卒業研究論文

トランザクション定義を簡略化した ハードウェア・トランザクショナル・メモリの 性能評価

指導教員　　津邑 公暁 准教授
松尾 啓志 教授

名古屋工業大学 工学部 情報工学科
平成 21 年度入学　21115078 番

鈴木 大輝

平成 25 年 2 月 12 日

トランザクション定義を簡略化した ハードウェア・トランザクショナル・メモリの性能評価

鈴木 大輝

内容梗概

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の調停にロックが広く用いられている．しかしロックを用いた場合には，デッドロックの発生や並列性の低下といった問題がある．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 (TM) が提案されている．TM は，データベースにおけるトランザクション処理をメモリアクセスに適用したものであり，複数のトランザクションによる同一アドレスへのアクセスによって競合が発生するまで，トランザクションを投機的に実行できるため，ロックを用いた場合よりも並列性が向上する．また，TM では競合の発生を検知したときに，実行中のトランザクションの整合性が保てない場合，あるいはデッドロック状態に陥りそうな場合に，途中結果を破棄し，そのトランザクションの実行を初めからやり直すことができる．このため，プログラマはこれらの問題を考慮せずに，並列プログラムを記述することができる．なお，この TM をハードウェア上に実装したものをハードウェア・トランザクショナル・メモリ (HTM) と呼ぶ．

さて，これまでに多くの研究により HTM のロックに対する処理性能の優位性が示されてきたが，その性能評価には，本来ならばクリティカルセクションとして扱うべき領域を，トランザクションとして定義したプログラムが用いられてきた．このため，TM の特徴であるプログラマビリティの高さを活かしつつ，HTM がロックよりも高速に並列実行可能であるか否か確認されていない．本論文では，この 2 つの優位性が両立可能であるかを検証するために，HTM のプログラマビリティを高めたプログラミングモデルを提案し，かつ，そのモデルの適用により予想される性能低下を抑制する手法を提案する．提案手法の有効性を検証するため，HTM の実装の一つである LogTM に実装し，シミュレーションによる評価を行った．GEMS microbench，SPLASH-2，および STAMP の 3 種のベンチマークプログラムを用いて評価した結果，ロックと比べて最大で 96.7%，平均で 26.5%実行サイクル数を削減し，処理性能を向上させることができた．この結果，提案手法を用いることで，従来の HTM を用いたプログラミングモデルよりも高いプログラマビリティと，ロックよりも高い並列実行性能の両方を同時に実現可能であることが確認された．

トランザクション定義を簡略化した
ハードウェア・トランザクショナル・メモリの性能評価

目次

1 はじめに **1**

2 ハードウェア・トランザクショナル・メモリ **2**

2.1 ハードウェア・トランザクショナル・メモリの概要 2

2.2 データのバージョン管理 4

2.3 部分ロールバック 6

3 研究動機と目的 **8**

4 トランザクショナル・メモリにおけるプログラマビリティの向上 **10**

4.1 プログラマビリティを高めたプログラミングモデル 11

4.2 性能低下の抑制 13

4.2.1 部分ロールバック適用の検討 13

4.2.2 競合アドレスを利用した部分ロールバック先学習法の適用 . . 14

4.2.3 実装に必要なハードウェアコスト 16

5 提案手法のロックに対する性能評価 **18**

5.1 ロックに対する提案手法の優位性の評価 18

5.2 部分ロールバック先学習法による性能低下抑制の評価 20

6 チェックポイントの排除に向けた方針 **22**

7 おわりに **23**

著者発表論文 **24**

参考文献 **24**

1 はじめに

マルチコア環境におけ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．このようなプログラミングモデルでは，共有リソースへのアクセスを調停する必要があり，その調停を行う機構として一般的にロックが用いられている．しかし，ロックを用いたプログラミングでは，デッドロックの発生を考慮する必要があり，また各プログラムで適切なロックの粒度を設定しなければ並列性を向上させるのは困難である．したがって，ロックはプログラマにとって必ずしも利用しやすい機構ではない．

そこで，ロックに代わる新たな並行性制御機構として，**トランザクショナル・メモリ (TM)** [1] が提案されている．TM は，データベースにおけるトランザクション処理をメモリアクセスに適用したものであり，複数トランザクションによる同一アドレスへのアクセスにより，トランザクションの整合性が保てなくなってしまう場合，そのアクセスを競合として検出する．競合を検出した場合，いくつのトランザクションの実行を停止する．これをストールという．さらに，複数のトランザクションがストールした場合，それらのトランザクションがお互いにストールさせ合い，デッドロック状態に陥る可能性がある．このような場合，そのうちのいずれかのトランザクションの途中結果を破棄する．これをアボートという．また，競合が検出されずにトランザクション処理を完了する場合，実行結果を元のメモリアドレスへ反映させる．これをコミットという．TM はこのように動作することで，競合が発生しない限りトランザクションを並列に実行することができる．また，この TM をハードウェア上に実現したものをハードウェア・トランザクショナル・メモリ (HTM) と呼ぶ．HTM では一般的に，各キャッシュラインに対してトランザクション内で発生したリードおよびライトアクセスの有無を記憶するための領域が追加されている．そして，キャッシュコヒーレンスリクエストを受け取ったときにこれらの領域を参照することで，競合検出を実現する．また，前述したメモリアクセスに対する操作により，デッドロック状態を回避できるため，プログラマビリティにおいてロックよりも優れた性質を持つ．

さて，これまで多くの研究により，並行性制御機構として HTM を用いることで，多くの場合においてロックよりも高速に並列処理可能であることが示されてきた．また HTM による並行性制御の高速化手法も提案されており，その優位性が注目されている．一方，HTM の特徴である，プログラマビリティの高さにおける優位性は，まだほとんど検証されておらず，HTM がロックよりもプログラマビリティに優れ，かつ高速

に並列実行可能であるかはまだ明らかではない.

そこで本論文では，そのプログラマビリティを高めた上で処理性能の優位性を再検証し，HTM がロックよりも利用しやすい機構であるかどうか確認する．そのために，トランザクションの定義を簡略化することで HTM を用いたプログラミングモデルのプログラマビリティを高め，併せて，このトランザクション定義の簡略化により発生しうる性能低下を予め抑制する手法を提案する．そして，この提案手法とロック機構，および従来のトランザクション定義を用いた HTM の処理性能を比較する．これにより，HTM を用いることで，ロックよりも高いプログラマビリティと高い並列実行性能の両立を実現可能かどうか検証する.

以下，2 章では本研究の研究対象である HTM について説明する．次に 3 章で HTM のロックに対する優位性について検証し，4 章ではに 3 章で得られた知見に基づいた提案手法について述べる．そして，5 章では提案手法を評価し，6 章では評価に基づいて，今後の研究方針について述べる．最後に，7 章で結論を述べる.

2　ハードウェア・トランザクショナル・メモリ

本章では, 本研究の対象となる HTM の概要について述べる.

2.1　ハードウェア・トランザクショナル・メモリの概要

マルチコア・プロセッサにおける並列プログラミングでは，複数のプロセッサ・コアが単一アドレス空間を共有する．したがって，異なるプロセッサ・コアによる同一メモリアドレスに対するアクセスを調停する必要があり，その調停を行う機構として一般的にロックが用いられている．しかし，ロックを用いたアクセス制御ではデッドロックが発生する可能性がある．そのため，プログラマはロックを用いた並列プログラムを記述する際に，デッドロックの発生を考慮しなければならない．また，並列に実行するスレッド数や使用するロック変数自体が増加した場合，ロックの獲得・解放操作に要するオーバヘッドも増加し，性能低下してしまう可能性がある．さらに，プログラムごとに適切なロックの粒度を設定することは難しい．例えば，粗粒度のロックを用いる場合では，プログラムの構築は容易であるがクリティカルセクションが大きくなるため並列性は損なわれる．一方細粒度のロックを用いる場合では，並列性は向上するが大規模なプログラムであるほど設計が複雑となる．以上のような特徴は，ロックを用いたプログラム設計が困難である要因となっている.

そこで，ロックを用いない並行性制御機構であるトランザクショナル・メモリ (TM)

が提案されている．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対して適用した手法である． またこの機構では，クリティカルセクションを含む一連の命令列が**トランザクション**として定義され，トランザクションは以下の 2 つの性質を満たす.

シリアライザビリティ(直列可能性):
: 並行実行されたトランザクションの実行結果は，当該トランザクションを直列，すなわち逐次的に実行した場合と同じであり，全てのスレッドにおいて同一の順序で観測される.

アトミシティ(不可分性):
: トランザクション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ず，各トランザクション内における操作は，トランザクションの終了と同時に観測される．そのため，操作の途中経過が他のスレッドから観測されることはない.

以上の性質を保証するために，TM はトランザクション内のメモリアクセスを監視する．あるトランザクション内でアクセスされたメモリアドレスと，他のトランザクション内でアクセスされたメモリアドレスが同一であった場合，実行されているトランザクションの整合性が保たれなくなることがある．このとき，これを**競合**として検出する．競合を検出した場合，片方のトランザクションの実行を停止する．これを**ストール**という．さらに，複数のトランザクションがストールした場合，それらのトランザクションがお互いにストールさせ合い，デッドロックのような状態に陥る可能性がある．このような場合，そのうちのいずれかのトランザクションの途中結果を破棄する**アボート**を行う．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，メモリおよびレジスタの状態をトランザクション開始時点の状態に戻す．この一連の処理を**ロールバック**という．また，競合が検出されずにトランザクション処理を完了する場合，実行結果を元のメモリアドレスへ反映させる．これを**コミット**という.

TM はこのように動作することで，ロックによる排他制御と同等のセマンティクスを維持しつつ，競合が発生するまでトランザクションを並列に実行することができる．これにより排他制御を行う場合よりもプログラムの並列性が向上するため，コア数に応じて性能がスケールすることが期待できる．また，TM で行われるこのような操作は，ハードウェア上あるいはソフトウェア上に実装される．このとき，ハードウェア上に実装された TM 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．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．一方，ソフトウェア上に実装された TM はソフトウェア・トランザクショナル・メモリ (STM) [2] と呼ばれる．STM では，HTM のように特別なハードウェア拡張を行う必要がない代わりに，TM 上で行われる操作は全てソフトウェア上で実現される．このため，HTM は STM と比較してこの操作に対するオーバヘッドが小さく，性能が高い．よって，本論文では HTM を対象とする．

2.2 データのバージョン管理

トランザクションの投機実行では，その失敗時にメモリの状態をトランザクション実行開始時の状態に戻さなくてはならない．このため，投機実行中に共有メモリのデータを更新する際には，更新前の値も保持しておく必要がある．そこで HTM では，トランザクション内で発生したストアアクセスにより更新したデータ，あるいは更新前のデータを，そのアドレスとともに別領域に保持する．このバージョン管理は，以下の 2 つの方式に大別される．

Eager Version Management:
書き換え前の古い値を別の補助記憶領域にバックアップし，新しい値をメモリに上書きする．コミットはバックアップを破棄するだけなので高速に行えるが，アボート時にはバックアップされた値をメモリにリストアする必要がある．

Lazy Version Management:
書き換え前の古い値をメモリに残し，新しい値を別の補助記憶領域に登録する．アボートは高速に行えるが，コミット時にメモリへの値のコピーが必要となる．

これら 2 つの方式のバージョン管理における，メモリおよび補助記憶領域の状態変化を**図 1** と**図 2** を用いて示す．これらの図中の Memory はメモリを表し，Assistant Memory は補助記憶領域を表す．

まず，メモリアドレス `0x100` に値 10 が格納されている状態で，トランザクションの実行が開始されたとする．その後，トランザクションの実行が進み，アドレス `0x100` へ値 15 の書き込みが行われるとする．すると，図 1(b) のように，Eager 方式では Memory の書き換え前に，ストアアクセスされたアドレス `0x100` と書き換え前の値である 10 が Memory から Assistant Memory へバックアップされ，ストアの結果である 15 が Memory に上書きされる．一方，Lazy 方式では，アドレス `0x100` とストアの結果である 15 が Assistant Memory に登録され，Memory の値は上書きされない．

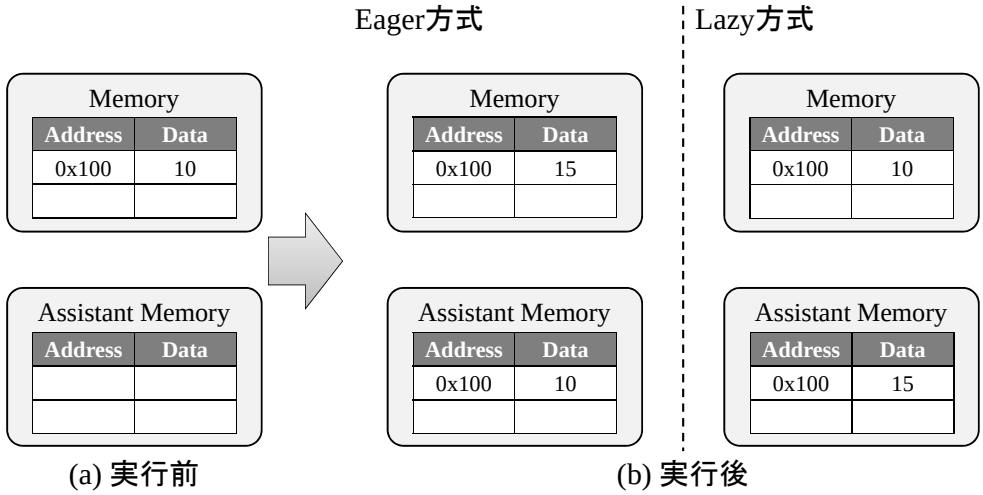

図 1: store 実行

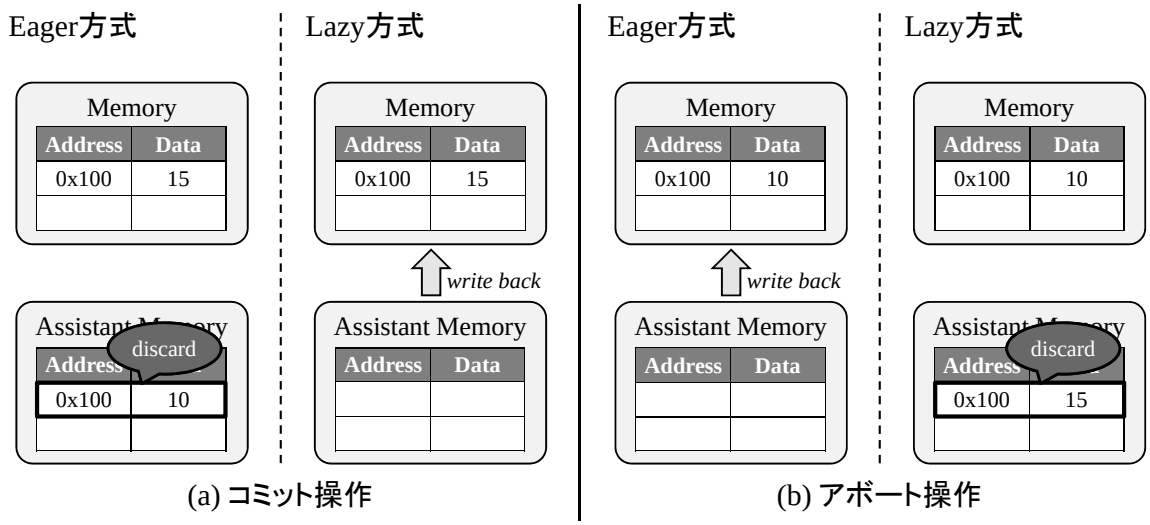

図 2: 投機実行成功または失敗時の操作

次に図 1 の状態からさらに実行が進み，投機実行が成功した場合には，トランザクションがコミットされる．Eager 方式の場合，ストアの結果である値 15 は既に Memory に保持されているため，図 2(a) に示すように Assistant Memory の内容を破棄するだけでコミットを実現できる．しかし Lazy 方式では，同図に示すように，Assistant Memory に保持されているストアの結果を Memory に上書きしなければならない．

一方で，投機実行が失敗した場合，トランザクションはアボートされる．このとき，Eager 方式では，図 2(b) に示すように，Assistant Memory に保持されている書き換え前の値 10 が Memory に書き戻される．これに対し，Lazy 方式は Assistant Memory に

保持されているアドレスと値を破棄するだけで，アボート操作を実現できる．このようにして，それぞれの方式ではトランザクション開始時の Memory の状態が復元される．

以上のように，Lazy 方式ではアボート操作が高速に処理されるのに対し，Eager 方式では，必ず行われるコミット操作が高速に処理される．そのため，Eager 方式ではアボートが繰り返し発生するようなプログラムでは処理が遅くなる場合もある．しかし，Lazy 方式におけるコミットのオーバヘッドは削減の余地が無いのに対し，Eager 方式では，スケジューリングによって競合やアボートの発生自体を抑えることで，アボートのオーバヘッドを削減できると考えられる．以上の理由から，本論文では Eager 方式のバージョン管理を用いる．

なお，後述する提案手法で用いる既存手法にしたがい，本論文ではアボート対象となるトランザクションの選択に，タイムスタンプ方式を用いる．この方式では，競合検出の際に各トランザクションの開始時刻を比較する．そして自スレッドが実行しているトランザクションの開始時刻が，競合相手のスレッドが実行しているトランザクションの開始時刻よりも遅い場合，自スレッドのトランザクションをアボートする．

2.3 部分ロールバック

TM では，投機実行に失敗しアボート操作を行った後，トランザクションを開始状態までロールバックし，再び投機実行を開始する．このとき，メモリの状態をトランザクション開始時の状態に戻すコストと，一度実行した命令を再実行するコストがかかる．これらのコストを削減するための手法として，部分ロールバック [3] が提案されている．**図 3** に示すように，TM ではトランザクションとして定義される領域内に，さらにトランザクションを定義することができる．このトランザクションを内部トランザクションとする．このとき，図中で使用されている `BEGIN_XACT(X)` は識別番号が `X` であるトランザクションの開始地点を，`COMMIT_XACT(X)` は同トランザクションの終了地点を指示するためのものである．

部分ロールバックでは，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とする．例えば，アボートの原因となった命令が内部トランザクションの中にある場合，このトランザクションの開始状態にロールバックする．よって，内部トランザクション開始前までに更新された値の書き戻しと，それらの更新を再実行するのにかかるコストを削減することができる．図 3 の例では，`x = 1;` の再実行および `x` の値の書き戻しの必要がなくなる．

この部分ロールバックの動作例を**図 4** に示す．この図では，2 つのスレッド Thread1

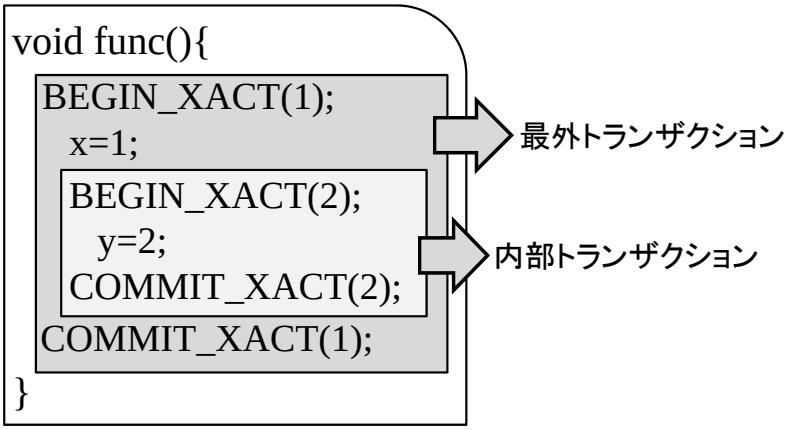

図 3: 部分ロールバックを用いるプログラム例

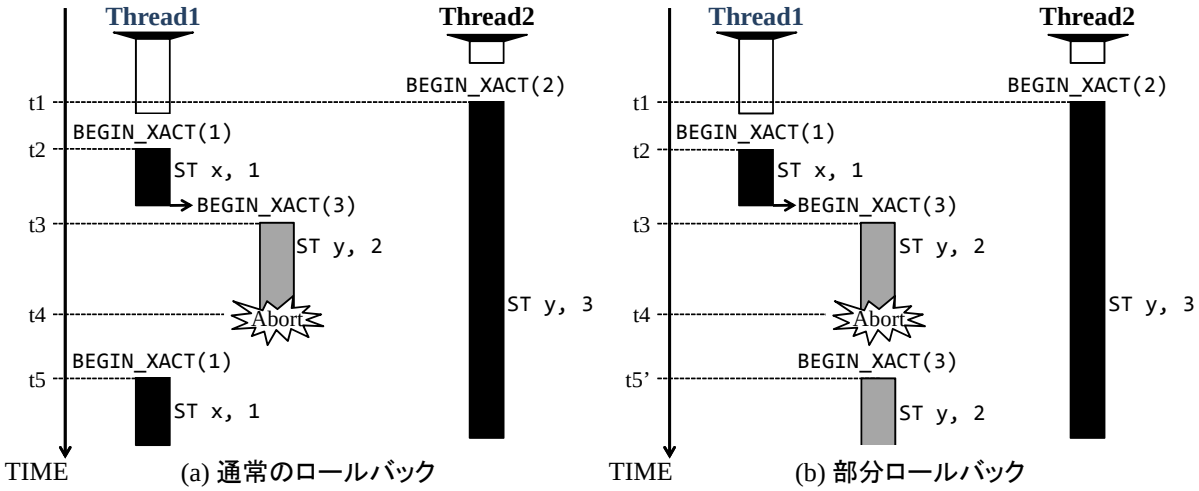

図 4: 部分ロールバック

と Thread2 が，トランザクションを投機的に実行している様子を表しており，下向きの軸が時間経過を表している．なお，以降では `BEGIN_XACT(X)` から開始されるトランザクションを Tx.`X` と表す．この例ではまず，Thread2 が Tx.2 の処理を開始した (時刻 t1) 後に，Thread1 が Tx.1 を開始している (t2)．次に．Thread1 がアドレス `x` へのストア命令 `ST x, 1` を実行し，続けてネストされたトランザクションである Tx.3 を開始する (t3)．そして，Thread1，Thread2 の順にアドレス y へ値を書き込むとする．このとき，Thread1 がアドレス y へ先にストアアクセスするため，Thread2 のアドレス y へのアクセス時に競合が発生する．競合を検出した Thread1 は，自身のトランザクション開始時刻と Thread2 のトランザクション開始時刻とを比較する．この例では，Thread2 の方がトランザクション開始時刻が早いため，Thread1 は実行中の Tx.3 をア

ボートする (t4)．通常のロールバックでは，図 4(a) のように，最外トランザクションである Tx.1 の開始状態にロールバックする (t5)．このため，Tx.3 の開始前に Tx.1 で実行された `ST x，1` によって更新された値も書き戻さなくてはならない．これに対し，部分ロールバックでは図 4(b) のように，Tx.3 の開始時の状態にロールバックする (t5')．これにより，`ST x，1` によって更新された値は書き戻されないため，通常のロールバックと比べると，アドレス `x` の指す領域への値の書き戻しと `ST x，1` の再実行にかかるコストが削減される．

3　研究動機と目的

本論文ではまず，HTM のロックに対する処理性能の優位性を確認するために，シミュレーションによる実行サイクル数の計測を行った．シミュレータには，トランザクショナルメモリの研究で広く用いられている，Simics[4] 3.0.31 と GEMS[5] 2.1.1 の組み合わせを用いた．**表 1** に詳細なシミュレータ構成を示す．また，評価対象のプログラムとしては，GEMS 付属の microbench から Btree，Contention，Deque，Prioqueue，Slist，マルチプロセッサ用のベンチマークである SPLASH-2[6] から Raytrace，TM 評価用のベンチマークである STAMP[7] から Kmeans の計 7 種のベンチマークプログラムを用い，それぞれのプログラムを 16 または 31 スレッドで実行した．

なお，各コアが 1 スレッド実行するため全体で 32 スレッド実行となるが，これらのコアの内，1 つが OS 用に占有されるため，31 スレッドによる評価を行った．ただし，STAMP ベンチマークは 2 のべき乗数のスレッドでしか動作しないため，このベンチマークに限り 16 スレッドで評価した．実行に用いた入力パラメータを**表 2** に示す．

図 5 は，HTM の実装の一つである LogTM[8] を用いて評価した各ベンチマークプログラムの実行サイクル数を，ロックを用いた場合の実行サイクル数で正規化したグラフである．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 [9]．したがって，各評価対象につき 10 回試行し，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．

図中の，7 つのベンチマークプログラムのうち 6 つのプログラムで，ロックよりも HTM の方が実行サイクル数が少ないという結果が得られた．また，評価結果から算出したロックに対する HTM の平均サイクル削減率は，約 32.6 %であった．この結果から，HTM はロックに対して処理性能において優位であることが認められた．

さて，この性能評価に用いたベンチマークプログラムは，ロックを用いる場合には

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
IPC	non-memoryIPC=1
D1 cache	32 KBytes
ways	4 ways
latency	3 cycle
D2 cache	8 MBytes
ways	8 ways
latency	20 cycles
Memory	4 GBytes
latency	450 cycles
Interconnect network latency	14 cycles

表 2: ベンチマークプログラムの入力パラメータ

GEMS	
Btree	priv-alloc-20pct
Contention	config 1
Deque	1024ops 32bkoff
Prioque	8192ops
Slist	500ops 64len
SPLASH2	
Raytrace	teapot
STAMP	
Kmeans	random-n2048-d16-c16.txt

クリティカルセクションとして定義される領域を，そのままトランザクションとして定義したものを用いている．このため，TM の特徴であるプログラマビリティの高さ

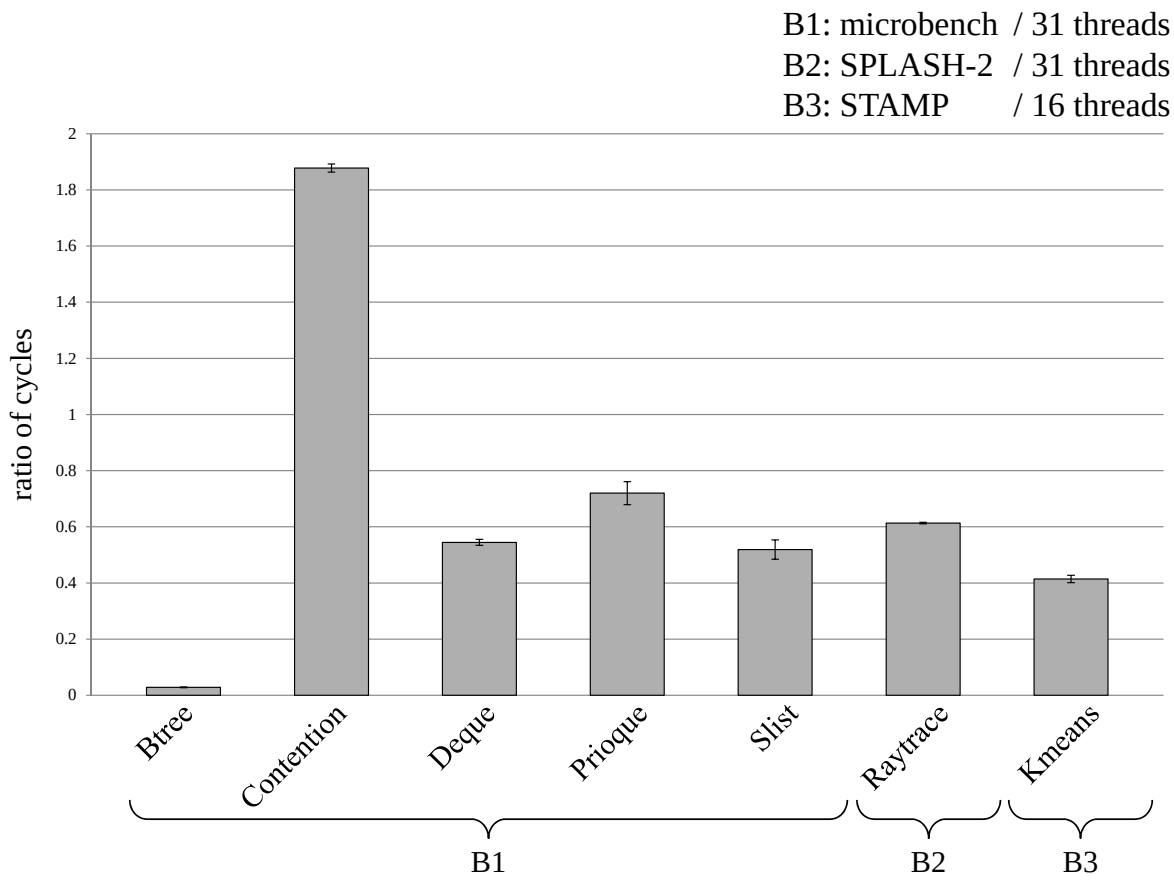

図 5: ロックに対する LogTM のサイクル数比

における優位性は検証されているとは言えず，この優位性とロックに対する HTM の処理性能の優位性が両立できることは確認されていない．そこで本論文では，トランザクションの開始と終了の定義を簡略化することにより，プログラマビリティを高めたプログラミングモデルを提案する．併せて，このプログラミングモデルの適用により発生すると予想される性能低下を，部分ロールバックを組み合わせて抑制する手法を提案する．そして，この提案手法と，ロック機構および従来のプログラミングモデルによる HTM の処理性能を比較する．これにより，得られた評価結果から，HTM を用いることで，ロックよりも優れたプログラマビリティと高い速度性能の両立が実現可能かどうか検証する．

4　トランザクショナル・メモリにおけるプログラマビリティの向上

本章では, 前章で述べた研究目的を達成するために, トランザクション定義を簡略化することでプログラマビリティを高め，併せてそれにより発生すると予想される性能

低下の抑制手法を提案する．

4.1 プログラマビリティを高めたプログラミングモデル

従来の HTM を用いた並列プログラムでは，トランザクションの開始およびトランザクションの終了をプログラム中に明示しなければならない．ここで，**図 6**(a) に従来の HTM を用いたプログラムの例を示す．このプログラム中の 3，8，12 行目の記述がトランザクションの開始を表し，5，10，14 行目の記述がトランザクションの終了を表す．このとき，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，厳密にトランザクションの処理範囲を定義しなければならない．このため，ロックを用いる場合と比較してプログラマビリティにおいて優れているとは言い難い．これは，トランザクションの開始と終了の二つを明示しなければならないことに起因する．そこで，提案するプログラミングモデルではこれらの定義を簡略化した上で，トランザクションとして処理する範囲を示すための新たな定義を設ける．この定義は，同一ブロック上におけるトランザクション間，あるいは非トランザクションとトランザクション間の境界を示す．なお，ブロックとは C 言語で用いられるブロックと同義とする．C 言語で定義するブロックは，中括弧で囲まれた領域であり，このブロックにより変数のスコープが決定される．また，ブロックは入れ子にすることもできる．

本論文では，提案するプログラミングモデルで用いる，トランザクションの開始および終了の定義に代わる新たな定義を**チェックポイント**と呼ぶ．なお，以降の図中ではこのチェックポイントを `CHECKPOINT(X)` と表し，この式で識別番号が `X` であるトランザクションの境界点であることを指示する．各コアはチェックポイントが宣言されたブロックに記述されている最初の式から最初のチェックポイントまで (図 6(b)2 行目から 4 行目までと 9 行目) と，あるチェックポイントから次のチェックポイントまで (6 行目から 13 行目まで) をそれぞれトランザクションとして処理する．また，上記の 2 種類以外の領域は非トランザクションとして処理する．なお，あるブロック内に別のブロックが存在し，両ブロック上でチェックポイントがそれぞれ記述されている場合，トランザクションとして処理する領域内に，別のトランザクションとして処理する領域が生じる．スレッドはこの領域をネストされたトランザクションとして処理する．図 6(b) の例では，スレッドは 9 行目をネストされたトランザクションとして処理する．

一般に，クリティカルセクションとして扱うべき領域は，トランザクションとして扱うべき領域の部分領域である．したがって，提案するプログラミングモデルでは，

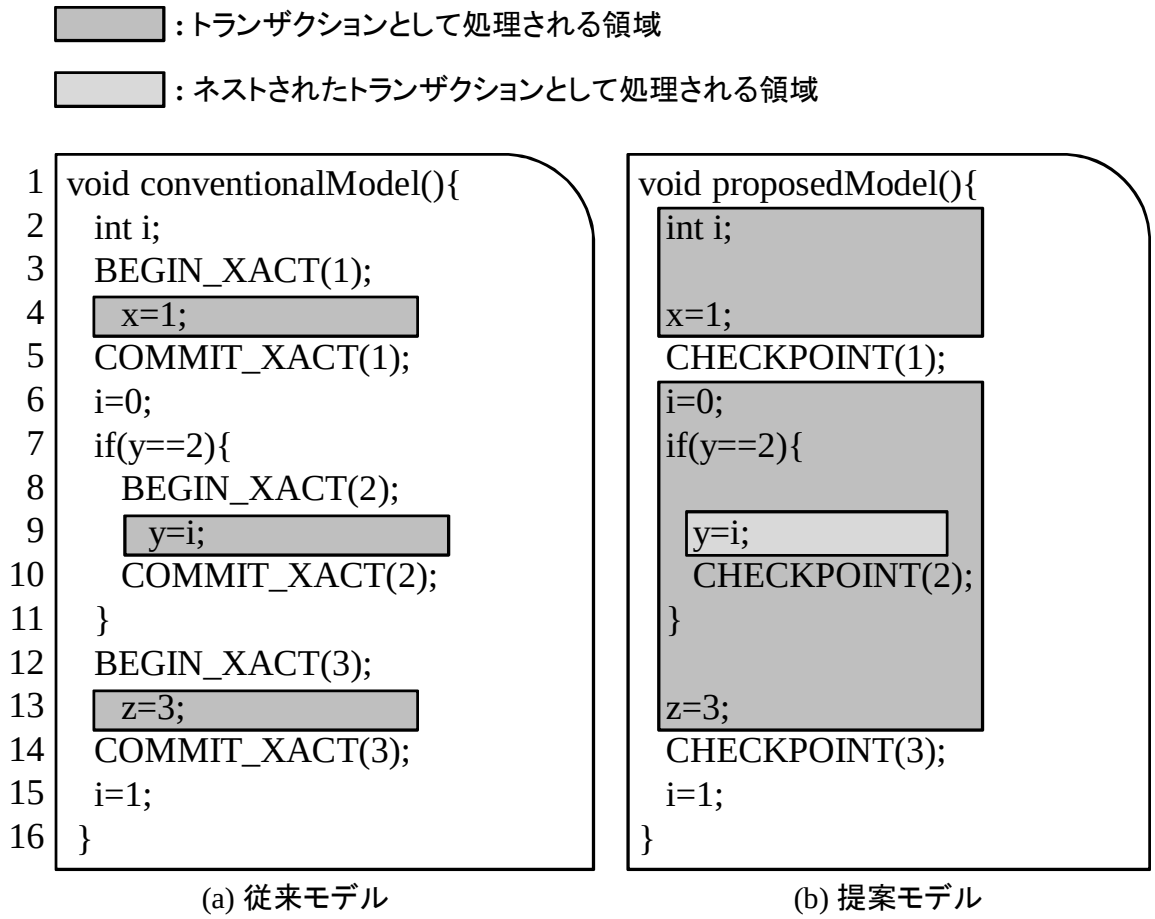

図 6: HTM を用いたプログラム例

チェックポイントを設定するだけでクリティカルセクションを単一のトランザクション処理領域内に含むことができる．このため，厳密にトランザクションとして処理する領域を定めなくても，HTM を用いた並行性制御が可能となる．よって，提案するプログラミングモデルは従来のプログラミングモデルよりもプログラマビリティを高めることができている．

しかし，このプログラミングモデルでは，本来ならば非トランザクションとして処理可能な領域がチェックポイント間に存在していても，トランザクションとして処理されてしまう．このため，提案するプログラミングモデルは従来のプログラミングモデルよりもトランザクション処理領域が大きくなってしまう．例えば図 6(a) の従来モデルでは，ローカルな変数である `i` を含む式はトランザクション外にある式として扱われる．これに対し，(b) の提案モデルでは，2 行目から 4 行目までと 6 行目から 13 行目までがトランザクションとして処理されるため，`i` を含む式もトランザクション内に

ある式として扱われてしまう．これにより，アボート時に書き戻さなければならないデータおよび再実行にかかるコストが増加してしまうと考えられる．そこで，この性能低下を予め抑制するために，これらのコストを削減する技術である部分ロールバックを提案モデルに組み合わせる．

4.2 性能低下の抑制

本節では，まず，提案するプログラミングモデルに組み合わせる部分ロールバックとしてはどのような手法が適切かを検討し，次に，部分ロールバックを LogTM に適用する方法について述べる．

4.2.1 部分ロールバック適用の検討

前節では，トランザクション定義を簡略化し，トランザクションとして処理する範囲を定める新たな定義を設けることで，プログラマビリティを高めるプログラミングモデルを提案した．これによりプログラマは，厳密にトランザクションとして処理する範囲を定義しなくても，HTM のための並列プログラムを記述できるようになる．しかし，このプログラミングモデルは従来のプログラミングモデルよりもトランザクションとして処理する範囲が拡大すると考えられる．そのため，アボート後に再実行しなければならない命令数が多くなり，その実行コストが大きくなってしまうことが予想される．そこで，この性能低下を抑制するために，部分ロールバックを提案モデルに組み合わせる．部分ロールバックを用いると，アボート時のメモリに値を書き戻す量を削減し，再び実行しなければならない命令数も削減できる．これにより，提案モデルにより発生する性能低下の抑制に効果をもたらすと考えられる．

ところで，提案モデルではトランザクション定義を簡略化し，新たに定義したチェックポイントを用いることはすでに述べた．このチェックポイントは，同一ブロック上の，トランザクションとして処理する領域を定めるものである．そのため，単にこれを設定するだけでは，部分ロールバックに必要な内部トランザクションを記述することができない．そこで，通常の部分ロールバックを用いるためには，プログラマが図 6(b)9 行目のようにチェックポイントを設定する必要があるため，プログラマビリティが低下してしまう．そこで，組み合わせる部分ロールバック手法として，競合アドレスを利用した部分ロールバック先学習法 [10] を用いる．この部分ロールバック先学習法では，ある条件に従い，再実行開始点の候補となるリスタートポイントを自動的に設定する．これにより，プログラマがブロックを入れ子にして内部トランザクションを記述しなくても部分ロールバックが可能となるため，プログラマビリティの低下を

防ぐことができる．

4.2.2　競合アドレスを利用した部分ロールバック先学習法の適用

提案モデルで発生する性能低下を予め抑制するために，競合アドレスを利用した部分ロールバック先学習法を HTM に適用する．この部分ロールバック先学習法では以下の手順によってリスタートポイントを自動的に設定する．

1. アボート時にその原因となったアドレスを保持する．
2. 再実行開始後，記憶したアドレスに初めてアクセスするとき，その処理の直前にリスタートポイントを設定する．

以上のようにしてリスタートポイントを設定して，処理が進み再びトランザクションがアボートされるとする．このとき，アボートの原因となった命令がリスタートポイント設定後に実行された命令だとする．すると，アボートされたトランザクションはリスタートポイントへ部分ロールバックすることができる．

次に，この部分ロールバック先学習法を提案モデルに組み合わせた場合の動作例を，**図 7** を用いて説明する．図中の AddressBuffer は，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表す．また，図中の `RESTART_POINT` は部分ロールバック学習法により設定されたリスタートポイントを表す．まず，図 7(a) のように，`ST y，2` でトランザクションがアボートされるとする．このとき Thread1 は，アボートの原因となったアドレス y を AddressBuffer に記憶させた後，トランザクションをアボートする．次に，トランザクションの再実行後初めてアドレス y にアクセスするとき，図 7(b) のように，Thread1 はこのストアアクセスを実行する前に `RESTART_POINT` を設定し，続けて `ST y，2` を実行する．その後，処理が進み，y へのストアアクセスが原因でトランザクションがアボートされるとする．このとき，y へのアクセスは `RESTART_POINT` が設定された後に実行されているため，`Thread1` は図 7(c) のように部分ロールバックし，図 7(b) で設定された `RESTART_POINT` から再実行する．このように，部分ロールバック先学習法では，トランザクション処理中に必要に応じて，リスタートポイントを動的に設定する．

なお，本論文ではこの手法を LogTM に適用するために，リスタートポイント設定後の処理をトランザクション内のトランザクションとして処理する．以降，このトランザクションを中間トランザクションと呼ぶ．このトランザクションは，処理中にチェックポイントに達した場合，最外トランザクションまたは内部トランザクションがコミットされるまでコミット操作を繰り返す．例えば，図 7(d) に示すように，中間トランザクションの処理中に `CHECKPOINT(2)` に達したとする．このとき，中間トランザクショ

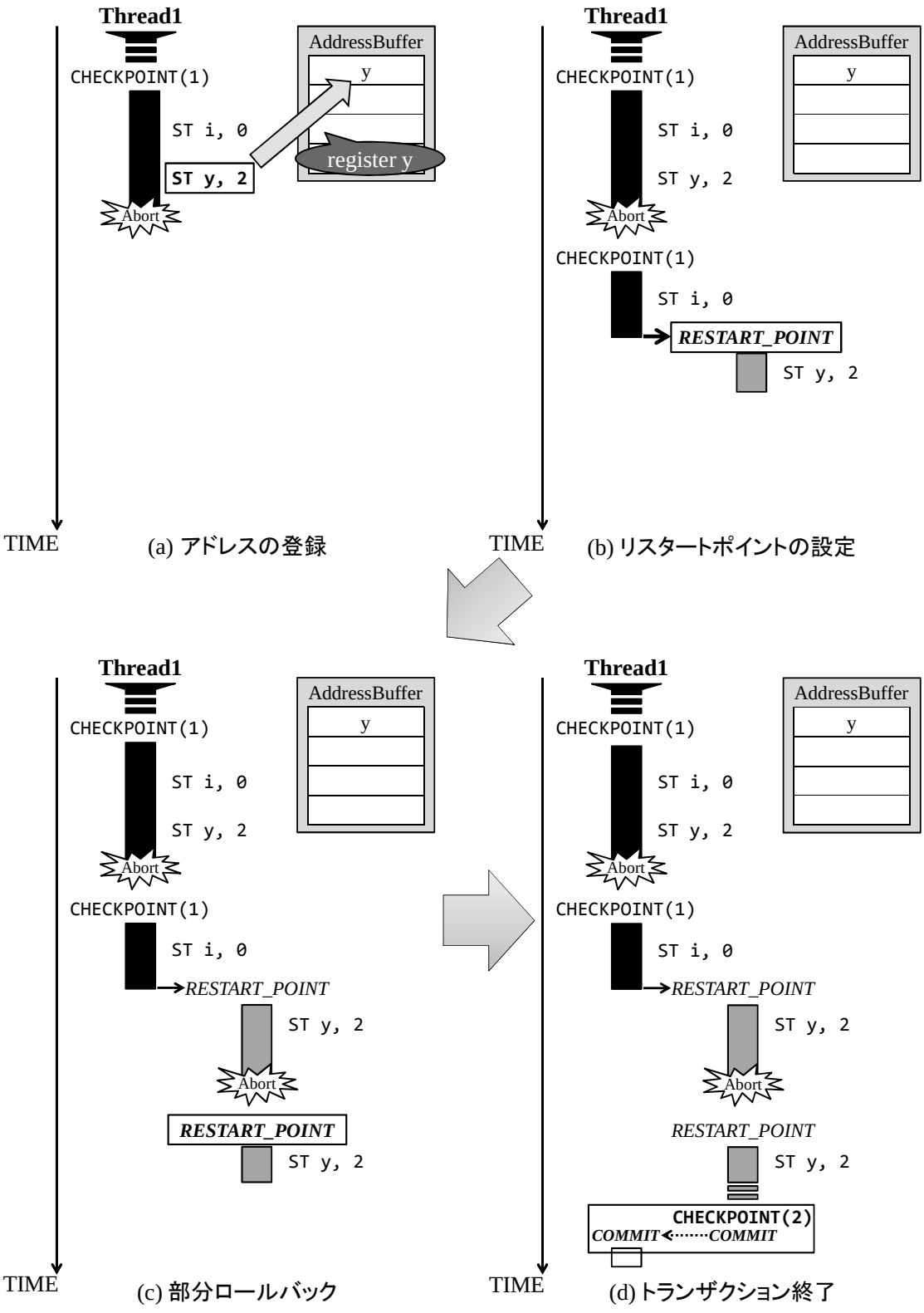

図 7: 競合アドレスを利用した部分ロールバック先学習法の適用

ンをコミットし，CHECKPOINT(1) から開始されたトランザクションをコミットする．

ところで，提案するプログラミングモデルでは，チェックポイントの設定箇所によってプログラム中に内部トランザクションが記述される場合がある．例えば，前節で述べたように，あるチェックポイントを含むブロックが，別のチェックポイントを含むブロック内に記述されている場合が挙げられる．このとき，スレッドはこの内部トランザクションとリスタートポイントの設定による中間トランザクションとを識別することができない．このため HTM は，ネストされたトランザクションの処理中にチェックポイントに達した場合，コミット操作を何回繰り返せば良いのか判断できない．そこで，ネストされたトランザクションが開始された回数を深さとして保持する，LevelBuffer というバッファを追加する．スレッドは，このバッファに最外トランザクションと内部トランザクションの開始時の深さを登録する．この値が，コミット操作の繰り返し回数を特定するための情報となる．そして，チェックポイントに達したときには，スレッドは保持している深さの中から最も大きい値を取り出し，その値と同じ深さのトランザクションが終了するまでコミット操作を繰り返す．これにより，HTM が行うべきコミット操作の繰り返し回数を特定することができる．なお，プログラマは入れ子状態のブロックの内部にもブロックを記述することができるため，記述可能な内部トランザクションの深さに上限を設けることはできない．そこで，LevelBuffer のバッファサイズを越える数の深さを保持しなければならなくなった場合には，バッファ内の古い値からメモリに退避させていくこととする．

このコミット操作の例を**図 8** に示す．このとき，図中の横軸は実行されるトランザクションの深さを表す．また，深さ 1 のトランザクションは内部トランザクションであり，深さ 2 のトランザクションは中間トランザクションであるとする．この例は，深さ 2 のトランザクション処理中にチェックポイントに達したときの様子を表している．この場合，LevelBuffer に保持されている値のうち，最も大きい値である 1 を取り出し，現在実行中のトランザクションの深さとこの値を比較する．すると，実行中のトランザクションの深さの方が大きいので，スレッドはこのトランザクションをコミットする．その後同様にして，深さ 1 のトランザクション処理をコミットするまでこの操作を繰り返す．

4.2.3　実装に必要なハードウェアコスト

競合アドレスを利用した部分ロールバック法を LogTM に適用するために必要なハードウェアを追加する．**図 9** には，追加ハードウェアが示されており，各ハードウェアの役割は以下の通りである．

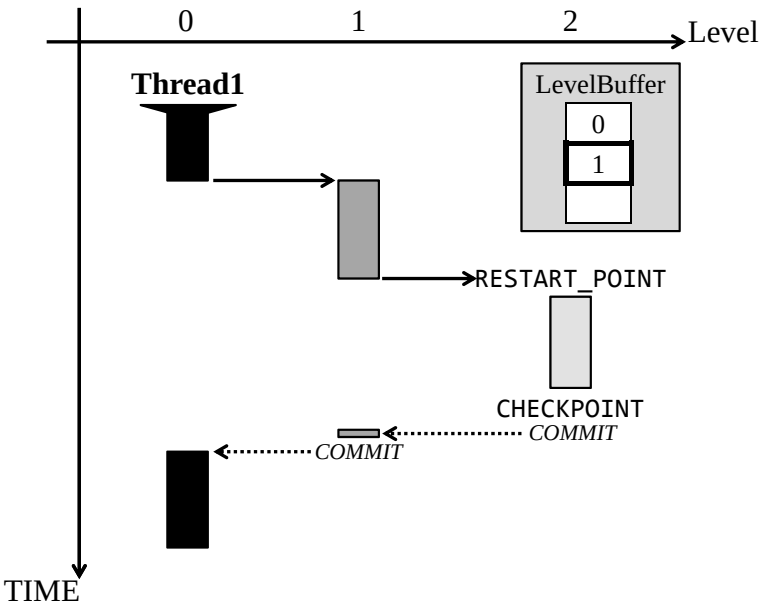

図 8: 部分ロールバック先学習法適用時のコミット操作

AddressBuffer:

アボートの原因となったアドレスを記憶する．バッファが溢れた際のアドレスの置き換え方式には，LRU 方式を用いる．

IsSetBit:

AddressBuffer に保持しているアドレスに対し，リスタートポイントの設定の有無を記録する．

LevelBuffer:

内部トランザクションが開始されたときの，トランザクションの深さを保持する．保持している値と同じ深さのトランザクションがコミットされたときに，その値を破棄する．

既存研究 [3] に従い，AddressBuffer には 4 つのアドレスを保持させる．それに伴い，チェックポイントの設定の有無を記録するビットを 4 つ用意する．アドレスの長さを 64bit とすると，AddressBuffer のサイズは 32byte となり，IsSetBit は 4bit となる．また，仮に LevelBuffer に 5 つの深さを保持させるとすると，そのサイズは 20byte となる．よって 1 コア当たり，52.5byte のハードウェアを追加する．

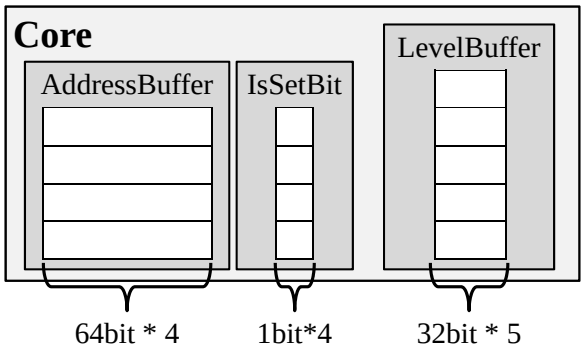

図 9: 追加ハードウェア

5　提案手法のロックに対する性能評価

これまで述べた拡張を実装し，シミュレーションによる評価を行った．想定するシミュレータ構成および使用するベンチマークプログラムは 3 章で述べたものに準ずる．また，各ベンチマークプログラムに対して，10 回ずつ試行し，得られた結果から 95 %の信頼区間を求めた．

5.1　ロックに対する提案手法の優位性の評価

評価結果を**図 10** および**表 3** に示す．図中では，各ベンチマークプログラムと実行スレッド数との組合せによる結果が，各 3 本のグラフで表されている．これら 3 本は左から順に，それぞれ

(L)　ロック：クリティカルセクションを排他制御するモデル．

(T)　従来の LogTM：クリティカルセクションとして扱うべき領域をトランザクションとして処理するモデル．

(P)　提案したプログラミングモデル＋部分先ロールバック学習法：各ベンチマークプログラムに提案プログラムモデルを適用し，かつ競合アドレスを利用した部分ロールバック先学習法を LogTM に実装したモデル．

の実行サイクル数の平均を表しており，モデル (L) の実行サイクル数を 1 として正規化している．

図 10 より，7 つのベンチマークプログラムのうち 6 つのプログラムで，提案モデル (P) におけるプログラムの実行サイクル数がロックを用いるモデル (L) よりも減少し，表 3 に示すように，最大で 96.7%，平均で 26.5%のサイクル数を削減できた．次に，モデル (P) を従来の LogTM を用いるモデル (T) と比較すると，Btree，Contention，Slist，

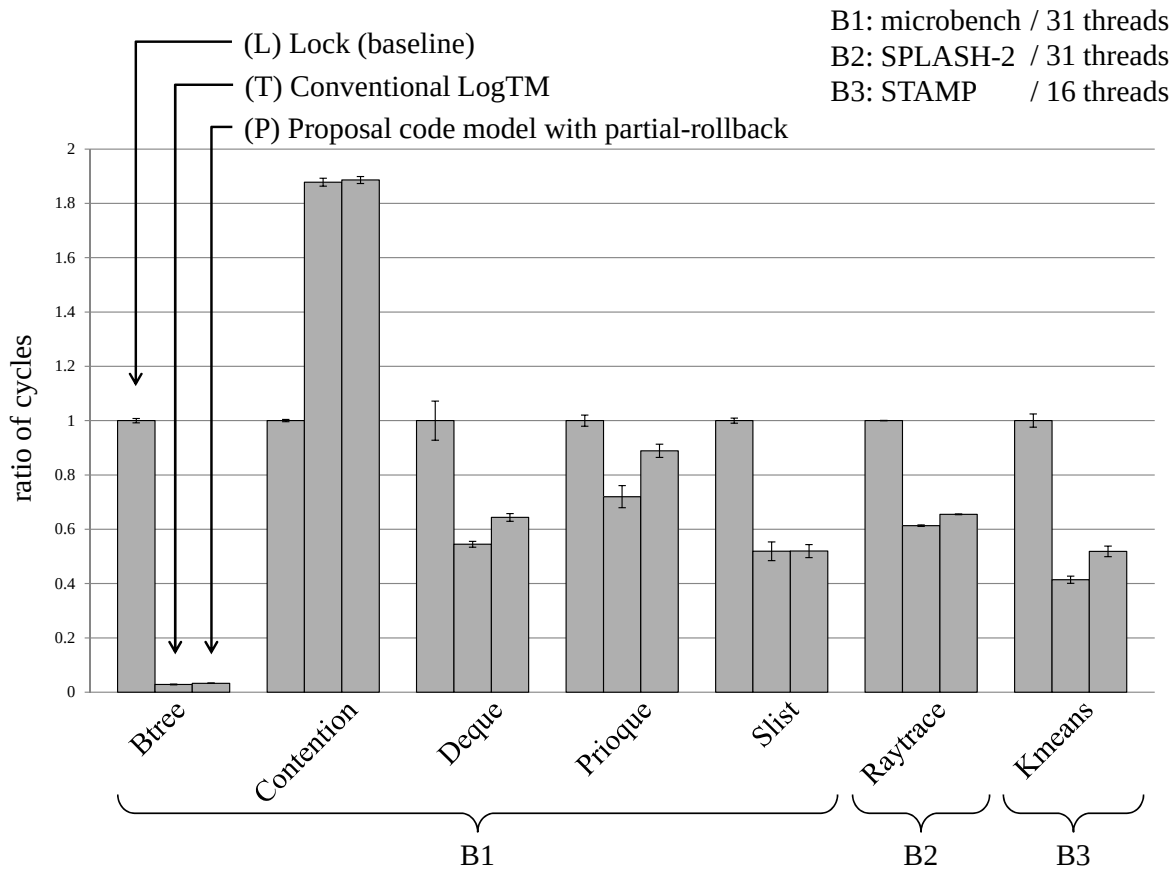

図 10: 各モデルにおける実行サイクル数比

表 3: 削減サイクル率

	最大	平均
(T)	97.2%	32.6%
(P)	96.7%	26.5%

Raytrace においては，ほぼ同じ実行サイクル数となった．これは，発生すると予想された性能低下を，競合アドレスを利用した部分ロールバック先学習法の適用によって抑えることができたためだと考えられる．その他のベンチマークプログラムではサイクル数がやや増加してしまっているものの，モデル (L) に対する各モデルのサイクル削減率は，モデル (P) が平均 26.5%であったのに対し，モデル (T) は平均で 32.5%であり，著しい性能低下は見受けられなかった．

これらの結果より，提案手法において発生が予想された性能低下を抑制し，ロックに対して十分な性能向上を果たせたことがわかった．よって，従来のプログラミング

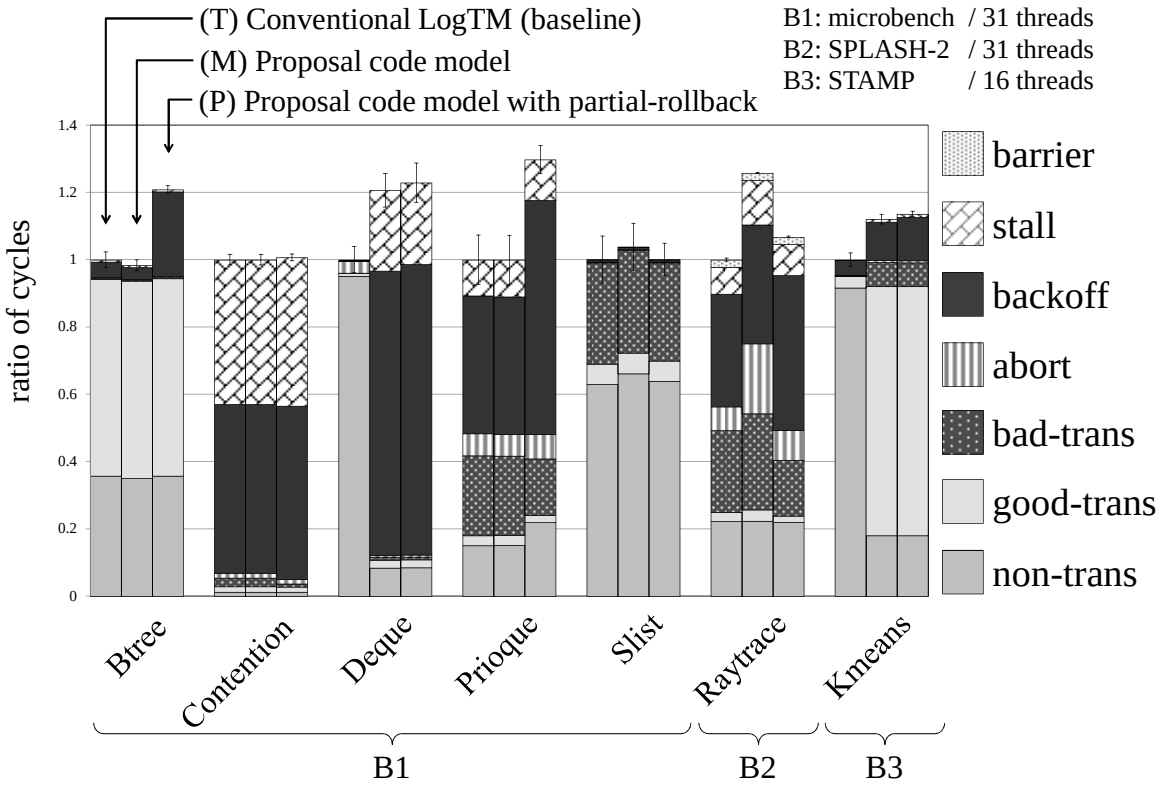

図 11: 各モデルにおけるスレッドの平均実行サイクル数比

モデルよりもプログラマビリティを高めつつ，ロックよりも高い処理性能を実現することは可能であることが確認された．

5.2 部分ロールバック先学習法による性能低下抑制の評価

前節の結果から，提案手法はロックに対して，プログラマビリティと処理性能の両面において優位であることが確認された．しかし，いくつかのベンチマークプログラムにおいて既存の LogTM よりも性能が低下してしまった．そこで，各ベンチマークプログラムで並列実行されたスレッドの平均実行サイクル数を測定し，競合アドレスを利用した部分ロールバック先学習法の有効性を検証した．**図 11** と**表 4**，および**表 5** に評価結果を示す． 図中では，各ベンチマークプログラムと実行スレッド数との組合せによる結果が，各 3 本のグラフで表されている．これら 3 本は左から順に，それぞれ

(T) 既存の LogTM

(M) 提案プログラムモデルのみ : 各ベンチマークプログラムに，プログラマビリティを高めたプログラムモデルを適用したモデル．

(P) 提案プログラムモデル + 部分ロールバック先学習法

表 4: 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

表 5: アボート回数に対する部分ロールバック回数比

	Btree	Contention	Deque	Prioque	Slist	Raytrace	Kmeans
(P)	7.9%	92.8%	18.3%	77.6%	0.06%	89.9%	0.08%

の，各スレッドの実行サイクル数の平均を表しており，モデル (T) の実行サイクル数を 1 として正規化している．ただし，モデル (M) では性能低下抑制のための部分ロールバック先学習法は LogTM に実装していない．また，図 11 中の凡例はサイクル数の内訳を示しており，それぞれは以下の通りである．

barrier: バリア同期に要したサイクル数

stall: ストールに要したサイクル数

backoff: アボート後に実行開始までランダム時間待つのに要したサイクル数

aborting: メモリへの値の書き戻しなどのアボート処理に要したサイクル数

bad-trans: 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: 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: トランザクション外の実行サイクル数

なお，LogTM では，アボート直後にトランザクションを再開した場合，そのアボートの原因となったアドレスへのアクセスにより，競合が再度発生することを防ぐため，アボート後にランダムサイクル待機する機能を備えている．backoff はこの待機に要したサイクル数を表す．モデル (M) に対して，部分ロールバック先学習法を追加実装したモデル (P) は，Slist と Raytrace において性能が向上した．これは，部分ロールバックによるメモリへの書き戻しコストおよび再実行コストの削減により aborting と bad-trans が減少したためである．しかし，それ以外の 5 つのプログラムでは平均実行サイクル数が増加してしまった．このとき，この性能低下したプログラムでは，表 4 のようにアボート回数が増加していた．これは，アボート後に部分ロールバックし，一度競合したアドレスの直前からトランザクション処理を再実行すると，同アドレスによる競合が再び起こってしまうことが原因だと考えられる．この再競合によりトランザクショ

ンがアボートされると，同様の動作が繰り返されてしまう．そのため，これらのプログラムではstallの増加や，アボート回数の増加に伴うabortingとbackoffの増加が引き起こされてしまった．

以上のことから，部分ロールバックによって性能低下を抑制するためには，メモリへの書き戻しと再実行のコストを削減するだけでなく，一度競合したアドレスで再び競合しにくくする必要がある．今回適用した，競合アドレスを利用した部分ロールバック学習法では，アボートの原因となりうる競合アドレスの直前にロールバックする．よって，部分ロールバック後にこのアドレスが原因で再び競合が発生する可能性は高いため，一度競合したアドレスで再び競合することを防ぐのは困難である．このことから，表5に示すアボート回数に対する部分ロールバック回数比が，実行サイクル数と相関を持たず，Prioqueのように部分ロールバック率は高いけれども，実行サイクル数は増加してしまうという結果が得られた．

これらの結果から，プログラマビリティを高めた場合に発生が予想される性能低下の抑制に対して，部分ロールバックが十分な効果を発揮するとは限らないとわかった．さらに，4.2.3項で示したように，部分ロールバックを適用するためにはハードウェアを追加しなければならない．これらの要因から，提案するプログラミングモデルに部分ロールバックを組み合わせることは，必ずしも適切ではないと結論づけられる．ただ，提案手法において部分ロールバックを組み合わせなかったとしても，図11に示すように，モデル(M)はモデル(P)と比較して，Raytraceなどの一部の例外はあるものの，ほとんどのベンチマークプログラムでは大きく性能低下していない．このため，前節で得た結論の通り，従来のプログラミングモデルよりプログラマビリティを高めつつ，ロックより十分に高い処理性能を実現可能であることに変わりはない．

6　チェックポイントの排除に向けた方針

本論文では，トランザクションの開始と終了の定義を簡略化し，トランザクション処理領域を定める新たな定義を用いて，プログラマビリティを高める手法を提案した．これにより，厳密にトランザクションとして処理する領域を定めなくても，HTMを用いた並列プログラムを記述することが可能となった．また，前章の評価結果から，提案したプログラミングモデルに部分ロールバックを適用しなかったとしても，ロックと比較して十分な性能向上を果たせたことも確認された．そこで本章では，提案モデルで定義したチェックポイントを簡略化し，よりプログラマビリティを高めるためのアイデアについて述べる．

現在検討中の手法では，プログラム解析によりトランザクションとして処理する領域を検出する．ところが，一般的にクリティカルセクションとして扱うべき領域はそれを含むプログラムの処理内容に依存する．このため，トランザクションとして処理すべき領域とそうでない領域を，この解析によって識別することは困難である．そこで，プログラマから変数間の依存関係に関する情報を得て，解析によりクリティカルセクションとして扱うべき領域外の処理部分を見つけ出し，トランザクションとして処理すべき領域を特定する．これにより，プログラマがHTM専用の記述を行う必要がないため，HTMを理解していなくても，HTMを利用したプログラムを記述することが可能となる．さらに，並列性を意識しなくても，並列実行可能なプログラムを記述することもできるようになる．

7　おわりに

本論文では，HTMのプログラマビリティを高めつつ，高速に並列実行可能であるかを検討した．そのために，HTMのプログラマビリティを高めたプログラミングモデルを提案し，併せてそれにより発生すると予想される性能低下を抑制する手法を提案した．提案したプログラミングモデルでは，トランザクションの開始と終了の定義を簡略化し，トランザクションとして処理する領域を定める新たな定義を用意した．これにより，プログラマがクリティカルセクションとして扱うべき範囲を特定しなくても，HTMのための並列プログラムを記述することを可能にした．また，予想される性能低下の抑制のために，競合アドレスを利用した部分ロールバック先学習法を組み合わせ，再実行コストの削減を図った．提案手法の有効性を確認するため，GEMS microbenchおよびSPLASH-2 ベンチマークプログラムを用いて評価した結果，ロックと比較して最大で96.7%，平均で26.5%実行サイクル数を削減した．この結果より，提案手法を用いることでHTMはロックよりもプログラマビリティに優れ，かつ高い並列実行性能を維持可能であることが確認された．

今後の課題として，チェックポイントの定義の簡略化によるプログラマビリティの向上が挙げられる．また，提案するプログラミングモデルで予想される性能低下の抑制に，部分ロールバックを用いることは必ずしも適さないことがわかった．そこで，プログラマビリティを高めたプログラムモデルに，これまでに提案されてきたHTMの高速化手法を適用した場合，どの程度の性能向上が得られるか今後検証していきたい．

謝辞

本研究のために，多大な御尽力を頂き，御指導を賜わった名古屋工業大学の松尾啓志教授，津邑公暁准教授，斎藤彰一准教授，松井俊浩准教授，梶岡慎輔助教に深く感謝致します．また，本研究の際に多くの助言，協力をして頂いた松尾・津邑研究室および齋藤研究室，松井研究室の方々に深く感謝致します．特に、江藤正通氏，堀場匠一朗氏，橋本高志良氏には研究を進めるにあたって多大な助言を頂きました．ここに深く感謝致します．

著者発表論文

報文

1. 鈴木 大輝, 江藤 正通, 橋本 高志良, 堀場 匠一朗, 津邑公暁, 松尾 啓志: “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”, 情処研報, Vol.2013-ARC-204, (*toappear*) (Mar. 2013)
2. 橋本 高志良, 鈴木 大輝, 堀場 匠一朗, 江藤 正通, 津邑公暁, 松尾 啓志: “Read-after-Read アクセスを制御するハードウェアトランザクショナルメモリ”, 情処研報, Vol.2013-ARC-204, (*toappear*) (Mar. 2013)

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] Shavit, N. and Touitou, D.: Software Transactional Memory, *Proc. 14th ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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).

[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] 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).