Vol.2013-ARC-204 No.5
2013/3/26

# ハードウェアトランザクショナルメモリにおける競合パターンに応じた競合再発抑制手法の適用

鈴木 大輝[1]　江藤 正通[1]　橋本 高志良[1]　堀場 匠一朗[1]　津邑 公暁[1,a)]　松尾 啓志[1]

**概要**：マルチコア環境における並列プログラミングでは，共有リソースへのアクセス制御にロックが広く利用されてきたが，その場合，デッドロックの発生や並列性の低下などの問題がある．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・メモリが提案されている．これをハードウェアで実現する HTM では，一般的にフラグを用いることで競合解決を実現する．しかしその方法ではスレッドスケジューリングの効率が悪く，トランザクションのアボートやストールが頻発する場合がある．本稿ではその代表的な例を 2 つ取り上げ，各問題の解決手法をそれぞれ提案する．提案手法の有効性を検証するためにシミュレーションによる評価を行った結果，2 つの手法を組み合わせたモデルは既存の LogTM に比べて最大で 72.2%，平均で 28.4%の性能向上が得られることを確認した．

## 1. はじめに

マルチコア環境において一般的な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に対する排他制御機構として一般にロックが用いられてきた．しかしロックを用いた場合，ロック操作のオーバヘッド増大に伴う並列性の低下や，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難しく，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 では，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することで，共有変数へのアクセスにおいて競合が発生しない限り，投機的に実行を進めることができ，ロックを用いる場合よりも並列性が向上する．なお，トランザクションの実行中においては，その実行が投機的であるがゆえ，共有変数に対する更新の際に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）．トランザクショナル・メモリのハードウェア実装であるハードウェア・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，上述のとおり HTM では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，トランザクションは並列に実行される．その実行の際に競合が発生した場合，一般的な HTM では，フラグを用いることでアボート対象のトランザクションが選択されるが，その対象はトランザクション間で発生した競合のパターンとは関係なく決定される．そのため，ある種の競合パターンが発生した場合に性能が低下してしまう可能性がある．なお HTM には，アボート後に待機時間を設ける exponential backoff や，その待機時間を競合相手のトランザクションがコミットされるまでとする magic waiting という競合抑制技術が存在する．しかし，HTM の性能に悪影響を及ぼす競合パターンが発生した際にこれらを用いるだけでは，競合解決までに多くの時間がかかってしまう．そこで本稿では，その競合パターンに着目し，一部のトランザクションをあえて逐次的に実行することで競合を抑制する手法を提案する．

## 2. Futile Stall 防止手法

本章では，futile stall の発生という問題について述べ，これを防止する手法の提案と実装方法について述べる．

### 2.1 Futile Stall の発生

競合を頻繁に引き起こす複数のトランザクションが並列に実行されるとき，開始時刻の遅いトランザクションに

[1] 名古屋工業大学
Nagoya Institute of Technology, Nagoya, Aichi, 466-8555, Japan
[a)] tsumura@nitech.ac.jp

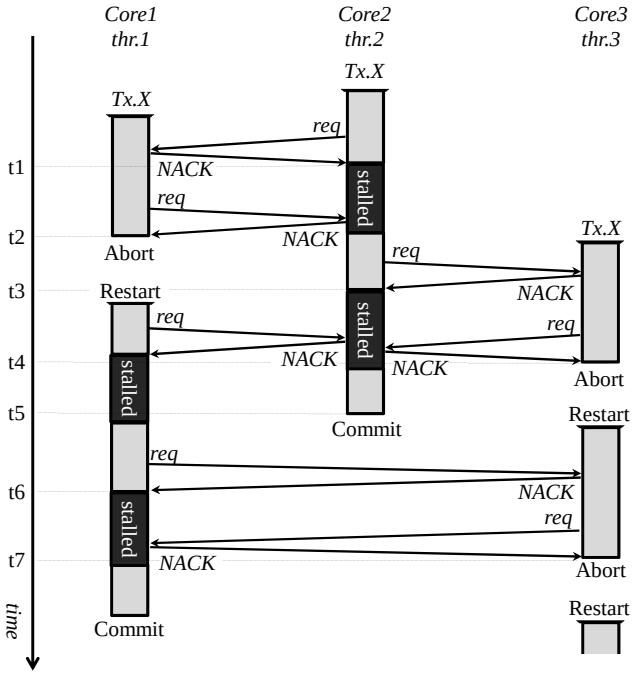


図 1　Futile Stall の発生

よって開始時刻の早いトランザクションがストールさせられる状況が頻繁に発生する．この競合の後，これらのトランザクション間で再度競合が発生した場合，開始時刻の遅いトランザクションがアボートされる．この場合，結果的には開始時刻の早いトランザクションをストールさせる必要がなかったことになる．このようなストールを futile stall という．

ここで**図 1** のように，3 つのスレッド（*thr.1*〜*3*）が，それぞれ同じトランザクション *Tx.X* を実行する例を用いて futile stall を説明する．なお，futile stall はロード命令とストア命令に依存せず発生するため，命令の表記を省略する．まず，*thr.1* によって既にアクセスされたアドレスに *thr.2* がアクセスしようとした場合，*thr.2* が実行する *Tx.X* はストールする（t1）．次に，*thr.2* が既にアクセス済みのアドレスに *thr.1* がアクセスしようとした場合，*thr.1* の実行する *Tx.X* の方がその開始時刻が遅いため，このトランザクションがアボートされる（t2）．このとき，*thr.1* が実行していたトランザクションのアボートにより資源が開放されたため，*thr.2* は *Tx.X* の実行を再開する．しかし，*thr.2* はトランザクションをストールさせている間，*thr.1* の実行するトランザクションがコミットされるのを待っていたにも関わらず，このトランザクションはアボートしてしまったため，結果的には *thr.2* の実行するトランザクションをストールさせる必要がなかったこととなる．このようなストールを futile stall という．

この後，*thr.2* が *thr.3* と競合（t3 および t4）した場合，さきほどと同様に *thr.2* の実行する開始時刻の早い *Tx.X* のストールは futile stall となり，その競合相手である *thr.3* の実行する *Tx.X* はアボートされる（t4）．続いて，*thr.2* が *Tx.X* をコミットした場合（t5），競合相手のトランザクションである *thr.1* および *thr.3* の *Tx.X* 同士で競合が発生し，*thr.1* の *Tx.X* はストールし（t6），*thr.3* の *Tx.X* はアボートされてしまう（t7）．

このように，競合を頻繁に引き起こす可能性のある複数のトランザクションが並列に実行される場合は futile stall が多く発生する．また，*thr.3* の実行するトランザクションのように，アボートを繰り返してしまうトランザクションも発生する．なお，並列実行されるトランザクションの増大にともない，競合が発生する可能性がさらに増える．その結果としてアボートや futile stall の発生も増えるため，性能が大きく低下すると考えられる．

### 2.2　Futile Stall の防止

前節で述べた futile stall が多く発生する場合，並列実行されるトランザクション間で競合が頻繁に発生していると考えられる．その場合，図 1 の *thr.3* の実行するトランザクションのようにアボートの繰り返しが発生する．そこで本節では，アボートを一定回数繰り返したトランザクションを競合が頻発するトランザクションと判断し，そのトランザクションを逐次的に実行する手法を提案する．

これにより，本来そのトランザクションの実行によって発生するアボートの頻発を防ぐことができ，futile stall も抑制できると考えられる．しかし，実行命令数の多いトランザクションを逐次的に実行した場合には，逐次実行の時間が長くなり，性能が悪化する可能性がある．したがって，ある程度実行命令数の少ないトランザクションのみを逐次実行の対象とするよう考慮しなければならない．これらを踏まえ，あるトランザクションが以下に示す 2 つの条件を満たした場合，そのトランザクションを逐次実行の対象とする．なお，これらの条件で示されている *A-tx* および *L-inst* はそれぞれトランザクションのアボート回数およびトランザクションの実行命令数の閾値を表している．

- 条件 F-I：トランザクションをコミットするまでに，閾値 *A-tx* 以上アボートを繰り返す
- 条件 F-II：トランザクションをコミットするとき，そのトランザクション内で実行された命令数が閾値 *L-inst* 以下である

これらの条件を満たすトランザクションが複数存在する場合，それらのトランザクションを逐次的に実行する．ただし，トランザクションの実行中に OS による割り込み処理が発生した場合，トランザクションの開始からコミットまでに実行された命令の数はトランザクション内の実行命令数より多くなってしまう．そのため，あるトランザクションの開始からコミットまでに実行された命令数が閾値 *L-inst* より多い場合があったとしても，そのトランザクションを逐次的に実行することで性能向上が得られる可能

性がある．

そこで，割り込み処理が原因で条件 F-II を満たさなかったトランザクションを救済するために，閾値 *L-inst* より値の小さい閾値 *S-inst* を設ける．たとえば，あるトランザクションが実行され，そのトランザクションは条件 F-II を満たさなかったとする．その後，同じトランザクションが再び実行され，その実行命令数が今度は *S-inst* よりも少なかった場合，そのトランザクションの実行命令数が多いと判定されたのは OS による割り込み処理が原因であると判断し，そのトランザクションは条件 F-II を満たしたと判定する．

### 2.3 追加ハードウェアと動作モデル

前節で述べた動作を実現するために，まず逐次実行の対象トランザクションを決める必要がある．そして，その対象となったトランザクションの実行順序を制御しなければならない．本節では，これらの処理を実現するために追加したハードウェアおよびその動作モデルについて述べる．

#### 2.3.1 逐次実行対象トランザクションの決定

逐次実行の対象トランザクションを決めるために，以下に示す 5 つの機構を各コアに追加する．なお，HTM では，トランザクション毎に ID が割り振られており，この ID は個々のトランザクションを識別するために用いられる．下記に示す R-flags，Stx-flags および Ltx-flags はその ID をインデクスとしてトランザクション毎に管理される．

**Abort Counter（A-Counter）**： トランザクションの実行を開始してからコミットするまでの間に発生したアボートの回数をカウントする．このカウントされた値は，トランザクションのコミット時にリセットされる．

**Recurrence flags （R-flags）**： トランザクションの ID をインデクスとし，そのインデクスに対応するトランザクションがアボートを繰り返したかどうかを記憶する．前述した A-Counter の値が閾値 *A-tx* 以上になった場合，実行中のトランザクションの ID に対応するビットがセットされる．なお，コミット後に再び同一のトランザクションが実行される可能性があるため，一度セットされたフラグはプログラムが終了するまでクリアされない．

**Instruction Counter（I-Counter）**： トランザクション内で実行された命令数をカウントする．制御フローの変化により，同一のトランザクションでも実行のたびにその命令数が変化する可能性がある．したがって，カウントされた値はアボートおよびコミット時にリセットされ，トランザクションの実行毎に命令数がカウントされる．

**Short Tx flags（Stx-flags）**： トランザクションの ID をインデクスとし，そのインデクスに対応するトランザクションの実行命令数が少ないかどうかを記憶する．前述した I-Counter の値が，閾値 *L-inst* 以下の状態でコミットが発生した場合，実行中のトランザクションの ID に対応するビットがセットされる．これらのフラグはトランザクションを実行する毎に参照され，プログラムの終了までクリアされない．

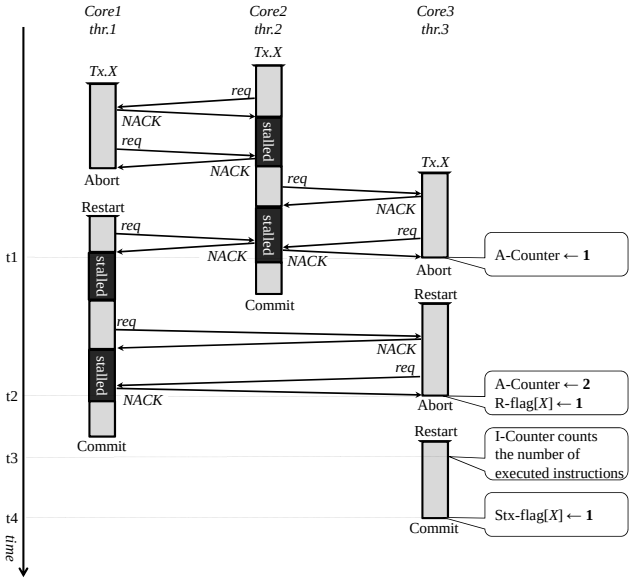


**図 2** 逐次実行対象トランザクションの決定

**Long Tx flags（Ltx-flags）**： トランザクションの ID をインデクスとし，そのインデクスに対応するトランザクションの実行命令数が多いかどうかを記憶する．前述した I-Counter の値が，閾値 *L-inst* より大きい状態でコミットが発生した場合，実行中のトランザクションの ID に対応するビットがセットされる．これらのフラグは前述した Stx-flags と同様にトランザクションを実行する毎に参照される．なお，前節で述べたように，トランザクションの実行命令数が少ないにもかかわらず，割り込み処理により誤って Ltx-flags がセットされる場合がある．そのような状況に対応するため，I-Counter の値が閾値 *S-inst* 以下の状態でコミットが発生した場合，実行中のトランザクション ID に対応するフラグをクリアする．

これらの追加 H/W のコストがどのくらいの規模になるのかをここで見積もる．一般的なベンチマークには多くて 10 強のトランザクションが含まれており，また，各トランザクションで実行される命令数は数百程度が一般的である．その場合，必要な H/W は A-Counter が 2 bit，I-Counter が 9 bit，R-flags，Stx-flags および Ltx-flags はそれぞれ 16 bit あれば十分である．

競合が発生するトランザクションの実行例について，**図 2** を用いて *thr.3* における追加ハードウェアの動作を説明する．なお，説明を簡単にするため，閾値 *A-tx* の値を 2 とする．

まず，*thr.3* の実行する *Tx.X* が *thr.2* との競合によりアボートされるとする（t1）．このとき，1 回目のアボートが発生したため，A-Counter の値は 1 となる．次に，*thr.3* は *Tx.X* を再実行し，今度は *thr.1* との競合により *Tx.X* をアボートするとする（t2）．このとき，A-Counter の値は 2 となり，閾値 *A-tx* と等しくなる．ここで，R-flag のうち当該トランザクションの ID に対応するインデクスを持つビット R-flag[$X$] の値をセットすることで，*Tx.X* がアボートを頻発させたトランザクションであることを記憶する．

続いて *thr.3* は *Tx.X* を実行するが，R-flag[$X$] にビットがセットされているため，トランザクション内で実行される命令数が I-Counter でカウントされる．そしてこの I-Counter の値が，閾値 *L-inst* より小さい状態でトランザクションがコミットされるとき，実行するトランザクションの ID が $X$ であるため，Stx-flag[$X$] のビットがセットされる（t4）．このようにして，*Tx.X* 中で実行された命令数が少なかったと記憶される．一方で，I-Counter の値が *L-inst* より大きい状態でトランザクションがコミットされる場合は，Ltx-flag[$X$] のビットがセットされる．その後，Stx-flag[$X$] のみがセットされている状態で *Tx.X* が実行される場合，このトランザクションは過去に競合を頻繁に引き起こし，かつ実行命令数が少ないと判定され，逐次実行の対象となる．

### 2.3.2 トランザクションの実行順序制御

前項で述べた逐次実行対象トランザクションの実行順序を制御するために，以下に示す機構を各コアに追加する．

**Sequential flag（S-flag）：** 他スレッドが実行を試みたトランザクションを待機させる必要があるか判定する．このフラグは，自スレッドが逐次実行対象となるトランザクションの実行を試みた場合にセットされる．また，自身が実行するトランザクションのコミットを他のスレッドが待機し始めた場合にクリアされる．

**ID of Opponent Thread（O-id）：** 自スレッドが実行するトランザクションのコミットを待つ相手スレッドの番号を記憶する．この値はコミット時に参照され，どのスレッドが自身のコミットを待っているかを知るのに用いられる．なお，この値の参照により待機スレッドが特定された後，その待機スレッドに実行の許可が与えられるため，この値は参照後にクリアされる．

ここで，これらの追加 H/W のコストを見積もる．コア数および総スレッド数が 32 の場合では，S-flag は 1 bit，O-id は 5 bit となり小量で十分であることが分かる．また，前項で述べた追加 H/W コストと合計すると，260B となり，futile stall 防止手法を小容量の H/W を追加することで実現できることが分かる．

次に，**図 3** の，逐次実行対象トランザクションの実行順序が制御される場合の例を用いて，3 つのスレッド（*thr.1*〜*3*）における追加ハードウェアの動作を説明する．なお，

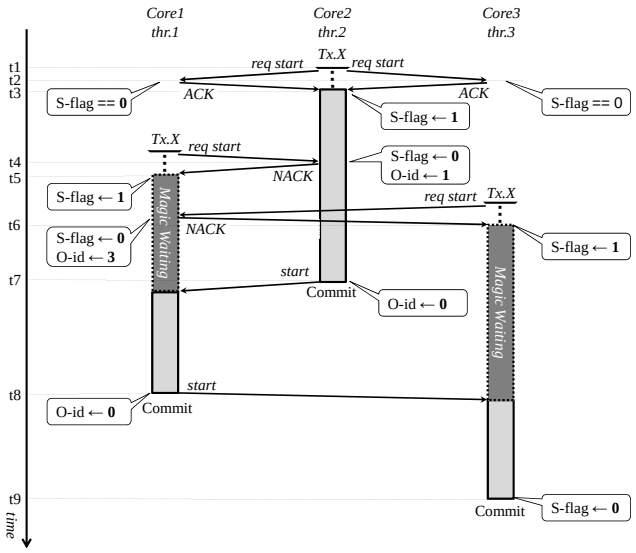


**図 3** トランザクションの実行順序制御

図 3 の *Tx.X* は既に逐次実行の対象とされているとする．

まず，*thr.2* が逐次実行の対象である *Tx.X* の実行を開始しようとした場合（t1），実行の許可を求めるリクエストが *thr.1* と *thr.3* に送信される．このリクエストを受信した *thr.1* および *thr.3* はそれぞれ S-flag を参照する（t2）．*thr.1* と *thr.3* は時刻 t1 までに逐次実行対象トランザクションの実行を試みていないため，S-flag はセットされていない．したがって，*thr.1* および *thr.3* から *ACK* が *thr.2* に返信される．この *ACK* を受信した *thr.2* は，*Tx.X* の実行を開始するとともに，自身の S-flag をセットする（t3）．

続いて，*thr.1* が *Tx.X* の実行を開始しようとした場合，さきほどと同様に実行の許可を求めるリクエストが他スレッドに送信される．なお，*ACK* が返信される場合のリクエストは省略している．このリクエストを受信した *thr.2* は，自身の S-flag がセットされていることから *thr.1* によるトランザクションの実行を待機させる必要があると判断する（t4）．したがって，*thr.2* は *NACK* を *thr.1* に返信し，待機させた相手スレッドの番号である 1 を O-id にセットする．ここで，もし *thr.2* が実行するトランザクションのコミットをさらに別のスレッドが待機した場合，*thr.2* のコミット後に複数のスレッドが同時に待機状態から復帰してしまう．この復帰したスレッドはそれぞれトランザクションの実行を開始するため，それらのスレッド間で競合が発生する可能性がある．そこで，*thr.2* は S-flag をクリアすることで自身のコミットを待機するスレッドを *thr.1* のみに限定する．

一方で *thr.2* から *NACK* を受信した *thr.1* は，自身の S-flag をセットし（t5），*thr.2* からトランザクションの実行許可が得られるまで待機し続ける．これにより，他のスレッドがトランザクションの実行を開始しようとした場合に *thr.1* のトランザクションがコミットするまで待機させることができる．次に *thr.3* が *Tx.X* の実行を開始しよう

表 1　シミュレータ諸元

| | |
|---|---|
| Processor | SPARC V9 |
| number of cores | 32 cores |
| frequency | 1 GHz |
| issue width | single-issue |
| 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 |

とした場合も同様の処理が行われる（t6）．

その後，*thr.2* の実行する *Tx.X* がコミットされるとする（t7）．このとき，*thr.2* は O-id を参照することで，自身の実行するトランザクションがコミットされるのを待機しているのは *thr.1* だということを知ることができる．したがって，*thr.2* は *thr.1* にトランザクションの実行許可を与える（t7）．また，*thr.2* のコミットを待機しているスレッドが存在しなくなるため，*thr.2* は自身の O-id をクリアする．続いて *thr.1* および *thr.3* の実行する *Tx.X* がコミットされる場合もさきほどのコミット時と同様の処理が行われる（t8，t9）．以上のようにして，トランザクションの逐次実行が制御される．

## 3. 評価

2.3 節で述べた拡張を，HTM の研究で一般的に用いられている LogTM[2] に実装し，シミュレーションによる評価を行った．また，我々は starving writer という競合パターンを解決する手法を提案しており**?**，この手法と本論文で提案した手法を組み合わせた場合に性能にどのような影響が現れるのかを調査するため，これも併せて評価した．

### 3.1 評価環境

評価には HTM の研究で広く用いられている Simics [3] 3.0.31 と GEMS [4] 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．プロセッサは 32 コアの SPARC V9 とし，OS は Solaris10 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 付属 microbench，SPLASH-2 [5]，および STAMP [6] から計 12 個を使用した．

表 2　実行サイクル数の削減率

| | GEMS | SPLASH-2 | STAMP | all |
|---|---|---|---|---|
| (S) 平均 | 8.5% | 10.3% | 1.7% | 7.5% |
| 最大 | 17.3% | 18.7% | 1.9% | 18.7% |
| (F) 平均 | 31.7% | 26.8% | 0.9% | 19.8% |
| 最大 | 72.7% | 71.5% | 1.8% | 71.5% |
| (H) 平均 | 36.6% | 34.0% | 2.1% | 28.4% |
| 最大 | 72.2% | 70.4% | 3.1% | 72.2% |

### 3.2 評価結果

**図 4** および**表 2** に実行サイクル数比を示す．図 4 の 4 本のグラフはそれぞれ左から順に

**(B)**　既存の LogTM（baseline）
**($S_3$)**　Starving writer 解消手法
**($F_4$)**　Futile stall 防止手法
**(H)**　Hybrid モデル：2 つの手法を組み合わせたモデル．

の実行サイクル数比を表しており，既存手法 (B) の実行サイクル数を 1 として正規化している．また，凡例は内訳を示しており，Non_trans はトランザクション外，Good_trans，Bad_trans はそれぞれ結果的にコミット/アボートされたトランザクション内の実行サイクル数．Aborting，Backoff，Stall，Barrier，Magic_Waiting はそれぞれ，アボート，exponential backoff，ストール，バリア同期，magic waiting に要したサイクル数である．なお，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しなければならない [7]．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表している．

結果を手法別に見ると，まず starving writer 解消手法 (S) では，解決すべき対象とした競合の再発，およびそれにともなうアボートの頻発をほとんどのプログラムが含んでいたため，(S) によりこれらを解決することで性能が向上した．なお，Slist は競合の繰返しがほとんど発生しないプログラムであるが，(S) はそのような特徴を持つプログラムに対して性能に悪影響を及ぼさないため，(B) とほぼ同等の結果となっている．

プログラムを個別に見ると，まず Contention，Genome，Kmeans，Vacation では，ほぼすべての手法で提案手法によりわずかに高速化している．これは主にアボートの抑制によるもので，アボート回数は既存手法 (B) に対し最大 72.9%（Kmeans），最低でも 17.1%（Genome）削減されている．また，全実行サイクルに占める magic waiting の割合は，たとえば Kmeans では 0.1%以下となっており，本提案によって新たに加えられた待機処理が短時間で済んでいることが分かる．しかしこれらのプログラムでは，元来アボートが実行サイクルに与える影響は小さかったため，高速化率は小さくなっている．

次に，Btree，Deque，Prioque，Barnes，Radiosity につ

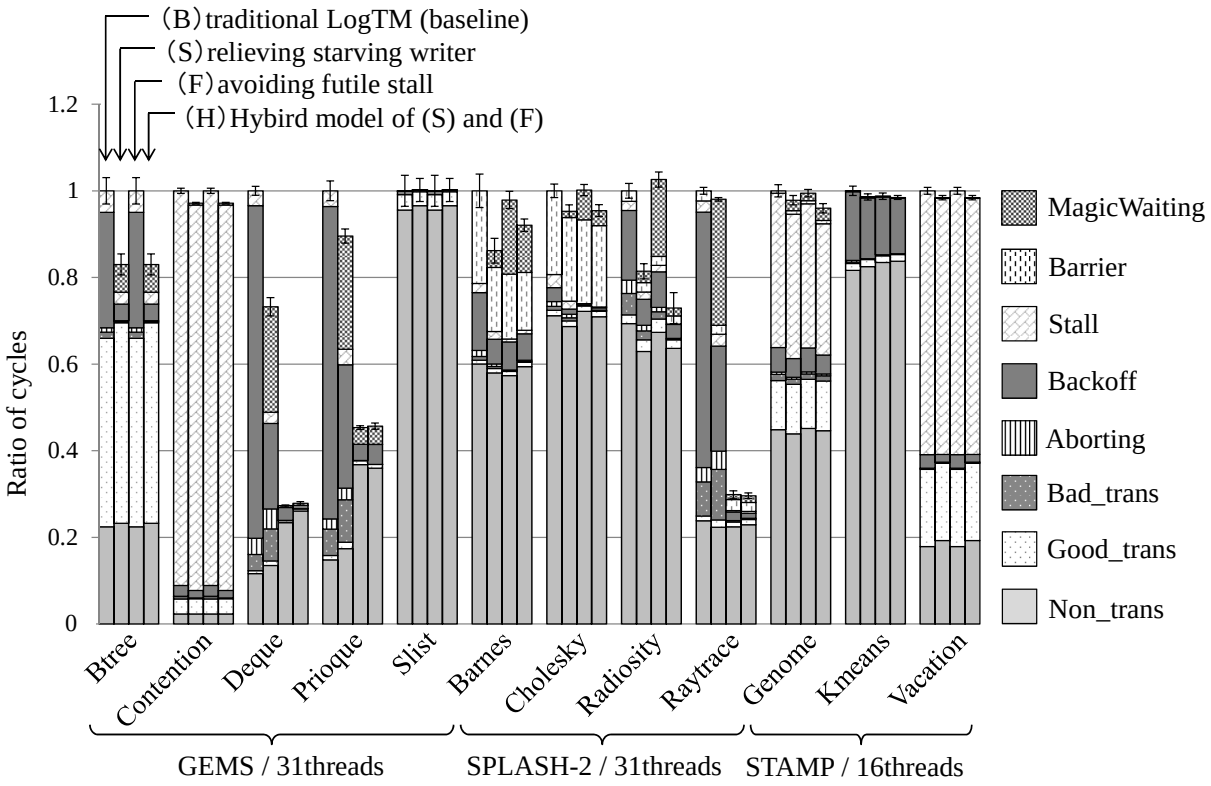


図 4　実行サイクル数比

いては，アボート回数の削減による Bad_trans や Aborting サイクルの減少，競合自体の削減による Stall サイクルの減少，アボートの繰返しを抑制したことによる Backoff サイクルの減少などにより大きく高速化しており，提案手法の有効性が確認できた．なおこれらのうち Deque および Prioque については Magic_Waiting の占める割合が比較的大きいことから，writer が早期にコミットできたというよりむしろ，magic waiting によって exponential backoff よりも適切な待ち時間が reader に設定された結果であると考えられる．しかし残り 3 つのプログラムについては，実行サイクル数全体に占める Magic_Waiting の割合が少ないため，writer が従来手法より早期にコミットに至ったことの効果が大きいと考えられる．

なお，これらの中でも Btree は最も starving writer が発生するプログラムであるが，starving writer 発生時のアボート抑制および Backoff 削減も高速化に寄与している．Btree において，(S) のアボート回数を調査したところ，既存手法に対し 86.8% も削減できていることが確認できた．また，最長のアボート繰返し回数についても，約 1/4 程度に削減されていた．

また Radiosity では，(S) により 30 スレッドが一時的に magic waiting を有効にする状況が見られた．これはすなわち，非常に競合を起こしやすいトランザクションが存在するということである．そして (S) は，そのトランザクションの実行をコミットまで優先的に進行させることで，(B) で発生していたアボートの頻発を抑制したと考えられる．

続いて futile stall 防止手法 (F) の結果を見てみると，まず Btree，Contention，Slist，Vacation については競合を引き起こしたトランザクションの実行命令数が閾値 *L-inst* よりも多く，これにより逐次的な実行が行われなかったため，既存モデルと同じ結果となった．次に，Cholesky と Radiosity について見ると，一部の提案モデルの性能がわずかに低下している．これは，アボートが時折繰り返されるものの基本的には多く発生しないトランザクションを逐次実行したためである．

一方，Genome と Kmeans では適切に逐次実行の対象が選択されたが，(B) においてそのトランザクションで発生していたアボートが性能に与える影響は少なかったため，大きな性能向上が得られなかったと考えられる．また，(F) による効果が最も大きかった Deque，Prioque および Raytrace は，アボートを頻繁に引き起こすトランザクションもしくは全くアボートを引き起こさないトランザクションのどちらかしか存在しないプログラムであり，(F) ではアボートを頻発させるトランザクションのみが逐次実行の対象とされたため性能が大きく向上した．

なお，Deque および Prioque の内訳を詳細に見てみると，どの提案モデルも既存モデルと比べて Non_trans が大きく増加したことが分かる．これら 2 つのプログラムでは，トランザクション外の処理の大部分を random 関数の実行が占めていた．そこでこの random 関数が，Non_trans 増大の原因に関係しているか調べるため，この関数を線形合同法によって乱数を発生させる関数に置き換えて評価した．

その結果，既存モデルと提案モデルの Non_trans が同程度となったため，random 関数が各提案モデルに何らかの影響を与えていると考えられる．

最後に，2 つの手法を組み合わせたモデル (H) の結果では，Barnes 以外のプログラムにおいて，2 つの提案手法のうちの結果が良い方と同等の結果となったことが分かる．なお，Barnes では，(S) によって十分に高速化が得られる一部のトランザクションをプログラムの早い段階から逐次的に実行してしまったため，(S) ほど性能が向上しなかったと考えられる．

一方 Radiosity と Genome では，(H) の結果が (S) および (F) の結果を上回った．これらのプログラムでは，starving writer が発生するトランザクションと競合が頻発するトランザクションの両方が存在するため，トランザクション毎に適切な手法が適用され，(H) の性能向上率が最も高い結果になったと考えられる．

## 4. 関連研究

トランザクションのアボート後，その実行を最初からやり直すのではなく，途中から再実行することにより，その地点までの再実行を省略する部分ロールバックに関する研究 [8], [9], **?** やアプリケーションの振る舞いによってバージョン管理や競合検出の方式を動的に変更する研究**?** など数多くの HTM に関する研究が行われてきた．前者の研究のように部分ロールバックを適用した場合，実行再開後，競合しやすいアドレスにアクセスする箇所まで到達するのに要する時間が短縮されるため，競合がより再発しやすくなる．一方で後者の研究のように競合検出やバージョン管理の方式を変更する場合，競合の起こり方を考慮していないため，本論文で述べた競合パターンによって悪影響を受けてしまう．そこで本稿では，このような競合パターンを考慮しスレッドのスケジューリングを改良することで，従来の問題の解決を図った．

本研究と同様に，スレッドスケジューリングに着目した改良手法もこれまで数多く提案されてきた．Titos ら [10] は Eager 方式と Lazy 方式を組み合わせた新しい競合解決手法を提案した．彼らの手法では，基本的には Eager 方式が採用されている．しかし競合が発生した場合には，その競合対象のキャッシュライン上のデータを Lazy 方式のように扱う実行に切り替えることで，競合を引き起こしたデータを含むトランザクションのアボートを抑制した．

Yoo ら [11] は並列実行そのもののパフォーマンスを向上させるために，HTM に adaptive transaction scheduling（ATS）を実装することで，高い競合率によって並列性が欠落するようなワークロードのパフォーマンスを向上させる手法を提案した．ATS はアプリケーションからのフィードバックを用いて並列に実行するトランザクションの数を動的に制御することができる．この手法では既存の HTM に対して最大で 97%の実行速度向上を達成していたが，ほとんどのベンチマークでは速度向上しておらず，提案手法の効果が得られたプログラムは一部であった．

Akpinar ら [12] 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た．それらのポリシーでは，アボートされたトランザクションの数などの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，彼らは既存の backoff アルゴリズムよりも適切な待機時間を設定するアルゴリズムを考え，提案したポリシーと組み合わせることで競合の抑制を図った．しかし，性能向上が得られた手法は多くなく，最も性能向上率の高い手法でも最大で 15%の速度向上しか実現できておらず，その効果は大きいものではない．

Geoffrey ら [13] は複数のトランザクション内で共通してアクセスされるアドレス数に注目し，共通の度合を Similarity と定義した．彼らは bloom filter を用いてこの Similarity を見積もる手法を提案し，Similarity が一定の閾値を超えた場合，トランザクションを逐次的に実行することで競合の再発を抑制する．しかし，この手法によって，既存の HTM に対して速度性能がどの程度向上したのか示されていない．

Gaona ら [14] は，消費エネルギーを削減するための新たな手法を提案した．この手法では，複数のトランザクション間で競合が発生した場合，それらのトランザクションに優先度が設定され，逐次的に実行される．優先されたトランザクションが実行されている間，優先されなかった残りのトランザクションを待機させることで消費電力が抑制される．しかし，既存の HTM に対して消費電力の削減が 10%しか実現できておらず，また速度性能は既存の HTM と同程度であった．

以上に述べた手法はいずれもアボートや競合発生の有無などの情報に基づいてスレッドの振る舞いを決定している．しかしこれらの情報を用いるだけでは，単に競合が発生しただけでもスレッドの振る舞いに変化が生じる可能性がある．また，スレッドの振る舞いを決める際に，著しく性能が低下する際の競合パターンに注目していないため，適切に競合を解決できない可能性もある．

一方本稿は，性能に悪影響が及ぼされる場合のアクセスパターンやアボートが繰り返される動作に注目した．また，トランザクションのアボートが繰り返される場合，そのトランザクションは再びアボートされる場合が多く，この動作により HTM の性能が大きく低下する可能性が高い．したがって，アボートの繰り返しを引き起こすような競合を抑制することが重要であると考え，これを解消することで性能向上を図る手法を提案した．

## 5. おわりに

本稿では，HTM におけるトランザクションの実行時に

悪影響を及ぼす可能性のある競合パターンを 2 つ述べ，これらを解決するための手法をそれぞれ提案した．提案した 2 つの手法の有効性を検証するために，既存の LogTM に提案手法をそれぞれ実装し，GEMS 付属の microbench，SPLASH-2，および STAMP の 3 種のベンチマークを用いてシミュレーションによる評価を行った．評価の結果，2 つの手法を組み合わせたモデルでは競合再発に起因するアボートの繰り返しおよびストールが抑制されることを確認した．その結果，既存の LogTM に比べて実行サイクル数が最大で 72.2%，平均で 28.4%削減されることを確認した．また，これら 2 つの提案手法の各モデルを実現するにあたり必要なハードウェアコストを見積もった結果，各提案手法のハードウェアコストはそれぞれ 384B，260B であり，少量のハードウェアを追加することでこれらの手法が実現できることを示した．

今後の課題として次の 2 つが挙げられる．まず 1 つ目の課題として，starving writer 解消手法の改良である．たとえば，starving writer 解消手法に部分ロールバックを組み合わせることで，再実行コストを削減することが考えられる．その際，特定の命令の実行に優先度を設ける，もしくは backoff のための期間を改善することで，競合の再発を防止する必要がある．

次に 2 つ目の課題として，futile stall 防止手法でより適切に逐次実行対象のトランザクションを選択することが挙げられる．これを実現するために，futile stall 防止手法で用いた閾値 *A-tx*，*L-inst* および *S-inst* をプログラムに合わせて動的に設定することもしくは，別の指標を設けることなどが考えられる．

## 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*, pp. 289–300 (1993).

[2] 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*, pp. 254–265 (2006).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[9] Waliullah, M. M. and Stenstrom, P.: Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems, *Proc. Int'l Symp. on Parallel and Distributed Processing (IPDPS)*, pp. 1–11 (2008).

[10] Titos, R., Negi, A., Acacio, M. E., García, J. M. and Stenstrom, P.: ZEBRA:A Data-Centric, Hybrid-Policy Hardware Transactional Memory Design, *Proc. Int'l Conf. on Supercomputing (ICS'11)*, pp. 53–62 (2011).

[11] 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).

[12] 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).

[13] Blake, G., G., R., Dreslinski and Mudge, T.: Bloom Filter Guided Transaction Scheduling, *Proc. 17th International Conference on High-Performance Computer Architecture (HPCA-17 2011)*, pp. 75–86 (2011).

[14] Gaona, E., Titos, R., Acacio, M. E. and Fernández, J.: Dynamic Serialization' Improving Energy Consumption in Eager-Eager Hardware Transactional Memory Systems, *Proc. Parallel, Distributed and Network-Based Processing 2012 20th Euromicro International Conference (PDP'12)*, pp. 221–228 (2012).