Vol.2010-ARC-190 No.9
2010/8/3

最適なロールバック・ポイントを選択するトランザクショナル・メモリ

伊藤悠二[†1]　塩谷亮太[†1,†2]
五島正裕[†1]　坂井修一[†1]

並列プログラミングにおいて，ロックを用いた同期機構が広く用いられている．しかし，ロックを用いると，デッドロックや不要なロックによる並列性の低下が生じることがある．一方で，ロックを用いるよりも容易に速いプログラムを書ける，トランザクショナル・メモリを用いた同期機構が提案されている．トランザクショナル・メモリを用いた並列プログラミングでは，プログラマが排他制御したい一連の処理をトランザクションとして指定する．多くのトランザクショナル・メモリの手法では，トランザクションは一つのスレッドで投機実行され，不可分に実行されているかのように実行される．もし並列に実行されている他スレッドの処理とトランザクションの処理が競合した場合，トランザクションをロールバックし，初めから再実行する．

トランザクションのロールバックでは，再実行される命令数が多いと，大きなペナルティとなることがある．そこで，トランザクションの開始点に戻るのではなく，その途中に戻る部分ロールバックを行うことでペナルティを削減する手法が提案されている．しかし，これらの手法では，常に最適なチェックポイントへロールバックするわけではない．

本稿では，過去の競合アドレスへのアクセスをチェックポイントとし，最適なチェックポイントを選択する手法を提案する．数の制限なしにチェックポイントを取り，最適なチェックポイントを選択することで，投機失敗時のペナルティを削減する．

開始点のみをチェックポイントとした最適なチェックポイントを選択する手法の評価では，最大 9.2 倍の性能向上を達成できた．

Transactional Memory Selecting the Optimal Rollback Point

Yuji Ito,[†1] Ryota Shioya,[†1,†2] Masahiro Goshima[†1] and Shuichi Sakai[†1]

Transactional Memory is proposed for programmability and performance. In most Transactional Memory systems, programmers specify a sequence which should be synchronized as a transaction. A thread executes a transaction speculatively at a time as if it was executed atomically. If a transaction accesses an address which another parallel thread accesses, the system does a rollback and restarts the transaction.

When a long transaction does a rollback, the penalty is large for many instructions to be re-executed. Therefore, proposals of partial rollback were proposed. Partial rollback is rollback into a transaction to reduce penalty instead of to its start point. However, these proposals can't always do a rollback to the optimal checkpoint.

In this paper, we propose transactional memory selecting the optimal checkpoint from checkpoints which are taken on the point of a memory access to a conflicting address. The transactional memory system can take checkpoints without limit and reduce penalty of restart.

The evaluation of the scheme which selects the optimal checkpoint from start points showed up to a 9.2 times speedup.

1. はじめに

近年では，複数のプロセッサ・コアを 1 つのチップ上に集積したマルチコア・プロセッサが広く普及しており，共有メモリ型の並列プログラムを実行するためのインフラは既に整ったと言ってもよい．にもかかわらず，並列プログラムの普及は遅々として進んでいない．その原因の一つには，同期通信に用いられる**ロック**の存在が挙げられよう．ロックを用いたプログラミングでは，プログラマは，デッドロックや不要なロックによる性能低下など，プログラムの本質ではない問題に多くの注意を払わなければならない．

そのため，ロックを用いない同期通信手法として，**トランザクショナル・メモリ**[1)–8)] が有望視されている．ソース・コード上で**トランザクション**と指定された部分は，**不可分** (atomic) に実行される．より正確には，実際に不可分に実行された場合と同じ結果を与えることが保証される．したがって一般には，いわゆるクリティカル・セクションを，ロック—アンロックで挟む代わりにトランザクションと指定すればよい．トランザクショナル・メモリでは，デッドロックは原理的に発生しない．また，不要な部分をトランザクションとして指定したとしても，以下に述べる投機処理を行えば，大きな性能低下は起こらない．

†1 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†2 日本学術振興会特別研究員 DC
JSPS Research Fellowships for Young Scientists DC

トランザクションの投機実行とペナルティ

トランザクショナル・メモリの実行系の多くは，トランザクションを投機的に実行する．投機を行う実行系では，トランザクションは，別のトランザクションと同期などをとることなく開始される．すると，あるトランザクションと別のトランザクションが同一アドレスにアクセスしてしまうことがあり，それを放置すると各トランザクションが不可分に実行されたのと同じ結果を残すことができなくなってしまう．そこでそのような場合には，それらのアクセスを**競合**として検出し，どちらか一方のトランザクションを「なかったことにする」．そして（一方のトランザクションが終了した後）やり直すのである．このようにして実行系は，トランザクションが実際に不可分に実行されたのと同じ結果を残すことができる．また競合が発生しない場合には，トランザクション同士は並列に実行され，同期のためのオーバヘッドは生じない．

競合の検出は，通常， キャッシュ・コヒーレンス・プロトコルをわずかに拡張するだけで実現することができる．拡張のために，キャッシュ・ラインごとにビットが付加される（2.2 節参照）．

競合検出後に，一方のトランザクションを「なかったことにする」処理を**ロールバック**という．あり得るロールバックに備えて，トランザクション内における状態の更新は可逆的に行う必要がある．そのため（最も基本的な手法では）トランザクションの開始点においてチェックポインティングを行い，ロールバック時にはチェックポイントへと状態を回復する．

図 1 にトランザクションが投機的に実行される様子を示す．同図では，スレッド T_1/T_2 でトランザクション X_1/X_2 がそれぞれ実行され，変数 x に対して，X_1 はライトを，X_2 はリードを行っている．

このとき，X_2 は X_1 の書き込んだ x の値をリードすると，X_1 の実行の途中経過を X_2 が観測することになり，X_1 のアトミシティが満たされない．したがって，これらのアクセスを競合として検出し，いずれか一方のトランザクションをアボートする．ここでは，X_2 をアボートし，X_2 内の命令を再実行する．一方，X_1 では，そのまますべての実行が確定され，コミットされる．

ロールバック時に取り消され（再実行され）る命令の数が，トランザクショナル・メモリの投機失敗の**ペナルティ**として現れる．たとえば，長いトランザクションの終了直前で競合が発生した場合などには，ペナルティは大きくなる．

部分ロールバック

このペナルティ小さくするためには，トランザクションの開始点より下流にロールバックを行う**部分ロールバック** (**partial rollback**) が有効である．最も基本的はトランザクショナル・メモリでは，トランザクションの開始点においてのみチェックポインティングを行い，競合発生時にはトランザクションの開始点までロールバックする．しかし実際には，必ずしも開始点まで戻る必要はない．トランザクショナル・メモリの実行系に対する要請は，トランザクションを実際に不可分に実行した場合と同じ結果を与えることである．したがって，開始点まで戻らなくても，競合を起こした命令より前に戻れば十分である[9),10)]．部分ロールバックを行ったほうが当然ペナルティが小さく，より高い性能が見込まれる．

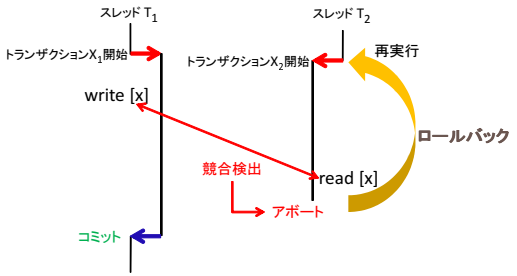

図 1 トランザクションの投機実行
Fig. 1 Speculative Execution of Transactions

部分ロールバックでは，現在の状態からチェックポイントの状態に戻す．チェックポイントによって，各アドレスの値は異なる．そのためにトランザクションによって書き換えられる値のマルチバージョン管理を行う．各チェックポイントの状態は，書き換えられる直前にその値とアドレスを保存しておくことで保持される．この保存先を**ログ**という．ログは，手法によって異なるが，バッファやアドレス空間上の領域である．部分ロールバック時には，ログにある値を用いてチェックポイントの状態を回復する．

部分ロールバックの手法のポイントは，以下の 2 つに分けられる：

チェックポイントの位置 チェックポイントを（トランザクションの開始点以外の）どこに予め設定しておくか．

チェックポイントの管理 競合を起こした命令の直前のチェックポイントを特定するため，複数のチェックポイントをどのようなデータ構造によって管理するか．

以下，それぞれについて説明する．

チェックポイントの位置

チェックポイントの位置としては，1. ネスティッド・トランザクションの開始点 と，2. 過去の競合アドレスへのアクセス が提案されている．

1. ネスティッド・トランザクションの開始点

トランザクション中で別のトランザクションが開始されることをトランザクションの**ネスト**と呼び，ネストされているトランザクションを**ネスティッド・トランザクション**と呼ぶ．ネスティッド・トランザクションは，トランザクション内で呼び出した関数内にトランザクションがあった場合などに現れる．

ネスティッド・トランザクショナル・メモリでは，ネストの内側のトランザクションで競合が発生したとき，最外側のトランザクションの開始点まで戻るのではなく，内側のトランザクションの開始点まで戻ることができる．実際，6)，7)，10) は，内側のトランザクションの開始点に戻ることができる．

2. 過去の競合アドレスへのアクセス

9) では，学習によりチェックポイントを設定する方法が提案されている．この方法では，競合を起こしたアドレスを記録しておき，次にトランザクションが実行された時にはそのアドレスへの初めてのアクセス直前でチェックポインティングを行う．

チェックポイントの管理

前述したように，競合の検出は，キャッシュ・ラインに付加されたビットによって行われる．このビットを更に拡張することによって，競合より前のチェックポイントを特定することができる．

a. キャッシュ・タグによるチェックポイントの特定

既存の手法[7),9)] は，キャッシュ・ラインに付加されたビットによって，① 競合の検出 と，② 競合を起こす前のチェックポイントの特定 の 2 つの処理を行う．このビットは，チェックポイントごとに用意する必要があるため，設定できるチェックポイントの数には強い制限がかかる．また，キャッシュ・ラインがリプレースされ，キャッシュから追い出されると，キャッシュ・ラインに付加されたビットによって ② 競合を起こす前のチェックポイントの特定 することはできない．これらの手法については，2 章で詳しく述べる．

b. ログによるチェックポイントの特定

それに対して我々の提案した手法[10)] では，キャッシュ・ラインに付加されたビットによっては ② 競合を起こす前のチェックポイントの特定 は行わない．② 競合を起こす前のチェックポイントの特定 はアドレス空間上にとられたログによって行う．キャッシュ・ラインに付加されたビットは，① 競合の検出 と ③ ログの作成の補助 のために用いる．キャッシュ・ラインに付加されたビットによっては ② 競合を起こす前のチェックポイントの特定 を行わないため，設定できるチェックポイントの数には制限がない．また，リプレースされたキャッシュ・ラインがあっても，その競合の検出さえできれば，競合を起こす前のチェックポイントを特定できる．この手法については，3 章で詳しく述べる．

10) では，チェックポイントの位置 については，1. ネスティッド・トランザクションの開始点 を採用していた (1+b)．本稿では，2. 過去の競合アドレスへのアクセス を組み合わせる方法 (2+b) について述べる．

開始点のみをチェックポイントとした最適なチェックポイントを選択する手法[10)] をマルチプロセッサ・シミュレータ GEMS に実装し，最適なチェックポイントを選択する効果を評価した結果，最大 9.2 倍の性能向上を達成することができた．評価に関しては，4 章で述べる．

2. 関 連 手 法

本章では，関連手法として，LogTM[6)] と学習によるチェックポインティング手法[9)] について述べる．

2.1 チェックポイントの位置

LogTM と学習によるチェックポインティング手法では，チェックポイントの位置が異なる．

LogTM

LogTM では，すべてのトランザクション開始点をチェックポイントとする．実際には，必ずしも開始点に戻る必要はないが，トランザクションを不可分に実行するという意識によるものである．

学習によるチェックポインティング手法

過去に競合したアドレスへの初めてのアクセス直前にチェックポイントを取る．一度競合したアドレスは，ロールバック後でも並列に実行されているトランザクションによってアクセスされている可能性が高く，もう一度競合する可能性が高い．そこで，あるアドレスで競合が起きた場合，次もそのアドレスで競合が起きると予想し，次のアクセス前にチェックポイントを取っておく．ただし，最も外側のトランザクション開始点では，チェックポイントを取っておく．

2.2 チェックポイントの管理

チェックポイントの管理方法は，LogTM と学習によるチェックポインティング手法とも

に，キャッシュにより① 競合の検出 と，② 競合を起こす前のチェックポイントの特定 の 2 つの処理を行う．まず，通常の競合検出について述べ，次に関連手法でのチェックポイントの特定について述べる．

① 競合の検出

キャッシュ・コヒーレンス・プロトコルを拡張し，リード/ライト・ビットを用いて競合を検出する．リード/ライト・ビットとは，トランザクションによる投機状態を表すキャッシュ・ラインごとに設けられた以下の 2 ビットである．

リード・ビット　トランザクションによるリード・アクセスが行われたらセットされる

ライト・ビット　トランザクションによるライト・アクセスが行われたらセットされる

キャッシュ・コヒーレンス・プロトコルにより，リード・ビットがセットされているラインへの無効化の要求や，ライト・ビットがセットされているラインへの共有，無効化の要求があれば，競合を検出する．これらのビットは，コミットやロールバック時にクリアされる．

キャッシュ・ラインがリプレースされた場合，LogTM では，メモリに書き戻し，拡張したディレクトリ・プロトコルを用いて，ディレクトリがそのラインを投機状態を管理する．ディレクトリがそのラインの投機状態を管理しているので，そのラインへの要求によって競合を検出できる．

学習によるチェックポインティング手法では，リプレースされたキャッシュ・ラインは，メモリに書き戻さず，すべてのリード/ライト・ビットごと専用のバッファに保存される．他スレッドからアクセス要求があった場合，キャッシュだけでなく，そのバッファの中のラインについてもリード/ライト・ビットを調べて競合を検出する．

② 競合を起こす前のチェックポイントの特定

関連手法では，**図 2** のようなキャッシュを用いる．競合時，リード/ライト・ビットを調べることでチェックポイントを特定する．各手法でのリード/ライト・ビットは以下のような意味を持つ．

LogTM　各深さでの投機状態がそれぞれのリード/ライト・ビットにセットされる．例えば，R_2 がセットされているラインは，深さ 2 のトランザクションによってリード・アクセスされたことを示す．

学習によるチェックポインティング手法　あるチェックポイントから次のチェックポイントまでの投機状態がそれぞれのリード/ライト・ビットにセットされる．例えば，3 つ目のチェックポイントを取ったら，4 つ目のチェックポイントを取るまでリード/ライトは，それぞれ R_3，W_3 にセットする．

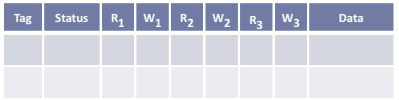

図 2　チェックポイントを選択するキャッシュ
Fig. 2　Cache selecting the optimal checkpoint

チェックポイントとチェックポイントの間の部分をセクションと呼ぶと，トランザクションはチェックポイントによって複数のセクションに分割される．それぞれのリード/ライト・ビットはそれらのセクション 1 つ 1 つに対応する．従って，リード/ライト・ビットを調べることで，競合を起こした命令がどのセクションで実行されたのかが判別できる．判別したセクションの頭のチェックポイントにロールバックすることで，競合を起こした命令以前に戻ることができる．

例えば，2 つ目のチェックポイントから 3 つ目のチェックポイントまでのセクションは，R_2，W_2 によって競合の検出を行う．R_2 のみがセットされたラインに無効化要求があれば，2 つ目のチェックポイントへロールバックすればよい．

2.3　関連手法の問題点

LogTM の部分ロールバック

LogTM では，終了した内側の開始点に部分ロールバックすることはできない．なぜなら，内側のトランザクションが終了した場合，外側のトランザクションにマージするためである．マージを行うのは，後に同じ深さのトランザクションが実行されることがあるからである．そのために，終了した深さのリード/ライト・ビットは，外側のリード/ライト・ビットとそれぞれ OR を取ってマージし，クリアされる．以降，すでに終了した内側の命令は，外側のトランザクション内で実行された命令として扱われる．トランザクションのネストを意識して深さ別で管理する限り，この問題を解決することは難しい．

学習によるチェックポインティング手法のログ

学習によるチェックポインティング手法では，ログのサイズが有限であり，大きなトランザクションは部分ロールバックできない．ログの中でチェックポイントのバージョン管理をしているので，ログでその状態を保持できなくなれば，チェックポイントの状態を回復できない．つまり，メモリ・アクセスや競合の多い大きなトランザクションについては，部分ロールバックできない．

チェックポイント数の制限

2 つの関連手法でのチェックポイントの数は，リード/ライト・ビットの数に制限される．チェックポイントの特定のためには，どのセクションで競合が起きたかを調べなければならない．それには各セクションに対応したリード/ライト・ビットがそれぞれ必要である．しかし，LogTM ではネスティッド・トランザクションの深さが，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の数がリード/ライト・ビットの数を越えた場合，新たなセクションに対応するリード/ライト・ビットがない．従って，以降のセクションの競合を検出することはできないため，チェックポイントを取ることはできない．

キャッシュ・ラインのリプレース

2 つの関連手法ともに，投機状態にあるキャッシュ・ラインのリプレースによって，チェックポイントの特定が不可能になることがある．

LogTM では，ディレクトリによってリプレースされたキャッシュ・ラインが投機状態にあることは認識できる．しかし，深さ別の投機状態の区別は失われ，そのラインの競合による部分ロールバックは不可能である．

学習によるチェックポインティング手法では，リプレースされたラインはすべてのリード/ライト・ビットごとバッファに保存されているため，どのセクションで競合したかはまだ判別できる．しかし，そのバッファは有限であるため，バッファからも投機状態にあるラインが溢れてしまう場合は，部分ロールバックどころかトランザクションを投機的に実行することも不可能となる．

3. 提 案 手 法

3.1 提案手法の目的

提案手法では，学習によるチェックポインティング手法[9] と同様のチェックポイントの設定を行い，ログによってチェックポイントを特定する．過去の競合アドレスへの初めてのアクセス直前にチェックポイントを取る．競合するかもしれない命令の履歴をログに取りながらトランザクションを実行することで，競合時にその最も古い競合命令を検索し，その直前のチェックポイントを選択する．本手法により，再実行される命令数を最小にできる．

図 3 では，各手法でのロールバックの様子を表している．同図では，他スレッドにより x の値を書き換えられてロールバックを行っている．内側のトランザクションを最外側のトランザクションの一部として扱っている**平坦化 (flattening)**[2)–6)] では，トランザクション内の命令全てを再実行する．当然，そのペナルティは大きい．LogTM では，終了した内側

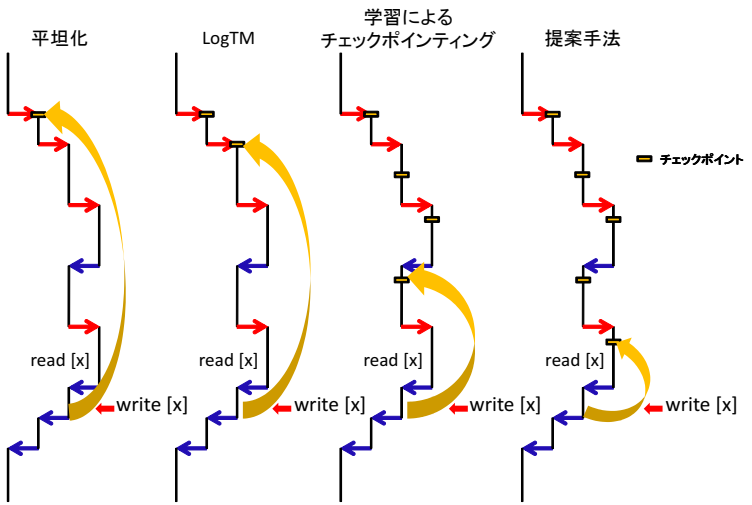

図 3 提案手法の目的
Fig. 3 Our goal

の開始点に部分ロールバックをすることはできないため，再実行される命令が多くなってしまう．また，チェックポイントを取れる深さに制限があるため，複雑にネストしたトランザクションでは，最適なロールバックができない．チェックポイントが 4 つまでの学習によるチェックポインティングでは，5 つ目のチェックポイントを取ることができず，4 つ目のチェックポイントにロールバックする．また，トランザクションのサイズが小さくないと，2.3 節で述べたように，チェックポイントの数の制限だけでなく，そもそもトランザクションが投機的に実行できなくなることがある．提案手法では，学習により x の直前にチェックポイントを取る．チェックポイント数に制限がないため，それ以前にチェックポイントをいくつでも取ることができる．その中から最適なチェックポイントとして x の直前のチェックポイントを選び，再実行される命令数を最小にする．

3.2 ログによるチェックポイントの特定

ログを用いて，競合する最も古い命令とその直前のチェックポイントを検索することで最適なチェックポイントを選択する．

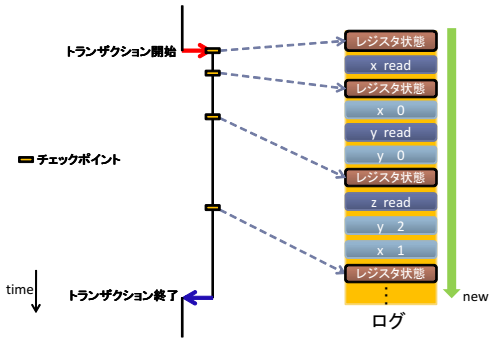

図 4 履歴の取り方
Fig. 4 Logging

チェックポイント特定の方針

まず，競合する最も古い命令になり得るすべての命令の履歴と，チェックポイントの履歴を実行順にログに取りながらトランザクション内の命令を実行する．ここでの履歴とは，いつその命令を行い，いつチェックポイントを取ったかの記録のことである．また，ログは，LogTM と同様でアドレス空間上の領域である．競合する命令とはメモリ・アクセスのみであるので，命令の履歴はアドレスを，チェックポイントの履歴はその時のレジスタ状態を保存しておく．また，ライト・アクセスについては，LogTM と同様に，ロールバックのために書き換える前の古い値も保存しておく．最適なチェックポイントの選択は，ログを検索することで行う．

図 4 に履歴の様子を示す．同図では，チェックポイントとメモリ・アクセスの履歴を下が新しいものとなるように記録している．例えば，ログの上から 2 行目は，x がリードされたことを示し，上から 4 行目は，x がライトされ，そのライト以前の値が 0 であったことを示している．同図で，他スレッドによる x のリードがあった場合，古い順に，ここでは上からログを調べれば，上から 4 行目の x へのライトが競合する最も古い命令であるとわかる．そして，その命令から新しい順に検索し，2 つ目のチェックポイントが最適であることを識別する．

ログを用いるメリット

キャッシュを用いてチェックポイントを特定する既存手法に対して，ログを用いる提案手法のメリットを述べる．

- チェックポイント数に制限なし
  キャッシュのリード/ライト・ビットの数によって設定できるチェックポイントの数が制限されることはない．本手法のキャッシュの役割は，詳細は 3.4 節で述べるが，競合の検出とキャッシュのリード/ライト・ビットを用いてログの作成の補助である．ログの検索によって競合した命令の実行されたセクションを識別するため，キャッシュのリード/ライト・ビットを用いて行わない．また，ログはアドレス空間上の領域であるため，保存するチェックポイントの状態がログから溢れることはない．
- キャッシュ・ラインのリプレース
  本手法では，チェックポイントの特定に影響はなく，キャッシュ・ラインをリプレースできる．あるキャッシュ・ラインがリプレースされた場合，LogTM と同様に，メモリに書き戻し，ディレクトリがその投機状態を管理することで対応する．以降，キャッシュの役割の 1 つである競合の検出はディレクトリによって行われる．また，履歴はアクセス時に取られるので，リプレース後には，もう 1 つのキャッシュの役割であるログ作成の補助は必要ない．従って，キャッシュ・ラインのリプレースには何の問題もない．

3.3 履歴を取るメモリ・アクセス

最適なチェックポイントを選択するための命令の履歴と，チェックポイントの状態に戻すための履歴を実行順に取る必要がある．これらの取るべき命令の履歴は，以下の 3 種類のメモリ・アクセスである．

- 競合する最も古いメモリ・アクセスになり得るメモリ・アクセス
  初アクセスであるリード 初アクセスならば，そのアドレスが競合した場合，競合する最も古いアクセスになり得る．初アクセスでない最初のリードについては，履歴をとる必要はない．なぜなら，そのリード以前にライトしたならば，必ずそのライトがより古い競合するアクセスであるからである．
  初ライト 初ライトとは，あるアドレスに対する最初のライトである．あるアドレスに対する初ライトは，それが初アクセスでなくとも，競合する最も古いメモリ・アクセスになり得る．「初アクセスであるリード」で述べたように，初ライト以前にリードがあっても，他スレッドによるリードは自スレッドのリードと競合しないからである．
- ロールバックのために履歴を取るべきメモリ・アクセス
  チェックポイント後の初ライト あるチェックポイントから次のチェックポイント直前

までに値を書き換えられたアドレスに対しては，そのチェックポイント時の値を履歴として取っておかなければならない．

あるラインについて履歴を取った後にそのラインがリプレースされ，同じトランザクションが再び同じラインにアクセスした場合，もう一度履歴を取ることがある．再びアクセスしたそのラインのリード/ライト・ビットはクリアされているために，過去にアクセスしたことがあるかは不明であり，初アクセスとして扱うからである．余分な履歴によってログのサイズや検索時間は増えるが，選択される最適なチェックポイントが変わることはない．

3.4 実　　装

履歴の取り方

キャッシュのリード/ライト・ビットを用いて履歴を記録するか識別し，各履歴はアドレス空間上にとられたログに保存される．このキャッシュの役割が既存手法と異なる．キャッシュは，① 競合の検出 と ③ ログの作成の補助 のためにリード/ライト・ビットを設ける．**図 5** に履歴を取るべきメモリ・アクセスを識別するキャッシュの構成を示す．同図のキャッシュは，直前のチェックポイントの前と後それぞれのリード/ライト・ビットを持つキャッシュである．直前のチェックポイント前のリード/ライト・ビットを R_p，W_p とし，直前のチェックポイント後のリード/ライト・ビットを R_c，W_c としている．このとき，3.3 節で述べた履歴を取るメモリ・アクセスは，以下のように認識される．

- 初アクセスであるリード：すべてのリード/ライト・ビットがセットされていないキャッシュ・ラインへのリード・アクセス
- 初ライト：W_p，W_c ともにセットされていないキャッシュ・ラインへのライト・アクセス
- 開始点後の初ライト：W_c がセットされていないキャッシュ・ラインへのライト・アクセス

処理の流れ

トランザクションの実行は以下のように行われる．

(1) トランザクション開始
最も外側となる開始点では，その時のレジスタ状態をログに保存後，トランザクション内の命令を開始する．

(2) メモリ・アクセス
アクセスするキャッシュ・ラインのリード/ライト・ビットを見て，必要なら履歴をログに取る．あるアドレスへの初めてのアクセスなら，保存してある過去の競合アドレスの中にそのアドレスがあるか調べる．そのアドレスがあったらチェックポインティングを行う．その後，アクセスを行い，リードならば R_c を，ライトならば W_c をセットする．

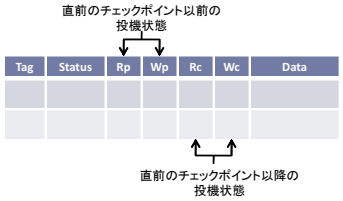

図 5 ログ作成のためのキャッシュ
Fig. 5 Cache for making a log

(3) チェックポインティング
その時のレジスタ状態をログに保存する．今の R_c と W_c は，新たなチェックポイントよりも以前のものとなるので，R_p と W_p にそれぞれ OR を取ってマージ後，クリアされる．

(4) コミット
トランザクション内のすべての命令が終了したら，ログを初期化し，リード/ライト・ビットをすべてクリアすることでトランザクションをコミットする．

(5) ロールバック
競合したアドレスを保存する．次に，3.2 節で述べたように，ソフトウェアにより，ログを古い順に調べて競合する最も古い命令を探し，最適なチェックポイントを選択する．そのチェックポイントのメモリ状態をログに取ってある古い値を用いて回復する．そして，R_c，W_c をクリアして，ログを古い順に調べて R_p，W_p をチェックポイントの状態に戻す．最後に，レジスタ状態を回復し，チェックポイントから命令を再実行する．

4. 評　　価

ここでは，チェックポイントが開始点のみであるが，提案手法と同様の最適なチェックポイントを選択する手法の評価を行う．

4.1 評価環境

OS も含めた機能シミュレータ Simics と実行駆動型マルチプロセッサ・シミュレータ GEMS

表 1 パラメータ
Table 1 Parameter

processor	IPC 1(in-order), 16cores
L1D cache (private)	32kB, 4way, 64Bytes line
L1 cache latency	1cycle
L2 cache (shared)	8MB, 8way, 64Bytes line
L2 cache latency	20cycles
Memory latency	200cycles
Directory latency	6cycles
Interconnection network latency	3cycles

を合わせて用いた．GEMS では，Ruby モジュールを用いて LogTM のメモリ・シミュレーションが可能であり，これを修正して最適なチェックポイントの選択する手法を実装した．各パラメータは表 1 の通りである．平坦化，LogTM，最適なチェックポイントを選択する手法のそれぞれについて，全スレッドで最初のトランザクション開始から最後のトランザクション終了までの処理時間を計測した．トランザクション同士が競合した場合，新しい方のトランザクションをロールバックするものとした．ベンチマークは，GEMS 付属の microbenchmarks と SPLASH-2 Benchmarks の raytrace，barnes を用いた．raytrace，barnes については，ロック部分をトランザクションに修正した．

4.2 評 価 結 果

評価結果を**図 6** に示す．縦軸は平坦化を 1 とした相対速度，横軸はスレッド数である．slist 以外のプログラムでは，LogTM と同等か性能低下が見られた．slist では，最大 9.2 倍の性能向上を確認した．

4.3 考　　察

今回測定した多くのプログラムでは，ログ・サイズ増大によるキャッシュ・ヒット率低下やログ検索時間が性能低下を引き起こしている．トランザクション内の命令数が少なかったり，開始点の間に命令を挟むことがないなど，内側の開始点をチェックポイントとしても効果がないトランザクション構造であるためである．また，raytrace，barnes は，トランザクションがネストしておらず，部分ロールバックすることができないため，性能差がない．今後，このようなトランザクションに対して，過去に競合したアクセス直前にチェックポインティングする評価を行う必要がある．

性能の向上が見られた slist は，外側のトランザクションで共有リストを検索し，内側のトランザクションで共有カウンタをインクリメントしている．トランザクションの指定が適切でなく，あまりいい例ではないが，内側のトランザクションが終了してから共有カウンタについての競合が検出されることが多いプログラムである．LogTM では，終了した内側

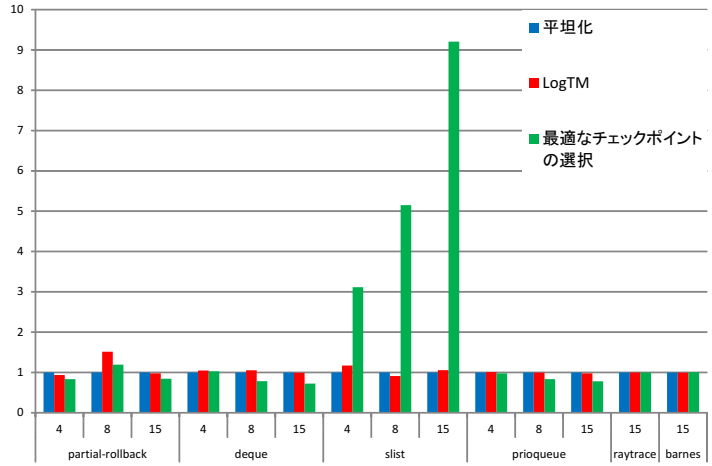

図 6 評価結果
Fig. 6 Result

のトランザクションの開始点に部分ロールバックすることはできないため，トランザクション内すべての命令を再実行している．一方，最適なチェックポイントを選択する手法では，内側の開始点を最適なチェックポイントとして選択し，リスト検索をやり直すことがないため，再実行される命令が少ない．

5. おわりに

本稿では，過去の競合アドレスへのアクセスをチェックポイントとし，ログによってチェックポイントを特定する手法を提案した．開始点以外のチェックポイントを数の制限なく取り，最適なチェックポイントを選択し，投機失敗時のペナルティを削減する．既存手法と比較して，チェックポイント数に制限はなく，キャッシュ・ラインがリプレースしても最適な部分ロールバックできる．

評価では，チェックポイントは開始点のみである最適なチェックポイントを選択する手法の評価を行った．最適なチェックポイントの選択の効果として，最大 9.2 倍の性能向上が見られた．

今後の課題として，提案手法の評価や，ログによるチェックポイントの特定のオーバヘッ

Vol.2010-ARC-190 No.9
2010/8/3

ドの削減を検討する必要がある．

参 考 文 献

1) Herlihy, M., Eliot, J. and Moss, B.: Transactional Memory: architectural support for lock-free data structures, *Proceedings of the 20th Annual International Symposium on Computer Architecture* (1993).
2) Moore, K.E., Hill, M.D. and Wood, D.A.: Thread-level Transactional Memory, Univ. of Wisconsin Computer Sciences Technical Report CS-TR-2005-1524, Dept. of Computer Sciences, University of Wisconsin (2005).
3) Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.：Unbounded Transactional Memory, *Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (2005).
4) Blundell, C., Devietti, J., Lewis, E.C. and Martin, M. M.K.: Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory, *Proc. of the 34th Annual Intnl. Symp. on Computer Architecture* (2007).
5) Ravi, R., Maurice, H. and Konrad, L.: Virtualizing Transactional Memory, *Proceedings of the 32nd Annual International Symposium on Computer Architecture* (2005).
6) Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D. and Wood, D.A.: LogTM: Log-based Transactional Memory, *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture* (2006).
7) 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, *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems* (2006).
8) Ceze, L., Tuck, J., Torrellas, J. and Cascaval, C.: Bulk Disambiguation of Speculative Threads in Multiprocessors, *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (2006).
9) Waliullah, M.M. and Stenstorm, P.: Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems, *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium* (2008).
10) 伊藤悠二，塩谷亮太，五島正裕，坂井修一：最適なロールバック・ポイントを選択するネスティッド・トランザクショナル・メモリ，情報処理学会研究報告 2009-ARC-184 (2009).