Vol.2009-ARC-184 No.5
2009/8/4

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

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，他スレッドから見れば，アトミックに実行されているかのように実行される．そのため，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，トランザクションの処理をすべて破棄し，初めから再実行する．

プログラムの構成上，トランザクション実行中にトランザクションが呼び出されることがある．このようなネスティッド・トランザクションについてもトランザクションとして正しく実行される必要がある．最も外側のトランザクションのアトミシティのためにロールバック時のロールバック・ポイントを最も外側のトランザクション開始点とすれば，その内に含まれるトランザクションもアトミシティが保たれる．しかし，競合とは関係のない処理もまた再実行されることがある．

本稿では，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・メモリを提案する．最も外側のトランザクション開始点にロールバックする必要がない場合，再実行される処理が最小限になるようなロールバック・ポイントをネスティッド・トランザクション開始点の中から選択することができる．

Nested Transactional Memory Selecting the Optimal Rollback Point

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

Lock-based synchronization is common in parallel programming. However, lock-based synchronization may make deadlocks and reduce parallelism with unnecessary locks. On the other hand, 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 one transaction speculatively at a time as if it was executed atomically. If a transaction accesses an address which another parallel thread accesses, the system discards all updates by the transaction and restarts the transaction.

Software composition allows a transaction to invoke other transactions. These transactions should be executed correctly. The most outer transaction is corrected by selecting its starting point as a rollback point when an inner transaction conflicts. However, the system may also restart sequences unrelated to the conflict.

In this paper, we present Nested Transactional Memory Selecting the Optimal Rollback Point. If the most outer transaction doesn't need rollback, the system improves performance by selecting a rollback point with minimum penalty for restart.

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いる．複数のスレッドを並列に実行することができるマルチコア・プロセッサが普及したことで，並列プログラムを実行するインフラが整った．このような環境を最大限利用するために，並列プログラミングの重要性が高まっている．

マルチコア・プロセッサでは，メッセージ・パッシング型に比べ，並列に実行されるス

†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

レッドがデータを共有して扱う共有メモリ型の並列プログラムが用いられることが多い．しかし，共有メモリの扱い方は非常に複雑であり，効率的な並列プログラムを書くことは難しい．その原因の一つは，並列プログラミングで良く用いられているロックを使った手法にある．プログラマはデッドロックや不要なロックによる並列性の低下などの問題に常に意識を配らなければならない．そこで，ロックを用いない同期通信機構として，**トランザクショナル・メモリ (Transactional Memory)** と呼ばれる手法が提案されている[1)–8)]．

トランザクションとは，一つのスレッドで実行される一連の処理をまとめたものである．トランザクションは，他スレッドから見れば，アトミックに実行されているかのように実行される．

トランザクショナル・メモリの多くの手法では，トランザクションを投機実行する．他のスレッドのメモリ・アクセスと競合してトランザクションをアトミックに実行できない可能性がある場合は，トランザクションを停止してそれまでの処理を破棄して再実行する．詳細は，2.1 節で述べる．

トランザクショナル・メモリの問題点の一つとして，**ネスティッド・トランザクション (nested transaction)** の扱い方がある．ネスティッド・トランザクションとは，内側にトランザクションがネストされたトランザクションである．

ネスティッド・トランザクションの最も簡単な扱い方としては，内側のトランザクションを外側のトランザクションの一部として扱うことが挙げられる．これにより，外側のトランザクションのアトミシティが保たれる．しかし，このような実装を行った場合，内側のトランザクションの競合により，外側のトランザクションの処理までが破棄されてしまい，実行効率が低下する．

本稿では，トランザクションがロールバックする際に，再実行される処理が最小限になる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・メモリを提案する．トランザクションを最初から再実行せず，ある処理以降から再実行してもトランザクションのアトミシティを保てる場合がある．このとき，トランザクションの途中にロールバックする**部分ロールバック (partial rollback)** を行い，戻り先であるロールバック・ポイントから再実行することで，再実行時のペナルティを最小限にすることができる．提案手法では，必要なメモリ・アクセスのログを取り，ロールバックする際に，そのログを基に最適なロールバック・ポイントを選択し，部分ロールバックして，再実行を行うことができる．

本論文の構成を以下に示す．第 2 章では，背景として，トランザクショナル・メモリについての概要や，その実装について述べる．また，ネスティッド・トランザクションについてもここで述べる．第 3 章では，部分ロールバックをサポートする既存手法について述べ，第 4 章では提案手法について述べる．最後に，第 5 章でまとめを行う．

2. トランザクショナル・メモリ

本章では，提案手法の背景として，トランザクショナル・メモリとその問題点の一つであるネスティッド・トランザクションについて説明する．

2.1 トランザクショナル・メモリの概要

トランザクショナル・メモリにおけるトランザクションとは，第 1 章で述べたように，一つのスレッドで実行される一連の処理をまとめたものである．トランザクションは以下のような性質を持つ．

アトミシティ (atomicity) トランザクションはアトミックに実行される．他のスレッドからトランザクション内の処理の途中経過は観測されない．

トランザクショナル・メモリの多くの手法では，トランザクションは投機的に実行される．トランザクションの投機実行中，他のスレッドで並列に実行されるメモリ・アクセスが，トランザクション中で行われたメモリ・アクセスと同一アドレスであった場合，これを競合として検出する．競合が検出された時には，トランザクションのアトミシティを保つために，片方のトランザクションを停止させ，それまでの処理をすべて破棄して再実行する．また，停止されることなくトランザクション中の処理をすべて終了したトランザクションは状態を確定させる．トランザクションを停止させ，それまでの処理をすべて破棄する操作を**アボート (abort)**，処理をすべて終了したトランザクションの状態を確定する操作を**コミット (commit)** と呼ぶ．

図 1 に，トランザクションの実行の様子を示す．同図では，スレッド 1 でトランザクション 1 が実行され，スレッド 2 でトランザクション 2 が実行されている．アドレス A に対して，トランザクション 1 はライト・アクセスを行い，トランザクション 2 はリード・アクセスを行っている．このとき，トランザクション 2 はトランザクション 1 の書き込んだ値をリードする．しかし，トランザクション 1 内の処理の途中経過をトランザクション 2 が観測することになり，トランザクション 1 のアトミシティが満たされない．従って，アドレス A に対するトランザクション 1 のライト・アクセスとトランザクション 2 のリード・アクセスを競合として検出し，どちらかのトランザクションをアボートしなければならない．ここでは，トランザクション 2 をアボートし，再実行する．一方，トランザクション 1 では，そのまますべての処理が確定され，コミットされる．

Vol.2009-ARC-184 No.5
2009/8/4

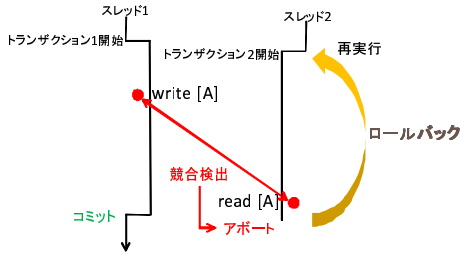

図 1 トランザクションの実行
Fig. 1 Transaction execution

2.2 トランザクショナル・メモリの利点

ロックを用いた同期機構と比較した時のトランザクションを投機実行するトランザクショナル・メモリを用いた同期機構の利点は，

- デッドロックなし：トランザクションは並列に投機実行されるので，複数のリソースに対するロックの取得はなく，デッドロックしない．
- 高い並列性：ロックを用いた場合，競合する可能性が少しでもあるならば，プログラマは排他制御部分にロックをかけなければならない．ロックをかけた処理はシリアライズされるため，競合しないほとんどの場合においても並列に実行できない．一方，トランザクショナル・メモリを用いると，競合してアボートされ，再実行するトランザクションが一部あるものの，トランザクションは並列に実行される．
- プログラマの負担の軽減：ロックを用いた場合，プログラムを正しく動かすためにデッドロックには大きな注意を払わなければならない上，より速いプログラムのためには並列可能な部分を正しく認識し，適切な粒度でロックをかける必要がある．トランザクションが並列に投機実行されるトランザクショナル・メモリでは，このようなプログラマの負担は大きく減る．

2.3 トランザクショナル・メモリの実装例

以下では，後に述べるトランザクショナル・メモリの手法のために，通常のトランザクショナル・メモリの実装例を説明する．

トランザクションの実行の概要を以下に示す．

(1) レジスタ状態の保存
開始点へのロールバックのためにトランザクション開始時のレジスタ状態を保存しておく．

(2) 投機状態をキャッシュに保持
トランザクションによるメモリ・アクセスは，投機状態としてキャッシュに保持する．

(3) 競合検出
トランザクションは投機実行されるので，他スレッドとの競合を検出しなければならない．この競合はキャッシュ・コヒーレンス・プロトコルを用いて検出する．

(4) コミットまたはアボート
トランザクション内のすべての処理が終了したらコミットする．競合してアボートする場合は，開始時の状態にロールバックし，トランザクションを再実行する．

投機状態

キャッシュ上に保持されるトランザクションによるメモリ・アクセスは，投機状態をキャッシュ・ライン毎に以下の 2 つのビットをつけて管理する．

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する．

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する．

これらのビットを用いて，どのキャッシュ・ラインがトランザクションによって投機的なアクセスをされたかを識別し，競合の検出やアボート時の投機状態の破棄を行う．

競合検出

投機実行中のトランザクションがアクセスしたアドレスは，他のスレッドからアクセスされることがある．このとき，トランザクションのアトミシティが満たされないアクセスを競合として検出しなければならない．そのようなアクセスのパターンは，以下の 3 パターンがある．

read after write あるアドレスが，トランザクションによってライト・アクセスされ，その後，他のスレッドによってリード・アクセスされる．トランザクションが変更した値が他のスレッドによって観測されることになり，アトミシティを満たさない．

write after read あるアドレスが，トランザクションによってリード・アクセスされ，その後，他のスレッドによってライト・アクセスされる．トランザクションが実行中であるにも関わらず値が変更されることになるため，アトミシティを満たさない．

write after write あるアドレスが，トランザクションによってライト・アクセスされ，その後，他のスレッドによってライト・アクセスされる．write after write と同様に，アトミシティを満たさない．

こうした競合するメモリ・アクセスのパターンの検出は，キャッシュ・コヒーレンス・プロトコルを用いて行われる．あるプロセッサがあるアドレスにアクセスする場合，キャッシュ・コヒーレンス・プロトコルにより，そのアドレスに対応するキャッシュ・ラインの状態を変えるよう他のプロセッサに要求が送られる．他のプロセッサにおいてライト・アクセスがあると，そのプロセッサからそのラインに対する無効化要求が送られる．トランザクションを実行しているプロセッサは，その無効化要求を受け取り，そのラインのリード/ライト・ビットを参照することで write after read，write after write の競合を検出する．また，あるラインへトランザクションがライト・アクセスした後は，他のプロセッサにおけるラインは無効化されている．それらのプロセッサがそのラインへアクセスすると，トランザクションを実行しているプロセッサへそのラインに対するデータ要求が送られる．この要求を受け取ったプロセッサは，そのラインのライト・ビットを参照することで read after write の競合を検出する．

コ ミ ッ ト

トランザクション中のすべての処理を終了したら，リード/ライト・ビットをすべて 0 にすることでクリアして，コミットする．

ア ボ ー ト

競合を検出して，アボートする時は，開始時の状態にロールバックする．レジスタ状態は，あらかじめ保存してあった開始時のレジスタ状態により回復する．メモリ状態は，ライト・ビットがセットされているキャッシュ・ラインを無効化することで，トランザクションによって書き換えられる前のメモリ状態を回復する．リード/ライト・ビットについてはコミット時と同様にすべてクリアする．

2.4 ネスティッド・トランザクション

ネスティッド・トランザクションは，関数やメソッドをトランザクションとして指定した場合に実行されることがある．トランザクショナル・メモリを用いた並列プログラムを書く場合，プログラマは排他制御が必要な一連の処理をトランザクションとして指定する．トランザクションは，関数やメソッドといったものに適合しやすく，関数やメソッドがトランザクションとして指定されることが多いと考えられる．実際 Java では，synchronized メソッドを用いて，メソッドを単位とした排他制御を行うことができる．例えば，synchronized メソッドをトランザクションとして実装した場合，synchronized メソッドを実行中に synchronized メソッドの呼び出しがあると，ネスティッド・トランザクションを実行することになる．

ネスティッド・トランザクションを扱うトランザクショナル・メモリである**ネスティッド・トランザクショナル・メモリ (Nested Transactional Memory)** では，深さによらずネスティッド・トランザクションに対応する必要がある．その深さが制限されると，プログラマは呼び出される関数にトランザクションが含まれているか逐一チェックしなければならず，プログラミングの大きな負担となるからである．

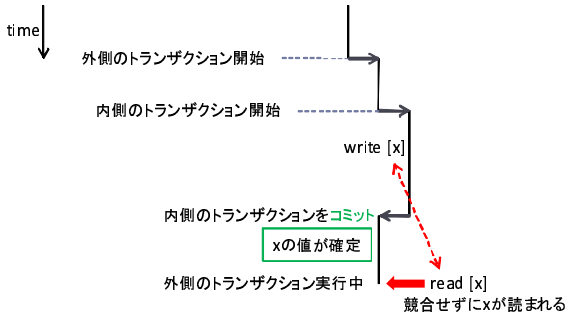

図 2 内側のトランザクションのコミット
Fig. 2 Commit an inner transaction

2.5 ネスティッド・トランザクションの投機

ネスティッド・トランザクションもまたトランザクションであるから，アトミシティを保つように実行されなければならない．しかし，ネスティッド・トランザクションを投機実行した場合，通常のトランザクションとは違って，内側のトランザクション中の処理をすべて終了してもコミットすることはできない．なぜなら，**図 2** に示すように，内側のトランザクションをコミットすると，外側のトランザクションが実行中であるにもかかわらず，コミットによって確定された状態が他のスレッドによって観測されてしまうからである．

既存のトランザクショナル・メモリの多く[2)–6)] は，第 1 章で述べたように，内側のトランザクションを最も外側のトランザクションの一部として扱っている．その様子を**図 3** に示す．最も外側となるトランザクションの実行中は，その内側のトランザクション中の処理も最も外側のトランザクションの処理として実行される．つまり，実際にはネストされていない一つのトランザクションとして実行されることになる．この内側のトランザクションが外側のトランザクションの一部として実行されることを**平坦化 (flattening)** されているという．

図 4 に，平坦化のデメリットとなるロールバックの様子を示す．平坦化を行う場合，競合

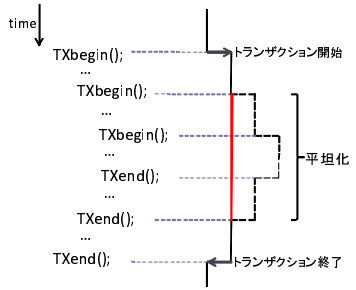

図 3 ネスティッド・トランザクションの平坦化
Fig. 3 Flattening a nested transaction

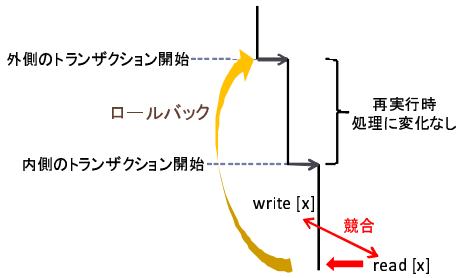

図 4 平坦化のデメリット
Fig. 4 Demerit of flattening

によるロールバック時には，最も外側のトランザクションごとアボートして最初から再実行しなければならない．しかし，その外側のトランザクションが競合していないならば，競合した内側のトランザクションの直前までは当初と全く同様の処理を行うので，それらの処理を繰り返す分だけオーバヘッドとなる．

3. 部分ロールバックをサポートするトランザクショナル・メモリ

2.5 節で述べたように，平坦化では再実行時のペナルティが大きいので，部分ロールバックを検討する．

Moore らは，トランザクションによって上書きされてしまう書き換えられる前の古い値をログと呼ばれる領域へ退避させる，LogTM という手法を提案している[6]．Moravan らは，この LogTM を拡張し，部分ロールバックをサポートした手法を提案している[7]．本章では，まず LogTM について述べ，部分ロールバックをサポートした手法とその問題点について述べる．

3.1 LogTM

LogTM は，トランザクションによって上書きされてしまう書き換えられる前の古い値をログと呼ばれる領域へ退避させる．投機状態と競合検出については，2.3 節で述べた手法と同様に，キャッシュにリード/ライト・ビットを設けて管理する．

ロ　　グ

ログとは，仮想メモリが割り当てられたスレッド毎の領域である．ログには，トランザクションによって書き換えられるラインの仮想アドレスと古い値を保存する．2.3 節での手法の問題点の一つとして，投機状態にあるラインがライトバックした場合，メモリ上にあるトランザクション開始時の値が上書きされる．これではロールバックすることができない．しかし，LogTM では，ログに古い値を保持することで，このような場合でもロールバックすることを可能にしている．

トランザクションの実行

LogTM でのメモリ・アクセスとコミットとアボートについて述べる．

- メモリ・アクセス
  メモリ・アクセス時には，リード/ライト・ビットをセットする．ライト・アクセス時には，上で述べたように，ログへ仮想アドレスと古い値を保存する．このとき，複数回同じアドレスの古い値を取る必要はない．なぜなら，開始点へロールバックするのに必要な分だけで十分だからである．LogTM では，ライト・アクセス時に，そのラインのライト・ビットを調べ，ログへ取る量を減らしている．これは，そのラインのライト・ビットがすでにセットされているならば，古い値はすでにログへ退避済みだと認識できるためである．
- コミット
  コミット時には，すべてのリード/ライト・ビットをクリアし，ログを初期化する．
- アボート
  競合しアボートする場合，ソフトウェアにより，ログにある値を用いて，新しい方から順に値をトランザクション開始時の状態へ戻す．その後，すべてのリード/ライト・ビットをクリアし，ログの初期化を行う．最後に，レジスタ状態を戻し，トランザクションを最初から再実行する．

3.2 部分ロールバックのサポート

Moravan らの手法では，トランザクションは投機的に実行され，トランザクションの投機状態，ロールバックのためのトランザクション開始時のレジスタやメモリの状態をトランザクションの深さによって区別して管理する．これにより，どの深さのトランザクションが競合し，再実行されれば良いのか判断でき，そのトランザクション開始時のプロセッサやメモリの状態を回復することができる．

トランザクションの投機状態については，深さ別のリード/ライト・ビットを用いて管理する．ロールバックのためのトランザクション開始時のレジスタやメモリの状態については，LogTM のログを拡張して保存する．

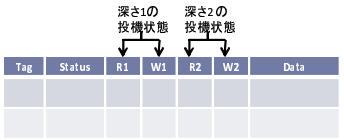

図 5 深さ毎の投機状態を管理するキャッシュ
Fig. 5 Cache managing speculative states for each depth

深さ別の投機状態

投機状態の管理には，**図 5** のようなキャッシュを用いる．同図のキャッシュは，通常のキャッシュに各深さ別にリード/ライト・ビットを設けたものである．ここでは，最も外側のトランザクションの深さを 1 とし，深さ i のトランザクション実行中に開始されたトランザクションの深さを $i+1$ とする．例えば，R_1 がセットされているラインは，深さ 1 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．この深さ別のリード/ライト・ビットとキャッシュ・コヒーレンス・プロトコルにより，どの深さのトランザクションが競合したのかがわかる．

ログの拡張

ここでのログは，LogTM と同様に，トランザクションにライト・アクセスされた仮想アドレスとその変更される前の値を保持する．しかし，LogTM のログと異なり，ログ・フレームを階層化したものである．ログ・フレームとは，対応するトランザクション開始時のレジスタ状態とすぐ外側のトランザクションに対応したログ・フレームのヘッダを指すポインタを格納する固定長のヘッダと，対応するトランザクション開始時の状態に戻すための可変長のレコードから成る．レコードは，対応するトランザクションによって書き換えられたキャッシュ・ラインの仮想アドレスと変更される前の値から成る．このログ・フレームは，新たな深さのトランザクションが開始される毎に作られ，深さによって区別されている．

3.3 ネスティッド・トランザクションの実行

トランザクションは投機実行され，キャッシュ・コヒーレンス・プロトコルによって競合を検出する．競合があった場合，競合したメモリ・アクセスを行った深さのトランザクションの開始点に部分ロールバックし，再実行する．以下では，深さ i のトランザクションの実行，終了，ロールバックについてそれぞれ述べる．

深さ i のトランザクションの実行

開始時には，新規ログ・フレームのヘッダとして現在のレジスタ状態を保存する．ライト・アクセス時は，該当するキャッシュ・ラインの仮想アドレスと書き換えられる前の古い値をログに退避させる．次に，書き換える新しい値を該当するキャッシュ・ラインへ書き込み，そのラインの深さ i のライト・ビットをセットする．その命令以前に深さ i のライト・ビットがセットされている場合は，すでに値は退避されているので，値を書き換えるだけであとは何もしない．リード・アクセス時は，値が書き換わることはないので，ログに仮想アドレスや値を退避させず，値をロードして，該当するキャッシュ・ラインに深さ i のリード・ビットをセットする．すでに立っている場合はセットされたままにしておく．

深さ i のトランザクションの終了

競合がないまま深さ i のトランザクション内の処理がすべて終了したら，キャッシュの深さ i のリード/ライト・ビットを深さ $i-1$ のリード/ライト・ビットに OR を取ってマージして，深さ i のリード/ライト・ビットをクリアする．最も外側の深さ 1 のトランザクションが終了した時は，キャッシュのリード/ライト・ビットを他の深さも含めすべてクリアしてコミットし，ログを初期化する．

深さ i のトランザクションのロールバック

ロールバック時には，対応するログ・フレームを用いて，そのトランザクションが書き換えた値を書き換えられる前の値に戻す．その後，レジスタ状態を深さ i のトランザクション開始時のレジスタ状態へ戻し，キャッシュの深さ i 以上のリード/ライト・ビットをクリアする．

3.4 問 題 点

Moravan らの手法の問題点として，深さの制約と，必ずしも部分ロールバックが最適でない場合がある．以下では，それらについてそれぞれ述べる．

- 深さの制約
  この手法では，部分ロールバックできる深さに制約がある．これは，投機状態を管理できる深さがキャッシュによって予め決まっているからである．よって，キャッシュで管理できない深さのトランザクションは平坦化される．図 5 のキャッシュでは，深さ 2 までしか対応できず，深さ 3 以上のトランザクションは深さ 2 のトランザクションに平坦化される．
- 最適でない部分ロールバック
  この手法における部分ロールバックは必ずしも最適であるとは限らない．その例を**図 6** に示す．同図は，深さ 3 のトランザクションが終了した後に，深さ 3 のトランザクションのライト・アクセスと他のスレッドのリード・アクセスが競合した場合である．深さ

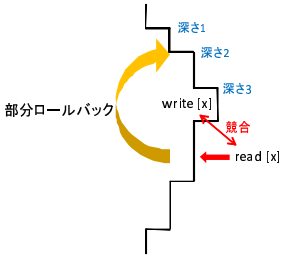

図 6 最適でない部分ロールバック
Fig. 6 Not optimal rollback

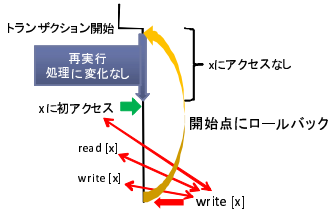

図 7 競合する最も古いメモリ・アクセスが初アクセスである場合
Fig. 7 The oldest conflict is the first access

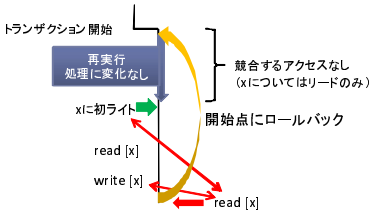

図 8 競合する最も古いメモリ・アクセスが初アクセスでない場合
Fig. 8 The oldest conflict is not the first access

3 のリード/ライト・ビットは，すでに深さ 2 のリード/ライト・ビットとマージされいる．従って，深さ 2 のトランザクションで行ったメモリ・アクセスが競合したと判断される．このため，ロールバック・ポイントは，深さ 3 ではなく，深さ 2 のトランザクションの開始点となる．このとき，再実行時にはロールバック・ポイントから深さ 3 の開始点までの処理を繰り返さなければならない．

4. 最適なロールバック・ポイントの選択

2.5 節や 3.4 節で述べたように，平坦化や Moravan らの手法[7] では，再実行時のペナルティが大きいことがある．これに対し，提案手法では，最適なロールバック・ポイントを選択することで，いかなるネスティッド・トランザクションについても再実行される処理を最小限にすることができる．以下では，まず，最適なロールバック・ポイントの背景として，どのような部分ロールバックが可能なのかを説明する．次に，最適なロールバック・ポイントについて述べ，最適なロールバック・ポイントを選択する手法とその実装について述べる．

4.1 部分ロールバック

ロールバック・ポイントは，トランザクション中で実行された競合するメモリ・アクセスの中で最も古いメモリ・アクセスより前でなければならない．なぜなら，トランザクションのアトミシティを保つために，競合したメモリ・アクセスすべてをやり直す必要があるからである．

以下では，競合する最も古いメモリ・アクセスより前に部分ロールバックした場合でもトランザクションのアトミシティが保たれることを示す．競合する最も古いメモリ・アクセスが，そのトランザクション中で初アクセスである場合と，初アクセスではない場合，それぞれについて証明する．

- 競合する最も古いメモリ・アクセスが初アクセスである場合
  競合する最も古いメモリ・アクセスが初アクセスである場合，そのアドレスに対してはそれ以前にアクセスがない．その例を**図 7** に示す．同図では，x へのアクセスが競合する最も古いメモリ・アクセスであり，この競合によってトランザクションの開始点から再実行している．このとき，x に初めてアクセスするまでの処理は，初めてトランザクションを実行する時と全く同様である．これは，x の値が変更されていても，アクセス直前まではその値を扱うことがないためである．よって，初アクセスより前へロールバックしても，そのロールバック・ポイントの状態は，開始点へロールバックして再実行した場合と同じなので，トランザクションのアトミシティは保たれる．
- 競合する最も古いメモリ・アクセスが初アクセスでない場合
  競合する最も古いメモリ・アクセスが初アクセスでない場合とは，**図 8** のように，競合する最も古いメモリ・アクセス以前のそのアドレスに対するメモリ・アクセスはリードのみで，他のスレッドによるそのアドレスに対するメモリ・アクセスはリードである場合しかない．このため，開始点にロールバックして再実行したとしても，そのアドレスの値は変更されないから，競合する最も古いメモリ・アクセス直前までの処理は初めてトランザクションを実行した場合と全く同様である．よって，そのロールバック・ポイントの状態は，開始点へロールバックして再実行した場合と同じなので，競合した最も古いメモリ・アクセスより前にロールバックしてもトランザクションのアトミシティ

は保たれる．

以上より，一般のトランザクションにおいて，競合する最も古いメモリ・アクセスより前に部分ロールバックできることが示された．

4.2 最適なロールバック・ポイント

最適なロールバック・ポイントは，競合する最も古いメモリ・アクセス直前の開始点である．4.1 節での証明からすれば，最適なロールバック・ポイントは，競合する最も古いメモリ・アクセス直前である．しかし，競合する最も古いメモリ・アクセス直前にロールバックするには，競合する最も古いメモリ・アクセスになり得るすべてのメモリ・アクセス直前でプロセッサ状態を逐一保存する必要があり，これは現実的でない．従って，プロセッサ状態の保存頻度を考慮し，競合する最も古いメモリ・アクセス直前の開始点が最適なロールバック・ポイントとなる．

4.3 最適なロールバック・ポイントの選択

4.3.1 方　　針

提案手法では，競合する最も古いメモリ・アクセスとその直前の開始点を検索することで，トランザクション開始点の中から最適なロールバック・ポイントを選択する．まず，競合する最も古いメモリ・アクセスになり得るすべてのメモリ・アクセスの履歴と，開始点の履歴を実行順に取りながらトランザクション内の処理を実行する．ここでの履歴とは，いつ各メモリ・アクセスや開始点があったかの記録のことである．競合検出時のロールバックは，その履歴を検索することで行う．**図 9** に履歴の様子を示す．同図では，各開始点とメモリ・アクセスの履歴を上から下に新しいものとなるように記録している．例えば，履歴の上から 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，ここでは上から履歴を調べれば，履歴の上から 4 行目の x へのライトが競合する最も古いメモリ・アクセスであるとわかる．そして，そのメモリ・アクセスから新しい順に検索し，深さ 2 の開始点が競合する最も古いメモリ・アクセスの直前の開始点であることを識別する．その開始点を最適なロールバック・ポイントとして選択する．

4.3.2 履歴を取るメモリ・アクセス

最適なロールバック・ポイントを選択するためのアクセス履歴と，ロールバックのための履歴を実行順に取る必要がある．この取るべき履歴は，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

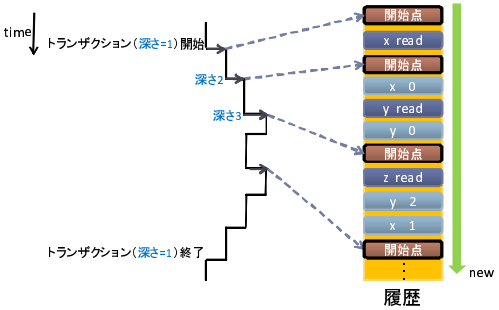

図 9 履歴の取り方
Fig. 9 Logging

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，競合する最も古いメモリ・アクセスになり得る．初アクセスでない初リードについては，履歴をとる必要はない．なぜなら，そのリードより前にライトがあるならば，必ずそのライトがより古い競合するメモリ・アクセスとなるからである．

初ライト　競合する最も古いメモリ・アクセスになり得る初ライトの例を**図 10** に示す．同図では，x への初ライト以前に x へのリードがある．しかし，x への他スレッドによるリードは競合しない．このとき，競合する最も古いメモリ・アクセスは x への初ライトである．このため，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．

- ロールバックのために履歴を取るべきメモリ・アクセス

開始点後の初ライト　開始点はロールバック・ポイントとなり得るので，その開始点から次の開始点直前までに値を書き換えられたアドレスに対しては，開始点時の値を履歴として取っておかなければならない．

これらの履歴の一部は，最適なロールバック・ポイントを選択するための履歴であると同時に，ロールバックのための履歴でもある．競合する最も古いメモリ・アクセスになり得るすべてのメモリ・アクセス中のライトは，そのアドレスへの初ライトであるので，トランザクションをロールバックさせるための履歴を兼ねる．従って，履歴の量を減らすために，初ライトと開始点後の初ライトは，合わせて一つの履歴を取る．

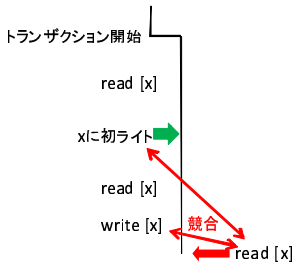

図 10 初ライト
Fig. 10 The first write

4.4 実　　装

ここでは，Moore らの LogTM[6] を拡張した提案手法の実装について説明する．

履歴の取り方

まず，Moravan らの手法[7] と同様に，各トランザクション開始点でロールバックに必要となるその時のレジスタ状態をヘッダとしてログに保存する．3.1 節で述べた LogTM のログを拡張して，最適なロールバック・ポイントを選択するための履歴とロールバックのための履歴を取る．**図 11** に，これを実現するキャッシュの構成を示す．同図のキャッシュは，直前の開始点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．ここでは，直前の開始点前のリード/ライト・ビットを R_p，W_p とし，直前の開始点後のリード/ライト・ビットを R_c，W_c とする．W_c がセットされていないキャッシュ・ラインは，直前の開始点後にライトされていない．つまり，このラインへのライトは開始点後の初ライトとして認識される．このとき，履歴を取るべきメモリ・アクセスを以下に示す．

- 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

これらのメモリ・アクセスについての履歴をログに取る．ロールバックのために，初ライト，開始点後の初ライトは，LogTM と同様に仮想アドレスと書き換える前の古い値をログに取る．また，初アクセスであるリードの場合，値を書き換えていないので古い値をログに取る必要はないが，リードであることは取らなければならない．

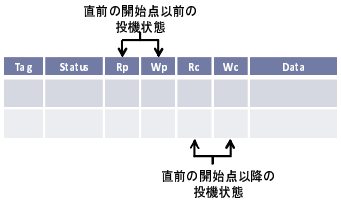

図 11 最適なロールバックのためのキャッシュ
Fig. 11 Cache for the optimal rollback

処理の流れ

トランザクショナル・メモリの多くの手法と同様に，トランザクションは投機実行され，キャッシュ・コヒーレンス・プロトコルにより各リード/ライト・ビットを用いて競合を検出する．

まず，最も外側となるトランザクションが開始されると，ログがない場合は割り当て，その時のレジスタ状態をログに保存する．各メモリ・アクセスでは，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って，リード・アクセスならば R_c を，ライト・アクセスならば W_c をセットする．このとき，各履歴は古い順にログに取られる．

トランザクション実行中にトランザクション開始点があった場合，その時のレジスタ状態をログに保存する．R_c と W_c は，R_p と W_p とそれぞれ OR を取ってマージ後，クリアされる．終了点では，最も外側のトランザクションの終了点でなければ，何もしない．最も外側の終了点では，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．

競合してロールバックすることになれば，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メモリ・アクセスを探し，最適なロールバック・ポイントを選択する．そのロールバック・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 を戻す．最後に，レジスタ状態を回復し，ロールバック・ポイントから再実行する．

図 12 は，簡単なネスティッド・トランザクションのコード例である．**図 13** は，図 12 の

```
x = 0; y = 0; z = 0;
TXbegin();
  x++;
  y = 2;
  TXbegin();          ← 深さ2
    y = z + 1;
  TXend();
  x++;
TXend();
```

図 12 ネスティッド・トランザクションの例
Fig. 12 Example of nested transaction

コードを深さ 1 のトランザクション終了直前まで実行した時のキャッシュとログの状態を変数名で表している．例えば，y については，直前の開始点である深さ 2 の開始点の前後でそれぞれライト・アクセスされている．そのため，y の W_p，W_c がともにセットされている．また，ログには，上で述べたようなメモリ・アクセス毎に，ライト・アクセスでは変数名と古い値が，リード・アクセスでは変数名とリードしたことが，履歴としてレジスタ状態とともに上から古い順に保持されている．例えば，x は深さ 1 のトランザクション開始直後にリード・アクセスされている．このため，ログには，上から 2 行目にその履歴が保持されている．この時点で他のスレッドによる z へのライト・アクセスがあり，競合してロールバックすることになったとする．まず，競合する最も古いメモリ・アクセスを検索する．ログを古い方，ここでは上から下へ履歴が検索され，競合する最も古いメモリ・アクセスとして z へのリードが検索される．そして，z へのリードから今度は上へログを検索し，レジスタ状態が保持されているヘッダを探し，深さ 2 のトランザクションの開始点を最適なロールバック・ポイントとする．**図 14** に，この例における最適なロールバックの様子を示す．Moravan らの手法[7] では，深さ 2 のトランザクションはすでに終了しているので，深さ 1 のトランザクション開始点までロールバックすることになる．一方，提案手法では，深さ 2 のトランザクションの開始点が最適なロールバック・ポイントとして選択され，トランザクションはそこから再実行される．

5. おわりに

本稿では，トランザクションがロールバックする場合に，破棄される処理が最小限になる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・メモリを提案した．トランザクションが競合しロールバックする際に，必ずしもトランザクション開始

図 13 最適なロールバック・ポイントを選択するためのキャッシュとログ
Fig. 13 Cache and log for the optimal rollback

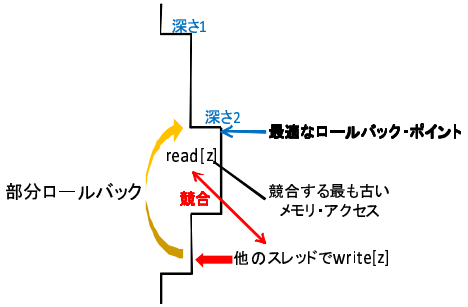

図 14 最適なロールバック・ポイントの選択
Fig. 14 Selecting the optimal rollback point

点から再実行しなくとも良い．トランザクション中の競合する最も古いメモリ・アクセスになり得るメモリ・アクセスに関して必要な履歴を取り，競合時にそれを検索することで，ロールバック・ポイントの候補である各トランザクションの開始点の中から最適なロールバック・ポイントを選択することができる．

今後の課題として，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・メモリの評価をする必要がある．また，大きなオーバヘッドを伴う各開始点でのレジスタ状態の保存についても考えて行きたい．

Vol.2009-ARC-184 No.5
2009/8/4

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).