Vol.2016-ARC-221 No.7
2016/8/8

トランザクショナルメモリにおける実行パスを考慮したスケジューリング手法

廣田 杏珠[1]　田渕 茉也[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，共有リソースへのメモリ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 ではトランザクションの実行が投機的であるため，競合が発生し性能が低下する可能性がある．この問題に対し，スケジューリングの改良により競合を抑制する研究が数多く行われてきた．しかし従来の手法はいずれも，分岐命令などに起因するトランザクション内の実行パスの変化について考慮していない．そこで本稿では，トランザクション開始時に実行パスを予測して実行時間を見積もり，それに基づいて競合を回避するスケジューリング手法を提案する．シミュレーションによる評価の結果，16 スレッドで最大 61.6 %，平均 13.8 %の実行サイクル数削減を確認した．

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いるが，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，共有リソース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，共有リソース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，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．この特徴を利用し，トランザクションの実行時間に基づいてスケジューリングを行うことで，競合を回避できると考えられる．しかし，分岐命令によりトランザクション内の実行パスが変化する場合，同一のトランザクションでも実行時間が変化するため，スケジューリングによる効果が十分に得られない可能性がある．そこで本稿では，トランザクション開始時に実行パスを予測することで，より正確に実行時間を見積もり，それに基づいて競合を回避するスケジューリング手法を提案する．

2. 関連研究

HTM に関しては，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，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

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

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，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 [9] 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．また，Akpinar ら [10] 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．

しかし，以上で述べた手法はいずれも，分岐命令などに起因するトランザクション内の実行パスの変化について考慮しておらず，それにより発生するトランザクションの実行時間やメモリアクセスパターンの変化に対応できていない．そこで本稿では，トランザクション開始時に実行パスを予測して実行時間を見積もり，それに基づいて競合を回避する手法を提案する．

3. 実行パスを考慮したスケジューリング手法

本章では，トランザクション開始時に，分岐予測と同様の考え方によりトランザクション内の実行パスを予測することで実行時間を見積もり，それに基づいて競合を回避するスケジューリング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点

本節では，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，`storeA` 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）．

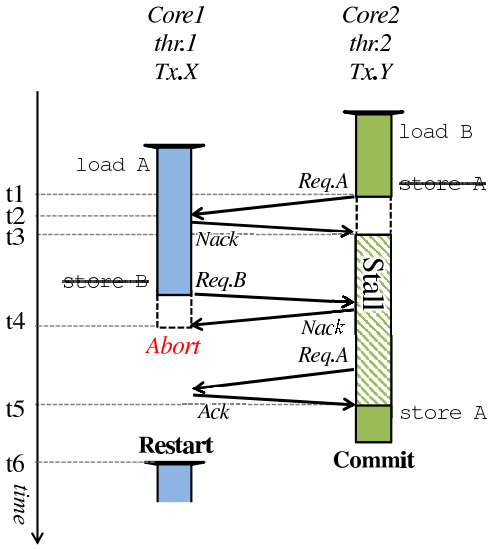


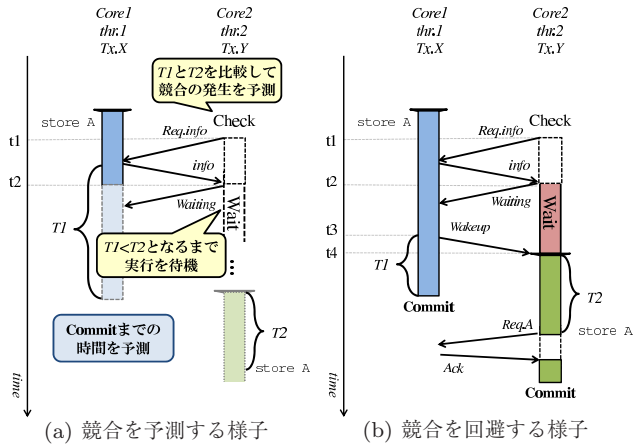
図 1　既存の競合解決と問題

3.2 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．そこで本稿ではこの特徴を考慮し，トランザクション開始時に一度競合したトランザクション同士による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を提案する．提案手法では，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 *T1* 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までの時間 *T2* 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のタイミングが，競合相手のコミット後であれば実際には競合は発生しないため，$T1 < T2$ 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，$T1 > T2$ であれば競合すると予測し，*thr.2* は $T1 < T2$ となるまで *Tx.Y* の実行を待機する．このため提案手法では，過去の競合相手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，図 2(a) はトランザクション開始時に競合の発生を予測し，コミットまでの実行時間を見積もった様子を表しており，図 2(b) はその後実行が進み，競合を回避する様子を表している．また，この例では *Tx.X* と *Tx.Y* が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保持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*



(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

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元に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 *T2* と，受信した *T1* とを比較する（t2）．比較した結果，*T1* 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，*T2* 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 *T2* よりも *T1* 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．

3.3　実行パスの変化を考慮した実行時間の予測

前節で述べたスケジューリング手法の有効性は，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の分岐命令の影響により，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，提案手法による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，その予測結果をスケジューリングに用いるためには，予測をトランザクションに進入する前に行う必要がある．そこで提案手法では，**広域分岐予測** [11] の考え方を応用する．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．提案手法ではこれを利用して，トランザクション開始時点に到達するまでに発行された，直近のロードおよび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内の実行パスを予測する．本稿では以降，この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**と呼ぶ．そして，グローバルロードストア履歴のパターンごとに実行時間を記憶することで，実行パスの変化に応じた実行時間を予測する．

```
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，本提案手法を**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`，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*〜*F* で呼ぶこととする．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

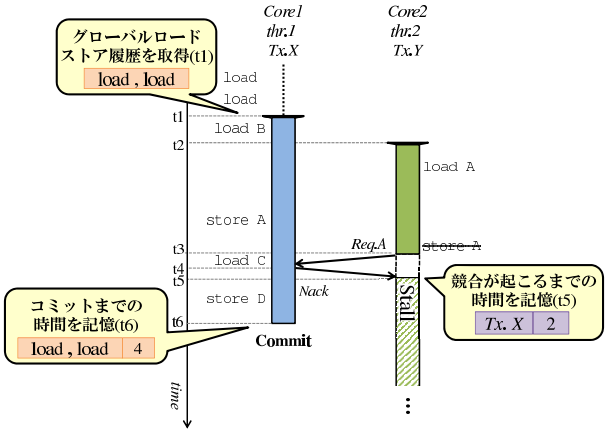


図 4　情報記憶時の動作

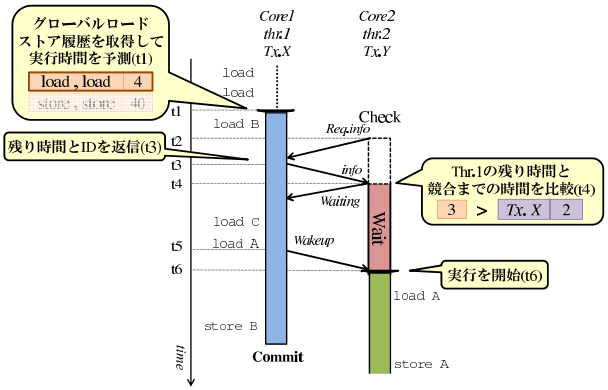


図 5　競合の発生を予測する場合の動作

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，再び同じプログラムが実行され，`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．これにより，トランザクションの実行時間を実行パスに関連づけた形で事前に予測できる．

4. 実装

本章では提案手法の具体的な動作モデルについて説明する．

4.1 情報記憶時の動作

本節では，提案手法である実行パスを考慮した競合予測に必要な情報を記憶する際の動作について，**図 4** 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同一の実行パスでも変動する可能性があるため，提案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る．

まず，*thr.1* が *Tx.X* の実行開始を試みる際に，グローバルロードストア履歴を取得する（t1）．いま，グローバルロードストア履歴として直近 2 回のメモリアクセス情報を用いるとし，この例では，*Tx.X* の開始前の直近 2 回のメモリアクセスがともに `load` であったとする．その後，*thr.2* が `store A` を試みたとすると（t3），*thr.1* は競合を検出し *thr.2* に *Nack* を送信する（t4）．*Nack* を受信した *thr.2* は，自身が *Tx.Y* の実行を開始してから競合を引き起こすまでにメモリにアクセスした回数を，競合相手のトランザクション ID と関連づけて記憶する．この例では，競合を引き起こした `store A` に，先に実行済みの `load A` を加えた 2 をメモリアクセス回数として，競合相手トランザクションの ID である X と併せて記憶する．そして，*Tx.Y* をストールさせる（t5）．一方，*thr.1* の実行が進み *Tx.X* のコミットに至ると（t6），*thr.1* は *Tx.X* の実行時間としてメモリアクセス回数を，実行開始前に取得したグローバルロードストア履歴と関連付けて記憶する．この例では，*thr.1* は *Tx.X* の実行中に `load B`，`load C`，`load A`，`store B` を実行しているため，メモリアクセス回数を 4 とする．そして，その値をグローバルロードストア履歴である load,load と関連づけて記憶する．以上で述べたように動作することで，実行パスを考慮した競合予測に必要となる情報を記憶する．

4.2 トランザクションの実行中の動作

本節では，各スレッドで記憶した情報をもとに競合の発生を予測する動作を説明する．まず，前節で述べた *thr.1*，*thr.2* がそれぞれトランザクション *Tx.X*，*Tx.Y* を並列に実行する**図 5** を例に，競合の発生を予測する動作について述べる．なお，前節で述べた手順を経て，*thr.1* および *thr.2* は競合予測に必要な情報を既に保持しているものとする．

まず，*thr.1* が *Tx.X* の実行開始を試みる際に，グローバルロードストア履歴を取得し，その履歴と関連付けて記憶した値から実行時間を予測する（t1）．この例では，*Tx.X* の開始直前にロードが 2 回実行されており，それに対応した値から実行時間を 4 と予測したとする．その後，*thr.2* が *Tx.Y* の実行開始を試みる際，競合予測を行うために，他のスレッドに対して，現在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（t2）．このリクエストを受信した *thr.1* は，先ほど予測した *Tx.X* の実行時間 4 から現在までのメモリアクセス回数 1 を減じることで，コミットまでの残り時間 3 を算出し，その値と自身のトランザクション ID である X を *thr.2* に返信する（t3）．*thr.1* のコミットまでの残り時間 3 と比べ

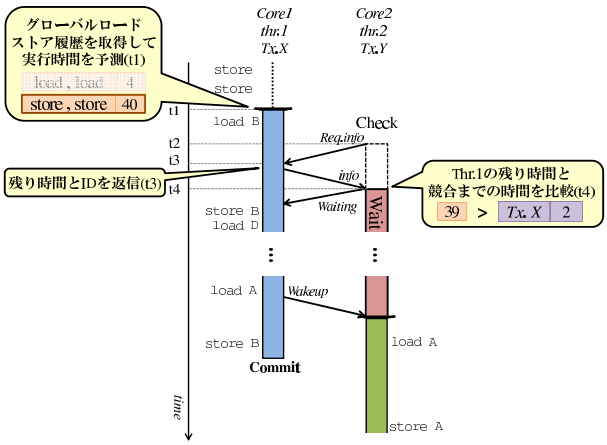


図 6　実行時間が変化する場合の動作

て，*thr.2* が *Tx.Y* の実行開始から *Tx.X* と競合を引き起こすまでの時間 2 の方が小さいため，*thr.2* は *Tx.Y* の実行を開始すると競合が発生すると予測し，実行を待機する．このとき，*thr.2* は *Waiting* メッセージと，競合を引き起こすまでの時間 2 を *thr.1* に送信する（t4）．その後 *thr.1* の実行が進み，*thr.2* が競合を引き起こすまでの時間より，*thr.1* のコミットまでの残り時間の方が短くなると，この時点で *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 *thr.2* に *Wakeup* メッセージを送信する（t5）．このメッセージを受信した *thr.2* は *Tx.Y* の実行を開始する（t6）．以上のように動作することで，トランザクション開始時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．

ここで，*thr.1* が実行する *Tx.X* において，分岐命令により図 5 の例とは異なるパスの命令が実行される場合に，*thr.2* がどのように競合予測を行うかについて**図 6** を用いて述べる．図 5 の例と同様に，まず，*thr.1* が *Tx.X* の実行開始を試みる際に，グローバルロードストア履歴を取得し，*Tx.X* の実行時間を予測する（t1）．この例では，図 5 の例とは異なり，*Tx.X* 開始直前にストアが 2 回実行されており，それに対応する値から *Tx.X* の実行時間を 40 と予測したとする．その後，*thr.2* が *Tx.Y* の実行を試みる際，競合予測に必要な情報を問い合わせるため *Req.info* を他のスレッドに送信する（t2）．リクエストを受信した *thr.1* は，予測した *Tx.X* の実行時間 40 から現在までのメモリアクセス回数 1 を減じることで，コミットまでの残り時間 39 を算出する．そして，*thr.1* は算出した残り時間 39 と，自身の実行するトランザクションの ID である *X* を *thr.2* に返信する（t3）．*thr.2* はこれらの情報に基づいて競合の発生を予測する（t4）．このとき，*thr.1* のコミットまでの残り時間 39 と比べて，*thr.2* が *Tx.Y* の実行を開始してから競合を引き起こすまでの時間 2 の方が短い．そのため，*thr.2* は *Tx.Y* の実行を開始すると競合が発生すると予測し，図 5 の例よりも長く *Tx.Y* の実行を待機する．このとき，トランザクション内の実行パスを考慮していなければ，

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

図 5 と同様に，*thr.1* のコミットまでの残り時間を 3 であると予測するため，*thr.2* は *Tx.Y* の実行を早期に開始し，競合が発生してしまう．しかし，提案手法では実行パスごとにトランザクションの実行時間を記憶しているため，図 6 に示す例のように，分岐命令によりトランザクション内の実行パスが変化したとしても競合予測を正確に行うことができる．以上のように動作することで，分岐命令により変化する実行パスに基づいて実行時間を予測し，競合を回避する．

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[12] 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[13] と GEMS 2.1.1[14] 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タ構成を示す．評価対象のプログラムとしては GEMS microbench，SPLASH-2[15]，および STAMP[16] から計 11 個を使用した．なお，各ベンチマークプログラムをそれぞれ 16 スレッドで実行した．

5.2 評価結果

評価結果を**図 7** および**表 2** 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 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**) 実行パスの変化を考慮せず，トランザクションの実

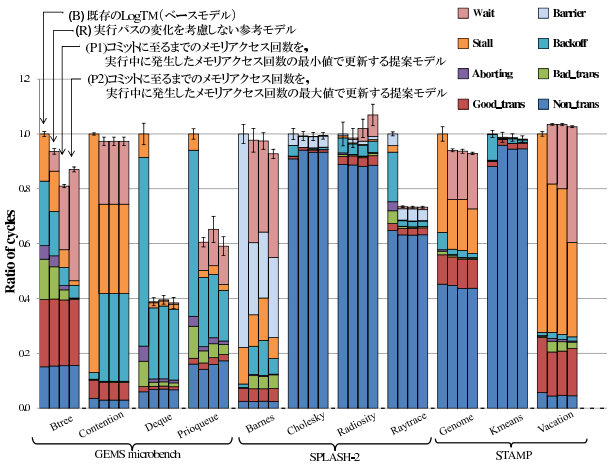


図 7　各プログラムにおける実行サイクル数比

行時間を，その実行中に発生したメモリアクセス回数の最小値とする参考モデル

(**P1**) コミットに至るまでのメモリアクセス回数を，実行中に発生したメモリアクセス回数の最小値で更新する提案モデル

(**P2**) コミットに至るまでのメモリアクセス回数を，実行中に発生したメモリアクセス回数の最大値で更新する提案モデル

の実行サイクル数を表しており，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．また，フルシステムシミュレータ上でマルチスレッドを用いた動作のシミュレーションを行うには，性能のばらつきを考慮する必要がある [17]．したがって，各評価対象につき試行を 10 回繰り返し，得られた結果から 95 %の信頼区間を求めた．信頼区間はグラフ中にエラーバーで示す．なお，トランザクションの実行開始時に取得したグローバルロードストア履歴と，トランザクション内の実行パスとの間に相関がない場合，グローバルロードストア履歴が同一であっても記憶されているコミットまでのメモリアクセス回数と実際にトランザクションの実行中に発生したメモリアクセス回数が異なる可能性がある．そこで，記憶するメモリアクセス回数を常に最小値で更新するモデル (P1)，最大値で更新するモデル (P2) の 2 つで評価した．提案手法では，(P1) の場合，コミットまでの残り時間が短く見積もられる場面が増え，無駄な待機時間を削減できると考えられる．一方 (P2) の場合，コミットまでの残り時間が長く見積もられる場面が増え，(P1) と比べて待機時間は増加するが，競合の発生回数をより削減できると考えられる．また参考モデル (R) においても同様に，最小値で更新する場合と最大値で更新する場合の 2 つの評価結果を比較したところ，多くのベンチマークプログラムで最小値で更新する場合の方が性能が向上していため，これを (P1) および (P2) の比較対象とする．なお今回の評価では，トランザクション開始直前までの直近の 8 回分のロードおよびストアをグローバルロードストア履歴として記憶することとした．

表 2　各プログラムにおけるサイクル削減率

		GEMS	SPLASH-2	STAMP	All
(R)	平均	27.4 %	7.7 %	1.2 %	13.1 %
	最大	61.1 %	26.5 %	6.0 %	61.1 %
(P1)	平均	29.2 %	7.0 %	1.4 %	13.6 %
	最大	60.5 %	26.7 %	6.3 %	60.5 %
(P2)	平均	29.5 %	6.9 %	2.1 %	13.8 %
	最大	61.6 %	26.7 %	7.1 %	61.6 %

図中の凡例はサイクル数の内訳を示しており，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，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．

評価の結果，Radiosity および Vacation をのぞく全てのベンチマークプログラムにおいて 2 つの提案モデル (P1) および (P2) で既存モデル (B) と比べて性能が向上しており，最大 61.6 %，平均 13.8 %の実行サイクル数削減を確認した．また，実行パスを考慮しない参考モデル (R) に対しても，全てのベンチマークプログラムにおいて 2 つの提案モデル (P1) および (P2) のいずれかの性能が向上している．このことから，競合予測において実行パスを考慮することの重要性が確認できた．

また，提案手法において実行パス予測に用いる，過去の実行パスと関連づけた実行時間情報について，これを記憶するのに必要な容量を概算したところ，1 コアあたり 1.6KByte となった．このことから，十分に小さい追加ハードウェアコストで提案手法が実現できることが確認できた．

5.3　考察

まず Prioqueue を見ると，提案モデル (P1) で参考モデル (R) に対して，Bad_trans，Aborting，Stall の増加により性能が低下している．このプログラムには，トランザクションの実行開始時に取得したグローバルロードストア履歴が同一でも実行時間が 10 倍以上変化するようなトランザクションが含まれており，提案モデル (P1) においてトランザクションの実行中に発生したメモリアクセス回数の最小値を用いると，競合が発生する場合でも，誤って競合が発生しないと予測してしまう場合があった．一方，提案モデル (P2) では参考モデル (R) に対して性能が向上している．提案モデル (P2) では最大値を用いるため，Wait は参考モデル (R) や提案モデル (P1) と比較して増加するが，競合の発生を抑制することで，Bad_trans，Aborting，Backoff，Stall を削減し，既存モデル (B) と比べて約 40 %の性能向

上が得られた．

一方 Btree では，2 つの提案モデル (P1) および (P2) で他のプログラムと比べて Wait の割合が大きいものの，Aborting，Stall，Backoff などのサイクル数を大幅に削減できている．このプログラムには 2 種類のトランザクション（それぞれを仮に *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，store 命令の両方が含まれており，発生する競合のほとんどは *Tx.Insert* 同士で発生していた．*Tx.Insert* は分岐命令などにより実行パスが変化することで競合の有無が変化するため，(P1) および (P2) で実行パスの変化に対応した競合予測を行うことで，(R) と比較して競合の発生を抑制できたと考えられる．しかし，(P2) では (P1) に対して性能が低下している．これは，(P2) では (P1) と比較して，無駄な待機時間が発生したことで Wait の割合が増大したためである．

また，Deque はトランザクション内に含まれる実行命令数が少なく，競合相手トランザクションが比較的早期にコミットされる．よって，トランザクションの実行を待機する時間が短いため，最大値を用いる提案モデル (P2) でも Wait が増大せず，競合の発生を抑制することができたと考えられる．

Barnes においては，(P1) および (P2) で (B) に対して実行サイクル数を削減できた．このプログラムでは主に Barrier が減少している．これは，提案手法の適用により競合の発生を抑制することで，各スレッドで発生するアボートの回数が減少し，実行を早く終えたスレッドが同期をとるために他のスレッドを待ち続ける時間が少なくなったためだと考えられる．また，(P1) および (P2) を比較すると，Wait の割合にあまり差がない．この原因を調査したところ，Barnes には最大で約 18 万回メモリアクセスするトランザクションが含まれていた．このトランザクションの実行時間が非常に長く，早期にトランザクションの実行を開始したとしても待機時間を抑制できる余地が少ないため，(P1) および (P2) において Wait の割合がほぼ同一となったと考えられる．

Cholesky と Kmeans では，(P1) および (P2) において Bad_trans，Aborting，Backoff，Stall を削減できている．しかし，実行サイクル数の大半を Non_trans が占めており，全体として大きな性能向上は得られなかった．また，Kmeans ではプログラムに含まれるトランザクションの実行時間は，実行パスによらずほぼ一定であり，(R) に対して 1 %以下の性能向上にとどまった．

Radiosity と Vacation では，Prioqueue 同様，取得した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トランザクションが含まれていたため，実行時間の予測を誤る場合が多く確認された．この原因を調査したところ，Vacation には，ランダムに生成された値に基づいて実行時間が決まるトランザクションが含まれていた．一方 Radiosity では，実行が繰り返される度に実行時間が短くなるトランザクションが含まれていた．このようなトランザクションに対しては，提案手法で採用したような，過去の情報に基づいた競合予測では対応できず，無駄な待機時間が発生したと考えられる．このような問題に対応するため，グローバルロードストア履歴の値が共通している場合でも実行時間が大きく変化することを観測した際には，提案手法による待機の適用を中止するなどの対策を今後検討する必要がある．

Contention，Raytrace，Genome では，(P1) および (P2) で，(B) と比べて Stall が減少したものの，(R) と比べると Wait の割合が大きく，1 %以下の性能向上にとどまった．これらのプログラムでは，実行パスによらず実行時間が一定であるトランザクションが，実行されたトランザクションの 8 割以上を占めていた．そのため，実行パスを考慮したスケジューリングを行っても，全体に対する効果が少なかったと考えられる．

6. おわりに

本稿では，既存の HTM を拡張して，分岐命令により変化する実行パスを考慮してトランザクションの実行時間を予測し，その予測結果を用いて競合を回避するスケジューリング手法を提案した．提案手法では広域分岐予測の考え方を利用し，トランザクション実行開始直前のグローバルロードストア履歴のパターンごとに実行時間を記憶しておくことで，実行パスの変化に対応した競合予測を行う．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 61.6 %，平均 13.8 %の実行サイクル数を削減できることを確認した．

しかし，今回のベンチマークプログラムではトランザクションの実行開始時に取得した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した．このようなトランザクションが実行されると，提案手法を用いたとしても競合予測に失敗し性能が低下することがあった．したがって今後，グローバルロードストア履歴として，ロードおよびストアの出現パターンだけでなく，他の情報の利用も検討する必要があると考えられる．例えば，ロードおよびストア対象アドレスも記憶し，これを活用してより詳細に実行パスを区別することで，競合を予測することがあげられる．

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 Struc-

tures, *Proc. 20th Annual Int'l Symp. on Computer Architecture (ISCA'93)*, pp. 289–300 (1993).

[2] 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).

[3] 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).

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

[5] 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).

[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] 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).

[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] Yeh, T.-Y. and Patt, Y. N.: Two-level adaptive training branch prediction, *Proc. 24th Annual IEEE/ACM Int'l Symp on Microarchitecture(MICRO'24)*, ACM, pp. 51–61 (1991).

[12] 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).

[13] 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).

[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] 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).

[16] 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).

[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).