低く，*Tx*.10 に対しては競合予測手法がうまく働いていると考えられる．

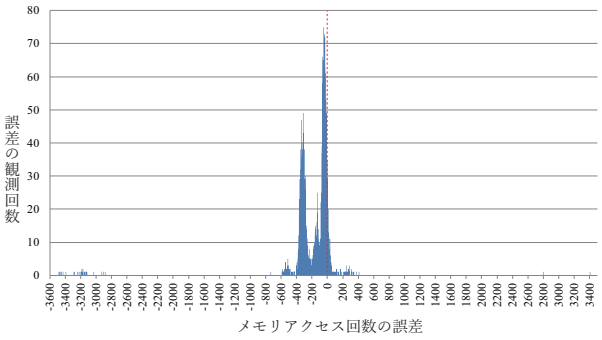


図 6 Vacation の予測精度

Vacation

Vacation に含まれる 3 種類のトランザクションのうち，最も実行される回数が多かったトランザクション *Tx*.0 の実行時間予測の精度を**図 6** に示す．

Vacation の *Tx*.0 では，トランザクションの実行全体のうち，−500 から 0 の誤差が生じている場合が多い．*Tx*.0 の平均実行時間は 478.5 であるため，この誤差は比較的大きく，誤差が負となっている回数も多いことから，競合回避に失敗しやすいと考えられる．また，図 4 から分かるように，Vacation で性能向上が得られていないのは，待機時間の増加に依るところが大きい．よって，競合を回避するために待機したにも関わらず，競合を回避することができず，その待機が無駄になってしまっていると考えられる．一方で，絶対値が 3,000 を超える誤差が発生する場合も観測された．先述したように，この手法では過去の最短の時間を記憶して予測に使用していることから，トランザクションの各実行パスの出現初期に大きな値を記憶してしまい，それを予測に用いた場合において，このような大きな誤差が発生していると考えられる．

そこで本稿では，この Vacation の *Tx*.0 に見られるように，誤差の絶対値が非常に大きくなってしまう場合を回避することで，性能の向上を図る．

4. 記憶する実行時間に制約を与える手法

本章では，Vacation のトランザクションが持つ特徴を述べ，その特徴と予測精度をふまえた改良方針を示す．

4.1 Vacation の概要

Vacation は，旅行の予約システムを模したプログラムであり，レンタカー，飛行機，ホテルの予約情報と，サービスを利用する顧客情報とをそれぞれテーブルで管理している．Vacation が持つ 3 種類のトランザクションは，これらの 4 つのテーブルのうちいずれかに予約情報を読み書きする．そのため，これらのテーブルが複数のスレッドからアクセスされることで，競合が引き起こされる可能性がある．ここで，*Tx*.0 のコードを**図 7** に示す．まず，2 行目の for

Vol.2017-ARC-225 No.1
Vol.2017-SLDM-179 No.1
Vol.2017-EMB-44 No.1
2017/3/9

```
1  BEGIN_TRANSACTION( 0 );
2    for( n = 0; n < numQuery; n++ ){
3      long t = types[n];
4      switch(t){
5      case RESERVATION_CAR;
6      /* レンタカーの予約価格の問い合わせ */
7      break;
8      case RESERVATION_FLIGHT;
9      /* 飛行機便の予約価格の問い合わせ */
10     break;
11     case RESERVATION_ROOM;
12     /* ホテルの予約価格の問い合わせ */
13     break;
14     }
15   }
16   if( isFound ){
17     /* 予約対象が存在した場合,顧客情報を登録 */
18   }
19   if( Id[RESERVATION_CAR] > 0 ){
20     /* レンタカーの予約確定 */
21   }
22   if( Id[RESERVATION_FLIGHT] > 0 ){
23     /* 飛行機便の予約確定 */
24   }
25   if( Id[RESERVATION_ROOM] > 0 ){
26     /* ホテルの予約確定 */
27   }
28 COMMIT_TRANSACTION(0);
```

図 7　Vacation 内のトランザクション $Tx.0$

文でレンタカー，飛行機，ホテルのうち予約を希望する対象の価格を問い合わせるため，テーブルへアクセスする．このとき，トランザクション開始前にランダム生成された配列 `types` の値が変数 `t` に格納され，問い合わせる予約対象はランダムに決定される．問い合わせの結果，予約できる対象が存在する場合，16 行目で顧客情報を登録するために，テーブルへアクセスする．さらに，19 行目，22 行目，25 行目でそれぞれレンタカー，飛行機，ホテルの予約を確定するために，テーブルへアクセスする．

さて，このトランザクションではアクセス先のテーブルがランダムに変化するため，グローバルロードストア履歴が同一であっても，実行時間が大きく変化する．そのため，あるグローバルロードストア履歴に対する初回実行時間の値が大きい場合，次回の実行時間の値が記憶した値と比べて小さいとしても，最初に記憶した値に基づいて待機時間を設定してしまい，無駄な待機時間が発生する．前節で述べたように $Tx.0$ の平均実行時間が約 478.5 であるのに対し，誤差が 3,000 を超える場合があるため，この待機時間は無視できないものである．

そこで，実行時間を記憶する際に，その値が一定の基準を超える場合にその値を記憶しないという単純な対処方法の効果を確認する．Vacation の $Tx.0$ の実行時間を調べたところ，最大値が 3,755 であったため，記憶する実行時間の上限値を 2,000，1,000，500，250 と変化させて評価した．

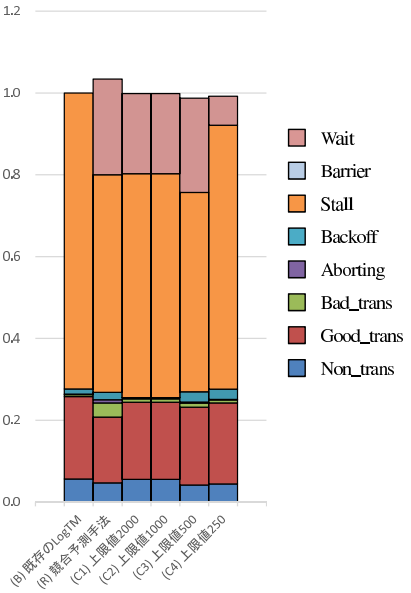


図 8　記憶する実行時間の上限値を設定した場合の実行サイクル数比

4.2　評価結果

前節で述べた改良案の有効性を確認するため，シミュレーションにより評価を行った．本節では，その評価結果を示し，考察する．なお本評価は，3 章に示した評価と同一の環境で行った．

評価結果を**図 8** に示す．図 8 では，Vacation の評価結果が図 4 で示した 2 本のバーに加えて，左から順に記憶する実行時間の上限値を 2,000**(C1)**，1,000**(C2)**，500**(C3)**，250**(C4)** に設定したモデルを加えた計 6 本のバーで示されている．また，図 4 同様，それぞれのバーは実行サイクル数を表しており，既存の LogTM (B) の実行サイクル数を 1 として正規化している．評価結果より，記憶する実行時間の値を制限するモデルでは，無駄な待機時間を削減できたことで，どの値で制限した場合でも既存の LogTM と同等程度まで実行サイクル数を抑制できている．

特に，上限値を 500 に設定したモデル (C3) では，全てのモデルのうち最もサイクル数が削減されており，既存の LogTM と比較して約 1.3%，上限値を設定しない既存の競合予測手法 [2] と比較して約 4.5%の性能向上が得られた．(C3) は (C1) や (C2) と比較して Stall サイクルを有意に抑制できているが，これは，大きい実行時間値が予測に用いられる機会が減ったことで，競合が予測されるトランザクション同士が並列に動作できる時間が増え，他のトランザクションとの競合機会が減少したためではないかと考えられる．実際 (C3) では，(C1) や (C2) に対し，Stall 回数も約 2.5%減少していた．

一方，上限値を 250 に設定したモデル (C4) では，平均実行時間より待機時間が短く設定される場合が多くなり競合が発生していた．そのため，競合予測を行う他のモデルと比べて Stall に要するサイクル数が増加したが，待機時間の抑制により，既存の LogTM と同等のサイクル数となった．

記憶する実行時間を 2,000，および 1,000 未満に制限したモデル (C1 および C2) では，長く待機したにもかかわらず，誤差が発生し競合する場合の待機時間を抑制したことで，競合予測を行う手法 (R) と比較して，待機処理に要したサイクル数を削減できた．なお，$Tx.0$ の実行時間は 1,000 以上 3,000 以下の値をとることが少なく，誤差も 1,000 ～ 3,000 の範囲の値をとることが少なかったため，この 2 つのモデルは同等の結果となった．

以上の結果から，トランザクションの実行時間が著しく変化する場合，記憶する実行時間に一定の制約を設けることで，競合の発生による性能悪化を抑えつつ，無駄な待機時間を抑制し性能を向上できることが分かった．しかし，今回の予備評価では単一のプログラムに対して，記憶する値の範囲を静的に決定したため，今後はそれぞれのプログラムごとに適切な上限値を動的に決定する手法などを検討する必要がある．

5. 関連研究

HTM に関して，アボート後の再実行コストを抑える，部分ロールバックに関する研究 [10–12] や，トランザクションが持つ様々な特徴情報に基づいて競合を抑制する研究 [13–15] など，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 [16] 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，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．また，Blake ら [17] は複数のトランザクション内におけるアクセスの局所性を Similarity と定義し，この Similarity がある一定の閾値を越えた場合に，当該トランザクションを逐次的に実行することで競合を抑制する手法を提案している．Akpinar ら [18] は HTM 向けに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．さらに，Armejach ら [19] はアボートの再発抑制を目的として Hardware Adaptive Recurrence Predictor（HARP）と呼ばれるアボートの発生を予測する機構を提案している．

このように，様々な手法が提案されているが，本稿で解析した Vacation をはじめとして，いずれの手法を用いても依然として大きな性能向上を実現できていないベンチマークプログラムが存在する．本稿では，トランザクション実行時間の変動という視点からこの Vacation について調査し，我々が既に提案している競合予測手法 [2] に対する改良指針を示すことで，これを高速化する道筋を示した．

6. おわりに

本稿では，HTM のスケジューリング手法の中でも多くのプログラムで高い性能を発揮している競合予測手法において，性能向上を得られていないプログラムの実行時間の予測精度を調査した．Vacation では誤差の絶対値が著しく大きくなる場合がある．これは，グローバルロードストア履歴が同一でも実行時間が大幅に変化するため，記憶された実行時間が平均実行時間より著しく大きい場合，待機時間が不必要に長く設定されてしまい，無駄な待機が発生していたためである．そこで，記憶する実行時間の上限値を設定し評価を行ったところ，上限値を 500 に設定した場合に，既存の LogTM と比較して約 1.3%，制限値を設定しない競合予測手法と比較して約 4.5%の性能向上を確認できた．

しかし，本稿の予備評価では，単一のプログラムに対して，記憶する実行時間の値を範囲を静的に設定した．したがって今後，それぞれのプログラムごとに適切な上限値を動的に決定する手法を検討する必要がある．

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] 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).
[3] Mashita, K., Tabuchi, M., Yamada, R. and Tsumura, T.: A Waiting Mechanism with Conflict Prediction on Hardware Transactinal Memory, *IEICE Trans. on Information and Systems*, Vol. E99-D, No. 12, pp. 2680–2870 (2016).
[4] Yeh, T.-Y. et al.: Two-level adaptive training branch prediction, *Proc. 24th Annual IEEE/ACM Int'l Symp on Microarchitecture(MICRO-24)*, ACM, pp. 51–61 (1991).
[5] Moore, K. E. et al.: LogTM: Log-based Transactional Memory, *Proc. 12th Int'l Symp. on High-Performance Computer Architecture (HPCA'06)*, pp. 254–265 (2006).
[6] Magnusson, P. S. et al.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).
[7] 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).
[8] 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).
[9] Minh, C. C. et al.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
[10] J.Moravan, M. et al.: Supporting Nested Transactional

Memory in LogTM, *Proc. 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 1–12 (2006).

[11] McDonald, A. et al.: Architectural Semantics for Practical Transactional Memory, *Proc. 33rd Annual Int'l Symp. on Computer Architecture (ISCA'06)*, pp. 53–65 (2006).

[12] Moss, E. et al.: Nested Transactional Memory: Model and Preliminary Architecture Sketches., *Science of Computer Programming*, pp. 186–201 (2006).

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

[14] Tomic, S. et al.: Eazyhtm, Eager-lazy Hardware Transactional Memory, *Proc. 42nd Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-42)*, pp. 145–155 (2009).

[15] Shriraman, A. et al.: Flexible Decoupled Transactional Memory Support, *Proc. 35th Annual Int'l Symp. on Computer Architecture (ISCA'08)*, pp. 139–150 (2008).

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

[17] Blake, G. et al.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th Int'l Conf. on High-Performance Computer Architecture (HPCA-17 2011)*, pp. 75–86 (2011).

[18] Akpinar, E. et al.: A Comprehensive Study of Conflict Resolution Policies in Hardware Transactional Memory, *Proc. 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11)* (2011).

[19] Armejach, A., Negi, A., Cristal, A., Unsal, O., Stenstrom, P. and Harris, T.: HARP: Adaptive abort recurrence prediction for Hardware Transactional Memory, *Proc. 20th Int'l Conf. on High Performance Computing (HiPC'13)*, pp. 196–205 (2013).