Vol.2011-ARC-196 No.22
2011/7/28

LogTM における適切な競合レベル選択による効率的ロールバック

江藤正通[†1]　浅井宏樹[†1]
津邑公暁[†1]　松尾啓志[†1]

マルチコア環境における並列プログラミングでは，一般的にロックを用いてメモリアクセスの調停がとられている．しかしロックを使用する場合では，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 LogTM が提案されている．LogTM ではプログラマがトランザクションの範囲を定義する必要があり，また，部分ロールバックを行うにはプログラムの特徴を理解した上でトランザクションをネストさせなければならない．そこで本稿ではトランザクションがネスト構造でなくても部分ロールバック可能にする手法を提案する．また，部分ロールバックをより有効にするため，最低限解決しなければならない競合のみを解消する位置へロールバックする改良を提案する．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，既存の部分ロールバック手法に比べて前者で最大 18.5%，後者で最大 16.7%のトランザクション内実行サイクル数を削減した．

Efficient Rollback for LogTM by Selecting Appropriate Conflict Levels

Masamichi Eto,[†1] Hiroki Asai,[†1] Tomoaki Tsumura[†1] and Hiroshi Matsuo[†1]

Lock-based thread synchronization techniques are commonly used in parallel programming on multi-core processors. However, lock can cause deadlocks and poor scalabilities. Hence, LogTM has been proposed and studied for lock-free synchronization. However, for using LogTM effectively, programmers should define transactions and nest them properly with deep knowledge of the runtime behavior. This paper proposes a method for partial-rollback without nesting transactions. Furthermore, for efficient partial-rollback, this paper proposes another method for selecting the restart levels. The result of the experiment shows that the former method improves the performance 18.5% in maximum and the latter improves the performance 16.7% in maximam.

1. はじめに

これまでのプロセッサ高速化技術は，スーパスカラといった命列レベル並列性 (Instruction Level Parallelism: ILP) に基づくさまざまな手法を中心としつつ，集積回路の微細化による高クロック化を半導体技術の向上により実現することで支えられてきた．しかしながらプログラム中の ILP には限界があり，また消費電力や配線遅延の相対的増大により，クロック向上も頭打ちになりつつある．この流れを受け，単一チップ上に複数のプロセッサ・コアを集積したマルチコア・プロセッサが広く普及してきている．マルチコア・プロセッサは，単一プログラムのマルチスレッドによる並列化や，スレッドレベル並列性 (Thread-Level Parallelism: TLP) の抽出により，スループットを向上させることで処理の高速化を目指すものである．

このような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．共有メモリ型並列プログラミングでは，共有リソースに対して調停をとる必要があり，排他制御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場合では，ロック操作のオーバヘッド増大に伴う並列性の低下，デッドロックの発生などの問題が起こりうる．さらに，プログラムごとに最適なロックの粒度を設定するのは難しいため，プログラマにとって必ずしも利用しやすいものではない．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[1)] が提案されている．

LogTM[2)] はハードウェア・トランザクショナル・メモリの一種であり，トランザクションを投機的に実行することでシリアライザビリティとアトミシティを保つ．トランザクションの投機実行中に競合が発生した場合は，メモリ及びレジスタをトランザクション開始時の状態に戻すロールバックを行う．また，トランザクションの内部に別のトランザクションが存在する場合は，内側のトランザクション開始位置から実行を再開する部分ロールバックを行うことができる．

しかし，トランザクションの開始位置はプログラマによって指定されているため，適切な位置に指定されていないと効率的に部分ロールバックを行うことができない．また，適切な

†1 名古屋工業大学
Nagoya Institute of Technology

© 2011 Information Processing Society of Japan

位置に指定するには，プログラマは競合しやすい命令を把握していなければならないが，これを事前に予測することは容易ではない．そこで，この問題を解決するために，トランザクションの途中に動的にチェックポイントを作成する手法を提案する．これにより，トランザクションの途中から実行が再開される可能性が生まれ，無駄な再実行の削減が期待できる．

また，既存の LogTM では自身以外の全てのスレッドとの競合を解消する位置にしかロールバックを行わないが，この条件を満たす場合は多くない．そのため，部分ロールバックが行われる割合が少なく，再実行する命令数が増大するという問題がある。そこで本稿では全ての競合ではなく，最低限解決しなければならない競合のみを解消する位置に部分ロールバックさせる手法も提案する．

以下，2 章では本研究の背景であるトランザクショナル・メモリ及び LogTM の概要について説明する．3 章では，LogTM の問題点及び提案手法の動作モデルについて説明し，4 章でその実装方法について説明する．5 章では提案手法を評価し，6 章で結論を述べる．

2. LogTM

本章では，本研究の対象となるトランザクショナル・メモリの基本的概念とトランザクショナル・メモリのハードウェア実装の一つである LogTM について説明する．

2.1 トランザクショナル・メモリの基本概念

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・コアが同一のメモリアドレスに対してアクセスするためには調停をとる必要がある．そのような排他制御機構として一般的にロックが用いられている．しかし，ロックを用いた調停ではデッドロックが発生する可能性がある．また，並列に実行するスレッド数が増加すると，ロックの獲得・解放に伴うオーバヘッドが増加し，性能が低下する可能性もある．さらに，プログラムごとに最適なロックの粒度を設定するのは難しい．例えば粗粒度ロックを用いる場合，プログラムの構築は容易であるが並列性は損なわれる．対して細粒度ロックを用いる場合，並列性は向上するがプログラムの設計が難しい．このような問題を解決するために，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 (以下 Transactional Memory: TM)** が提案されている．

TM はデータベースの一貫性を保つために用いら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る．TM では，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**と呼ぶ．このときトランザクションは以下の二つの性質を満たす．

シリアライザビリティ(直列可能性):

並行して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に実行した場合と同じである．

アトミシティ(原子性):

トランザクションの操作は完全に実行されるか全く実行されないかのいずれかである．

以上の性質を保証するために，TM はあるトランザクションが他のトランザクションと同じメモリアドレスにアクセスするかどうかを監視する．そして，他のトランザクションからアクセスされたメモリアドレスと，自身のトランザクション内でアクセスするメモリアドレスが同一であった場合を**競合**として検出する．競合が検出された場合，片方のトランザクションの実行を停止し，それまでの結果を全て破棄する**アボート**を行う．アボート操作を行ったスレッドはメモリ及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し，再実行する．この状態を復元する一連の処理を**ロールバック**という．一方で，トランザクションの終了まで競合が検出されなかった場合，トランザクション内で実行された結果を全てメモリに反映させる．これを**コミット**という．

このように TM を用いることで，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．これにより，ロックを用いる場合よりもプログラムの並列性が向上し，コア数の増大に応じて性能が向上しやすくなる．また，プログラマはロックの粒度を考慮する必要がなくなり，プログラマビリティが向上する．

2.2 データのバージョン管理

TM におけるトランザクションの投機的実行では実行結果が破棄される可能性があるので，アクセスするデータの古いバージョンを保持し管理する必要がある．我々がターゲットとする LogTM ではこのバージョン管理を仮想メモリ領域を用いることで実現している．LogTM は**ログ**と呼ばれる仮想メモリ領域をスレッドことに割り当て，トランザクション内のストア命令によって上書きされる前の値とそのアドレスをこのログに退避する．一方でストア命令の結果はメモリに書き込まれる．なお，トランザクション内で同じアドレスに対して複数回ストア命令が実行された場合，ログに保存するのは最初の 1 回だけでよい．なぜなら，トランザクションのロールバックにはトランザクション開始時点でのメモリ状態がわかれば十分だからである．

ここで，投機実行が失敗した場合と成功した場合における動作の違いについて説明する．投機的実行が失敗した場合はアボート操作を行った後，ログに保存された全ての値をメモリに書き戻すことでメモリ状態をトランザクション開始時点まで戻す．また，メモリ状態の復元後に，レジスタの状態をトランザクションの開始時点まで戻すことでトランザクションの再実行を可能にする．このレジスタ状態の情報はトランザクション開始時に**チェックポイン**

© 2011 Information Processing Society of Japan

ト(以下 CP) としてログに登録される．ロールバック時にログに登録された CP を参照することでレジスタ状態を復元する．一方で，投機的実行が成功した場合はコミット操作を行うが，全ての更新は既にメモリに反映されているため，ログの走査や退避した値の書き戻し等のメモリアクセスは必要なく，ログの内容を破棄するだけでよい．

2.3 競合検出

トランザクションのアトミシティを保つために，トランザクション内で実行される命令における競合の有無を監視する必要がある．そこで LogTM はキャッシュライン上に新しく read セット及び write セットという領域を追加している．各セットに対応する read ビット及び write ビットはトランザクション内でそれぞれ当該キャッシュラインに対するリードアクセスまたはライトアクセスが発生した場合にセットされ，トランザクションのコミット及びアボート時にリセットされる．

LogTM は一貫性モデルにディレクトリベース[3] の Illinois プロトコル[4] を採用し，これを拡張することで他のトランザクションを実行するスレッドとの競合を監視している．競合として検出されるのは以下の 3 パターンのアクセスが行われた場合である．

read after write: write ビットがセットされているアドレスに対するリードアクセス
write after read: read ビットがセットされているアドレスに対するライトアクセス
write after write: write ビットがセットされているアドレスに対するライトアクセス

例えば，あるスレッドがリード/ライト命令を実行する場合，トランザクション内の一貫性を保つためにコヒーレンスリクエストを他の全スレッドに送信する．このリクエストを受信したスレッドは，どのメモリアドレスへのアクセスが行われようとしているのか知ることができ，当該キャッシュライン上の read セット及び write セットを参照することで競合を検出することができる．競合が検出されなかった場合，リクエストを受信したスレッドから送信者に対して ACK が返信される．一方で競合が検出された場合は NACK が返信される．NACK を受信したスレッドは競合が発生したことを知り，競合したスレッドが終了するまで一時的に実行を停止する．これを**ストール**という．ストールしたスレッドは同じアドレスに対するリクエストを送信し続ける．競合したスレッドがそのトランザクションを終了した場合，そのスレッドから ACK が返信されるため，ストールしたスレッドは相手の終了を検知し，実行を再開できる．

しかし**図 1**(a) で示すように，複数のスレッドがストールするとデッドロック状態に陥る場合がある．この例では，2 つのスレッド Thread1 と Thread2 がそれぞれトランザクション T1 と T2 を投機的に実行している．まず，Thread1 が実行を開始した後に Thread2 が実行を開始しており，Thread1 が `ST 0x100` を実行し，その後に Thread2 が `ST 0x200` を実行済みである場合を考える．ここで，Thread1 が `LD 0x200` を実行しようとする場合，Thread1 は他のスレッドに対して request1 を送信する (時刻 t1)．request1 を受信した Thread2 は競合したことを検知するため NACK1 を返信し，NACK1 を受信した Thread1 はストールする (時刻 t3)．図中では省略しているが，Thread1 はアクセスの許可を受けるまで Thread2 に対して定期的に request を送信している．この後，Thread2 による `ST 0x300`，`ST 0x400` の実行を経て `LD 0x100` が実行される (時刻 t4) と，Thread2 は Thread1 と競合してストールする (時刻 t5)．このようにお互いにストールしてしまうとデッドロックに陥る．

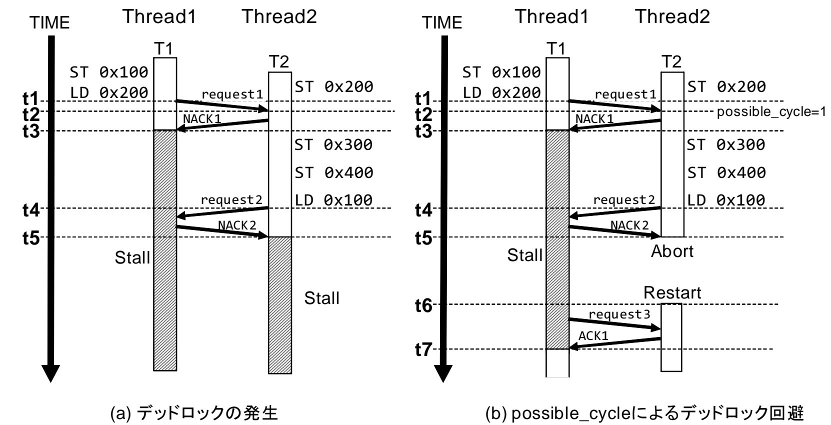

図 1 LogTM におけるトランザクションの競合解決

LogTM では，このデッドロックを回避するために各スレッドに possible_cycle flag を保持させている．possible_cycle flag は TLR's distributed timestamp method[5] に基づいた手法であり，図 1(b) のように，自身より早くトランザクションを開始したスレッドに NACK を返信するとセットされる (時刻 t2)．そして，possible_cycle flag がセットされているスレッドは，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 を受信した場合，デッドロックの発生を防ぐためにアボートする (時刻 t5)．こうして開始時刻の遅いトランザクションを実行しているスレッドがアボートの対象として選択される．アボート操作を行った Thread2 はトランザクション開始時の状態を復元し，トランザクションを再実行する (時刻 t6)．また，Thread2 がアボート操作を行ったことにより Thread1 は 0x200 番地にアクセスできるようになるため，Thread1 のストール状態が解消される (時刻 t7)．

© 2011 Information Processing Society of Japan

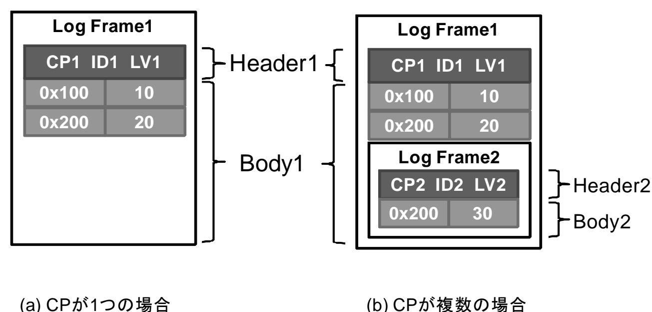

図 2 ログフレームの構成

2.4 部分ロールバック

LogTM では，アボートすることによって実行結果を破棄した状態から再実行可能な状態にロールバックするために，ログに退避した値をキャッシュに書き戻している．このとき，毎回トランザクション開始点までロールバックする場合では，再実行する命令が多くなってしまう．そこで LogTM はトランザクションの途中から実行を再開する**部分ロールバック**[6)]をサポートしている．なお，部分ロールバックを行うためには，予めプログラマはトランザクション内に別のトランザクションを定義しておく必要がある．

ロールバック時に走査されるログは**図 2**(a) に示すようにヘッダ部とボディ部から構成され，これらをまとめて Log Frame と呼ぶ．ログはスタック構造になっており，ヘッダ部には CP やトランザクションを識別するための **ID** 及び**トランザクションレベル**(以下 LV) が保存される．この ID はプログラマが各トランザクションに与える固有の ID である．また，LV はトランザクションのネストの深さを表している．具体的には，最初のトランザクションの実行を開始した時点で LV1 となり，そのトランザクションを実行中に新たなトランザクションの実行を開始すると LV2 となる．一方で，ボディ部には書き込み前の値及びそのアドレスが記録される．以降，この 1 つの登録をエントリと呼ぶ．

この例では，`ST 0x100`，`ST 0x200` が順に実行され，書き換え前の値 10 及び値 20 が退避されている．一方で，書き戻し処理はログへの退避とは逆順に行われるため，0x200 番地の値 20，次に 0x100 番地の値 10 が順にキャッシュに書き戻される．そしてログのヘッダ部に到達すると，これ以上書き戻し処理の必要がないと判断し，レジスタの状態をトランザクション開始時と同じ状態に戻す．一方で，トランザクションの実行中に新たなトランザクションの実行が開始されると図 2(b) のように新たに Log Frame2 が作成され，LV2 内における書き換え前の値及びそのアドレスが退避される．この新たな CP へロールバックするには，CP1 よりも後に登録されているアドレスが再び退避される場合があり，退避されるエントリの数が増加する．また，ロールバックの際にどの CP へロールバックすれば良いか判定するために，LogTM は**図 3** のようにキャッシュライン上の read セット及び write セットを LV ごとに割り当てる．図 3 中の R は read セット，W は write セットを表し，その後ろの数字は LV を表している．競合判定時にこれらのセットを参照することで，競合が発生した LV を検知できるため，その競合が解消される CP へロールバックすることが可能となっている．

Address	R1	W1	R2	W2	...	Value

図 3 拡張したリードライトセットの利用

3. 部分ロールバックの改良

本章では，既存手法である LogTM の問題点と，それを解決する提案手法について説明する．

3.1 動的な CP 作成

既存の LogTM において，部分ロールバックを行うためには，プログラマがトランザクション内に別のトランザクションを定義しておく必要がある．この内側のトランザクション開始位置は，トランザクション内の共有リソースがどのような条件下で競合するのか考えて決めなければ性能が悪化する可能性がある．なぜなら，競合する命令よりも後に指定した場合では，部分ロールバックができず，CP 自体や値復元のためのエントリの増加によりロールバック時の書き戻しコストが増大するためである．以降，これをロールバックコストと呼ぶ．また，競合する命令よりも前にトランザクション開始位置を指定した場合でも，多くのトランザクションをネストさせてしまうと，このロールバックコストが，削減できる再実行コストを上回ってしまい，性能が悪化する可能性がある．よって，部分ロールバックにより性能を向上させるためには，プログラマはこれらのコストの増減を考慮してトランザクション内の適切な位置に別のトランザクションの開始位置を指定しなければならない．

そこで本稿では，動的に CP の作成を行う手法を提案する．以降では，提案手法によって

© 2011 Information Processing Society of Japan

動的に作成される CP を動的 CP と呼び，プログラマにより指定されたトランザクション開始位置に作成される CP を静的 CP と呼ぶ．動的 CP を作成することで，プログラマはトランザクション内において競合を起こしやすい命令を考慮する必要がなくなる．

提案手法では，ロード/ストア命令の実行回数を用いて動的 CP を作成する．これは，競合がトランザクション内で実行されるロード/ストア命令によって発生するためである．また，動的 CP を作成しすぎないようにするため，CP を作成するたびに次に作成する動的 CP までの間隔を広げる．その方法として，CP を作成するたびに動的 CP 作成間隔を 2 倍に更新する．具体的には次のような式によって CP 作成間隔を定義する．

$$LSC(k) = 2^{k+i} \qquad (k \geq 1 \quad i \geq 0) \tag{1}$$

式 (1) の LSC は次の CP までに必要なロード/ストア命令の実行回数を示している．つまり，k 回目の CP は，前の CP から $LSC(k)$ 回のロード/ストア命令が実行されたときに作成される．また，i は間隔の初期値を設定するのに用いる．例えば，1000 回のロード/ストア命令を含むトランザクションを実行する際に，i の値を 4 とした場合では 32 ロード/ストアから動的 CP が作成され始め，合計 5 つの動的 CP が作成される，また，i の値を 8 とした場合にはひとつだけ動的 CP が作成される．

3.2 ロールバック先 LV の変更

既存の部分ロールバック手法で使用されているロールバック先 LV(以下 Restart Transaction Level: RLV) 選択手法では，プログラマがトランザクション内の適切な位置から再実行できるようにトランザクション開始位置を指定していたとしても，その位置にロールバックできない場合がある．なぜなら，自分以外の全スレッドとの競合を解消できる LV を RLV として選択しているためである．つまり，アボートに関係の無い競合にも影響される場合があり，RLV が必要以上に低くなってしまう可能性がある．RLV が低くなることで，実行を再開する位置がトランザクション終了地点から遠くなり，再実行する命令数が増加する．また，その結果としてトランザクション終了までに費す時間が増え，再度競合してアボートしてしまう確率も増える．

そこで本稿では，必ず解消しなければならない競合のみを RLV の設定に用いる手法を提案する．この必ず解決しなければならない競合とは，2.3 節で述べたデッドロックに陥る可能性のある 2 つの競合である．これらの競合はどちらも，自身よりもトランザクション開始時刻が早いスレッドとの競合である．これらの競合のうち，後に発生する競合のみを解消する場合では，競合相手のスレッドがストールし続けてしまい，再びアボートすることになってしまう．そのため，コミットまで実行を進めることができなくなってしまう．したがって提案手法では，これらの競合のうち，両方の競合を解消できる RLV を設定する．これにより，自身よりもトランザクション開始時刻の遅いスレッドに影響されることがなくなり，トランザクション開始時刻の先行しているスレッドが優先されることになる．

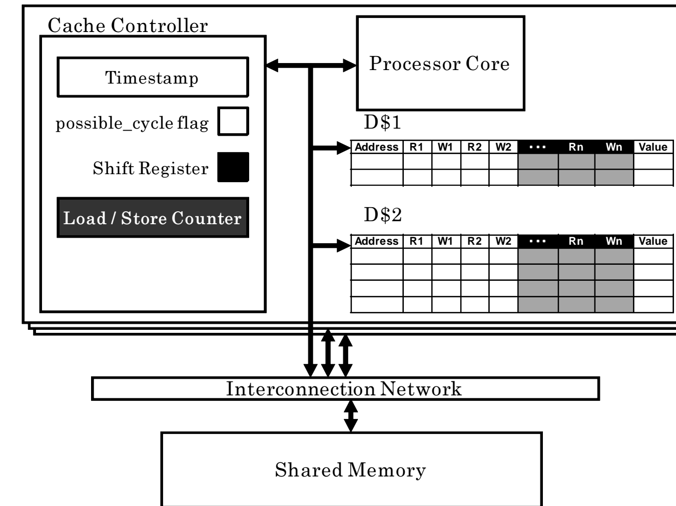

図 4 拡張した LogTM の構成

4. 実　　装

本章では提案手法の実装について説明する．

4.1 動的 CP の作成と削除

動的 CP の作成を実現するために既存の LogTM を拡張し，以下 3 つの機構を追加した．拡張した LogTM の構成図を**図 4** に示す．

Load/Store Counter:
トランザクション内におけるロード/ストア命令の実行回数をカウントする．

Shift Register:
次に作成する CP までの間隔を保持し，ビットシフトにより CP 間隔を変更する．

追加の read セットと write セット:
複数の CP が作成され，それぞれの CP にロールバック可能にする場合はその数だけ read セットと write セットが必要となる．そのため，各キャッシュライン上に各セットを追加する．

© 2011 Information Processing Society of Japan

提案手法では，動的 CP を作成するためにトランザクション内で実行されるロード/ストア命令の実行回数をカウントする．そのため，既存の LogTM を拡張してロード/ストア命令数のカウンタ (以下 Load/Store Counter) を設ける．この Load/Store Counter は各プロセッサ・コアのキャッシュコントローラ内に 1 つずつ設け，トランザクションを実行する各スレッドはそれぞれ固有のロード/ストア命令数を管理する．そして，Load/Store Counter の値が Shift Register の保持する値と等しくなった場合，動的 CP を作成する．このとき，CP の作成間隔を変動させるため，CP が作成されるたびに Load/Store Counter が保持する値をリセットする．また，Shift Register が保持する値を 1 ビット左シフトすることで，3.1 節で述べた CP の作成間隔を 2 倍に増加させることができる．

一方で，アボート操作及びロールバック操作を行う場合，Load/Store Counter が保持する値及び Shift Register が保持する値をロールバック先の CP を作成したときと同じ値に復元する必要がある．Load/Store Counter が保持している値は CP 作成時にリセットされているため，同様にその値をリセットするだけでよい．しかし，Shift Register が CP 作成時に保持していた値を得るためには，右シフトするビット数を計算しなければならない．そこで，提案手法ではアボート時の LV と RLV から右シフトしなければならないビット数を $LV - RLV$ として求める．

なお，既存の LogTM におけるコミット操作では，現在実行している LV の CP を 1 つ削除する．しかし，動的 CP を作成している場合に同様の操作を行うと，直近の動的 CP1 つ削除されるだけで，現 LV に対応する CP が削除されない．したがって，これらの CP を削除するための新たな操作を追加しなければならない．提案手法では，動的 CP にプログラマが指定した ID とは異なる ID を割り当て，コミット時には，当該 LV に対応する CP が現れるまで，全ての動的 CP に対してコミット操作を行うこととした．

4.2 ロールバック先 LV の設定条件

既存の部分ロールバック手法では，競合が発生した際にはその競合を必ず解消する LV を RLV として設定しており，他スレッドのトランザクションの開始時刻を全く考慮していない．そこで，提案手法では他スレッドのトランザクション開始時刻を考慮して RLV を設定するために，3.2 節で述べたアボートに関係する競合のうち，先に発生する競合を RLV の設定に利用する．この競合は図 1 で説明した possible_cycle flag のセットされる条件と同一であるため，このフラグのセット条件を利用する．

ここで，既存手法と提案手法の動作の違いを**図 5** を用いて説明する．まず既存手法の場合は，Thread2 が NACK1 を Thread3 に返信するときに Thread2 の RLV が設定されるが，このとき競合したアドレスは 0x200 であり，Thread2 は LV2 内でこのアドレスにアクセス

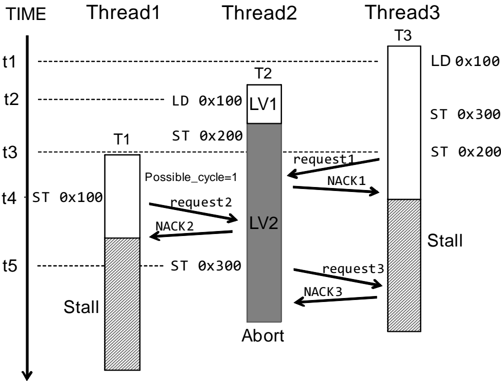

図 5 ロールバック先 LV の設定方法

しているため，RLV は 2 に設定される．その後，NACK2 を Thread1 に返信するときに RLV は 1 に更新される．このように，競合した相手スレッドのトランザクション開始時刻に関係無く，そのアクセスが行われたトランザクション内の競合を解消する LV で RLV を更新している．つまり，トランザクション開始時刻の遅いスレッドの影響を受けて RLV を設定し直してしまう．一方，提案手法では possible_cycle flag のセットに関係のある NACK1 を返信するときにのみ RLV の設定をするため，NACK2 を返信する際には RLV の更新はされない．つまり，トランザクションの開始時刻が早いスレッドが優先されるので，アボートするスレッドの部分ロールバックする可能性が増え，結果としてそのスレッドは既存手法よりも早くトランザクションを終わらせることができる．以上のようにして，NACK3 を受信した Thread2 は既存手法では L1 の開始位置にロールバックするのに対し，提案手法では L2 の開始位置に部分ロールバックすることが可能となる．

5. 評　　価

5.1 評 価 環 境

前章で述べた拡張を既存の LogTM に実装し，シミュレーションによる評価を行った．評価にはフルシステムシミュレータである Simics[7] と GEMS-2.1.1[8] を用いた．Simics は機能シミュレーション，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行う．プロセッサは 32 コアの SPARC V9 とし，各コアは単命令・インオーダ発行を行う．ま

© 2011 Information Processing Society of Japan

表 1　システムモデルパラメータ

Processor	SPARC V9
コア数	32 cores
周波数	4 GHz
発行形式	single-issue
発行順序	in-order
IPC	non-memoryIPC=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　ベンチマークプログラムとその入力

Contention	config 1
Prioque	8192ops
Slist	500ops 64len
Partial-rollback	128array 4depth
Btree	priv-alloc-20pct
Deque	1024ops 32bkoff

た，OS は Solaris10 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 付属のベンチマークプログラムである Btree，Contention，Deque，Partial-rollback，Prioque，Slist を用い，それぞれのプログラムを 16 及び 31 スレッドで実行した．また，それぞれの入力を**表 2** に示す．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，各評価対象につき試行を 10 回繰り返し，得られたサイクル数の平均値を比較する．

5.2 評 価 結 果

図 6 に Btree，Contention，Deque，Partial-rollback，Prioque，Slist の 16 スレッド及び 31 スレッドで実行したときの評価結果をに示す．それぞれのグラフは左から順に
(F) 必ずトランザクションの開始点までロールバックする手法 (Flat)
(P) 既存の部分ロールバック手法 (Partial)
(R) モデル (P) の RLV 変更手法
(C) モデル (R) に加えて動的 CP を作成する手法

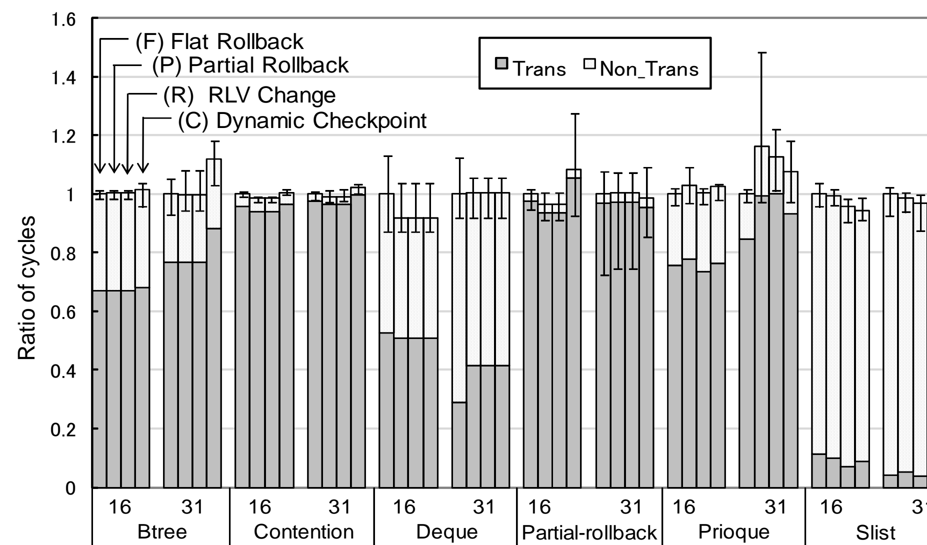

図 6　総実行サイクル数

の実行サイクル数の平均を表しており，モデル (F) の実行サイクル数を 1 として正規化している．なお，モデル (C) の最初の CP 作成間隔は 16 ロード/ストアとしている．また，凡例は内訳を示しており，Trans はトランザクション内で要した実行サイクル数，Non_Trans はトランザクション外で要した実行サイクル数を示している．

まず適切に RLV を変更する提案モデル (R) では，既存モデル (P) と比べて Slist 及び Prioque においてのみ変化が表れた．この理由として，Partial-rollback，Deque は，モデル (P) の場合で既に部分ロールバックが常に行われており，それ以上のコストを削減できる位置に CP が存在していないことが挙げられる．また Btree には複数の CP が存在しているが，モデル (P) 及びモデル (R) は共に部分ロールバックが行えていなかったため結果は変化していない．この理由は，必ず内側のトランザクションよりも前に発生した競合が原因となって Possible_cycle flag がセットされていたためである．なお，Contention においては CP が 1 つしか存在しておらず，必ずその CP がロールバックに利用されていた．モデル (R) において変化のあった Prioque 及び Slist においては，アボート回数が減少しており，

© 2011 Information Processing Society of Japan

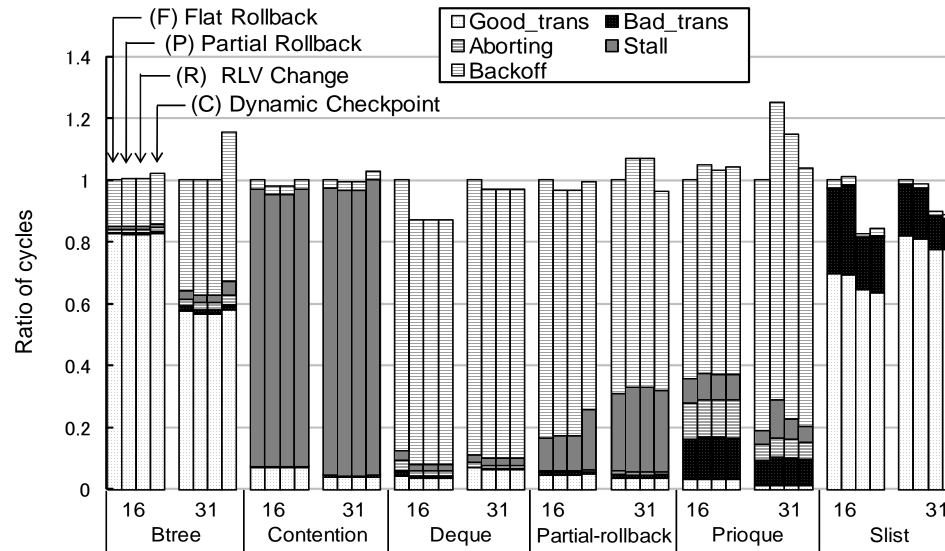

図 7 トランザクション内の実行サイクル数

これは部分ロールバックによって再度アボートする確率が低下したためだと考えられる．モデル (R) で大きくサイクル数が変化した Slist では既存モデル (P) と比べて 16 スレッド及び 31 スレッドでそれぞれ 4.2%，3.0%サイクル削減率が向上している．

一方で動的 CP を作成する提案モデル (C) では，既存モデル (P) と比べ Slist 及び 31 スレッドの場合において主に変化しており，Slist においては提案モデル (R) よりもさらにサイクル削減率が向上していた．その理由として，現在実行している命令により近い位置へ部分ロールバックすることで，2 次キャッシュミスが減少したことが考えられる．また，サイクル削減率が低下した 16 スレッドの Partial-rollback では，実行時のサイクル数に幅があり，最大で平均よりも 36%増加している場合があった．これは，トランザクション開始時刻の早いスレッドを早くコミットさせるために複数のトランザクション開始時刻の遅いスレッドがストールし続けたことが原因であると考えられる．

次に，提案手法は主にトランザクション内の実行に影響を与えるため，トランザクション内の各処理に要したサイクル数の，スレッド間平均を評価した．その結果を**図 7** に示す．なお，凡例はトランザクション内における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．

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

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

Aborting: アボート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サイクル数

ここで Backoff とはアボートから再開までの待ち時間である．アボート直後にスレッドを再開した場合，同じ競合が再度発生することで他スレッドの実行を妨げる可能性がある．このため LogTM では，アボート後にランダムサイクル待機した後，スレッドを再開する機能を備えている．なお，この待機するサイクル数は，アボートが発生するたびに増大するよう設定されている．

まずモデル (R) では，Slist において部分ロールバックを行うことでモデル (P) よりも再実行する量を削減できていることが，Bad_trans が削減されていることからわかる．その結果，トランザクション内のサイクル削減率は，モデル (P) と比べ 16 スレッドで 18.5%，31 スレッドで 8.9%向上した．また，Prioque の 31 スレッドでは，モデル (P) においてストールサイクルが大きく増加してしまっているが，提案モデル (R) 及び (C) ではこの増加を抑えることができている．

次にモデル (C) では，Slist の 16 のスレッド及び 31 スレッドにおいてモデル (R) に近い結果となっているが，主に Good_trans が減少している．この理由としては，動的 CP へ部分ロールバックした場合は再実行を開始する位置がアボートの位置により近いため，コミットされるトランザクションの実行時にキャッシュヒット率が向上したことが考えられる．しかし，16 スレッドにおいては提案モデル (R) と比べてアボート回数が増加しており，動的 CP に部分ロールバックすることを複数回繰り返したことで Bad_trans が増加する場合があった．結果としては，モデル (P) と比べて 16 スレッドで 16.7%，31 スレッドで 9.7%サイクル削減率が向上したことを確認した．

なお，今回のシミュレーション評価では Contention 及び Slist 以外において主に Backoff サイクルの変動が大きく影響しており，ランダムに設定される待機時間の値に幅がありすぎる，あるいは値が大きすぎることが問題であると考えられる．この点については今後詳細な調査が必要である．

6. おわりに

本稿では，既存のハードウェア・トランザクショナル・メモリである LogTM を拡張し，

© 2011 Information Processing Society of Japan

RLV を変更する手法及び動的に CP を作成する手法を提案した．拡張した LogTM では CP 作成条件にロード/ストア命令の実行回数を用いて，トランザクションの途中に動的 CP を作成する．また，トランザクションの長さに合わせて CP 間隔を動的に変動させる手法を提案した．提案手法の有効性を確認するため，GEMS 付属ベンチマークプログラムを用いて評価を行った．提案手法によって影響を受けるトランザクション内の実行サイクル数が，既存の部分ロールバック手法と比べると前者では最大 18.5%，後者では最大 16.7%削減されることを確認した．提案手法が有効となった要因としては，ロールバック時の書き戻しコストの減少や再実行サイクル数の削減が挙げられる．しかし，動的 CP 作成によって書き戻すエントリ数が増加したため，性能が悪化してしまう場合も見られた．それに加えて，ランダム時間待機する Backoff サイクルの増加が提案手法の性能の増減に大きく影響していた．したがって今後の課題として，まずは Backoff サイクルの詳細な調査が挙げられる．また，使用されていない CP を作成しないようにすることでログに退避するエントリ数を減らし，その書き戻しにかかるコストを削減することや，動的 CP の位置をより競合しやすい命令の直前に作成し，再実行コストを削減する手法を探ることが必要である．それに加え，部分ロールバックをより効率的なものとするためにアボート対象のスレッド選択方法についても今後検討していきたい．

参 考 文 献

1) Herlihy, M. and Moss, J. E.B.: Transactional Memory: Architectural Support for Lock-Free Data Structures, *Proc. of 20th Annual International Symposium on Computer Architecture*, ACM, 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. of 12th International Symposium on High-Performance Computer Architecture*, IEEE Computer Society, pp.254–265 (2006).
3) Sweazey, P. and Smith, A.J.: A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. of 13th Annual Int'l. Symp. on Computer Architecture (ISCA'86)*, pp.414–423 (1986).
4) Censier, L.M. and Feautrier, P.: A New Solution to Coherence Problems in Multicache Systems, *IEEE Transactions on Computers*, Vol.c-27, No.12, pp.1112–1118 (1978).
5) Rajwar, R. and Goodman, J.R.: Transactional Lock-Free Execution of Lock-Based Programs, *Proc of 10th Symposium on Architectural Support for Programming Languages and Operating Systems*, ACM, pp.5–17 (2002).
6) J.Moravan, M., Bobba, J., E.Moore, K., Yen, L., D.Hill, M., Liblit, B., M.Swift, M. and A.Wood, D.: Supporting Nested Transactional Memory in LogTM, *Appears in the proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, pp.1–12 (2006).
7) 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).
8) Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., E.Moore, K., Hill, M.D. and Wood., D.A.: Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset, *AMC SIGARCH Computer Architecture News*, Vol.33, No.4, pp.92–99 (2005).
9) Alameldeen, A.R. and Wood, D.A.: Variability in Architectural Simulations of Multi-Threaded Workloads, *Proc. of 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03)*, pp.7–18 (2003).

© 2011 Information Processing Society of Japan