Vol.2013-ARC-206 No.9
2013/7/31

ハードウェア・トランザクショナル・メモリにおけるトランザクション定義単純化モデルの検討

橋本 高志良[1]　鈴木 大輝[2]　堀場 匠一朗[1]　津邑 公暁[1]　松尾 啓志[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（TM）が提案されている．この機構においては，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ため，一般にロックよりも処理性能が向上する．この機構のハードウェア実装はハードウェア・トランザクショナル・メモリ（HTM）と呼ばれる．しかし，HTM を用いて並列プログラミングを行う場合，プログラマはトランザクションの開始地点と終了地点をプログラム中に明示する必要があるため，従来のロックに比べて並列プログラムの記述が容易になるわけではない．本稿では，HTM におけるトランザクション定義を単純化し，メモリ同期地点を示す定義を新たに導入したプログラミングモデルを提案する．

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を調停する機構として，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含む一連の命令列を**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ことができるため，一般にロックを用いる場合よりも性能が向上する．なお，TM ではトランザクションを実行するスレッド間において，共有リソースへのメモリ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**）．また，トランザクションの実行中には，その実行が投機的であるがゆえ，共有リソース中のデータ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**）．この 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，上述した競合検出およびバージョン管理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，これまで多くの既存研究により，HTM はロックと比較してその処理性能が優れていることが示されてきた．しかし，それら既存研究の性能評価では，これまでクリティカルセクションとして定義していた領域を，そのままトランザクションに書き換えたプログラムが利用されてきた．そのため現状では，プログラマがクリティカルセクションと同様の精度でトランザクション処理領域を定義することが期待されており，HTM がプログラマビリティを向上させつつロックよりも性能面において優れていることは，確認されていない．

そこで本稿では HTM におけるトランザクション定義を単純化し，メモリ同期地点を示す新たな定義を導入したプログラミングモデルを提案する．これにより，プログラマはトランザクションとして処理すべき領域を指定するという概念ではなく，チェックポインティングの概念に基づいた並列プログラミングを行うことができ，より簡潔に HTM を用いた並列プログラムを記述することが可能となる．

2. トランザクショナル・メモリの概要

本章では，本研究の対象であるトランザクショナル・メモリと，そのハードウェア実装である HTM の基本概念に

[1] 名古屋工業大学
Nagoya Institute of Technology
[2] 電気通信大学
The University of Electro-Communications

ついて述べる．

2.1 トランザクショナル・メモリ

ロックに代わる排他制御機構であるトランザクショナル・メモリ（TM）は，データベース上で行われるトランザクション処理をメモリアクセスに適用した手法であり，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，これを投機的に実行する．

このような投機実行において，TM はプログラムの整合性を保つためにトランザクション内で発生するメモリアクセスを監視する．具体的には，複数のトランザクションが同一のアドレスにアクセスを試みた際，TM はプログラムの整合性が保たれなくなる場合にこれらのメモリアクセスを競合として判定する．この競合の検出時に，TM ではデッドロックを回避するためにトランザクションのアボートが行われる．トランザクションをアボートしたスレッドは，トランザクションを再実行するために，自コアのキャッシュ状態をトランザクション開始時点の状態に戻す（**ロールバック**）．一方，競合が検出されることなくトランザクションの実行が完了する場合，トランザクション内で行われた変更をメモリ上に反映させるコミットが行われる．

TM はこのように動作することで，競合が検出されない限りトランザクションを投機的に並列実行することが可能であるため，一般にロックと比較して処理性能が向上する．なお，TM で行われる競合検出やデータのバージョン管理などの操作はハードウェアまたはソフトウェアにより実装される．ソフトウェア実装であるソフトウェア・トランザクショナル・メモリ（STM）[2] では，特別なハードウェア拡張は必要ないが，ソフトウェア処理のためのオーバヘッドが大きい．これに対し，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，競合を検出および解決する機構をハードウェアによってサポートしているため，STM に比べて速度性能が高い．

2.2 競合検出とその解決方法

アドレスに対する競合を検出するため，HTM では一般的に各キャッシュラインに追加された **Read/Write ビット**と呼ばれるフィールドを用いる．これらのビットは，実行トランザクション内で当該ラインに Read/Write アクセスが発生した場合にそれぞれセットされ，コミットおよびアボート時にクリアされる．これらのビットによって検出される競合には，Read after Write，Write after Read，Write after Write の 3 つがある．なお，HTM における競合検出方式は，そのタイミングによって以下の 2 つに大別される．

Eager Conflict Detection: トランザクション内でメモリアクセスが発生した時点で，そのアクセスに関する競合が存在しないか検査する．

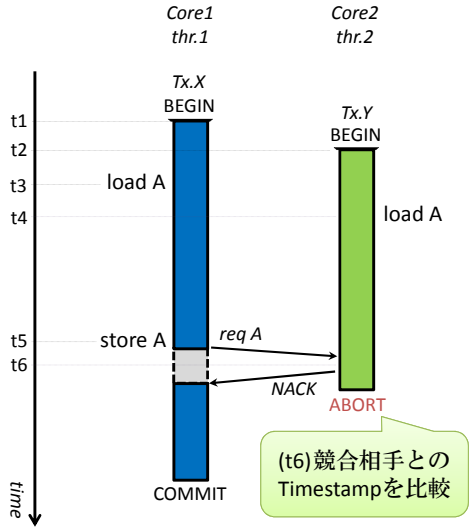


図 1　Timestamp 手法による競合解決

Lazy Conflict Detection: トランザクションをコミットしようとした時点で，そのトランザクション内で行われた全てのメモリアクセスに関して競合が発生していないか検査する．

これら 2 つの方式のうち，Lazy 方式ではトランザクション内で競合が発生してから検出されるまでの時間が Eager 方式に比べて長くなり，無駄な処理が増大して実行効率が悪くなる．そのため，一般的な HTM では競合検出方式として Eager 方式が採用されている．また，HTM では以上に示した方式によって検出される競合に伴って発生するデッドロックをトランザクションのアボートにより回避する．なお，HTM には複数のデッドロック回避手法が存在するが，後述の提案プログラミングモデルに組み合わせる，学習によるチェックポインティング手法 [3] に従い，本稿で取り扱う HTM では **Timestamp 手法**を採用する．

この手法の動作例を**図 1** に示す．この図において，*Core1* と *Core2* はコアを，*thr.1* と *thr.2* は各コア上で動作するスレッドを示している．ここで，*thr.1* と *thr.2* が共通のトランザクション *Tx.X* の実行を開始したとすると（t1，t2），これらのスレッドはそれぞれのトランザクション開始時刻を Timestamp として保持しておく．そして，両スレッドがあるアドレス A に対する Read アクセスを行った後に（t3，t4），*thr.1* が同じアドレス A に対して Write アクセスを試みたとすると（t5），*Core.2* のアドレス A に対する Read ビットがセットされているため，*thr.2* は Write after Read 競合を検出する（t6）．この時，競合を検出した *thr.2* は自身の実行するトランザクションの Timestamp と，競合相手の実行するトランザクションの Timestamp を比較する．この例では，*thr.2* の実行するトランザクションの開始時刻の方が遅いために *thr.2* が実行トランザクションをアボートする．これにより，これらのスレッド間の競合が解決されるため，*thr.1* は自身の *Tx.X* の実行を継続し，

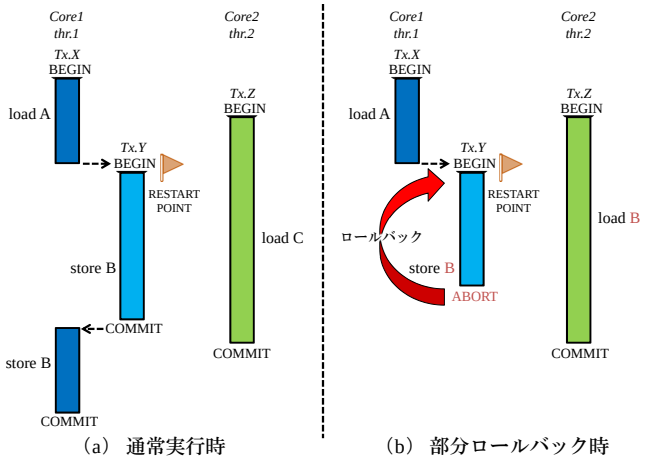


図 2　部分ロールバック適用時の動作例

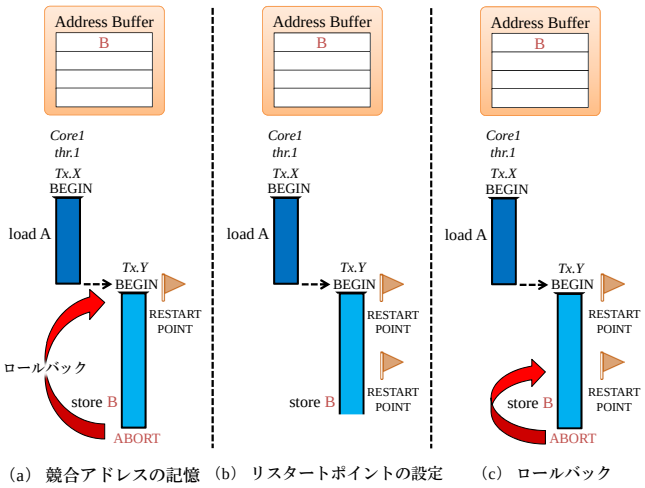


図 3　学習によるチェックポインティング手法

コミットに至ることができる．

2.3　データのバージョン管理

前節で述べたように，HTM におけるトランザクションの投機的実行では，各スレッドはデッドロックの回避のために自身の実行トランザクションをアボートしなければならない場合がある．このような場合に備えて，HTM ではデータのバージョン管理が行われる．こちらも以下の 2 つの方式に大別される．

Eager Version Management: 更新前の古い値をメモリ上の別領域に退避し，新しい値をメモリに上書きする．コミットは退避した値を破棄するだけであるため高速に行うことができるが，アボート時には，退避した値をメモリに書き戻す必要がある．

Lazy Version Management: 更新前の古い値をメモリに残し，新しい値をメモリ上の別領域に保持する．更新前の値がメモリに残っているためアボートは高速に行うことができるが，コミット時に，更新した値をメモリ上に反映させる必要がある．

これらのうち，一般的な HTM では必ず実行されるコミット処理を高速に行うことのできる Eager 方式が採用されている．しかし上記のとおり，この方式ではアボート処理に多くのコストを払う必要がある．

2.4　部分ロールバック

ここでは 2.3 節で述べたようなアボート処理のコストを抑えるために，HTM において従来から実装されている**部分ロールバック**[4] という手法について述べる．HTM ではトランザクションとして定義された領域内で，新たに別のトランザクションを入れ子状に定義することができる．このように入れ子状に定義された内側のトランザクションを内部トランザクションと呼び，外側のトランザクションの処理の一部として実行される．部分ロールバック手法では，この内部トランザクションの開始地点を，アボートされたトランザクションの再実行を始めるための候補地点（**リスタートポイント**）として設定する．このリスタートポイントが設定されている状態で，内部トランザクションがアボートされた場合，内部トランザクション開始前までに更新されたデータの書き戻しと，トランザクションの再実行のためのペナルティを削減することができる．

図 2 に部分ロールバック手法適用時の動作例を示す．これらの図において *thr.1*，*thr.2* はそれぞれ *Core1*，*Core2* 上で動作するスレッドを表しており，*thr.1* は *Tx.X* の実行を，*thr.2* は *Tx.Z* の実行をそれぞれ開始したとする．その後，*thr.1* が内部トランザクションとして *Tx.Y* の実行を開始したとすると，この内部トランザクション *Tx.Y* の開始地点にリスタートポイント（RESTART POINT）が設定される．この時，両スレッド間において競合が検出されずにトランザクションの実行が進む場合，図 2 (a) のようにそれぞれのトランザクションがコミットされる．一方で，*thr.1* が内部トランザクション *Tx.Y* の実行中に，*thr.2* と競合することで当該トランザクションをアボートする場合，図 2 (b) に示すように，*thr.1* は *Tx.Y* 開始地点に設定されたリスタートポイント時点のメモリ状態までロールバックするだけでトランザクションの再実行が可能となる．

また，この部分ロールバック手法に改良を施したものとして，学習によるチェックポインティング手法 [3] がある．先程も述べたように，従来の部分ロールバック手法ではアボート時に内部トランザクションの開始地点までロールバックして再実行を行う．しかし，その開始地点からアボートの原因となったメモリアクセスまでに，競合が発生し得ないような多くの操作が存在している場合，再実行すべき命令数も多くなってしまい，効率が悪い．

そこでこの学習によるチェックポインティング手法では，入れ子状態となった内部トランザクションの開始地点にリスタートポイントを設定することに加えて，過去に競合が発生したアドレスに対するメモリアクセスの直前にも

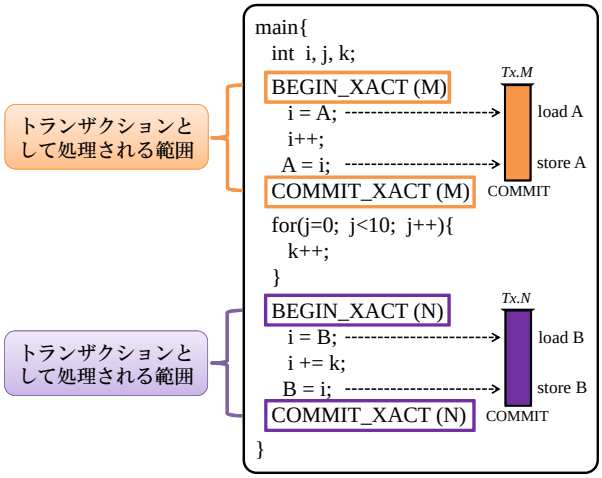


図 4　従来のプログラミングモデル

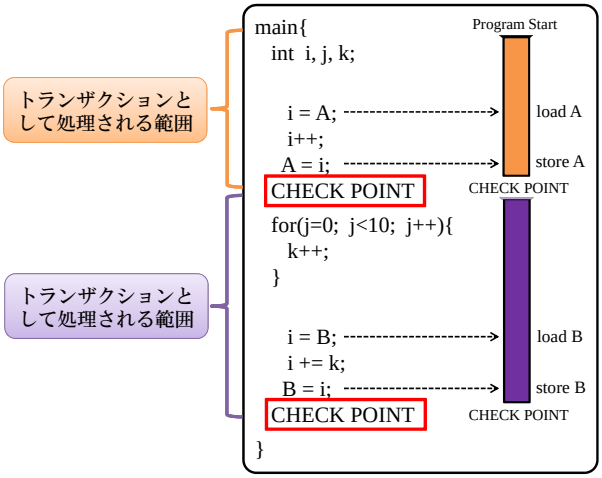


図 5　提案するプログラミングモデル

リスタートポイントを設定する．このように，リスタートポイントを設定することで，従来の部分ロールバック手法よりもアボート後の再実行命令数を削減できる．

図 2 の *thr.1* 上で実行されるトランザクションに対して，この手法を適用した場合の動作例を**図 3** に示す．これらの例において，Address Buffer はトランザクションのアボート時にその原因となったアドレスを保持するためのバッファを，そして RESTART POINT は再実行のためのリスタートポイントをそれぞれ示す．

まず図 3 (a) では，先程の図 2 の例と同様に，*thr.1* は内部トランザクションとして *Tx.Y* の実行を開始するとともに，*Tx.Y* の開始地点にリスタートポイントを設定する．そして，store B により発生する競合が原因となって *Tx.Y* がアボートされた場合，この時アクセスされたアドレスである B が Address Buffer に記憶される．その後，図 3 (b) のように *Tx.Y* が再実行され，Address Buffer に記憶されたアドレス B へのメモリアクセスである store B が実行される際，このアクセスの直前に新たにリスタートポイントが設定される．そして，この新たなリスタートポイントが設定された後に再び *Tx.Y* がアボートされた場合，図 3 (c) に示すように *thr.1* は *Tx.Y* の開始地点ではなく，新たに設定したリスタートポイントの位置からトランザクションを再実行することができる．

3. トランザクション定義単純化モデルの提案

本章では，HTM における既存のプログラミングモデルを利用するのみでは並列プログラムの記述が容易とはならないことを示し，そのような問題を解決する新たなプログラミングモデルを提案する．

3.1 提案モデルの基本方針

並列プログラミングにおいて HTM を利用する場合，プログラマはトランザクションとして定義すべき処理範囲をプログラム中に明示する必要がある．ここで，HTM を利用して従来通りにプログラムを記述した例を**図 4** に示す．この例における BEGIN_XACT() は，トランザクションの開始地点を表しており，COMMIT_XACT() はトランザクションの終了地点を表している．また，M と N はこの例で用いられているトランザクションの識別子をそれぞれ示している．この時，プログラマはクリティカルセクションとして扱うべき領域が単一のトランザクション内に含まれるように，トランザクションとして処理すべき領域を定義しなければならない．このように HTM における既存のプログラミング体系では，プログラム中にトランザクションの開始地点と終了地点の両方を明示しなければならない．そのため HTM を利用したとしても，プログラマはロックにおける排他制御すべき領域の指定と同様に，トランザクションとして扱うべき領域を指定する必要があるため，並列プログラムの記述が容易になるわけではない．

そこで本稿では HTM におけるトランザクションの定義を単純化し，プログラマが HTM を利用する際に，トランザクションとして処理すべき領域を指定せずとも，並列プログラムを記述することのできるプログラミングモデルを提案する．提案モデルでは，従来のトランザクション開始地点と終了地点の定義に代わる，**チェックポイント**という新たな定義を導入する．このチェックポイントは，プログラムの並行実行中にその時点までのメモリ状態を同期させたい地点に記述する．提案するプログラミングモデルを利用することで，プログラマはトランザクションとして処理すべき領域を指定するという概念ではなく，耐故障制御に用いられるようなチェックポインティングの概念に基づいた並列プログラミングを行うことができ，より簡潔に HTM を用いた並列プログラムを記述することが可能となる．

ここで，提案するプログラミングモデルに基づいて記述したプログラム例を**図 5** に示す．この例における CHECK POINT は，上述したチェックポイントを示している．こ

の例では，A = i; が実行された時点で共有変数 A に対する一連の操作が一旦完了するため，この A への書き込み操作の直後に最初のチェックポイントが記述される．また，同様に共有変数 B への書き込み操作の直後に次のチェックポイントが記述される．そして，各スレッドはプログラムの開始直後の部分から最初の CHECK POINT までの領域と，最初の CHECK POINT から次の CHECK POINT までの領域をそれぞれトランザクションとして実行し，これら以外の領域をトランザクション外の部分として実行する．

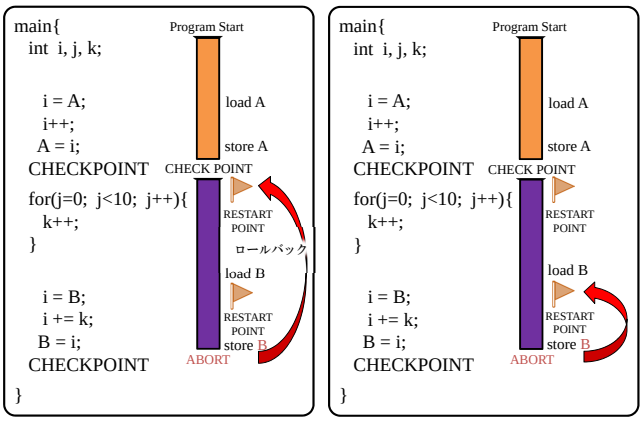
3.2 提案モデルの動作

ここでは，前節で示した図 4 および図 5 のプログラム例を比較することで，提案するプログラミングモデルを用いて並列プログラムを記述したとしても，実行するプログラムのセマンティクスには影響を及ぼさないことを確認する．なお，これらのプログラム例では A，B は共有変数であるとする．

図 4 の例では，BEGIN_XACT(M) によって最初のトランザクションが開始され，まず共有変数 A から値の読み出しを行う．そして，読み出した値にインクリメント操作をして再び A に値を書き込んだ後に，COMMIT_XACT(M) によってトランザクションを終了する．続けて，トランザクション外の部分である for 文の操作が行われた後に，BEGIN_XACT(N) によって次のトランザクションが開始される．このトランザクション中では，共有変数 B から値が読み出され，その値に対して局所変数 k の値を足しあわせたものが共有変数 B に書き込まれた後，COMMIT_XACT(N) によってこのトランザクションが終了される．

一方で図 5 の例では，まずメインプログラムの開始とともに最初のトランザクションが開始され，共有変数に対して図 4 の *Tx.1* と同様の操作が行われる．そして，実行がプログラム中の最初の CHECK POINT に到達すると，共有変数 A に対する一連の操作が完了するため，これまでの共有変数に対する実行結果がコミット処理によってメモリ上に反映され，最初のトランザクションの実行が終了される．その後，そのチェックポイントから先の操作が次のトランザクションとして実行される．次のトランザクションでは，その処理範囲の中に for 文による操作が含まれているが，この for 文の操作によって更新される値は局所変数のみであるため，共有変数に対する影響はない．その後，図 4 の *Tx.2* と同様の操作が行われた後にプログラムの実行が次の CHECK POINT に到達すると，最初のトランザクションの場合と同様にコミット処理が行われた後に，このトランザクションの実行が終了される．

上記の例から，提案モデルではトランザクション処理領域を明確に指定しないため，従来モデルではトランザクション外として扱われていた局所変数に対する操作も，実行トランザクション中に包含されることが分かる．このように，提案モデルにおいて拡大されたトランザクション領域に含まれる処理は，主に局所変数に対する処理である場合がほとんどであり，これはプログラムのセマンティクスには変化を引き起こさない．また，トランザクションで保護される必要のない共有変数に対する処理が存在する場合は，その処理も新たにトランザクション内に含まれることになるが，並列プログラムを記述する際にそのような記述をすることはほとんどない．仮にその処理が，提案モデルによってトランザクション中に包含されることになったとしても問題はなく，性能に与える影響も大きくないと考えられる．



(a) 提案モデルのみの場合　(b) 学習によるチェックポインティング手法を組み合わせた場合

図 6 学習によるチェックポインティング手法を組み合わせた場合の動作

3.3 提案モデルに対する部分ロールバックの適用

本提案モデルでは，前節で述べたように HTM におけるトランザクション定義を単純化しているが，従来トランザクション外として扱われていた領域がトランザクション内として扱われることになるため，アボート時の再実行ペナルティが大きくなってしまう．そこで，このペナルティの増大を抑制するために 2.4 節で述べた学習によるチェックポインティング手法を利用する．この手法では，リスタートポイントの設定される箇所がトランザクション開始地点に限定されないため，今回我々が提案する，チェックポインティングの概念に基づいたプログラミングモデルに対し，親和性が高い．すなわち，提案するプログラミングモデルとこのチェックポインティング手法を組み合わせることにより，提案モデルにおいてトランザクションとして処理すべき領域が拡大することに起因する再実行ペナルティの増大を抑制することが可能になると考えられる．

ここで，提案モデルのみを用いた場合と提案モデルに学習によるチェックポインティング手法を適用した場合のそれぞれにおける，ロールバックの動作を**図 6** に示す．まず図 6 (a) では，最初のチェックポイントの位置にリスタートポイントが設定される．その後にトランザクションがア

ボートされたとすると，このリスタートポイントが設定されているチェックポイントの位置までロールバック処理が行われてから，トランザクションが再実行される．しかしこれでは，局所変数に対する処理である for 文も再実行する必要があるため，ペナルティが大きい．

これに対して図 6 (b) では，図 6 (a) と同様に最初のチェックポイントの位置にリスタートポイントを設定することに加えて，アボートの原因となった共有変数 B への Write アクセスの直前にもリスタートポイントが設定される．したがって，トランザクションがアボートされた際には，このリスタートポイントの設定されている Write アクセスの直前の位置までロールバック処理が行われてから，トランザクションが再実行される．このように，学習によるチェックポインティング手法を提案モデルに組み合わせた場合には，従来のトランザクション開始地点以降の箇所にリスタートポイントが設定されることになるため，トランザクション定義を単純化したことによる不利益を解消することができ，提案モデルのみを用いた場合に増大する，再実行時のペナルティを抑制することができる．

4. 評価

本章では，これまで述べた提案モデルを評価し，従来のプログラミングモデルやロックとの性能比較をした上で，提案モデルの今後の改良方針について述べる．

4.1 評価環境

提案モデルを HTM の研究で広く用いられている LogTM に適用し，シミュレーションによる評価を行った．評価には Simics[5] 3.0.31 と GEMS[6] 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には GEMS 付属 microbench，SPLASH-2[7] および STAMP[8] から計 7 個を使用した．なお，各コアが 1 スレッドを実行するため，プロセッサ全体で 32 スレッド実行となるが，OS 用に 1 コアを使用するため，31 スレッドで評価した．ただし，STAMP は 2 の冪乗数のスレッド数でしか動作しないため，STAMP に限り 16 スレッドで評価した．

4.2 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が 5 本のバーで表されており，これらのバーは左から順に，

(L) ロック： クリティカルセクションを逐次実行するモデル．

(T) 既存モデル： クリティカルセクションをトランザクションとして並列実行するモデル．

(M) 提案モデル： 提案するプログラミングモデルを用いてトランザクションを並列実行するモデル．

(P) 提案モデル＋チェックポインティング手法： 提案するプログラミングモデルと学習によるチェックポインティング手法を組み合わせて並列実行するモデル．

の実行サイクル数の平均を表しており，ロック (L) の実行サイクル数を 1 として正規化している．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する必要がある [9]．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．この信頼区間はグラフ中にエラーバーで示す．

評価結果から，ほぼ全てのベンチマークプログラムにおいて提案モデルを利用した (M) と (P) の実行サイクル数が，ロック (L) の実行サイクル数よりも削減されていることが分かる．ここで，各モデルのロック (L) に対する実行サイクル削減率の最大と平均を表 2 に示す．この表から，提案モデルを利用した (M) と (P) はトランザクションを定義するモデル (T) と比較して，平均で 5〜6%近く性能が劣るものの，ロックよりも十分高い性能が達成できることが確認できた．

表 1 シミュレータ諸元

Processor	SPARC V9
#cores	32 cores
clock	1 GHz
issue width	single
issue order	in-order
non-memory IPC	1
D1 cache	32 KBytes
ways	4 ways
latency	1 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	8 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

表 2 削減サイクル率

	最大	平均
(T)	97.2%	32.6%
(M)	97.5%	27.6%
(P)	96.7%	26.5%

4.3 考察

ここでは，前節で得られた評価結果から詳細な考察を行う．まず，提案モデル (M) は Btree, Contention, Prioque, Slist の 4 つのベンチマークプログラムにおいて，既存モデ

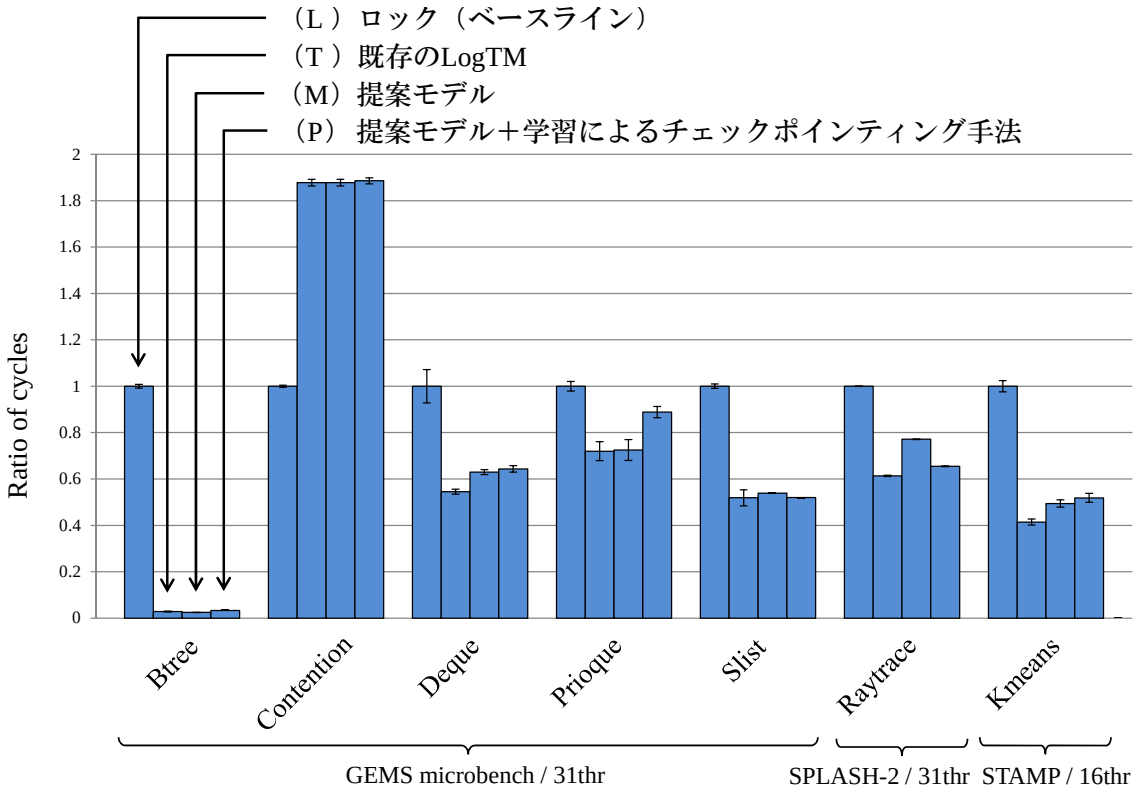


図 7 各モデルにおける実行サイクル数比

表 3 アボート回数

	Btree	Contention	Deque	Prioque	Slist	Raytrace	Kmeans
(M)	364	13,756	154	38,023	4,164	443,459	241
(P)	433	13,762	187	44,594	4,036	432,727	246

ル (T) とほぼ同じ性能となった．これらのベンチマークプログラムでは，従来のトランザクション定義を単純化することで，個々のトランザクションの処理領域に含まれることとなるトランザクション外の処理は少ない．そのため，アボート処理のコストや再実行のペナルティが増大することがなく，性能低下を引き起こさなかった．一方で Deque，Raytrace，Kmeans の 3 つのプログラムでは，提案モデル (M) は既存モデル (T) と比較して，その性能がある程度低下していることが分かる．これらのプログラム中で定義されていたトランザクション処理領域は，既存モデル (T) では非常に短いものであったが，提案モデル (M) ではトランザクション定義を単純化することで，その領域中にに多くのトランザクション外の処理が含まれていた．そのため，トランザクションをアボートした際のオーバヘッドが増大し，性能が低下してしまったと考えられる．

次に，提案モデルに学習によるチェックポインティング手法を組み合わせたモデル (P) について見ていく．このモデルでは，提案モデル (M) においてトランザクション処理領域の拡大によって性能が低下した Deque，Raytrace，Kmeans の 3 つのプログラムの性能が既存モデル (T) と同程度まで改善すると予想していたが，これらのうちで (M) よりも性能向上したプログラムは Raytrace のみであった．ここで，性能向上しなかった Deque や Kmeans では，トランザクションとして扱われる領域が拡大したことにより，一度のアボートで発生するペナルティが大きくなる一方で，部分ロールバックが行われた回数自体は少なかったため，その性能にほぼ変化がなかったと考えられる．一方で，Prioque では (P) は (M) に対して大きく性能低下している．この原因を特定するために，(P) と (M) で実行した場合の各ベンチマークプログラムにおけるアボート回数を計測した．計測結果を**表 3** に示す．この表から，性能が低下した Prioque ではそのアボート回数が大きく増加していることが分かる．

ここで学習によるチェックポインティング手法を用いた場合に，アボート回数が増加してしまう原因を考察する．この手法では，トランザクションがアボートされると，そのアボートの原因となったアドレスに対するメモリアクセスの直前からトランザクションが再実行される．そのため，再実行の直後に同じアドレスで即座に競合が再発し，アボートが繰り返されてしまう可能性が高い．HTM では一般に，アボートされたトランザクションに対して待機時間を設定するアルゴリズムとして，exponential backoff が用いられているため，このようなアボートの繰り返しによって非常に長い待機時間が設定されるようになってしまう．したがって，このアボートの繰り返しは，学習による

チェックポインティング手法が抱える根本的な問題となっており，これを解決しない限り，性能向上が見込めない可能性がある．

4.4 今後の方針

前節で述べたように，今回提案モデルに組み合わせた学習によるチェックポインティング手法では，競合したアドレスに対するメモリアクセスの直前からトランザクションが再実行されて競合が頻発してしまうことが根本的な問題となっている．この原因は，トランザクションがアボートされ，ロールバック処理が行われた後に即座に再実行されることにあり，再実行前に適切な待機時間を設けることが非常に重要である．HTM では，exponential backoff を始めとする待機アルゴリズムが用いられているが，これらのアルゴリズムでは必ずしも適切な待機時間を設定できていないことが改めて確認できた．我々は現在，トランザクションに適切な待機時間を設定する研究を別途進めており [10]，その研究で得た知見を本研究にも取り入れることで，本提案であるプログラミングモデルを用いた場合においても従来記法と同等以上の性能を実現すべく改良を進めていきたい．

また，これと併せて，今回提案したプログラミングモデルで導入したチェックポイントをも排除することで，HTM による並列プログラムの記述をさらに容易とすることも，今後検討していきたいと考えている．このチェックポイントの排除を実現するために，まず実行するプログラムを静的に解析し，プログラム中でクリティカルセクションとして処理されるべき領域を検出するという方法が考えられる．しかし，一般的にクリティカルセクションとして扱うべき処理領域は，その領域を含むプログラムの処理内容に依存するため，トランザクションとして処理すべき領域を単純なプログラム解析により定義することは困難であると考えられる．

よって，プログラム中で使用される共有変数間の依存関係に関する情報など，従来のトランザクション定義よりも簡潔な記述によって，トランザクションとして扱うべき処理領域を正確に検出する方法を検討していきたい．

5. おわりに

本稿では，HTM における従来のトランザクション定義を単純化した，新たなプログラミングモデルを提案した．提案モデルでは，HTM で従来より用いられていたトランザクションの開始地点と終了地点の定義を単純化し，チェックポイントという新たな定義を導入した．提案するプログラミングモデルを利用することで，プログラマはトランザクションとして処理すべき領域を指定するという概念ではなく，チェックポインティングの概念に基づいた並列プログラミングを行うことができ，より簡潔に HTM を用いた並列プログラムを記述することが可能となる．

GEMS microbench，SPLASH-2 および STAMP ベンチマークより 7 個のプログラムを使用し，上記の提案モデルの評価を行った．評価結果より，提案モデルはロックよりも速度性能が優れており，かつ従来のトランザクション定義を用いたプログラミングモデルと比較しても大きく性能低下しないことが確認できた．

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] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995).
[3] Waliullah, M. M. and Stenstrom, P.: Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems, *Proc. 22nd IEEE Int'l Symp. on Parallel and Distributed Processing (IPDPS 2008)*, pp. 1–11 (2008).
[4] 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).
[5] Magnusson, P. S. et al.: Simics: A Full System Simulation Platform, *Computer*, Vol. 35, No. 2, pp. 50–58 (2002).
[6] 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).
[7] 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).
[8] Minh, C. C. et al.: STAMP: Stanford Transactional Applications for Multi-Processing, *Proc. IEEE Int'l Symp. on Workload Characterization (IISWC'08)* (2008).
[9] 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).
[10] 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，Vol. 6, No. 3(ACS44)（採録決定）(2013).