Vol.2014-ARC-208 No.22
2014/1/24

共有変数に対する複合操作を排他実行するハードウェアトランザクショナルメモリの改良

橋本 高志良[1]　井出 源基[1]　山田 遼平[1]　堀場 匠一朗[1]　津邑 公暁[1,a)]

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が提案されている．この機構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（HTM）では，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．しかし，共有変数に対する複合操作が行われるようなトランザクションが並行実行された場合，その際に発生するストールが完全に無駄となる場合がある．本稿では，このような同一の共有変数に対する Read→Write の順序でのアクセスを検出し，それに関与するトランザクションを排他実行することで，HTM の全体性能を向上させる手法を提案する．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 72.2%，平均 17.5%の性能向上を達成した．

1. はじめに

マルチコア環境において一般的となっている，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，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，この機構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 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**として定義し，共有リソース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ため，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，共有リソース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（**Version Management**; **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，共有リソース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（**Conflict Detection**; **競合検出**）．TM 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，これらの処理を高速化している．

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る．しかし，共有変数に対する複合操作が行われるようなトランザクションが並行実行された場合，その際に発生するストールが完全に無駄となる場合がある．そこで本稿では，このような同一の共有変数に対する Read→Write の順序でのアクセスを検出し，それに関与するトランザクションを排他実行することで，HTM の全体性能を向上させる手法を提案する．

2. 関連研究

アボートしたトランザクションを途中から再実行することで，その再実行コストを抑える部分ロールバック [2], [3] の研究や，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 [4], [5] など数多くの HTM に関する研究が行われてきた．特にスレッドスケジューリングに関しては，これまで主に 2 つの方向から改良手法が提案されてきた．

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，次の 3 つの手法が挙げられる．まず，Yoo ら [6] は HTM に Adaptive Transaction Scheduling（ATS）と呼ばれるシステムを実装し，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 [7] は複数のトランザク

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

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 [8] は HTM の性能を低下させるような競合パターンに対する，様々な競合解決手法を提案している．また消費電力を抑制するという観点から，Gaona ら [9] は複数のトランザクション間で競合が発生した場合に，その競合に関与したトランザクションに実行優先度を設定し，それらを逐次実行する手法を提案している．

以上に述べた手法は，いずれもアボートや競合の発生回数などの情報のみに基づいてスレッドの振る舞いを決定しており，それらのスレッドが共有変数にアクセスする順序を考慮していない．そのため，HTM の性能を低下させうる競合パターンの根本的な解決には至っておらず，目立った性能向上を得ることはできていない．一方，本稿では共有変数に対するアクセス順序に着目し，上述したスケジューリング手法では解決できていなかった競合パターンの効率的な解決を図る．

3. 共有変数に対する複合操作の排他実行

本章では，既存の HTM における問題点と，それを解決する提案手法について述べる．

3.1 共有変数に対する複合操作に起因する問題

一般に，共有変数への Read アクセスは，その後に Write アクセスをともなう場合が多く見られる．具体的には，複合演算子および複合代入式を用いる複合操作を実現する場合などがこれにあたる．この複合操作を含むトランザクションが複数のスレッドによって並行実行されると，それらのスレッドの Read アクセスが許可されたとしても，その後に実行される Write アクセスにより結局競合が発生し，これが HTM の性能低下を引き起こす可能性がある．

図 1 は，上述した共有変数に対する複合操作を含むトランザクション（*Tx.X*）を，2 つのスレッド *Thread1* および *Thread2* が並行実行する様子を示している．まず，双方のスレッドが `load A` を実行した後，*Thread2* が `store A` を実行しようとした際に，競合が検出される．ここで LogTM[10] に代表される，**Eager Conflict Detection** 方式を採用する HTM では一般に，競合を発生させた *Thread2* が *NACK* の受信にともない，自身の実行する *Tx.X* をストールする（時刻 t1）．その後，*Thread1* が `store A` を実行しようとする際（t2），*Thread2* は既に当該アドレスにアクセス済であるため競合を検出し，*Thread1* へ *NACK* を返信する．この時，*Thread1* は自身よりも早くトランザクションを開始したスレッドから *NACK* を受信するため，*Tx.X* をアボートすることになる（t3）．このアボートにより，*Thread2* は *Tx.X* を再開できるが，この間に *Thread1* の実行は一切進行しておらず，*Thread2* のストールは完全に無駄であったことになる．このように，結果的にアボートされてしまうようなトランザクションとの競合により発生する無駄なストールは **Futile Stall**[11] と呼ばれ，HTM のスループットを低下させる大きな要因となっている．

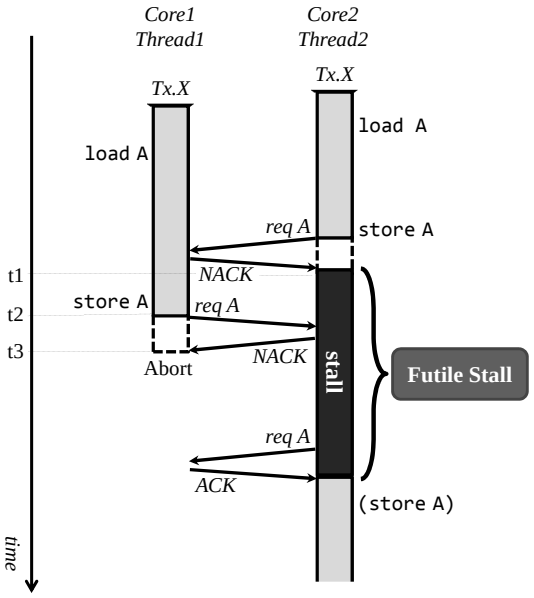

図 1 共有変数への複合操作に起因する Futile Stall

3.2 共有変数に対する複合操作の排他実行手法

前節で述べた Futile Stall が発生する要因として，ある共有変数に対して Read→Write の順でアクセスするスレッドが複数存在する場合に，それらのスレッドが共に Read のみを完了した状態となってしまうことが挙げられる．そこで，Read→Write の順序でアクセスされる共有変数に対する Read を実行する際に，他のスレッドが当該アドレスに Read アクセス済みであるか否かをチェックする．そして，他スレッドが当該アドレスに Read アクセス済みであった場合，この Read アクセスを即座には許可せず *NACK* を返信して待機させることで，共有変数に対する複合操作を排他実行する手法を提案する．

ここで**図 2** に，提案手法を用いた場合の動作を示す．この例では，3 つのスレッド（*Thread1*〜*3*）がそれぞれ，共有変数に対する複合操作，つまり共有変数に対する Read→Write の順序でのアクセスを含む同一のトランザクション（*Tx.X*）を投機実行している．まず，*Thread2* が共有変数のアドレスである *A* に対して `load A` を実行した後，*Thread1* と *Thread3* が同様に `load A` の実行を試みたとする（時刻 t1, t2）．この際に，*Thread1* と *Thread3* は Read アクセスのためのリクエストを送信するが，この時点では *Thread2* が既にアドレス *A* に Read アクセス済みであるため，*Thread1* と *Thread3* のそれぞれに対して *NACK* が返信される．この *NACK* の受信により（t3, t4），*Thread1* と *Thread3* のアドレス *A* に対する Read アクセスの実行が待機させら

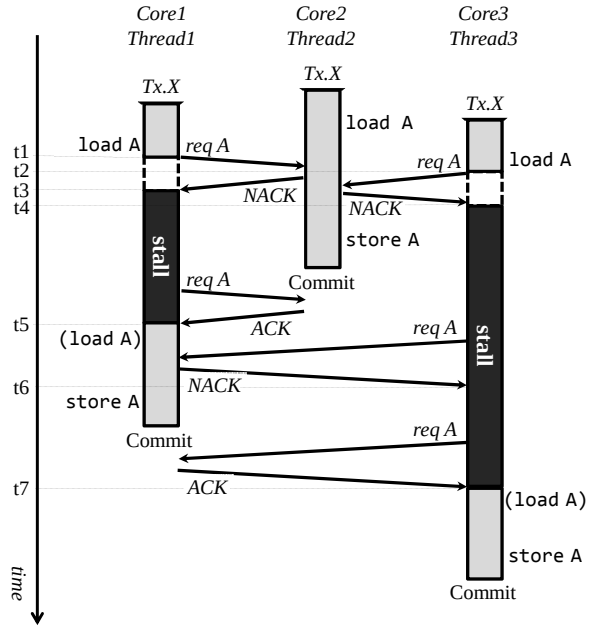

図 2　共有変数に対する複合操作の排他実行

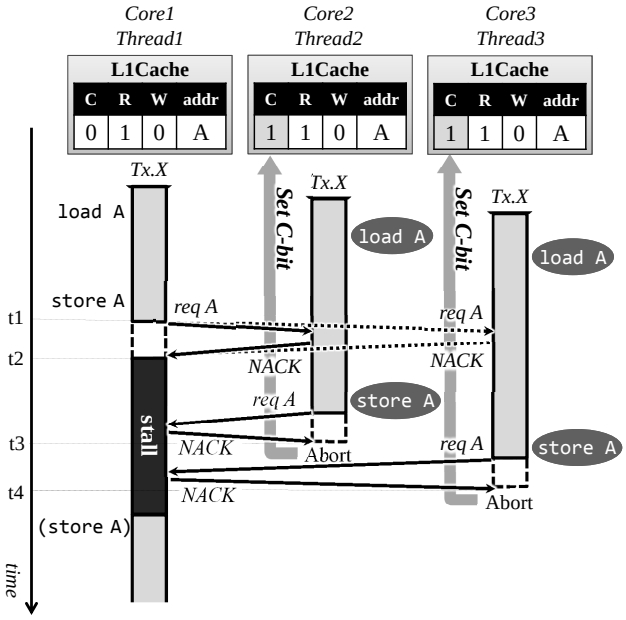

図 3　C ビットのセット

れる．そのため，*Thread2* はアドレス *A* に Write アクセスを試みたとしても，これらのスレッドと競合することなく *Tx.X* の実行を進めることが可能となる．これにより，図 1 で示した Futile Stall による無駄な待機時間が削減される．その後，*Thread2* が *Tx.X* をコミットすると，待機中であった *Thread1* のアドレス A に対する Read アクセスの実行が許可される（t5）．そして，*Thread3* は先程と同様に *NACK* を受信して実行を待機することとなる．続けて，*Thread1* が *Tx.X* をコミットすると，*Thread3* の Read アクセスが許可される（t7）．提案手法では，このように Read→Write の順序でのアクセスにおける Read アクセスの時点で *NACK* を送信することで，共有変数に対する複合操作の排他実行を実現する．

4. 実装

3 章で述べた複合操作の排他実行をキューによって実装する方法 [12] を我々は既に提案しているが，本章ではそれをさらに改良した実装方法と，その動作モデルについて述べる．

4.1 拡張ハードウェア構成

提案手法を実現するために，各コアの L1 キャッシュ内のキャッシュラインに以下のハードウェアを追加する．

Combined-Control bit（C ビット）：
当該キャッシュラインが Read→Write の順序でアクセスされたか否かを示すビット

各スレッドは，自身があるアドレスに対して Read→Write の順序でアクセスしたと判断した場合に，当該キャッシュラインの C ビットをセットする．そして他スレッドから Read アクセスのためのリクエストを受信したスレッドは，アクセス対象アドレスに対応する C ビットと R ビットを参照して *NACK* を返信すべきか否かを判定する．ここで R ビットとは，既存の HTM において W ビットとともに競合を検出するために各キャッシュラインに付加されるビットである．そしてこれらの R/W ビットは，そのアドレスに対する Read/Write アクセスが発生した際にセットされる．

ここで，追加した C ビットのハードウェアコストについて検討する．この C ビットは，当該キャッシュラインが Read→Write の順序でアクセスされたか否かを示すため，ライン 1 行あたり 1bit だけ必要となる．また，今回ターゲットとしている SPARC V9 プロセッサはコア数が 32，L1 キャッシュサイズが 32KBytes，キャッシュラインサイズが 64Bytes，そしてウェイ数が 4 であるため，提案手法を実装するために必要となるハードウェアコストは 1bit × 128 行× 32 コア=512Bytes と少量である．これに対して複合操作の排他実行をキューにより実装する方法 [12] では，必要となるハードウェアコストが少なくとも 768Bytes となっており，さらに既存の NACK とは異なる新たなメッセージおよび新たな待ち状態を定義する必要がある．そのため本稿で取り上げる実装方法の方が，より少ない追加ハードウェア量かつ既存の機構を利用した単純な動作で，複合操作の排他実行を実現できる．

4.2 共有変数に対する複合操作の排他実行の実現

本節では共有変数に対する複合操作を排他実行する動作モデルについて述べる．

4.2.1 C ビットをセットするタイミング

3 つのスレッド（*Thread1*〜*3*）がそれぞれ同一のトランザクション（*Tx.X*）を投機実行している**図 3** を例に，追

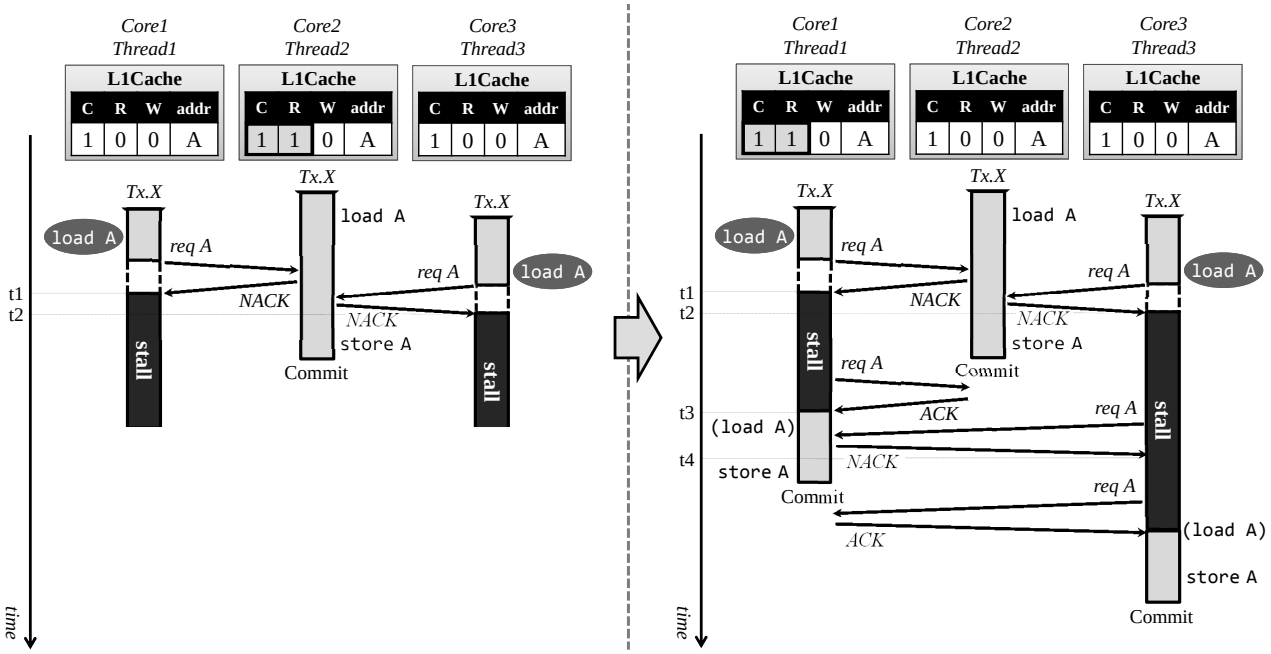

図 4　C ビットの利用

加した C ビットをセットする動作を説明する．まず，各スレッドが `load A` を実行した後，*Thread1* が `store A` を実行しようとする場合（時刻 t1），この Write アクセスによって Write-after-Read（WaR）と呼ばれる競合が発生する．この競合により，*Thread2* と *Thread3* から *NACK* が返信されるため，*Thread1* は自身の実行する *Tx.X* をストールする（t2）．続いて *Thread2* と *Thread3* が，それぞれ `store A` を実行しようとするが，*Thread1* との間でそれぞれ WaR 競合が発生するため，これらのスレッドは自身の実行する *Tx.X* のアボートを試みる．この時，*Thread2* と *Thread3* はアクセスしようとしていたアドレス `A` に対応する自身の R ビットをチェックする．当該アドレスの R ビットがセットされている場合，*Thread2* と *Thread3* は，自身が Write アクセスに先立ってアドレス `A` に Read アクセスしたことが分かるため，アドレス `A` に対応する C ビットをセットする（t3，t4）．

4.2.2　C ビットの利用

4.2.1 項で述べた動作によってセットされた C ビットの利用により，共有変数に対する複合操作が排他実行される様子を**図 4** に示す．はじめに，3 つのスレッド（*Thread1*〜*3*）は同一のトランザクション（*Tx.X*）を実行しており，さらに各コアの L1 キャッシュ上のアドレス `A` に対応する C ビットが既にセットされているとする．

この状態でまず，*Thread2* が `load A` を実行後，*Thread1* と *Thread3* が `load A` の実行を試みたとする．この時，*Thread1* と *Thread3* は *Thread2* へ，`A` に対する Read アクセスのためのリクエストを送信する．このリクエストを受信した *Thread2* は，アドレス `A` に対応する自身の C ビットと R ビットをチェックする．そしてどちらのビットもセット済みであるなら，当該アドレス `A` は Read→Write の順序でアクセスされ，かつ自身が Read アクセス済みであると判断できるため，*Thread2* は受信した *red A* に対して *NACK* を返信し，リクエストを送信したスレッドの Read アクセスの実行を待機させる（時刻 t1，t2）．これにより，*Thread2* は *Thread1* および *Thread3* と競合することなく *Tx.X* をコミットできる．その後，図 4 では実行を待機していたスレッドのうち，*Thread1* の Read アクセスの実行が許可されるため（t3），*Thread3* は先程と同様に *NACK* を受信し，Read アクセスの実行を待機し続ける（t4）．以上のように追加した C ビットを利用することで，共有変数に対する複合操作の排他実行を実現できる．

4.2.3　C ビットをクリアするタイミング

ここで，追加した C ビットをクリアするタイミングについて述べる．4.2.2 項でも述べたように，この C ビットは対応するキャッシュラインアドレスが Read→Write の順序でアクセスされたことを示すものであり，共有変数に対する複合操作を排他実行するために用いられる．しかしプログラム内容によっては，あるフェーズで Read→Write の順序でアクセスされるような共有変数が，別のフェーズでは Read アクセスしかされない可能性もある．そのため，この C ビットがセットされているアドレスに対して，Read アクセスしか行わないようなトランザクションが実行された場合，本来並行実行できるはずのトランザクションが逐次実行されることで，結果として性能低下を引き起こしてしまう可能性がある．後の 5 章で詳細に述べるが，あるデータ構造に対する検索処理などが，この Read アクセスしか行わないトランザクション処理に該当する．

そこで，各スレッドは実行トランザクションをコミット

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
L1 cache	32 KBytes
ways	4 ways
latency	3 cycle
line size	64 Bytes
L2 cache	8 MBytes
ways	8 ways
latency	34 cycles
line size	64 Bytes
L2 Directory	Full-bit vector sharers list
latency	6 cycles
Memory	4 GBytes
latency	500 cycles
Interconnect network	2D mesh topology
link latency	3 cycles
link bandwidth	64 Bytes

した際に，C ビットのセットされているアドレスが Read アクセスしか行われていないか否かを確認するために，当該アドレスの W ビットをチェックする．この W ビットがセットされていなかった場合，各スレッドは自身が当該アドレスに対して Read アクセスのみしか行わなかったと判断して C ビットをクリアする．以上のよう動作させることで，上述した Read アクセスしか行わないトランザクションの逐次実行による性能低下を防ぐことができる．

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，得られた評価結果から考察を行う．

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[10] に実装し，シミュレーションによる評価を行った．評価には Simics[13] 3.0.31 と GEMS[14]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，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ション環境を示す．評価対象のプログラムとしては GEMS 付属 microbench, SPLASH-2[15], および STAMP[16] から計 10 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．

5.2 評価結果

16 スレッドで実行した評価結果を**表 2** および**図 5** に示す．ここで図 5 中では，各ベンチマークプログラムの評価結果が 3 本のバーで表されており，左から順に，

(B)　既存の LogTM（ベースライン）
(Q)　キューを用いて実装した参考モデル
(P)　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．ここで，参考モデル (Q) は複合操作の排他実行をキューを用いて実装した方法 [12] の結果を示している．なお，STAMP ベンチマークプログラムである Kmeans と Vacation については，デフォルトで設定されている 2 種類の input を用いて評価しており，よりデータサイズの大きい input を用いた場合の結果を Kmeans+および Vacation+と図中で表記している．また，本稿の提案モデル (P) とキューによる実装を施した参考モデル (Q) の性能比較については 5.3 節で詳細に述べる．

表 2　各ベンチマークにおけるサイクル削減率（16 スレッド実行）

		GEMS	SPLASH-2	STAMP	All
(Q)	平均	26.7%	10.1%	2.8%	13.8%
	最大	53.6%	25.1%	7.1%	53.6%
(P)	平均	35.9%	9.7%	3.1%	17.5%
	最大	72.2%	25.5%	7.4%	72.2%

ここで，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 はバックオフ処理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，MagicWaiting は参考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．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．

評価結果より，提案モデル (P) は既存モデル (B) と比較して大幅な性能向上が得られていることが分かる．このことから，多くのプログラム中には同一の共有変数に対する Read→Write の順序でのアクセスを行うトランザクション処理が含まれており，Futile Stall を頻繁に発生させうる特徴があることが確認できた．この Futile Stall を提案手法によって効率的に解決することで，提案モデル (P) は各ベンチマークプログラムを 16 スレッドで実行した場合において，既存モデルに対して平均 17.5%，最大 72.2%の性能向上を得ることができ，参考モデル (Q) と比較しても平均 3.7%の性能向上を得ることができた．したがって本提案手法は参考モデル (Q) と比較して，少ないハードウェア量で

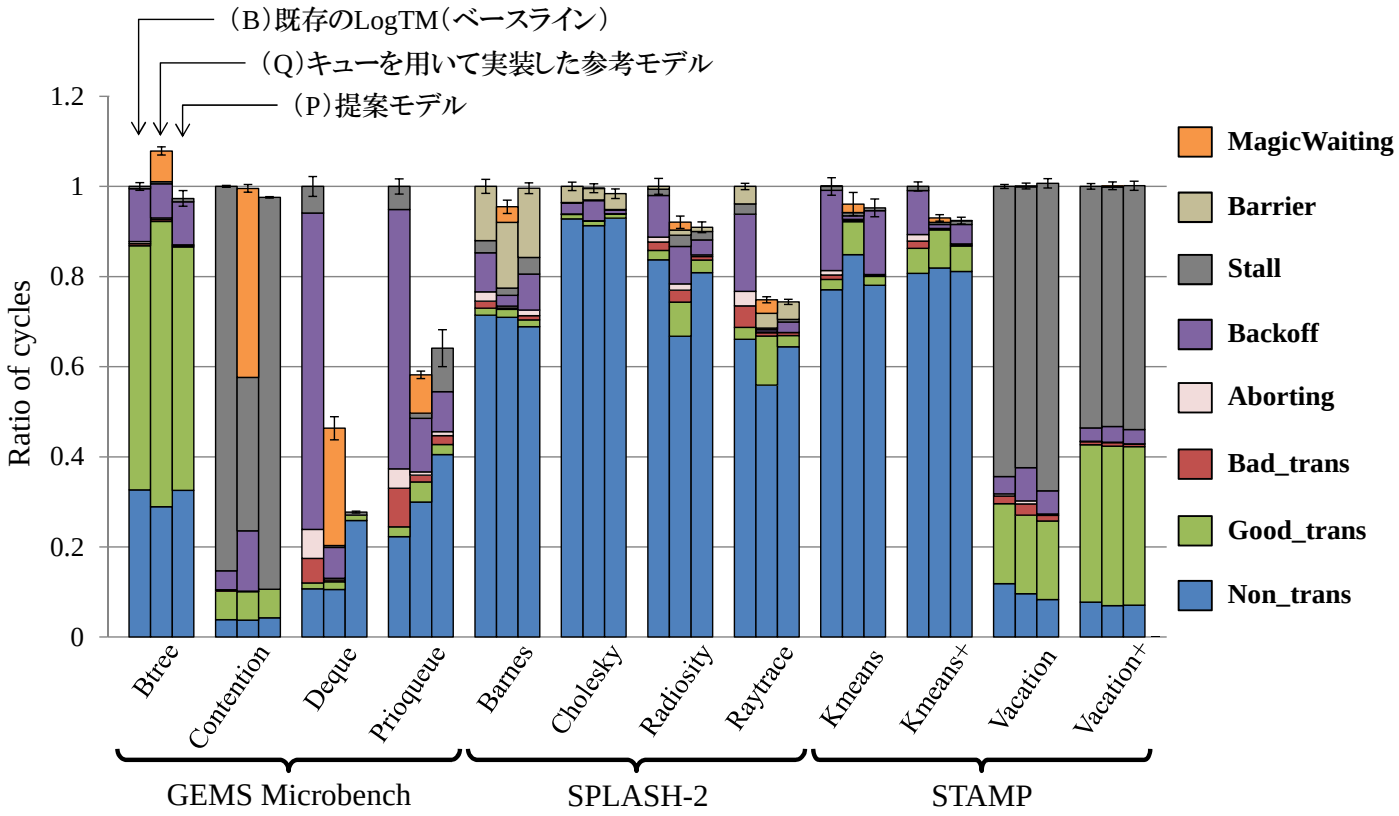

図 5　各プログラムにおけるサイクル数比（16 スレッド実行）

さらなる性能向上を達成できていることが分かる．次節では，各ベンチマーク別に詳細な検証を行う．

5.3 考察

5.3.1 GEMS Microbench

まず GEMS 付属の Microbench では，全てのプログラムで提案モデル (P) は既存モデル (B) に対して性能向上している．これらのうち，Deque および Prioqueue のプログラムでは，特に Backoff サイクル数の大幅な減少が目立つ．この 2 つのプログラム中には，共有変数に対する複合操作を行うトランザクションが含まれており，ある共有変数が Read→Write の順序で頻繁にアクセスされていた．本提案手法では，これにより発生していた Futile Stall やそれに起因するアボートを十分抑制することができたために，Backoff サイクル数の大幅な削減につながったと考えられる．さらに，Deque については提案モデル (P) と参考モデル (Q) でその性能に大きな差が生じていることが分かる．これは提案モデル (P) が，C ビットのセットされているキャッシュラインへの Read アクセスに対して NACK を返信してその実行を待機させるという，既存モデル (B) の競合検出時の動作をわずかに拡張した動作によって複合操作の排他実行を実現していることに起因する．その一方で参考モデル (Q) では，他のスレッドの Read アクセスの実行を待機させたスレッドがキューを用いて待機スレッドの再開順序を制御している．そして順序制御中は，その制御を担うスレッドは既存のストールとは別途に設けた待ち状態となるため，参考モデル (Q) は提案モデル (P) と比較して並列度が低下してしまう場面が多くなってしまうことから，その性能に差が生じたと考えられる．

また，Btree に関しては参考モデル (Q) は既存モデル (B) に対してわずかに性能低下している一方で，提案モデル (P) は性能が向上していることが分かる．この Btree というプログラム中には，ツリー構造に対する挿入および検索のための 2 種類のトランザクション（仮に *Tx.I*，*Tx.J* とする）が存在している．このうち，ツリー構造に対する挿入操作である *Tx.I* には同一の共有変数に対する Read→Write の順序でのアクセスが含まれているが，検索操作である *Tx.J* にはその共有変数に対する Read アクセスしか含まれていない．そのため，複数の *Tx.I* が並行実行される際には提案モデルおよび参考モデルが効果的であり，実際に内訳を比較してもこれら 2 つのモデルは既存モデルに対して Backoff サイクル数が減少している．しかし，複数の *Tx.J* が並行実行される場合には Write アクセスが発生しないため，Read アクセスの実行を待機させてしまうと，4.2.3 項で述べたような Read アクセスしか行わないトランザクションの逐次実行による性能低下が引き起こされてしまう．参考モデル (Q) では，この *Tx.J* の逐次実行を防ぐことができなかったためにわずかに性能が低下していた．こ

```
1 BEGIN_TRANSACTION( 16 );
2   ray->id = gm->rid++;
3 COMMIT_TRANSACTION( 16 );
```

図 6　Raytrace プログラム内のトランザクション．

れに対し提案モデル (P) では，Read アクセスしか行われなかった際に C ビットをクリアするようにしたことで性能低下を防ぎ，複合操作を排他実行したことによる性能向上を適切に得ることができたと考えられる．この Btree のように，あるデータ構造に対する挿入/検索処理を含むトランザクションが並行動作する際には，上述した逐次実行による性能低下が引き起こされる可能性が高く，データ構造が巨大でトランザクション処理時間がさらに長くなった場合に，特にその性能低下が顕著となってしまう．このことから，提案モデル (P) のようにトランザクションの処理内容を考慮してスレッドスケジューリングを行うことは重要であると考えられる．

5.3.2 SPLASH-2

SPLASH-2 ベンチマークでは，提案モデルにおいてほぼ全てのプログラムの実行サイクル数が減少した．これらの中でも Radiosity および Raytrace に関して，提案モデル (P) は既存モデル (B) に対して Backoff サイクル数が大幅に減少している．ここで Raytrace のプログラム中に存在する，共有変数に対する複合操作を含むトランザクションの 1 つを**図 6** に示す．この図 6 のトランザクションには，`gm->rid` 変数に対するインクリメント操作（複合操作）が行われており，これが同一の共有変数に対する Read→Write の順序でのアクセスに相当する．このインクリメント操作はユニークな ID を生成するために排他的に実行されなければならないが，複数のスレッドがこの操作を並行実行した場合には，既存モデル (B) において図 1 で示したような Futile Stall が発生し，なおかつ一方のトランザクションがアボートされることから Backoff サイクル数も増大する．提案モデル (P) では，図 6 に示したトランザクションを排他実行し，上記の問題の発生を抑制したことが Backoff サイクル数の減少に大きく寄与したと考えられる．

その一方で，Barnes のプログラムでは参考モデル (Q) で性能が向上しているのに対し，提案モデル (P) では性能向上が得られていないことが分かる．これは Barnes のプログラム中で，共有変数に対する複合操作を含むトランザクションの直後に，その共有変数に対して Read アクセスしか行わない可能性のあるトランザクションが存在していることが原因であると考えられる．このような挙動を示すトランザクションを**図 7** に示す．図中のトランザクションでは，最初の if 文で変数 `*qptr` のノードタイプが `LEAF` であるかどうかで条件分岐が発生するが（2 行目），仮に分岐が偽であった場合には変数 `*qptr` に対して Read アクセス

```
1  BEGIN_TRANSACTION( 3 );
2   if (Type(*qptr) == LEAF) {
3     le = (leafptr) *qptr;
4     if (le->num_bodies == MAX_BODIES_PER_LEAF) {
5       *qptr = (nodeptr) SubdivideLeaf(le, (cellptr) mynode, l
        , ProcessId);
6     } else {
7       Parent(p) = (nodeptr) le;
8       Level(p) = l;
9       ChildNum(p) = le->num_bodies;
10      Bodyp(le)[le->num_bodies++] = p;
11      flag = FALSE;
12    }
13  }
14 COMMIT_TRANSACTION( 3 );
```

図 7　Barnes プログラム内のトランザクション．

してからトランザクションが即座にコミットされる．そのため Barnes プログラム中の図 7 に示した部分の直前に存在するトランザクションで，変数 `*qptr` のアドレスに対応する C ビットがセットされたとしても，比較的短い期間のうちにこの C ビットがクリアされてしまい，複合操作の排他実行による性能向上が適切に得られなかった．

Cholesky では提案モデル (P) においてあまり大きな性能向上は得られていない．しかし，このプログラムではトランザクション外の実行サイクル数が，プログラムの総実行サイクル数の大半を占めているため，提案モデル (P) によって Futile Stall およびそれに起因するアボートを抑制できたとしても，性能向上の割合が小さくなってしまったと考えられる．

5.3.3 STAMP

STAMP ベンチマークでは，提案モデル (P) によって Kmeans と Kmeans+の実行サイクル数が減少した．このプログラム中には，Microbench および SPLASH-2 で性能向上していたプログラムと同様に共有変数に対する複合操作が含まれており，提案モデル (P) によってこれらの操作を排他実行したために性能向上につながった．ここで 5.2 節で述べたとおり，Kmeans+では Kmeans よりもデータサイズの巨大な input を用いており，これによって Futile Stall やそれに起因するアボートがより多く発生していたために，提案モデル (P) によるサイクル削減率が高くなったと考えられる．

一方，Vacation および Vacation+では提案モデル (P) において性能向上を得ることができなかった．これは，このプログラム中に共有変数に対する複合操作は含まれていないためである．なお，Vacation と Vacation+の評価結果の内訳を見ると，どちらもストールのサイクル数が総実行サイクル数の大半を占めている．この原因を調査したところ，これらのストールはほぼ Write-after-Write（WaW）競合

によって発生するものであることが分かった．したがって今後は，この WaW 競合によって発生してしまうストールを削減できる手法を考案する必要がある．

6. おわりに

本稿では共有変数に対する複合操作，つまり同一の共有変数に対する Read→Write の順序でのアクセスを検出し，このアクセスを含むようなトランザクション処理を排他実行するスレッドスケジューリング手法を提案した．これにより，既存の HTM の性能を低下させる競合パターンである Futile Stall やこれに起因するアボートを抑制した．また本稿では，複合操作の排他実行をごく少量の追加ハードウェアと簡単な動作で実現できることを示した．提案手法の有効性を確認するために GEMS 付属の Microbench，SPLASH-2 および STAMP ベンチマークより合計 10 個の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 スレッド実行時において最大 72.2%，平均 17.5%の実行サイクル数が削減されることを確認した．さらに，本提案手法と同様の発想をキューによって実装する方法と比較して，平均で 3.7%の性能向上を得ることができた．

なお本稿で提案したモデルは，共有変数に対する複合操作を排他実行することで性能向上を図ったが，Vacation のようなストールのサイクル数が総実行サイクル数の大半の割合を占めるプログラムでは性能向上は得られていない．したがって，今後はこのような WaW 競合によるストールがボトルネックとなるプログラムを高速化する必要がある．例えば，トランザクション中の Write アクセスが実行されるタイミングを考慮するなど，Write アクセスに対して焦点を当てたスレッドスケジューリング手法を考案していきたい．

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] 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] Moss, E. and Hosking., T.: Nested Transactional Memory: Model and Preliminary Architecture Sketches., *In OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Langauges*, pp. 186–201 (2005).

[4] M, L., G, M. and A, G.: A Dynamically Adaptable Hardware Transactional Memory, *Microarchitecture(MICRO), 2010 43rd Annual IEEE/ACM*, pp. 27–38 (2010).

[5] Shriraman, A., Dwarkadas, S. and Scott., M. L.: Flexible Decoupled Transactional Memory Support, *ISCA '08 Proceedings of the 35rd annual international symposium on Computer Architecture*, pp. 139–150 (2008).

[6] 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).

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

[8] 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).

[9] 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).

[10] 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).

[11] Bobba, J., Moore, K. E., Volos, H., Yen, L., Hill, M. D., Swift, M. M. and Wood, D. A.: Performance Pathologies in Hardware Transactional Memory, *Proc. 34th Annual Int'l Symp. on Computer Architecture (ISCA'07)*, pp. 81–91 (2007).

[12] 橋本高志良，堀場匠一朗，江藤正通，津邑公暁，松尾啓志：Read-after-Read アクセスの制御によるハードウェアトランザクショナルメモリの高速化，情報処理学会論文誌コンピューティングシステム (ACS44)，Vol. 6, No. 4, pp. 58–71 (2013).

[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).