Vol.2012-ARC-201 No.2
2012/8/1

過去の競合命令にチェックポイントを設定するトランザクショナル・メモリ

阿部高大[†1]　倉田成己[†1]　塩谷亮太[†2]　五島正裕[†1]　坂井修一[†1]

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．長いトランザクションでは，ロールバックが大きなペナルティとなる．トランザクション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できる．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，シグネチャ化したログによってロールバック・ポイントを選択することで, 部分ロールバックによるペナルティを最小にするような手法を提案する．

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], [2], [3], [4], [5], [6], [7], [8], [9] 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．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．

1.1 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 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 で詳しく述べる．

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**として現れる．例えば，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．しかし，トランザクションの長さを制限することはプログラマにとって大きな負担となるため，このペナルティを小さくする必要がある．

1.2 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．

部分ロールバックのためには，トランザクション中に予

†1 現在，東京大学大学院情報理工学系研究科
Presently with Graduate School of Information Science and Technology, The University of Tokyo
†2 現在，名古屋大学大学院工学系研究科
Presently with Graduate School of Engineering, Nagoya University

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．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されており [7]，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に用いるデータ構造　チェックポイントをどのようなデータ構造を用いて選択するか．

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 [10] 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 で述べる．

本稿の提案は，(III+b) である：

チェックポイントの位置　学習により過去に競合を起こした命令の直前に設定する

チェックポイントの選択　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．

提案手法については，4 で詳しく述べる．

2. トランザクショナル・メモリ

2.1 トランザクションの投機実行

1 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 で述べる．

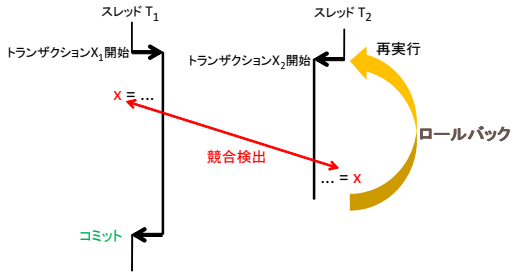

図 1　トランザクションの投機実行

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．チェックポイントの状態の保存については，2.3 で述べる．

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_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，そのまま全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．

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．

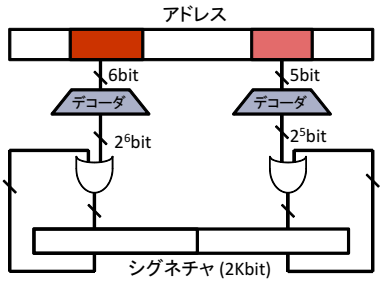

図 2 シグネチャの例
Fig. 2 Example of a signature

Waliullah らの手法 [10] 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** 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 ログ

ロールバックでは，競合時の状態からチェックポイントの状態に戻す．そのチェックポイントによってトランザクション中に書き換えられたアドレスの値やレジスタ状態は異なる．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，チェックポイント時のレジスタ状態と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とで保持される．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．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できる．

3. 関連手法

本章では，既存手法の部分ロールバックにおけるチェックポイントの位置やチェックポイントの選択について述べる．

3.1 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．

3.1.1 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．Moravan らが提案する部分ロールバックに対応した 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．

3.1.2 競合アドレスへのアクセス直前

Waliullah らの手法 [10] では，競合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．

3.2 チェックポイントの選択

本節では，チェックポイントの選択にキャッシュ・タグを用いる手法とシグネチャを用いる手法について述べる．

3.2.1 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 Waliullah らの手法 [10] では，競合検出はリード/ライト・ビットを行

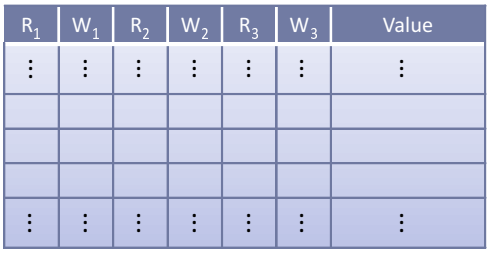

図 3 チェックポイントを選択するキャッシュ

い，チェックポイントの選択はさらにリード/ライト・ビットを拡張して**図 3** 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 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．

3.2.2 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．そのために，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．各チェックポイントで保存されたシグネチャは，トランザクション開始からそのチェックポイントまでのアクセスしたアドレスすべてを含む．これらシグネチャによって競合したアドレスがどのチェックポイントまでにアクセスされたか識別できる．

現在のシグネチャによって競合を検出したら，直前のチェックポイントまで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．

図 4 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．

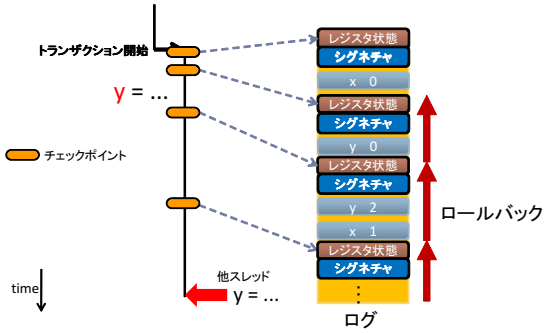

図 4 シグネチャによるチェックポイントの選択

3.3 関連手法の問題点

3.3.1 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．さらに，部分ロールバックに対応した LogTM[3] や LogTM-SE[5] では，終了した内側の開始点に部分ロールバックしない．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からである．以降，内側トランザクションの命令は，外側トランザクションで実行された命令として扱われる．そのため，内側トランザクションではなく，外側トランザクションごとロールバックすることになる．

3.3.2 キャッシュ・タグによるチェックポイントの選択

キャッシュ・タグでのチェックポイントの選択では，チェックポイント数が強く制限される．図 3 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．Waliullah らの手法 [10] 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．

3.3.3 チェックポイントの無効化

LogTM-SE では，トランザクション単位のロールバック

しか許していないため，最適な部分ロールバックができないことがある．終了した内側のトランザクションは外側のトランザクションの一部として扱われる．内側のトランザクションの終了時に，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．それら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．

4. 提案手法：過去の競合命令にチェックポイントを設定するトランザクショナル・メモリ

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，競合時にシグネチャを用いて選択を行う．また，トランザクション実行中にそれらのチェックポイントを無効化しない．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トランザクションのコミットまたはロールバック時まで保持し続ける．

図 5 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) 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-SE では，内側の開始点にしか部分ロールバックできず，終了した内側の開始点への部分ロールバックを許さず，チェックポイントを無効化するため，再実行される命令が多くなってしまう．Waliullah らの学習によるチェックポインティング手法 [10] 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，以前競合した x へアクセスする命令直前のチェックポイントを設定し，途中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，競合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ため，チェックポイント同士の間隔が短くな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，3.2 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．内側の終了点は，トランザクション全体において何の意味もなく，終了点でチェックポイントを無効化する必要は全くない．従って，ネスティッド・トランザクションの構造とチェックポイントは無関係であり，内側のトランザクションが終了してもログに保存されたシグネチャとレジスタ状態を無効化する処理は行わない．

5. 評価

5.1 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を実装した．各パラメータは**表 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・シグネチャを用いた．

評価対象として GEMS 付属のベンチマークプログラムである contention,partial-rollback,prioque と，STAMP[11] の kmeans を用い，GEMS 付属のベンチマークでは 15 スレッド,STAMP は 8 スレッドで実行し評価を行った．

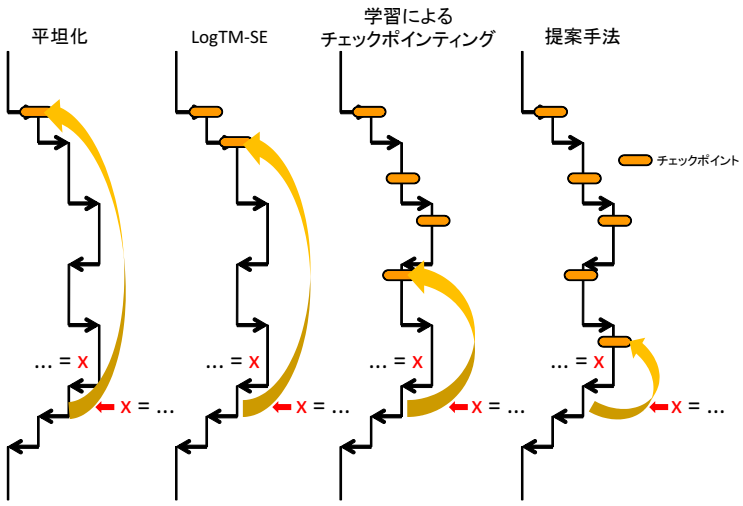

図 5 提案手法の目的

表 1 評価パラメータ

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

表 2 各ベンチマーク実行時のアボート数

ベンチマーク名	LogTM(BASE)	LogTM(TIMESTAMP)	提案手法
contention	210	10183	13649
partial-rollback	64	106	118
prioque	21244	22061	23205
kmeans	27	66	203

提案手法と LogTM-SE についてベンチマーク実行時の総サイクル数とアボート数を計測し比較を行う．LogTM-SE については競合検出時の動作についてのオプションを変え，提案手法とあわせて競合検出時にそれぞれ以下の様に動作する．

- LogTM-SE(BASE)：競合検出時に，後から競合したメモリアドレスにアクセスしたトランザクションがストールする．
- LogTM-SE(TIMESTAMP) ：競合検出時に，後からトランザクションを開始したスレッドがストールする．
- 提案手法 ：トランザクション同士が競合した場合，後からトランザクションを開始したスレッドがロールバックする．

5.2 評価結果

contention，partial-rollback,，prioque，を 15 スレッドで実行したときと，STAMP の kmeans を 8 スレッドで実行したときの評価結果のグラフを**図 6** に示す．また，評価対象実行時のアボートの回数を**表 2** に示す．

図 6 のグラフの縦軸は LogTM-SE(BASE) で評価対象を実行した時にかかった総サイクル数を 1 とした時の，LogTM-SE(TIMESTAMP) と提案手法での総実行サイクル数の比率，横軸はベンチマーク名ととスレッド数である．凡例は評価対象の実行時におけるサイクル数の内訳を示しており，それぞれの項目は以下の通りである．

- nontrans ： トランザクション外で要した実行サイクル数
- goodtrans ： コミットされたトランザクションの実行サイクル数
- badtrans ： 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
- barrier ： バリア同期に要したサイクル数

LogTM-SE(BASE) に対して提案手法では prioque で最大約 40%，kmeans で約 14% の総実行サイクル数を削減した．一方 contention で最大約 70%，partial-rollback で約 63 %総実行サイクル数が増加している．アボート回数については，contention で LogTM(BASE) に対して最大約 65 倍程度増加している．

5.3 考察

提案手法によって総実行サイクル数が減少した prioque，kmeans では，競合を起こした命令に設定されたチェックポイントへの部分ロールバックの効果が現れている．prioque おいては，アボート数に大きな差はないものの LogTM-SE(BASE) での実行時にはほぼ全ての競合が同一命令により引き起こされており，LogTM-SE(TIMESTAMP) や提案手法においては実行時には競合を起こした命令にばらつきがみられた．prioque 実行時に LogTM-SE ではトラン

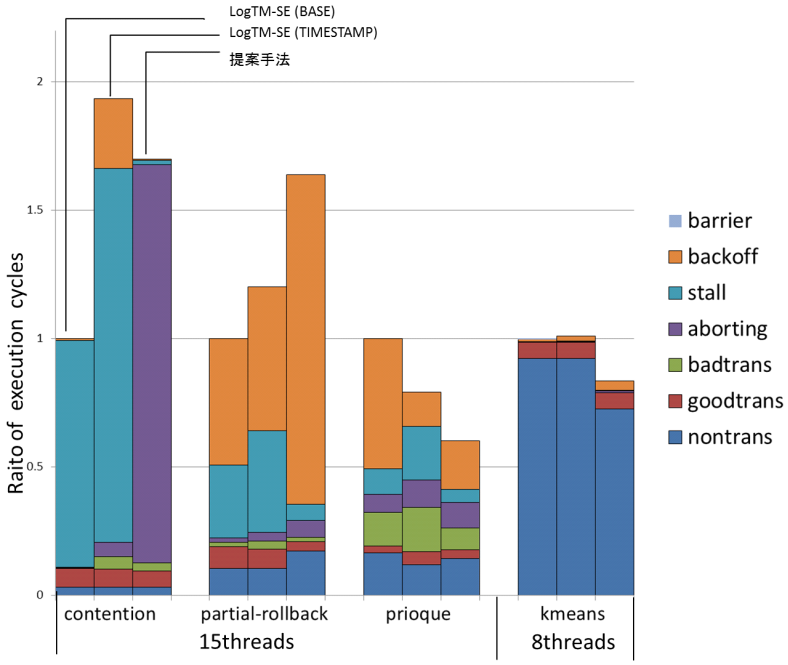

図 6 評価結果

ザクションの開始直後にチェックポイントが設定されるようになっており，提案手法でトランザクションの途中にチェックポイントを設定したため，ロールバックによるペナルティが LogTM-SE(TIMESTAMP) の場合よりも減少したと考えられる．

また，kmeans においても prioque の時と同様に，競合を起こした命令が LogTM-SE(BASE) での実行時には 2 種類であり，その他の結果については競合を起こした命令にばらつきが見られたため，部分ロールバックによる効果が現れたのではないかと考えられる．

一方，提案手法によって実行総サイクル数が増えてしまった contention ついては，LogTM-SE(BASE) に対し，LogTM-SE(TIMESTAMP) や提案手法の実行結果において競合の増加が見られた．提案手法では競合数の多さからチェックポイントを多く設定してしまい，競合時にログを検索するコストによりアボートに要するサイクル数がが増加してしまったのではないかと考えられる．

partial-rollback については提案手法での総実行サイクル数のうち，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が約 78 %を占めている．今後このランダム要素が実行結果にどれだけの影響を与えるのか調査する必要がある．

6. 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，最適なチェックポイントを選択する手法を提案した．シグネチャによって最適なチェックポイントを特定し，再実行される命令を最小にすることで効率的にトランザクションを実行できる．

本手法の評価を行った結果，LogTM-SE（BASE）の総実行サイクル数に対し，最大約 39 %の実行サイクル数を削減することができた．

今後の課題として，競合の判定方法についての詳細なデータを取って調査を行う必要や, 提案手法についてデータ量を増やし,LogTM-SE 以外の手法との比較を行うことが考えられる．

謝辞

本論文の研究は一部，文部科学省科学研究費補助金 No. 23300013 による．

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 and Jayaram Bobba and Michelle J. Moravan and 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 and Jayaram Bobba and Kevin E. Moore and Luke Yen and Mark D. Hill and Ben Liblit and 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 Inter-*

national 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] 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. 先進的計算基盤システムシンポジウム SACSIS2011,Vol. 2011, pp. 324-331, 2011.

[10] 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.

[11] 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.