修士論文

プログラミング上の制約を緩和するトランザクショナル・メモリの研究

Research on Transactional Memory Relaxing Restriction of Programming

平成23年2月9日提出

指導教員　坂井修一 教授

東京大学大学院 情報理工学系研究科

電子情報学専攻

48-096403 伊藤 悠二

概要

並列プログラミングにおいてロックを用いない同期機構として，トランザクショナル・メモリが提案されている．トランザクションは不可分に実行されているかのように投機実行される．もし他スレッドのアクセスと競合した場合，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．

トランザクショナル・メモリを用いた並列プログラムでは，トランザクション中でトランザクションを呼び出したり，トランザクションを実行するスレッドがコア数以上に生成されたりすることがあり得る．プログラマにとって，このようなトランザクションを書かないように制限を課すことは非常に大きな負担となる．しかし，既存手法では，トランザクション中でトランザクションを呼び出して長大になるトランザクションや，スレッド・スイッチして中断するトランザクションが適切にロールバックされないことがある．

本稿では，過去に競合した命令直前でチェックポイントを取り，それらを途中で無効化することなく最適なチェックポイントを選択する手法を提案する．さらに，中断トランザクションについても，コア別に競合検出を行い，最適なチェックポイントの選択を行う手法を併せて提案する．提案手法により，プログラマが書くことのできるトランザクションの制限を緩和できる．

本手法の部分ロールバックの評価では，部分ロールバックしない場合の最大 6.9 倍の性能向上を達成できた．

目 次

図 目 次

表 目 次

第1章　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, 2, 3, 4, 5, 6, 7, 8] が有望視されている．トランザクショナル・メモリでは，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．従って一般には，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．

トランザクションの投機　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．すなわち，トランザクションを投機的に開始し，複数のトランザクションが同一アドレスにアクセスしてした時，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**ロールバック**処理を行うのである．トランザクションの投機実行については，2 章で詳しく述べる．

競合の検出は，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．これにはキャッシュ・タグによる手法 [1, 2, 3, 7, 8] と，**シグネチャ**(signature) による手法 [4, 5, 6] が提案されている．競合検出手法は，2.2 節で詳しく述べる．

トランザクショナル・メモリの問題点　並列プログラミングでトランザクショナル・メモリを用いると，プログラマがすべてのトランザクションの挙動を把握するのは難しい．トランザクション中でトランザクションを含む関数を呼び出したり，トランザクションを実行するスレッドがコア数以上に生成されたりすることがあり得る．トランザクションを呼び出すようなトランザクションは長大となる．このような長いトランザクションのロールバック時には，再実行される命令数が投機失敗の**ペナルティ**が大きく現れる．トランザクションを実行するスレッドが多数となると，実行中にスレッド・スイッチが起き，トランザクションが中断することがある．既存のトランザクショナル・メモリでは，このようなトランザクションのロールバックが適切でない．つまり，プログラマにこのようなトランザクションを指定しないように制限が課せられ，非常に大きな負担となってしまう．

本稿では，実行中，中断中に関わらずトランザクションの最適なロールバックを行い，この制限を緩和する手法を提案する．提案手法では，ロールバック時のペナルティの削減と，さらにそれを中断トランザクションのロールバックへも適用する．以下でこれらを順に述べる．

部分ロールバック　ロールバック時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．

部分ロールバックのためには，トランザクション中に予めチェックポイントを取っておく必要がある．競合したトランザクションは，必ずしも開始点まで戻る必要はない．以前，競合を起こした命令より前に戻れば十分であることを証明した [7]．したがって，トランザクション中の適当な位置にチェックポイントを設定しておき，競合時にはその中から適切なチェックポイントを選択して部分ロールバックすればよい．したがって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　（トランザクションの開始点以外の）チェックポイントを予めどこに設定しておくか．

チェックポイントの選択　最適なチェックポイントをどのように特定するか．

部分ロールバックに対応した手法として，Yen らが提案する LogTM-SE[5] や Waliullah らの学習によるチェックポインティング手法 [9] が提案されている．これらの手法は，チェックポイントの位置と選択の 2 つの観点からは，以下のように分類できる．

チェックポイントの位置については，以下のような方法が考えられる：

I. ネスティッド・トランザクションの開始点．(LogTM-SE など)

II. 過去に競合が発生したデータに対するアクセスの直前．(Waliullah)

III. 過去の競合を起こした命令の直前．(提案手法)

一方，チェックポイントの選択については，以下のデータ構造を用いる方法が考えられる：

a. キャッシュ・タグ．(Walliullah など)

b. シグネチャ．(LogTM-SE など)

すなわち，LogTM-SE は (I+b)，学習によるチェックポインティング手法は (II+a) である．これらの手法については，3 章で述べる．

本稿の提案は，(III+b) である．学習により過去に競合を起こした命令の直前に設定し，シグネチャを用いてチェックポイントを選択する．Waliullah らも学習によチェックポインティングの方法を提案しているが，競合を起こしたデータに対するアクセスを検出する点が異なる．また，チェックポイントの選択では，シグネチャを用いることは LogTM-SE と同様だが，トランザクション実行中にチェックポイントを無効化しない点が異なる．提案手法の部分ロールバックについては，4 章で詳しく述べる．

中断トランザクションのロールバック　中断トランザクションについても実行中のトランザクションと同様に，競合検出を行い，競合したトランザクションのいずれかをロールバックする．また，中断トランザクションも中断前に上で述べたチェックポイントを設定している．このため，最適なチェックポイントを選択することで中断トランザクションについてもロールバック時のペナルティを削減することができる．中断トランザクションのロールバックのためには，以下の 3 つの処理を順に行う：

1. 競合検出.

2. 競合トランザクションの検索.

3. チェックポイントの選択.

上 3 つの処理について既存手法の LogTM-SE，FlexTM[6] と提案手法を比較した表を**表 1.1** に示す．当然，トランザクションの投機実行では 1. 競合検出 は行う．2. 競合トランザクションの検索 については，LogTM-SE では行わ

表 1.1: 中断トランザクションのロールバック

	LogTM-SE	FlexTM	提案手法
1. 競合検出	○	○	○
2. 競合トランザクションの検索	×	△	○
3. チェックポイントの選択	×	×	○

ずに競合を起こした実行中のトランザクションをロールバックする．FlexTM では，すべての中断トランザクションを検索するためそのオーバヘッドが大きい．3. チェックポイントの選択 については，FlexTM は競合した中断トランザクションの開始点を選択し，最初から再実行する．これらの手法については，5 章で述べる．

本稿の提案では，コア別に競合検出することで 2. 競合トランザクションの検索 のオーバヘッドを削減し，3. チェックポイントの選択 までの処理をすべて行う．提案手法により，検索を高速化して中断トランザクションについても最適な部分ロールバックを行うことができる．提案手法の中断トランザクションのロールバックについては，6 章で詳しく述べる．

提案手法の部分ロールバックをマルチプロセッサ・シミュレータ GEMS に実装，評価し，部分ロールバックしない既存手法に対して最大 6.9 倍の性能向上を確認した．評価に関しては，7 章で述べる．

第2章　トランザクショナル・メモリ

本章では，基本的なトランザクショナル・メモリの手法について述べる．

2.1　トランザクションの投機

1 章で述べたように，トランザクションは投機的に実行される．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．しかし，不可分に実行された場合の結果と同じ結果を残すために，競合を検出し，ロールバックを行う．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．競合検出については 2.2 節で述べる．

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トを取り，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．トランザクションの投機的な値の扱いやチェックポイントの状態の保存については，2.3 節で述べる．

図 2.1 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_1 の実行の途中経過を X_2 が観測することになり，X_1 が不可分に実行された場合と同じ結果にならない．従って，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ロールバックする．ここでは，X_2 をロールバックし，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．

2.2　競合検出

競合検出にキャッシュ・タグを用いる手法とシグネチャを用いる手法についてそれぞれ説明する．

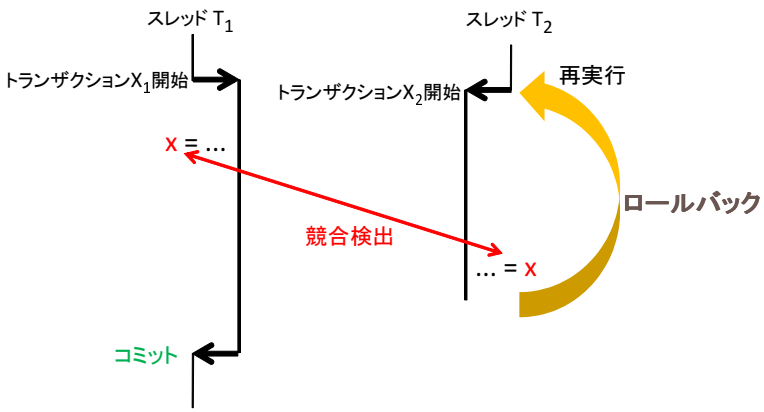

図 2.1: トランザクションの投機実行

2.2.1 キャッシュ・タグによる検出

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リード/ライトが行われたらセットされるキャッシュ・ラインごとに設けられた 2 ビットである．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時にクリアされる．

トランザクションによって書き換えられたキャッシュ・ラインがリプレースされた場合，LogTM[2] 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．トランザクションを実行しているコアがリプレース後もそのラインを保持しているものとしてディレクトリが管理し，そのラインへの要求によって競合を検出する．

Waliullah らの手法 [9] 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，そのバッファ中についてもリード/ライト・ビットを調べて競合を検出する．

2.2.2 シグネチャによる検出

キャッシュ・タグを用いず，アクセスしたアドレスを圧縮した**シグネチャ** (signature)[4, 5, 6] を用いて競合を検出する．シグネチャは，コアごとにリー

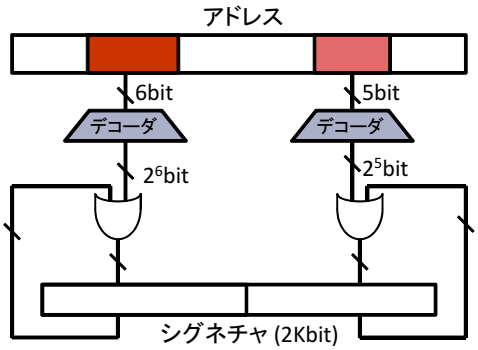

図 2.2: シグネチャの例

ド/ライト・アドレス用にリード/ライト・シグネチャそれぞれについて保持され，ハードウェアによって以下のように更新されるビット列である．トランザクションがアクセスしたアドレスのビット列の一部をデコードし，今までのシグネチャと OR を取ることで更新される．あるアドレスについて同様の部分をデコードしたものとシグネチャとの AND を取ったものが一致すれば，シグネチャを更新した要素の一つであると識別され,「ヒット」となる．キャッシュ・コヒーレンス・プロトコルによる共有要求されたアドレスがライト・シグネチャとヒットした場合や，無効化要求されたアドレスがリード/ライト・シグネチャいずれかにヒットした場合，そのアドレスについての競合であると検出する．コミット時にシグネチャのすべてのビットを 0 としてクリアする．

シグネチャを用いた競合検出では，複数のアドレスを固定長に圧縮しているため，本来競合ではないものを競合として誤検出することがある．アドレスのビット列の複数の部分を用いたり，ハッシュ関数を用いることで誤検出の少ないシグネチャを作ることができる．簡単な例として，**図 2.2** では，アドレスの一部それぞれ 6 ビット，5 ビットを用いて，2K ビットのシグネチャを作成している．

2.3 バージョン管理

本節では，トランザクション中の投機的な状態のバージョン管理について述べる．ここでは，ログ [2, 3, 5] を用いたバージョン管理について説明する．

トランザクション中のライトについては投機的な値をキャッシュの該当するキャッシュ・ラインに書き込む．その時，書き換えられる直前の値とそのアドレスを保存しておく．この保存先を**ログ**という．ログはアドレス空間上の領域

である．

ロールバックは，ログに保存された値をともに保存されていたアドレスへ書き戻すことで行われる．メモリには書き換えられる前の古い値が残っているので，ロールバック時にこれらのキャッシュ・ラインを無効化することで結果を破棄できる．しかし，ログを用いない場合，書き換えられたキャッシュ・ラインがリプレースされると，メモリ上の古い値が投機的な値に上書きされてしまう．それ以降では書き換える前の値が残っておらず，トランザクションをロールバックできなくなる．ログを用いれば，リプレース後も古い値を保持することができるため，トランザクションは引き続き実行を継続できる．

第3章　部分ロールバック

本章では，部分ロールバックにおけるログによるマルチバージョン管理，チェックポイントの位置の設定，チェックポイントの選択について述べる．

3.1　ログによるマルチバージョン管理

部分ロールバックでは，競合時の状態から途中のチェックポイントの状態に戻す．部分ロールバックを行う場合，チェックポイントは最初の開始点だけでなく，複数個取られる．このためにトランザクション中で書き換えられる値のマルチバージョン管理をログで行う．

各チェックポイントではその時点のレジスタ状態を保存する．ライト時には2.3節で述べたように書き換えられる前の値とそのアドレスを保存する．また，シグネチャを用いて競合検出を行う場合には，ロールバック後もそのチェックポイント以降の競合検出を続けて行えるように，各チェックポイントでシグネチャもログに保存しておく．

ロールバック時には，ログにある値やレジスタ状態，シグネチャを用いてチェックポイントの状態を回復する．ログではトランザクションによって書き換えられた値が実行順に保持されている．従って，ログの新しい方から順に値を書き戻すことで，チェックポイントの状態に戻すことができる．

3.2　チェックポイントの位置

チェックポイントの位置については，ネスティッド・トランザクションの開始点直前や競合したデータ・アドレスへのアクセス直前に設定することが提案されている．

3.2.1 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トランザクションが開始される**ネスト** (nest) 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** (nested transaction) と呼ぶ．Moravan らが提案する LogTM[3] や Yen らが提案する LogTM-SE[5] では，ネストされた内側のトランザクションの開始点ごとにチェックポイントを取る．内側のトランザクションで競合が発生した時，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．

3.2.2 過去の競合データ・アドレスへのアクセス直前

Waliullah らの学習によるチェックポインティング手法 [9] では，過去の競合データ・アドレスへのアクセス直前にチェックポイントを取ることを提案している．一度競合したアドレスは，ロールバック後でも再び競合する予測する．競合検出時にそのアドレスを保存し，ロールバック後にそのアドレスへのアクセスがあれば，その直前にチェックポイントを取る．予測したアドレスで競合が起きれば，その直前のチェックポイントに戻ることができる．

3.3 既存のチェックポイントの位置の問題点

ネスティッド・トランザクション開始点でのチェックポイント　内側のトランザクション開始点でチェックポイントを取ると，トランザクションの構造によってチェックポイントの位置が決まる．そのため，ネストしていないトランザクションでは，チェックポイントを取ることができない．また，競合とは関係なくチェックポイントが設定されたことで，部分ロールバックしてもペナルティの削減に効果がないことがある．

競合データ・アドレスへのアクセス直前　過去の競合データ・アドレスでチェックポイントを取る場合，同一の競合命令が複数のアドレスにアクセスするとその数だけチェックポイントを取ることになる．例えば，良く競合を起こす配列へ順にアクセスするような処理があると，チェックポイントが短い間隔で多く取られる．また，命令がアクセスするアドレスが判明するまでチェックポイントを取るか不明である．チェックポイントを取るにはレジスタ状態をログへ保存しなければならないため，アドレスが判明してからその命令の直前のレジスタ状態に戻さねばならず，オーバヘッドが大きい．従って，部分ロールバック

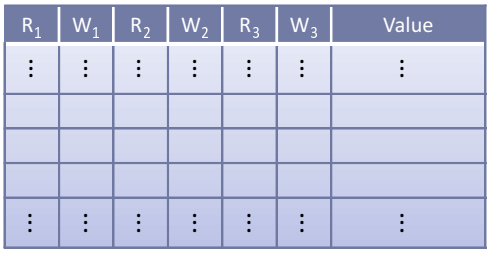

図 3.1: チェックポイントを選択するキャッシュ

によるペナルティの削減よりも多くのチェックポイントを取るオーバヘッドの方が大きくなってしまう．

3.4 チェックポイントの選択

競合の解決には，競合より前のチェックポイントに戻ればよい．本節では，その競合より前のチェックポイントの選択にキャッシュ・タグ，シグネチャを用いる手法についてそれぞれ述べる．

3.4.1 キャッシュ・タグによる選択

部分ロールバックに対応した LogTM[3] や学習によるチェックポインティング手法 [9] では，リード/ライト・ビットをさらに拡張して，どこで競合アクセスがあったかを検出する．

競合検出はリード/ライト・ビットを行い，チェックポイントの選択はリード/ライト・ビットを拡張した**図 3.1** のようなキャッシュを用いて行われる．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．図 3.1 における複数のリード/ライト・ビットは各セクションに対応する．例えば，3 つ目のチェックポイントを取ったら，そのセクションでリード/ライトが行われると，R_3/W_3 がセットされる．競合検出時に各リード/ライト・ビットを調べ，競合を起こしたアクセスが行われたセクションを特定する．そして，そのセクションの頭のチェックポイントに部分ロールバックを行う．例えば，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部分ロールバックする．

3.4.2 シグネチャによる選択

LogTM-SE[5] では，ロールバック後の競合検出のためだけでなく，チェックポイントの選択にも各チェックポイントで保存したシグネチャを用いる．チェックポイントで保存されるシグネチャは，各チェックポイントでその時点でのシグネチャをログに保存する．保存された各シグネチャは開始点からチェックポイントまでの累積のアクセス履歴となる．そのため，最も古い競合アクセス直前のチェックポイントを選択するためには，最も新しい競合のないシグネチャを検索する．新しい方から検索することはロールバックと同時に行えるため，LogTM-SE ではそのようにチェックポイントを選択している．

現在のシグネチャによって競合を検出したら，ログを用いてロールバックを行う．ログはアクセス順に取られるため，新しい方のログ・エントリから値とシグネチャを戻す．直前のチェックポイントまで回復させたら，一旦ロールバックを停止し，回復したシグネチャで再び競合検出を行う．競合が検出されなくなるまで直前のチェックポイントへのロールバックを繰り返すことで，競合の前のチェックポイントまでロールバックすることができる．

ここで，LogTM-SE では，終了した内側のトランザクションの開始点へ部分ロールバックを許していない．そのため，内側のトランザクションが終了すると，その開始点で保存したシグネチャとレジスタ状態を無効化することでチェックポイントを無効化する．内側のトランザクションでのライトは，外側のトランザクションで行ったものとするため，書き換えられる前の値については無効化しない．以降，ロールバックではそのシグネチャとレジスタ状態を無視してロールバックやチェックポイントの選択を行う．

図 3.2 では，LogTM-SE のログの様子を表している．各チェックポイントでその時のシグネチャがログに追加され，書き換えられる前の値を保持している．他スレッドによる変数 y へのアクセスで競合を検出したら，競合が検出されなくなるまで直前のチェックポイントへの部分ロールバックを繰り返す．同図では，2 つ目のチェックポイントにロールバック後のシグネチャでは y についての競合が検出されないため，このチェックポイントから実行を再開する．

3.5 既存のチェックポイントの選択の問題点

キャッシュ・タグによる選択　キャッシュ・タグを用いると，リード/ライト・ビットは予め設けられたものであるため，その数にチェックポイント数が強く制限される．例えば，図 3.1 のキャッシュでは，3 セクションまでしか管理できないため，4 つ以上のチェックポイントを取ることができない．また，投機

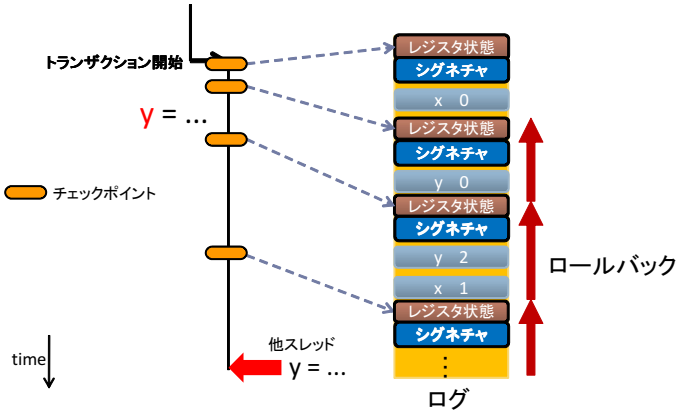

図 3.2: シグネチャによるチェックポイントの選択

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選択が不可能となる．

LogTM-SE のシグネチャによる選択　3.4 で述べたように，LogTM-SE では内側のトランザクションが終了すると，その開始点で保存したシグネチャを無効化する．そのシグネチャを残しておいてチェックポイントを選択することができるにも関わらず，終了した内側のトランザクション開始点に部分ロールバックすることを許さない．すでに終了した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い，外側のトランザクションが開始点で保存したシグネチャがあるために，内側のトランザクションで保存したシグネチャは必要ないものとされる．

第4章　提案手法： 最適なチェックポイントの選択

4.1　アプローチ

提案手法では，チェックポイントの位置を 過去の競合命令直前 で設定し，実行中にそれらのチェックポイントを無効化せず，競合時にシグネチャを用いて選択を行う．提案手法によって，あらゆるトランザクションにおいて既存手法より適切な部分ロールバックを行うことができる．

本手法のチェックポイントの位置の設定は，過去の競合命令直前にチェックポイントを設定することで，過去の競合データ・アドレスへのアクセス直前よりも競合に適したチェックポイントを設定する．

チェックポイントの選択について，1 章で述べたように，トランザクションは競合が起きたアクセスより以前のチェックポイントならばどこにでもロールバックすることができることを証明した [7]．そのため，LogTM-SE のように終了した内側のトランザクションの開始点に部分ロールバックできないという制限を設ける必要はない．そこで，本手法ではシグネチャを用いてチェックポイントを選択するが，チェックポイントを途中で無効化することなく，ロールバック先の候補として保持し続ける．

図 4.1 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) では，トランザクション内の命令すべてを再実行する．当然，そのペナルティは大きい．LogTM-SE では，内側の開始点にしか部分ロールバックできない上に，終了した内側の開始点のチェックポイントを無効化してしまうため，再実行される命令が多くなってしまう．学習によるチェックポインティング手法 [9] では，リード/ライト・ビットによりチェックポイント数が 4 つまでに制限されているとすると，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．提案手法では，取ったチェックポイントを無効化することなく，最適なチェックポイントとして x へアクセス

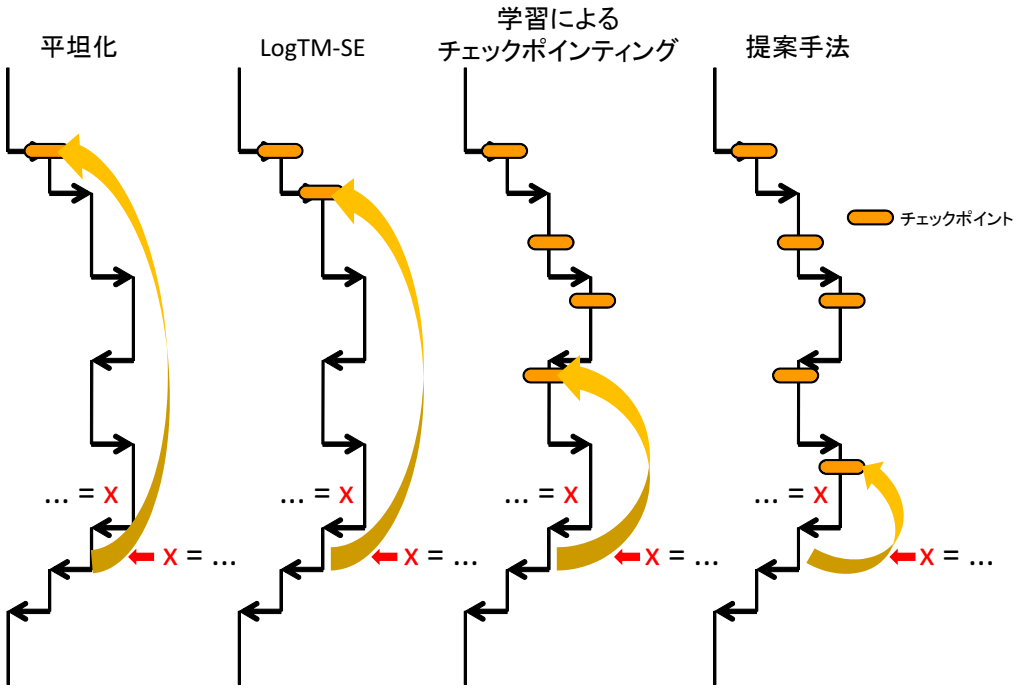

図 4.1: 最適なチェックポイントの選択

する命令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できる．

4.2 過去の競合命令直前でのチェックポイント

本手法では，過去の競合命令直前でチェックポイントを取る．過去の競合命令は，ロールバック後に再び競合を起こすと予測する．そのために競合検出時に競合した命令の PC を保存する．ロールバック後，保存してあった PC と同じ PC の命令を実行する直前にチェックポイントを取る．再びその命令で競合した場合，その命令から再実行できる．

この手法では，同一命令が異なるデータ・アドレスにアクセスするようなことがあっても，その命令については 1 つのみチェックポイントを取る．このため，チェックポイント同士の間隔が短いチェックポイントを多く取ることを避けることができ，多くのチェックポイントを設定するオーバヘッドを小さくできる．

4.3　シグネチャによるチェックポイントの選択

チェックポイントの選択については，3.4 節で述べたように，チェックポイント時にログに保存されたシグネチャを用いる．競合時にはログの新しいエントリから書き換えられる前の値を戻してロールバックを行う．シグネチャのエントリがあれば，そのシグネチャに競合アドレスが含まれるか調べる．シグネチャに競合がある場合，それを保存したチェックポイント以前に競合があるため，引き続きロールバックを行う．シグネチャに競合がなければ，それを保存したチェックポイントまでに競合がないことがわかるため，そのチェックポイントから再実行する．

本手法では，LogTM-SE の場合と異なり，トランザクション実行中にチェックポイントを無効化しない．チェックポイントは上で述べたように過去の競合命令直前で設定するので，ネスティッド・トランザクションの構造とは関係なく，内側のトランザクションが終了してもシグネチャを無効化する処理は行わない．

第5章　中断トランザクションのロールバック

本章では，既存手法におけるスレッド・スイッチへの対応として，中断トランザクションの状態管理，中断したトランザクションの競合検出とロールバックについて述べる．

5.1　中断トランザクションの状態管理

不可分な結果と同じ結果になることを保証するために，トランザクションは中断している間も競合検出とロールバックを行わなければならない．また，再開時に中断直前の状態を回復し，トランザクションを引き続き実行できなければならない．

競合検出については，キャッシュ・タグまたはシグネチャを用いる手法を 2.2 で述べた．しかし，スイッチした別のトランザクションが同じキャッシュにアクセスするため，キャッシュ・タグを用いて中断トランザクションとの競合検出することは困難である．このため，スレッド・スイッチに対応するトランザクショナル・メモリ [5, 6] では，シグネチャによる競合検出を行い，チェックポイント時だけでなく中断時にもシグネチャをログに保存する．再開時には中断時に保存したシグネチャを回復することで以降の競合検出を行う．

ロールバックのためには，チェックポイント時のレジスタ状態とトランザクションによって書き換えられる前の値が必要である．それらはログによって保持されているため，中断後でも残しておくことができる．

5.2　中断トランザクションの競合検出

実行中のトランザクションは，アクセスするたびに中断しているトランザクションとの競合を検出する必要がある．LogTM-SE[5] や FlexTM[6] では，通常のリード/ライト・シグネチャとは別に設けられた**サマリ・シグネチャ**(summary

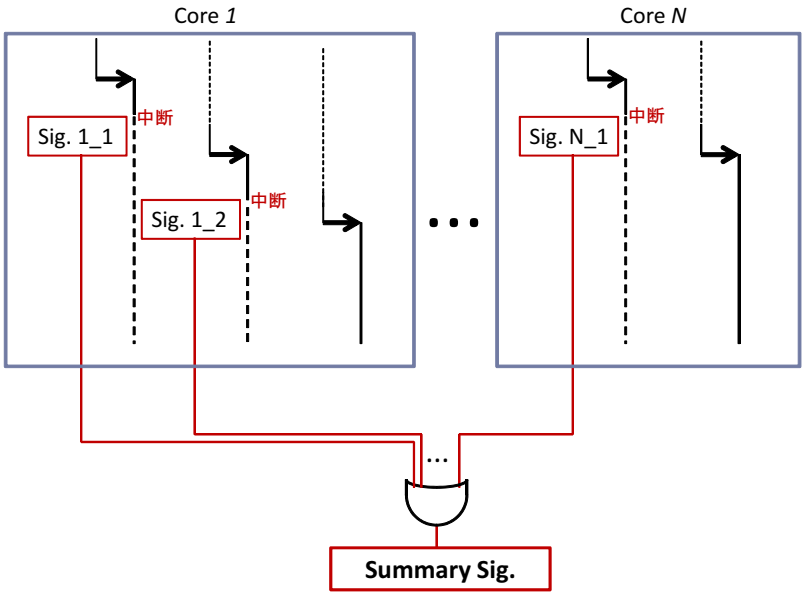

図 5.1: サマリ・シグネチャ

signature) を用いて中断トランザクションの競合検出を行う．サマリ・シグネチャとは，リード/ライトそれぞれ設けられる．リード/ライト用サマリ・シグネチャは，あるプロセス中で中断したすべてのトランザクションのリード/ライト・シグネチャについて OR を取ったシグネチャである．中断トランザクションのシグネチャはログに保持されているため，スレッド・スイッチごとにソフトウェアによってシグネチャがロードされ，サマリ・シグネチャが更新される．各トランザクションではアクセスごとに，他コアのリード/ライト・シグネチャだけでなく，サマリ・シグネチャも調べて中断トランザクションとの競合検出を行う．

図 5.1 では，各コアでトランザクションが実行され，一部のトランザクションが中断している．例えば，コア 1 では 3 つトランザクションのうち 2 つは中断し，3 つ目のトランザクションが現在実行されている．サマリ・シグネチャは，どのコアで中断しているかは関係なく，すべての中断トランザクションのシグネチャで OR を取って更新する．

LogTM-SE では，すべてのコアが同じサマリ・シグネチャを保持する．各中断トランザクションが再開してコミットした時にサマリ・シグネチャは更新される．一方，FlexTM は，サマリ・シグネチャをディレクトリに保持しておき，スレッド・スイッチごとにサマリ・シグネチャを更新する．

5.3 競合トランザクションの検索

競合したトランザクションのうち，実行/中断中に関わらず，どちらかのトランザクションをロールバックしなければならない．

LogTM-SE では，競合の検出のみを行い，競合した実行中のトランザクションをロールバックする．競合トランザクションを検索することがないため，すぐにロールバックの処理を開始することができる．

FlexTM では，中断トランザクションの中から競合したトランザクションを検索する．サマリ・シグネチャによって競合があることを確認し，予めログに保存してあった中断トランザクションすべてのシグネチャをソフトウェアによって調べることで競合トランザクションを特定する．その後，検索された競合トランザクションと競合トランザクションのいずれかをロールバックすることで競合を解決する．

5.4 既存の中断トランザクションのロールバックの問題点

上で述べた既存手法についての問題点を以下で述べる．

サマリ・シグネチャの誤検出　サマリ・シグネチャは，プロセス中すべての中断トランザクションのリード・ライト・アドレスを圧縮したものとなる．アクセスが多くなる長いトランザクションを多数実行するプログラムでは，誤検出率が非常に高くなる．

サマリ・シグネチャ更新のオーバヘッド　サマリ・シグネチャが最新のものに更新されるまでは，中断トランザクションとの競合検出ができない．このため，更新されるまでにすべてのトランザクションを実行することができない．つまり，更新のオーバヘッドによってトランザクションの実行効率が下がってしまう．しかし，更新には保存してある中断トランザクションのシグネチャをすべてロードしてくる必要があるため，オーバヘッドを小さくする必要がある．

競合トランザクション検索のオーバヘッド　競合した中断トランザクションが特定され，そのロールバックが完了するまで，競合した実行中のトランザクションは実行を続けることはできない．FlexTM では，その中断トランザクションを，サマリ・シグネチャの更新と同様に，保存されているすべてのシグ

ネチャをそれぞれロードして逐一調べることになる．これでは競合した中断トランザクションを特定するまでのオーバヘッドが大きい．

LogTM-SE でのロールバック　LogTM-SE では，中断トランザクションをロールバックしない．中断トランザクションとの競合時に必ず実行中のトランザクションをロールバックする．検索する必要がない一方で，中断トランザクションが再開しない限り，競合したトランザクションは繰り返し同じ競合を検出しロールバックする．このため，中断トランザクションが再開するまで競合したトランザクションはその競合する命令以降を実行することはできない．

中断トランザクションのロールバックのペナルティ　既存手法では，中断トランザクションについての部分ロールバックは考慮されていない．中断トランザクションは再開後，最初から再実行することになるため，ペナルティが大きい．

第6章　提案手法：中断トランザクションの最適な部分ロールバック

6.1　アプローチ

提案手法では，中断トランザクションの競合検出をコア別に行うことで競合トランザクションの検索オーバヘッドを削減し，中断トランザクションがロールバックする場合にも最適なチェックポイントの選択を行う．提案手法により中断トランザクションについても最適な部分ロールバックを効率的に行うことができる．

既存手法では全コアの中断トランザクションについてまとめて競合検出を行っていたが，5.4 節で述べたように，誤検出の増加や更新，検索のオーバヘッドが大きいといった問題がある．これに対し，コア別に競合検出を行うことで，シグネチャに含めるトランザクション数や検索の対象を削減する．

中断トランザクション最適なチェックポイントの選択を既存手法では考慮していないが，トランザクションは実行中にチェックポイントを取るため，その後中断したトランザクションも部分ロールバックすることができる．競合した中断トランザクションを検索後，最適なチェックポイントを選択することで再開後のペナルティを削減することができる．

図 6.1 は，各手法における中断トランザクションと競合した実行中のトランザクションを表している．競合したトランザクションは変数 x についてリードを行った後に中断している．その後，トランザクションが変数 x についてアクセスを行った場合に，各手法での競合の解決の様子を表している．LogTM-SE では，競合した中断トランザクションを検索せず，アクセスを行った実行中のトランザクションをロールバックする．このため，中断トランザクションが再開してコミットするまで，繰り返しロールバックして実行が進まない．FlexTM では，競合後にすべての中断トランザクションについて競合したトランザクションを検索する．すべての中断トランザクションを検索するため，オーバ

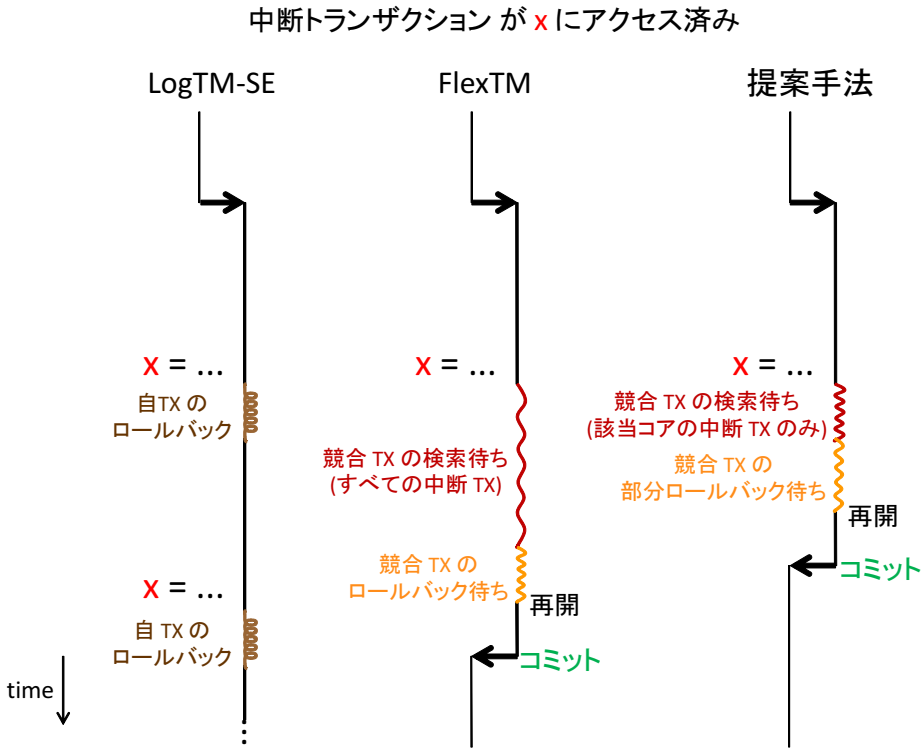

図 6.1: 中断トランザクションのロールバック

ヘッドが大きい．また，ロールバックによって中断トランザクションは最初から再実行されるため，競合した中断トランザクションの再実行時のペナルティは大きい．提案手法では，中断トランザクションの競合検索のオーバヘッドを削減し，競合した中断トランザクションも部分ロールバックを行い，再実行される命令を削減する．このように，中断トランザクションについても最適な部分ロールバックを行うことができる．

6.2 サスペンド・シグネチャ

コア別に中断トランザクションの競合検出を行うために，既存のシグネチャと別に，あるコアで中断したトランザクションすべてのシグネチャの OR を取ったシグネチャを用いる．このシグネチャを**サスペンド・シグネチャ**と呼ぶ．あるトランザクションがアクセスしたアドレスがサスペンド・シグネチャにヒットした場合，そのコアで中断しているトランザクションのいずれかが競合していることを検出できる．区別のため，ここではアクセスごとに更新される

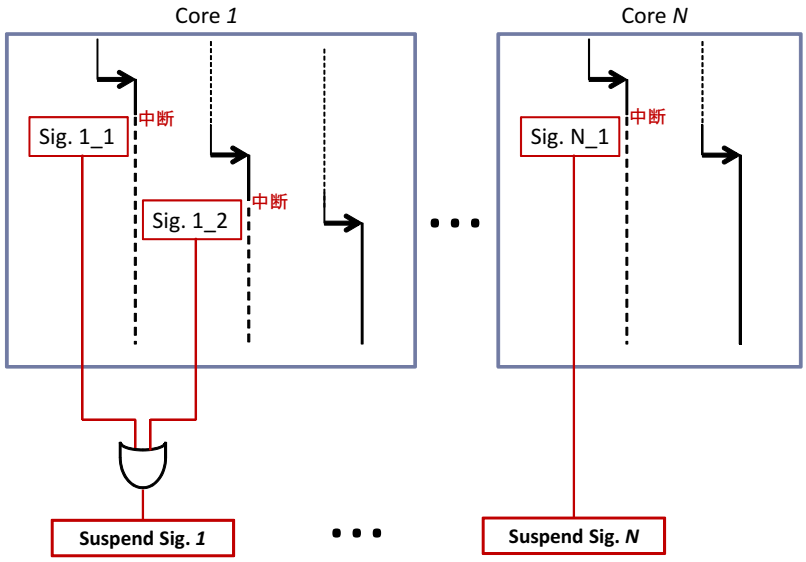

図 6.2: サスペンド・シグネチャ

既存のシグネチャをアクティブ・シグネチャと呼ぶ．各コアはアクティブ・シグネチャとサスペンド・シグネチャの両方をそれぞれリード/ライト用に 1 組ずつ保持する．既存手法のサマリ・シグネチャは全コアの中断トランザクションのシグネチャすべて OR を取っていたのに対し，サスペンド・シグネチャではそのコアの中断トランザクションのシグネチャのみを用いて更新される．

図 6.2 にサスペンド・シグネチャの例を示す．同図では，図 5.1 と同様に各コアがトランザクションを実行し，一部のトランザクションを中断している．コア 1 のサスペンド・シグネチャは，中断している 2 つのトランザクションのシグネチャの OR を取ったシグネチャとなる．コア N では，中断トランザクションが 1 つしかないため，そのトランザクションのシグネチャがコア N のサスペンド・シグネチャとなる．

サスペンド・シグネチャはスレッド・スイッチ時に更新される．各トランザクションは，既存手法と同様に，中断時にその時点のシグネチャをログへ保存しておく．サスペンド・シグネチャの更新は，一旦サスペンド・シグネチャのすべてのビットを 0 にすることでクリアし，そのコアで中断しているすべてのトランザクションのシグネチャを各ログよりロードして OR を取ることで行う．

サスペンド・シグネチャでは，サマリ・シグネチャよりも誤検出が少ない．なぜなら，既存手法のサマリ・シグネチャはすべての中断トランザクションによってアクセスされたアドレスのシグネチャであるのに対し，サスペンド・シグネチャはそのコアで中断したトランザクションだけにアクセスされたアドレ

スのみのシグネチャであるからである．また，サスペンド・シグネチャは 1 コアの中断トランザクションしか対象としないため，各ログから中断時のシグネチャをロードしてくる数が少なく，サマリ・シグネチャよりも更新のオーバヘッドは小さい．

6.3 中断トランザクションの最適な部分ロールバック

中断トランザクションの最適な部分ロールバックのために行う競合検出，競合トランザクションの検索，チェックポイントの選択の 3 つの処理について順に述べる．

6.3.1 競合検出

トランザクション中でのアクセス時に，そのトランザクションが実行されているコアのサスペンド・シグネチャと，他コアのアクティブ・シグネチャ，サスペンド・シグネチャにそのアドレスを含まれるか調べることで競合検出する．いずれかのコアのサスペンド・シグネチャで競合を検出した場合は，そのコアで実行しているトランザクションを中断し，後で述べる競合トランザクションの検索を行う．一方，他コアのアクティブ・シグネチャとの競合が検出された場合は，実行中トランザクション同士の競合であるので，4 章で述べたチェックポイントの選択を行う．

6.3.2 競合トランザクションの検索

シグネチャを用いて検索を行う．ログはトランザクションが中断してもそのトランザクションがコミットするまで保持されている．サスペンド・シグネチャによる競合検出時には，そのコアで中断したトランザクションのログに保存されている中断時のシグネチャに競合アドレスを含まれるか調べればよい．この処理はソフトウェアによって行われる．そのシグネチャに競合アドレスが含まれれば，そのシグネチャのトランザクションは競合した中断トランザクションであると認識される．

6.3.3 チェックポイントの選択

競合トランザクションを検索後，その中断トランザクションをロールバックする場合は最適なチェックポイントを選択する．サスペンド・シグネチャによ

る競合検出し，中断時のシグネチャに競合があった場合，そのトランザクションのログを用いてロールバックを行う．4 章で述べたように，実行中に予め過去の競合命令直前でチェックポイントを取っておき，ロールバックを行いながら各チェックポイントでのシグネチャが競合アドレスを含むか調べる．競合を含まないシグネチャが見つかれば，そのシグネチャを記録したチェックポイントが最適なチェックポイントであることがわかる．

第7章　評価

7.1　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 を合わせて用いた．GEMS では，Ruby モジュールを用いて LogTM-SE のメモリ・シミュレーションが可能であり，これを修正して提案手法の部分ロールバックを実装した．各パラメータは**表 7.1** の通りである．シグネチャは誤検出のないパーフェクト・シグネチャとバルク・シグネチャ[4] を用いた．LogTM-SE(部分ロールバックあり/なし) と提案手法の部分ロールバックについて，全スレッド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．また，スレッド・スイッチは発生しない．ベンチマークは GEMS 付属の microbenchmarks と，STAMP[10] の kmeans，vacation，labyrinth を用いた．

表 7.1: 評価パラメータ

processor	IPC 1(in-order)，16core
L1D cache (private)	32 KB，4 way，64 Byte line
L1 cache latency	1 cycle
L2 cache (shared)	8 MB，8 way，64 Byte line
L2 cache latency	20 cycle
Memory latency	200 cycle
Directory latency	6 cycle
Interconnection network latency	3 cycle
Signature size	2 Kbit

7.2 評価結果

microbenchmarks パーフェクト・シグネチャによる結果を**図 7.1** に，バルク・シグネチャによる結果を**図 7.2** に示す．縦軸は部分ロールバックなしの LogTM-SE を 1 とした相対速度，横軸はプログラム名とスレッド数である．slist では，提案手法が 15 スレッドで最大 6.9 倍の性能向上を確認した．バルク・シグネチャを用いる場合では，提案手法の性能向上がパーフェクト・シグネチャの場合よりも小さく，15 スレッドの prioqueue では 18.1% の性能低下が見られた．

STAMP パーフェクト・シグネチャによる結果を**図 7.3** に，バルク・シグネチャによる結果を**図 7.4** 示す．縦軸は部分ロールバックなしの LogTM-SE を 1 とした相対速度，横軸はスレッド数である．また，各プログラム名の後ろの low/high は，競合の頻度を表す．トランザクションが短く，競合の少ない kmeans では，チェックポイントを設定することが少なく，LogTM-SE とほぼ同等の性能であった．kmeans よりトランザクションが長く，競合の多い vacation では，8 スレッドの実行において性能向上が確認できた．最大でパーフェクト・シグネチャを用いた 8 スレッドの vacation-high で競合命令直前にチェックポイントを設定した手法では，10.0% の性能向上が見られた．vacation よりさらにトランザクションが長く，競合が多い labyrinth では，28.8% の大きな性能低下が見られた．バルク・シグネチャでは競合の多い場合の性能低下がパーフェクト・シグネチャの時より大きく，4 スレッドの vacation-low で最大 9.7% の性能低下を確認した．

7.3 考察

大きく性能向上した slist では，処理時間の長いトランザクションの終盤の同じ命令でほとんどの競合が起きている．ここでの LogTM-SE の部分ロールバックでは内側の開始点に戻ることができず，最初の開始点にロールバックするため，すべての処理を再実行することになっている．それに対し，提案手法ではチェックポイントがその競合命令直前で取られ，そのチェックポイントを選択できるので部分ロールバックの効果が大きく出ている．STAMP の vacation でも，スレッド数を多くすると競合が増えるが，提案手法の競合に適したチェックポイントの位置と選択によって部分ロールバックによる効果が現れる．

部分ロールバックのためのチェックポイント選択の処理よりも再実行される処理の方が少ない場合，提案手法では性能低下を起こす．prioqueue のような短いトランザクションでは部分ロールバックの効果が小さく，チェックポイン

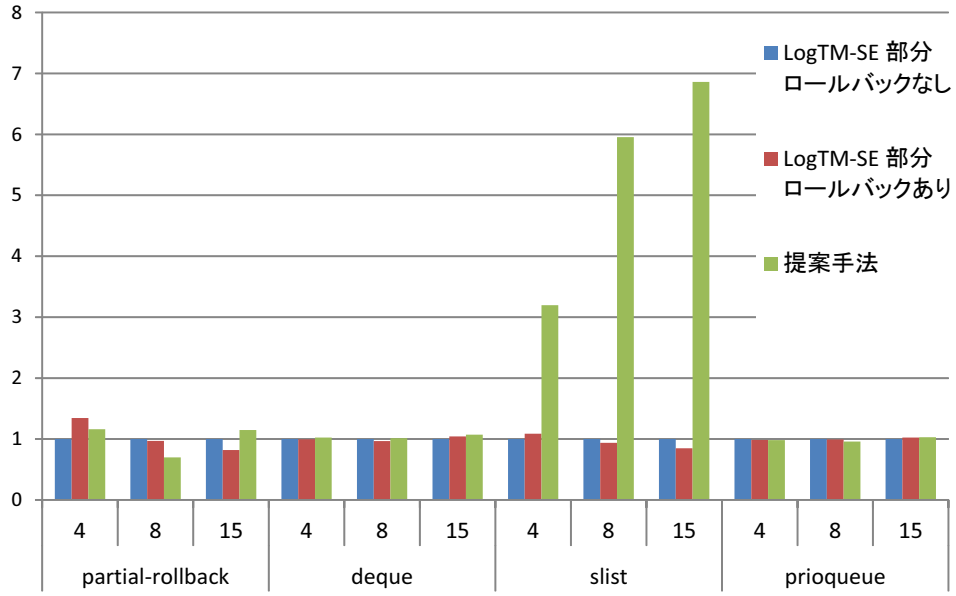

図 7.1: パーフェクト・シグネチャを用いた評価結果 (microbenchmarks)

ト選択のために性能低下が大きくなる．トランザクションの長い labyrinth でも，競合がトランザクションの前半の方に集中しているため，部分ロールバックの効果がほぼなく，チェックポイント選択のオーバヘッドが出ている．

シグネチャによるチェックポイントの選択について，バルク・シグネチャでの 8 スレッドの vacation は性能低下していることから，チェックポイント選択におけるシグネチャの誤検出によって適切な部分ロールバックができていないことが考えられる．

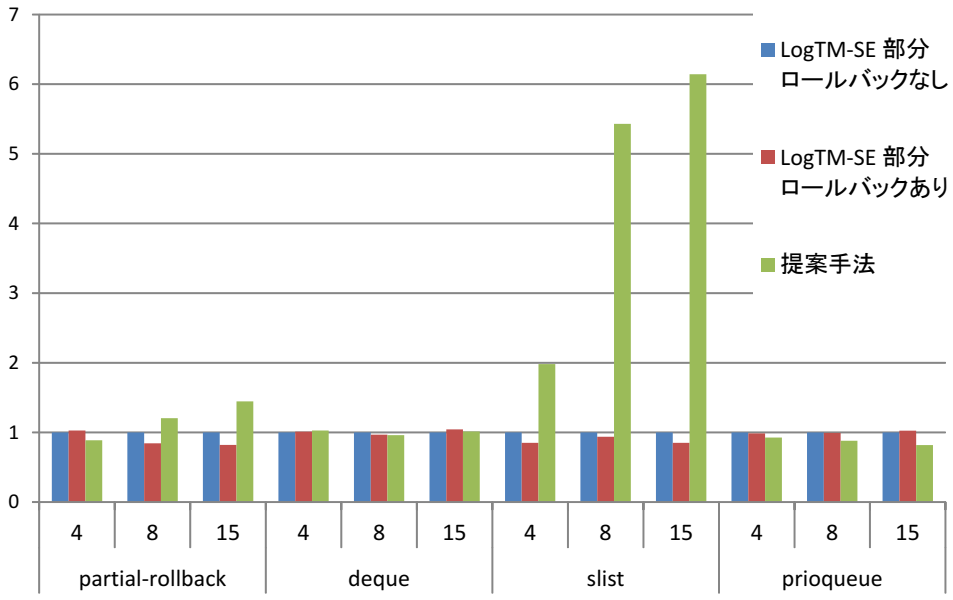

図 7.2: バルク・シグネチャを用いた評価結果 (microbenchmarks)

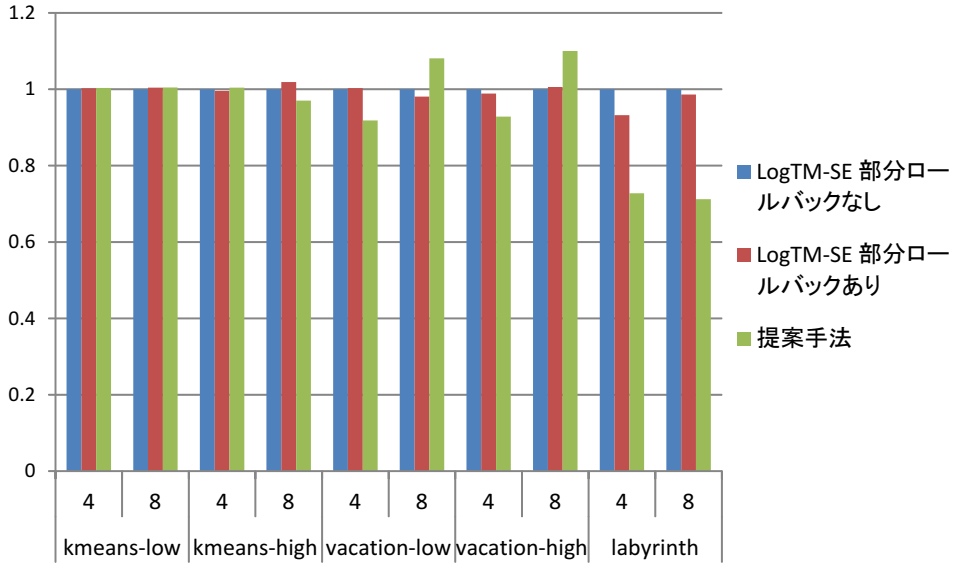

図 7.3: パーフェクト・シグネチャを用いた評価結果 (STAMP)

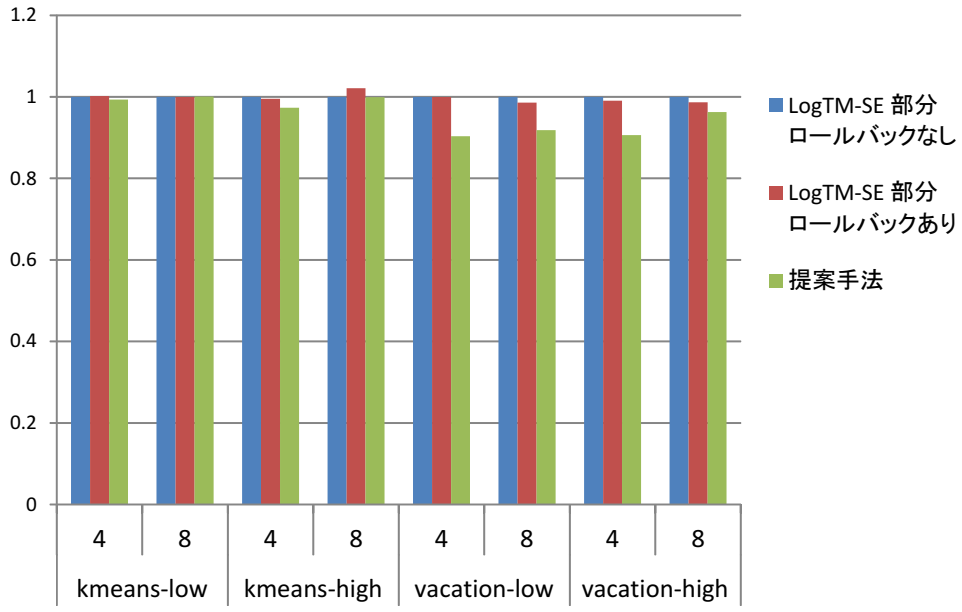

図 7.4: バルク・シグネチャを用いた評価結果 (STAMP)

第8章　おわりに

本稿では，最適なチェックポイントを選択する手法と，スレッド・スイッチによって中断したトランザクションの部分ロールバックを行う手法を提案した．最適なチェックポイントの選択は，過去の競合命令直前にチェックポイントを設定し，シグネチャを用いて選択することで行う．中断トランザクションの部分ロールバックは，サスペンド・シグネチャを用いてコア別に競合検出を行い，該当するコアのみの中断トランザクションを検索して競合トランザクションを検索し，最適なチェックポイントを選択することで行う．本手法により，あらゆるトランザクションで最適なロールバックを行い，プログラマに対するトランザクションの長さやトランザクションを実行するスレッド数の制限を緩和することができる．シミュレーションによる評価では，提案手法の部分ロールバックによって最大6.9倍の性能向上したことを確認した．

今後の課題として，トランザクション中のスレッド・スイッチが起きる場合の提案手法の評価を行うことが挙げられる．

参考文献

[1] Maurice Herlihy, J. Eliot, and B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[2] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, 2006.

[3] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[4] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[5] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[6] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, 2008.

[7] 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.

[8] 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トランザクショナル・メモリ. 情報処理学会研究報告 2010-ARC-190, 2010.

[9] M. M. Waliullah and Per Stenstorm. Intermediate checkpointing with conflicting access prediction in transactional memory systems. In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*. 2008.

[10] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, 2008.

発表文献

主著論文

1. 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，塩谷 亮太，五島 正裕，坂井 修一
   情報処理学会研究報告 2009-ARC-184, (2009).

2. 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
   伊藤 悠二，塩谷 亮太，五島 正裕，坂井 修一
   情報処理学会第 72 回全国大会, pp. 1-187-1-188 (2010).

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，塩谷 亮太，五島 正裕，坂井 修一
   情報処理学会研究報告 2010-ARC-190, (2010).

4. 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，塩谷　亮太，五島 正裕，坂井 修一
   先進的計算基盤システムシンポジウム SACSIS2011 (投稿中)

謝辞

本研究を進めるにあたり, 多くの方々にご指導，ご協力いただき，大変お世話になりました.

指導教員である坂井修一教授には，相談会などにおいて多くのご指導いただきました．また，五島正裕准教授には，本研究の進行から論文の添削まであらゆる面においてご指導いただくなど，特にお世話になりました．本当にありがとうございました.

塩谷亮太氏には，研究に関する指摘や助言などをいただきました．ありがとうございました.

八木原春水さん，伊世知代さん，長谷部環さんには，研究を行う上での各種事務手続きなどで，お世話になりました.

坂井・五島研究室の皆様には，研究室での生活や研究へのアドバイスなど，様々な面でご協力いただきました。心より感謝いたします。