Vol.2018-ARC-231 No.6
2018/6/14

トランザクショナルメモリにおける競合誤検出の影響調査とその改善手法

二間瀬 悠希[1]　林 昌樹[1]　多治見 知紀[1]　塩谷 亮太[2]　五島 正裕[3]　津邑 公暁[1]

概要：トランザクショナルメモリ（TM）は，クリティカルセクションを含む一連の命令列をトランザクションとして定義し，これを投機的に並列実行することで，粗粒度ロックと同程度の記述性と，細粒度ロックと同等以上の性能とを両立しうるパラダイムとして期待されている．しかし，TM では同一共有変数へのアクセスが頻発する場合に性能が低下してしまう．TM 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では一般的にキャッシュライン単位で共有変数に対するアクセス競合の検出を行うが，この検出方法では，複数のスレッドが異なる共有変数に対してアクセスを試みたとしても，これらの変数が同一キャッシュライン上に配置されている場合，誤って競合が検出される．本稿では，このような競合の誤検出がどの程度の頻度で発生するのかを調査し，誤検出が頻発するプログラムについて，原因となるデータ構造とそれに対する処理とを分析した．調査の結果，平均 27.4%，最大 99.9%の誤検出率を確認した．また，競合検出のためのハードウェアを追加し，キャッシュライン単位よりも細粒度に競合を検出できるようにすることで，誤検出を抑制する手法を提案し，評価を行った結果，平均 17.7%，最大 36.5%の性能向上を達成した．

1. はじめに

マルチコアプロセッサの普及に伴い，スレッドレベル並列性を活用した並列プログラミング環境の重要性が増大している．並列プログラミングでは，複数のプロセッサ・コア間で単一アドレス空間を共有する共有メモリ型並列プログラミングが一般的である．このプログラミング方式では，共有変数に対するアクセスを調停する必要があり，ロックによって排他制御する方法がしばしば用いられる．しかしロックを用いたプログラミングは，プログラムごとに適切なロックの粒度を設定することが困難であることや，デッドロックの発生などの問題があるため，プログラマにとって必ずしも利用しやすい仕組みではない．

そこで，ロックよりも容易に並列プログラムを記述可能なプログラミングパラダイムとして**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，データベースの更新・検索操作に用いられている**トランザクション（Transaction: Tx）** の概念をメモリアクセスに適用したものである．TM における Tx とは，従来ロックによって排他制御していたクリティカルセクションを含む一連の命令列であり，共有変数に対するアクセスが競合しない限りこの Tx を投機的に並列実行することで，ロックと比較して高い並列性を実現する．TM では，投機的な並列実行を実現するために，共有変数が更新される際に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，Tx を実行するスレッド間で同一共有変数に対するアクセスが競合しているか否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）**[2], [3] は，これらのための機構をハードウェアで実現することで，Tx 操作のためのオーバヘッドを抑制している．

HTM では一般に，キャッシュライン単位で競合を検査するため，複数のスレッドがそれぞれ異なる変数にアクセスしたとしても，それらの変数が同一キャッシュライン上に配置されている場合に，誤って競合と判断されてしまう．また，共有変数と非共有変数が同一キャッシュライン上に配置される場合もあり，Tx 外で非共有変数へアクセスする場合であっても競合としてみなされてしまうという問題がある．本稿では，HTM におけるこのような競合の誤検出が発生する頻度を調査した．また，誤検出が発生するベンチマークプログラムを分析し，誤検出を引き起こしやす

[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

いデータ構造とこれに対する一連の操作を明らかにする．その上で，変数の情報を保持するためのハードウェア拡張により細粒度に競合検出することで，このような競合の誤検出を抑制する手法を提案する．

2. トランザクショナルメモリとその問題

本章ではまず，トランザクショナルメモリの概要について述べる．次に，既存の HTM における競合検出の問題について述べる．

2.1 トランザクショナルメモリ

共有メモリ型並列プログラミングにおいて，TM はメモリアクセスが競合しない限り Tx を投機的に並列実行するため，Tx を粗粒度に定義したとしても並列性が損なわれることが少なく，記述性に優れる．さらに，プログラマはロックを使用するうえで考慮しなければならなかったデッドロックなどの問題を意識する必要がないため，容易に並列プログラムを設計することができる．TM における Tx は以下の 2 つの性質を満たす必要がある．

Serializability（直列化可能性）：
並列実行された Tx の実行結果は，当該 Tx を逐次実行した場合と同一であり，全てのスレッドにおいて同一の順序で実行されたように観測される．

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．また，各 Tx 内における処理の途中経過が他のスレッドから観測されることはなく，Tx の終了と同時に処理結果が観測される．

これらの性質を保証するために，Tx 中のメモリアクセスを監視する．複数の Tx 内で同一アドレスに対するアクセスが確認され，そのアクセスによって上記の Tx の性質を満たさなくなる場合に，この状態を**競合（Conflict）**として検出する．TM では競合が検出された場合，いずれかのスレッドが Tx の途中までの実行結果を破棄する．この操作を**アボート（Abort）**という．自身が実行する Tx をアボートしたスレッドは，バージョン管理の機構によって保持していた Tx 開始前のメモリおよびレジスタの状態を復元し，Tx を再実行する．

これら競合検出とバージョン管理のための機構はハードウェアもしくはソフトウェアによって実装される．このうちハードウェアによって実装した HTM は，これらの機構によるオーバヘッドが小さい．一方，ソフトウェアによって実装したソフトウェアトランザクショナルメモリ [4] は，ハードウェア拡張を必要としないが，ソフトウェアによる競合検出とバージョン管理のオーバヘッドが大きい．

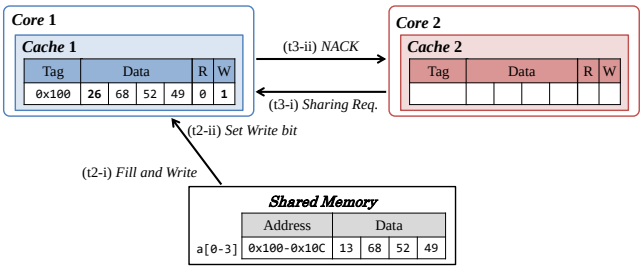

図 1 異なる変数へのアクセスによる誤検出

時刻	*Thread*1	*Thread*2
t1	BEGIN_TRANSACTION;	BEGIN_TRANSACTION;
t2	a[0]=26;	
t3		tmp=a[2];
⋮	⋮	⋮
	COMMIT_TRANSACTION;	COMMIT_TRANSACTION;

図 2 プログラム例とその実行スケジュール

2.2 HTM における競合検出方式の問題

一般的な HTM では，競合検出のために各キャッシュラインに **Read/Write ビット**と呼ばれるフィールドを追加している．そして，これらのビットがセットされているか否かを検査することで，競合検出を行う．そのため，同一キャッシュライン上に存在する異なる変数に複数のスレッドがアクセスする場合に，誤って競合として判断してしまうという問題がある．

この問題について，**図 1** および**図 2** のように，2 つのスレッドが Tx 中に同一キャッシュライン上の異なるメモリアドレスにアクセスする場合を例に説明する．図 1 中の *Core* と *Cache* はそれぞれプロセッサ・コアと各コアが持つローカルキャッシュを簡略化したものを表しており，*Shared Memory* は 2 つのコアが共有している主記憶を表している．なお，各コアと共有メモリとを繋ぐ通信路は省略する．また，*Core*1 および *Core*2 にはそれぞれ *Thread*1 および *Thread*2 が割り当てられており，各スレッドでは図 2 に示すようなプログラムが実行されているとする．なお，図 2 中の BEGIN_TRANSACTION および COMMIT_TRANSACTION はそれぞれ Tx の開始および終了を表している．また，プログラム内の各式が実行される時刻を図中左に示している．まず，2 つのスレッドが Tx の実行を開始し（t1），*Thread*1 が a[0] へ値を代入する（t2）．このとき，*Cache*1 には a[0] がキャッシュされていないため，共有メモリの 0x100 番地から始まるデータがキャッシュされた後，a[0] の値が更新される（図 1 t2-i）．なお，このとき当該キャッシュラインの更新前のデータおよびそのアドレスである 0x100 がバックアップ領域に退避されるが，図中では省略している．ここで，Tx 内で Write アクセスするため，0x100 番地に対応するキャッシュラインの Write ビットがセットされる（図 1 t2-ii）．

次に，*Thread*2 が a[0] と同一のキャッシュライン上に存在する a[2] の読み出しを試みる（t3）．このとき，*Cache*2

は a[2] をキャッシュしていないため，一貫性プロトコルに従って *Cache*1 と 0x100 番地のラインを共有するためのリクエストが送信される（図 1 t3-i）．リクエストを受信した *Thread*1 は 0x100 番地に対応するラインの Read/Write ビットを参照する．このとき，Write ビットがセットされているため競合が検出され，*Thread*1 は *Thread*2 に対して *NACK* を返信する（図 1 t3-ii）．

このように，既存の HTM では各キャッシュラインに付与された Read/Write ビットによって競合を検査するため，複数のスレッドがそれぞれ異なる変数にアクセスしたとしても，それらが同一キャッシュライン上に存在している場合，誤って競合を検出してしまう．

3. 競合誤検出に関する調査

競合が誤検出されることにより，本来は必要ではない Tx のアボートや再実行に係るオーバヘッドが発生し，性能が低下する．本章では，競合の誤検出頻度と誤検出が性能に及ぼす影響とを調査した結果を示し，考察する．なお，性能に及ぼす影響に関しては，誤検出が発生しない調査モデルを作成し，このモデルと既存の HTM とで速度性能の比較調査を行った．

3.1 評価環境

以上の内容を HTM の研究で広く用いられている LogTM[5] 上に実装し，シミュレーションにより評価した．シミュレータは Simics [6] 3.0.31 と GEMS [7] 2.1.14 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．詳細なシミュレーション環境を**表 1** に示す．なお，LogTM では競合検出のための機構を実現するために，ディレクトリベースのキャッシュコヒーレンスプロトコル [8] である Illinois プロトコル [9] を拡張している．評価対象のプログラムは，GEMS microbench[7]，SPLASH-2[10]，および STAMP[11] から計 8 個を使用した．**表 2** に各ベンチマークプログラムの入力パラメタを示す．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．

表 1 シミュレータ構成

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

表 2 ベンチマークパラメタ

GEMS microbench	
Btree	priv-alloc-20pct
Contention	config 1
Deque	4096ops 128bkoff
Prioqueue	8192ops
Slist	1024ops 64len
SPLASH2	
Cholesky	tk14.0
Radiosity	-p 31
STAMP	
Kmeans	-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt
Vacation	-q10 -u80 -r65536 -t4096

3.2 評価結果および考察

評価結果を**図 3** および**表 3** に示す．図 3 では，各ベンチマークプログラムの評価結果をそれぞれ 2 本のバーで表しており，左から，

(B) 既存の LogTM（ベースライン）

(S) (B) に変数分の Read/Write ビットを追加した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．ここで，(S) は本調査のために用意したモデルであり，各キャッシュラインに対して，内包される変数の数だけ Read/Write ビットを追加している．そして，これらのビットを用いて変数単位で競合検出を行うことにより，誤検出が発生しない理想的な状況をシミュレートする．

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，表 3 は (B) において各プログラムで発生した競合誤検出率を示している．なお，すべてのスレッドにおける *NACK* 送信回数のうち，それが競合の誤検出に起因するものであった回数の割合として誤検出率を定義し，算出した．評価の結果，最大 99.9%，平均 27.4%の誤検出率を確認した．また，競合誤検出が発生しない理想的なモデルである (S) では，(B) と比較して最大 87.9%，平均 24.7%，実行サイクル数が低減することが分かった．

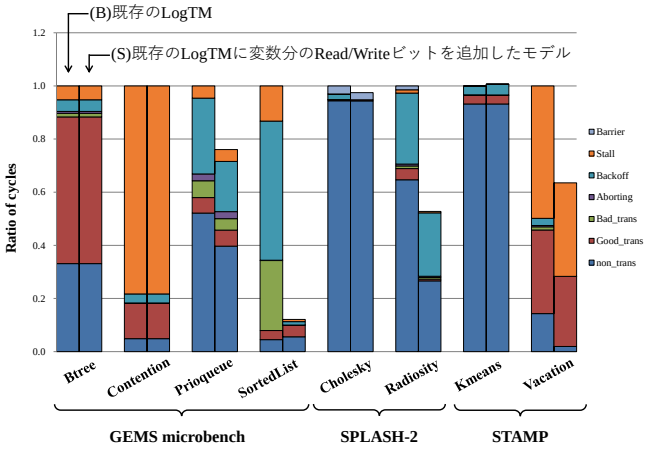

図 3 評価結果

表 3 誤検出率

ベンチマーク	Btree	Contention	Prioqueue	SortedList
誤検出率	0.6%	0.0%	45.8%	99.9%

ベンチマーク	Cholesky	Radiosity	Kmeans	Vacation
誤検出率	3.7%	66.5%	0.3%	29.8%

3.3 考察

Prioqueue および SortedList が特に高い誤検出率を示しており，(S) の実行サイクル数も (B) と比較して大幅に低い．これらのプログラムでは，グローバル変数として定義された構造体内の，複数のメンバ変数に頻繁にアクセスするトランザクションが含まれている．隣接したメモリ空間に配置されたこれらの変数へアクセスを試みる際に，競合の誤検出が発生していた．このような場合は構造体内の変数間にパディングを適切に施すことによって，誤検出を回避することができるが，キャッシュ効率の低下による性能悪化という弊害が起こりうる．

また，これらとは異なる特徴を持つプログラムに，Vacation がある．Vacation は，Stall や Abort に要するサイクル数だけでなく，Non_trans の実行サイクル数も (B) と (S) とで大きく異なっていることが分かる．Vacation は，旅行者のオンライン予約システムをエミュレートするプログラムである．この Vacation 内に含まれる 3 種類の Tx のうち，総実行回数の約 98%を占める Tx とその前後のコードを**図 4** に示す．

この Tx は，データベースを検索して，空きがあれば予約を取り，データベースを更新する．データベースの各予約情報は**図 5** に示す構造体として保持される．シミュレータ上で，long int 型は 4 バイトであるため，このデータ構造のデータサイズは 16 バイトである．この構造体の多くは，プログラム開始時に連続したメモリ領域に確保されるため，キャッシュラインサイズが 64 バイトとすると，最大で 4 つのデータが同一キャッシュライン上に配置される

```
1 for (i = 0; i < numOperation; i++) {
2   action = /* 乱数生成 */
3   switch (action) {
4     case MAKE_RESERVATION: {
5       BEGIN_TRANSACTION(0);
6       for(n = 0; n < numQuery; n++){
7         types[n] = /* 乱数生成 */
8       }
9       for (n = 0; n < numQuery; n++) {
10        switch (types[n]) {
11          case RESERVATION_CAR:
12            /* レンタカーの予約価格の決定 */
13            break;
14          case RESERVATION_FLIGHT:
15            /* 飛行機便の予約価格の決定 */
16            break;
17          case RESERVATION_ROOM:
18            /* ホテルの予約価格の決定 */
19            break;
20        }
21      }
22      ... /*いずれかの予約*/
23      COMMIT_TRANSACTION(0);
24      break;
25    }
26    ... /* その他の case */
27  }
28 }
```

図 4 Vacation 内の Tx

```
1 typedef struct reservation {
2   long id;
3   long numUsed;
4   long numFree;
5   long price;
6 } reservation_t;
```

図 5 予約情報を表す構造体

```
1 #define N 624
2
3 typedef struct random {
4   unsigned long (*rand) (unsigned long*, unsigned
      long*);
5   unsigned long mt[N];
6   unsigned long mti;
7 } random_t;
```

図 6 乱数生成器を表す構造体

可能性がある．これにより，異なるデータに対して検索または更新をしたとしても，これらが同一キャッシュライン上に配置された場合に競合が誤検出される．

さらに Vacation では，このような Tx 間の誤検出だけでなく，Tx 内でのアクセスと Tx 外でのアクセスとが競合と誤って検出される場合もあることが観測できた．詳細について調査したところ，トランザクション内外で行っている乱数生成が原因でこの誤検出が発生していた．Vacation

では，スレッドごとに図 **6** に示すような乱数生成器を確保しており，乱数生成時にこの乱数生成器に対する読み出しと書き込みが発生する．この構造体はプログラム開始時にスレッド数分だけまとめて生成されており，これにより隣接しているこれらの構造体が同一キャッシュライン上に配置される場合がある．そして，同一キャッシュライン上の異なる構造体にアクセスしようと試みた際に競合が誤検出されていた．

この問題は，スレッドを開始する前にスレッド固有のデータを複数まとめて動的に生成していることに原因がある．そのようなコーディングスタイルは一般的であるため，このような競合の誤検出も一般に多く発生しうると考えられる．これが性能に与える影響が大きいことも確認できたことから，既存の HTM では，プログラマに注意深いコーディングが求められることになる．上述したように，データ構造にパディングを挿入することで誤検出を回避することも可能ではあるが，キャッシュ効率の低下につながる上，複雑なデータ構造の場合は適切にパディングを施すことも容易ではない．

そこで，本稿では HTM の競合検出方法を改変することで，誤検出を抑制する手法を提案する．

4. 細粒度競合検出手法の提案

本章では，キャッシュラインよりも細粒度な単位で競合を検出することで，競合誤検出を抑制する手法を提案する．

4.1 細粒度競合検出のためのハードウェア拡張

3 章で用いた調査用モデルでは，各キャッシュラインに対して，そこに内包される変数の数分の Read/Write ビットを用意した．この実装では，これらのビットを利用して細粒度に競合を検査することにより，誤検出を解消できることが分かった．しかし，この拡張は全てのキャッシュラインに対して行われるため，ハードウェアコストが高い．加えて，実際に競合が発生するキャッシュラインは限定されると考えられるため，全てのキャッシュラインに対してビットを追加する方法には無駄が大きい．

そこで，提案手法では，細粒度に検査すべきキャッシュラインに限定して，そこに内包される変数のアクセス情報のみを記録することとする．ただし，細粒度に検査すべきラインをプログラムの実行前に知ることは困難であるため，過去のトランザクション実行中に競合したキャッシュラインを，対象のラインとする．このためにまず，各キャッシュラインでは，過去に当該ラインにおいて競合が発生したか否かを記録する．その上で，過去に競合が発生したラインに関しては，別途 Tx のアクセス先アドレスを細粒度に記憶し，競合判定に用いることで，誤検出を抑制する．

なお，アクセス先アドレスの記憶には，**Bloom フィルタ** [12] を用いる．Bloom フィルタは，ある要素が集合に登録されているか否かをその集合を検索することなく判定可能なフィルタである．このデータ構造は固定長のビット列と一定数のハッシュ関数から構成される．値の追加は，追加したい値からハッシュ関数を通じてハッシュ値を生成し，ビット列の対応する位置のビットをセットする．検索時は，追加時と同様にハッシュ値を生成し，セットされているか否かを判定する．Bloom フィルタには，ハッシュ値の衝突に起因する判定の偽陽性が存在するが，空間効率が良く，ハードウェアコストを低く抑えることが可能である．

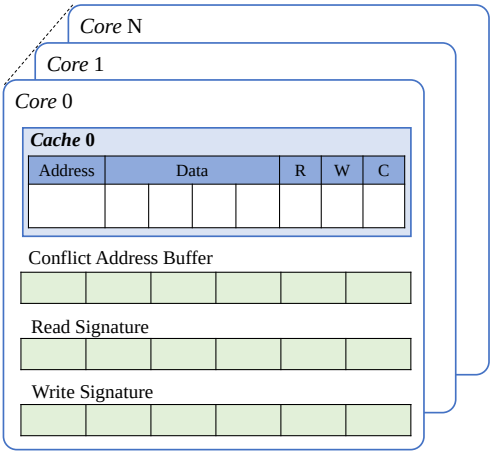

図 **7** 拡張したハードウェア

4.2 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 **7** に示す 3 つのハードウェアを各コアに追加する．追加したハードウェアに関する詳細を以下に示す．

Conflict Bit（C ビット）
各キャッシュラインごとに追加する 1 ビットのフラグで，当該キャッシュラインで過去に競合したか否かを保持する．

Conflict Address Buffer（C-Buffer）
Tx 中に競合したキャッシュラインのアドレス一覧を一時的に保持するバッファ．

Read/Write Signature（R/W-Signature）
変数のアドレスをハッシュ値とした Bloom フィルタ．

4.3 動作

4.3.1 初回競合時の動作

前節で示した追加ハードウェアに情報を登録する流れを図 **8** を用いて説明する．アクセスリクエストを受け取った *Thread*0 は (t1)，リクエストされたアドレスの Read/Write ビットを確認する（図 8 t1-i）．この例では Read ビットがセットされているため，このキャッシュラインに対するアクセスを競合として判定する．ここで当該ラインの C ビットを検査し，過去に競合が発生したか否かを確認する（図 8 t1-ii）．C ビットはセットされていないため，過

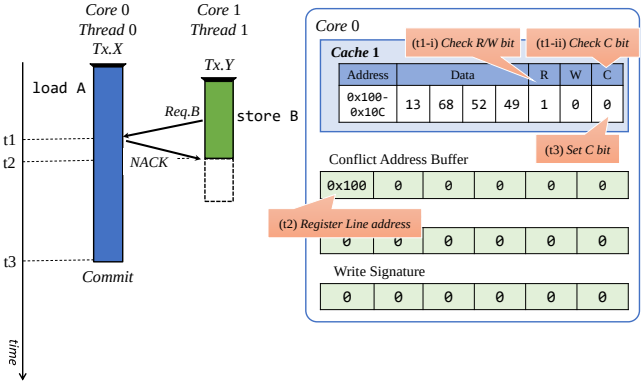

図 8　初回競合時の動作と追加ハードウェアの様子

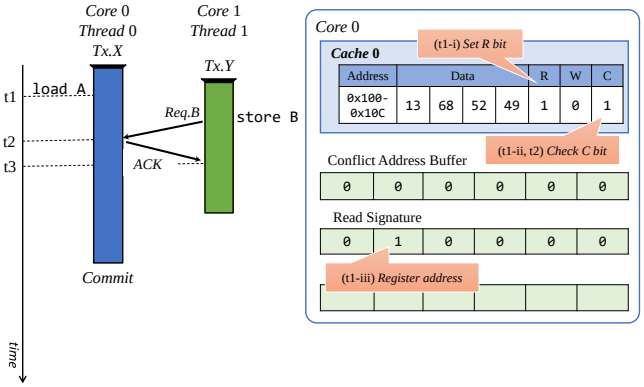

図 9　細粒度競合検出時の動作と追加ハードウェアの様子

去に競合したことがないキャッシュラインであると判断する．このような場合は，従来通りこのアクセスを競合と判定し，*Thread*1 へ *NACK* を送信する（t2）．このとき，当該ラインアドレスを C-Buffer へ登録し，競合したことを記録する（図 8 t2）．その後，*Thread*0 がコミットする際に，C-Buffer に登録されているキャッシュラインの C ビットをセットし，C-Buffer をクリアする（t3）．この操作により，以降のトランザクション実行時に，過去このキャッシュラインで競合が発生したことを知ることができる．

4.3.2　細粒度競合検出の動作

次に，再び *Thread*0 と *Thread*1 がそれぞれ同じ Tx を実行する場合を例に，細粒度に競合検出する流れを**図 9** を用いて説明する．t0 で *Thread*0 が A に load アクセスする際，まずアクセスリクエストを *Thread*1 に送信し，*ACK* を受け取る．その後，Read ビットをセットし（図 9 t1-i），次に C ビットがセットされているか否かを確認する（図 9 t1-ii）．前回の実行により，C ビットがセットされており，これにより過去のトランザクション実行中にこのラインで競合が発生したことが分かる．ここで，今後このラインを細粒度に競合検出するために，R/W-Signature に A のアドレスを登録する（図 9 t1-iii）．

次に，*Thread*1 から B のアクセスリクエストを受け取った *Thread*0 は，C ビットがセットされていることにより細粒度に競合検出するべきであると判断する（図 9 t2）．

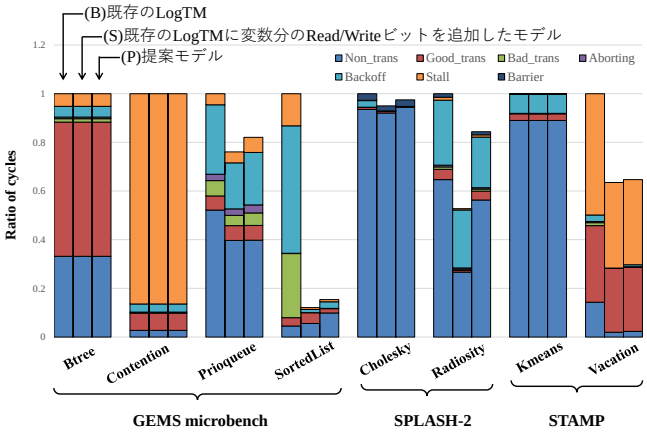

図 10　各プログラムにおけるサイクル数比

そして，R/W-Signature に B のアドレスが登録されているか否かを検索する．*Core*0 の R/W-Signature には B のアドレスは登録されていないため，B に対するアクセスは許可され，*Thread*0 は *Thread*1 に *ACK* を送信する（t3）．このように，細粒度に検査することで，キャッシュライン内の異なる変数に対するアクセスを許可することが可能となる．

なお，このように同一キャッシュラインの別変数に対してアクセスが許可された場合，それぞれで行われた更新がともに適切に反映されることを保証しなければならないが，これはキャッシュコヒーレンスプロトコルの枠組みで解決される．今回実装のベースとしている LogTM[5] では，MESI ベースのコヒーレンスプロトコルを採用しているが，他コア上で更新済みのキャッシュラインに対し別コアによる Write アクセスが許可されると，更新済みの値を含んだ当該ラインが共有されるとともにメモリに書き戻される．後者のコアでは，これに対し Write アクセスすることで，両更新を含んだキャッシュラインが保持される．一方，先に更新を行っていたコア上の当該ラインは invalidate されるため，一方のみの更新を含んだキャッシュラインが存在してしまうことはない．

5.　評価

本章では，前章で提案した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　評価結果

前章で述べた提案手法を LogTM 上に実装し，シミュレーションにより評価した．シミュレーション環境および使用したベンチマークプログラムは，3 章の調査で用いたものと同じである．**図 10** は，各ベンチマークのプログラムの評価結果を表している．図中の 3 本のバーは左から順に，

(B)　既存の LogTM（ベースライン）

(S)　(B) に変数分の Read/Write ビットを追加した参考モデル

(P) 提案モデル
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．図中の凡例は 3.2 節の図 3 と同様であり，(B) および (S) の結果については，図 3 の再掲である．

評価結果より，(P) は (B) に対して最大 36.5%，平均 17.7%の性能向上を達成した．

5.2 考察

3 章の調査で競合の誤検出が確認されたプログラムに対しては，提案手法により実行サイクル数を削減することができた．また，(P) を 3 章で用いた調査モデル (S) と比較すると，実装コストを低く抑えているにもかかわらず，Kmeans や Vacation では，誤検出が全く起こらない理想的なモデル (S) に匹敵する性能を達成できている．

一方 SortedList では，提案手法は (B) に対して大きく性能を向上できているものの，理想モデル (S) ほどの性能は得られていない．これは，(P) で誤検出を十分抑制できず，競合に起因するオーバヘッドが削減できなかったからであると考えられる．誤検出を抑制できなかった原因の一つとして，Bloom フィルタの偽陽性が考えられる．提案手法では Bloom フィルタを，アクセスした変数情報の保持のために用いている．これに対する検索時に，偽陽性により本来登録されていない，すなわちアクセスしていない変数であってもアクセスしたと判断される場合がある．提案手法で用いた Bloom フィルタでは，ハッシュ値生成に係るオーバヘッドを小さくするために，シフトと排他的論理和を用いた比較的単純なハッシュ関数を 1 つ使用している．このハッシュ関数では偽陽性率が十分に小さくならなかったため，結果として細粒度競合検出の誤検出率に影響を与えた可能性が考えられる．Bloom フィルタの偽陽性の発生確率は，ハッシュ値が一様に分布している場合に，ハッシュ関数の個数を増加させることで大幅に削減できることが知られている [13]．メモリアドレスから一様なハッシュ値が生成できるのであれば，ハッシュ関数を増加させることでより高い速度性能が達成できると考えられる．

6. おわりに

本稿では，HTM の性能に悪影響を及ぼす競合の誤検出に関して調査した．調査の結果，誤検出が頻発するプログラムが複数存在することが分かった．また，誤検出の中にはトランザクション間の競合だけでなく，トランザクション内のメモリアクセスとトランザクション外のメモリアクセスとが競合したと誤って判断される場合があることが分かった．これに関してプログラムを分析した結果，単一のスレッドからしかアクセスされない変数同士が隣接したメモリ領域に配置されることで発生する場合があることが分かった．調査結果を踏まえ，細粒度競合検出手法を提案し，評価を行ったところ，平均して 17.7%，最大 36.5%の性能向上を達成できることを確認した．今後の課題として，複数個のハッシュ関数を用意することによりブルームフィルタの偽陽性率を低下させることなどが挙げられる．

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] Knight, T.: An Architecture for Mostly Functional Languages, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 105–112 (1986).
[3] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C. and Olukotun, K.: Transactional Memory Coherence and Consistency, *Proc. 31st Annual Int'l Symp. Computer Architecture (ISCA'04)*, pp. 102– (2004).
[4] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995).
[5] 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).
[6] 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).
[7] 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).
[8] Sweazey, P. and Smith, A. J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. 13th Annual Int'l. Symp. on Computer Architecture (ISCA'86)*, pp. 414–423 (1986).
[9] Censier, L. M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Trans. on Computers*, Vol. C-27, No. 12, pp. 1112–1118 (1978).
[10] 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).
[11] 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).
[12] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors, *Commun. ACM*, Vol. 13, No. 7, pp. 422–426 (online), DOI: 10.1145/362686.362692 (1970).
[13] 倉田成己，塩谷亮太，五島正裕，坂井修一：ブルーム・フィルタを用いたメモリ・アクセス順序違反検出，情処研報，Vol. 2014-ARC-212, No. 17, 情報処理学会，pp. 1–15 (2014).