修士論文

競合トランザクションを逐次実行するトランザクショナル・メモリ

Transactional Memory That Sequentially Executes Conflicting Transactions

平成 26 年 2 月 6 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

48-126407 岩田 愛実

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．トランザクショナル・メモリではプログラマがトランザクションとして指定した命令列が不可分に実行されているかのように投機実行される．トランザクションの実行中にはスレッドのメモリ・アクセスを監視し，他スレッドのアクセスと競合した場合には，一方のスレッドを停止させて競合が解決するのを待つか，一方のスレッドの実行を破棄して再実行する．

競合が頻繁に発生するトランザクションは実行効率が悪くなる．トランザクショナル・メモリ用のベンチマークを解析すると，競合が発生しやすいアクセス・パターンとして，あるアドレスに対して短期的にリード・ライト・アクセスを行うものがあるとわかる．本研究では，このようなパターンに対してキューを用いることで，競合トランザクションを逐次実行させる手法を提案する．本手法の評価では，平均 6.6 %の性能向上を達成できた．

目次

第1章　はじめに **1**

第2章　トランザクショナル・メモリ **3**

2.1　トランザクショナル・メモリの概要 3
2.2　競合検出 5
2.3　競合解決 6
2.4　バージョン管理 7
2.5　従来の研究の問題点 8

第3章　STAMP **10**

3.1　STAMPの特徴 10
3.2　kmeans 10
3.2.1　kmeansの概要 10
3.2.2　競合の頻発 11
3.2.3　実装の問題点 13
3.3　vacation 14
3.4　genome 14

第4章　提案手法 **16**

4.1　アプローチ 16
4.2　キューを用いたアクセス管理 17
4.2.1　コアとメモリ間の通信 17
4.2.2　競合頻発箇所での実行例 18
4.2.3　キューのデッドロック 21

第5章　評価 **23**

5.1　評価環境 23
5.2　評価結果 24
5.3　考察 24

第 6 章　おわりに 28

参考文献 29

図 目 次

2.1 トランザクションの投機実行 4
2.2 possible_cycle によるデッドロックの解消 7

3.1 kmeans のあるトランザクションの実行の様子 11
3.2 競合が頻発するトランザクション 12
3.3 CHUNK を 3 または 1 としたときの kmeans での実行サイクル数の比率 13

4.1 コアとメモリ間の通信 17
4.2 競合が頻発する例 19
4.3 提案手法でのトランザクションの実行の様子 20
4.4 キューで生じるデッドロックの例 21

5.1 CHUNK が 3 のときの評価結果 25
5.2 CHUNK が 1 のときの評価結果 26

表 目 次

5.1　評価パラメータ 23

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．その原因の一つとして，同期通信に用いられる**ロック**の存在が挙げられる．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．

一方，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**[8]が提案されている．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．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，投機処理を行えば，大きな性能低下は起こらない．

トランザクショナル・メモリの実装の多くは，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の実行を停止して競合が解決されるまで待つか，トランザクションの投機状態を破棄し再実行を行う．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．トランザクションを再実行するには，トランザクション実行前の状態を保持しておく必要がある．トランザクションの投機状態および実行前の状態を保持しておくための仕組みを**バージョン管理**という．

トランザクショナル・メモリの従来の研究では，競合検出とバージョン管理の各手法を組み合わせることで，特定のベンチマークに対する性能向上を図る傾向があった．そこで，本論文では着目点を変え，トランザクショナル・メモリで実行するアプリケーションを解析し，その上でトランザクションの特徴に合わせた性能向上手法を検討した．

本論文では，トランザクショナル・メモリの実行時に競合が頻発するアドレスに対して，通常のトランザクションでの投機実行ではなく，逐次的にアクセスを行う手法を提案する．提案手法では，該当する変数に対してキューを用意し，1 つのスレッドがアクセスしている際には他のコア上で動作しているスレッドをキューに追加して待機させる．トランザクションのコミット時に，キューの先頭のコア上で動作しているスレッドのアクセスを許可する．これを繰り返すことで，競合頻発トランザクションの逐次実行を実現する．

本論文は，次のように構成される．まず，2 章でトランザクショナル・メモリの概要と従来のトランザクショナル・メモリの研究の問題点を説明する．次に，3 章でトランザクショナル・メモリのベンチマークとして用いられる STAMP について，各アプリケーションの特徴とアプリケーション固有の問題点を説明する．4 章では，2 章と 3 章で指摘した問題点を踏まえた上で，提案手法を説明する．5 章で評価結果を述べる．最後に 6 章でまとめと今後の課題を述べる．

第2章　トランザクショナル・メモリ

本章では，トランザクショナル・メモリの基本的な仕組みと，従来のトランザクショナル・メモリの研究における問題点を述べる．

2.1　トランザクショナル・メモリの概要

トランザクションとは，トランザクショナル・メモリにおいて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．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．

トランザクションは以下に示す2つの性質を持つ．

アトミシティ(atomicity)　トランザクションは不可分に実行した場合と同じ結果になる．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．

シリアライザビリティ(serializability)　トランザクションは逐次的に実行した場合と同じ結果になる．

これらの性質を満たすため，トランザクショナル・メモリでは競合検出を行う．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．競合が検出されたときには，トランザクションのアトミシティを保つために，競合解決を行う．競合解決の1つの手法としては，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．別の競合解決法として，競合相手のトランザクションが終了するまで

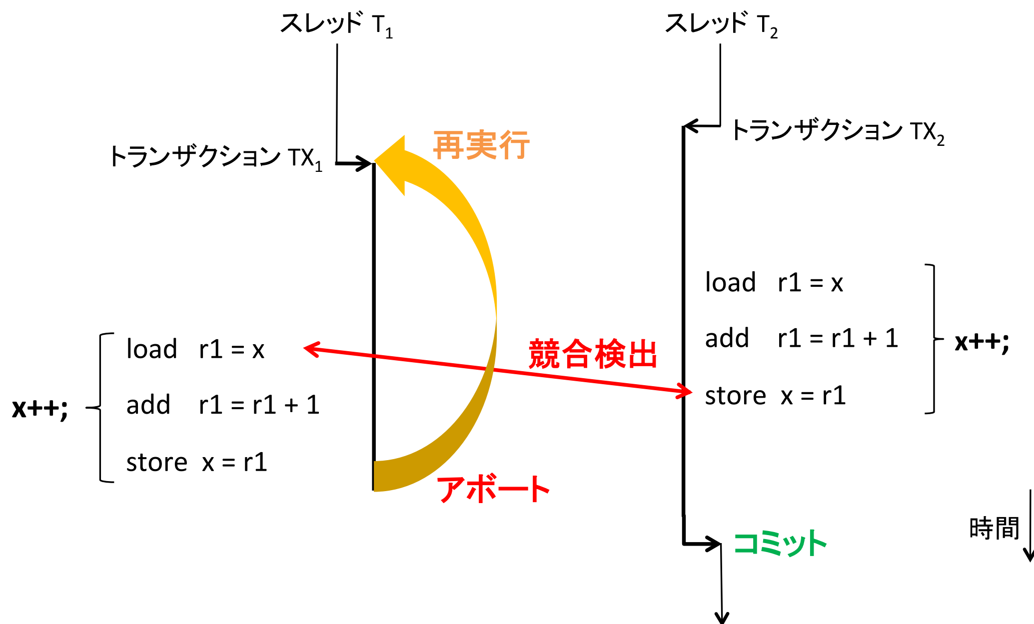


図 2.1: トランザクションの投機実行

実行を一時停止させ，競合解決後に再開する手法がある．トランザクションの実行を一時的に停止する操作を**ストール** (stall) と呼ぶ．

図 2.1 に，トランザクションの実行の様子を示す．図 2.1 では，スレッド T_1 でトランザクション TX_1 が実行され，スレッド T_2 でトランザクション TX_2 が実行されている．変数 x について，それぞれのスレッドにおいてトランザクション中でインクリメントを行っている．TX_2 で変数 x をロードしたあと，TX_1 でも同様に変数 x をロードしている．続いて，TX_2 がストアを実行すると，競合が検出される．もし競合を検出しなければ，TX_1 は TX_2 によるインクリメントの結果を反映させずに変数 x のインクリメントを行うことになるため，逐次的に実行した場合と異なる結果になり，シリアライザビリティが満たされない．競合が検出されたら競合解決を行うことになるが，2.3 節で説明するように競合解決にはいくつかの手法がある．ここでは，TX_2 をアボートし，再実行している．一方，TX_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．

2.2 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．

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を説明する．

キャッシュ・タグによる検出

キャッシュ・タグによる検出では，キャッシュ・ラインごとにリード・ビットとライト・ビットを用意し，トランザクションによるリード/ライト・アクセスが行われたら，該当ビットをセットする．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する．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．コミット時には，リード/ライト・ビットを全て 0 にすることでクリアする．一方，アボート時にはライト・ビットがセットされているキャッシュ・ラインを無効化し，コミット時と同様にリード/ライト・ビットを全てクリアする．

競合検出にキャッシュ・タグを用いるトランザクショナル・メモリには LogTM[10] などがある．

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[4] を用いて競合を検出する．シグネチャとは，スレッドへのアクセス情報を，ハッシュ関数を用いることでコンパクトにまとめて保持するためのビット列である．コアごとにリード/ライト・アドレス用にリード/ライト・シグネチャが用意される．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，そのアドレスに付いての競合であると検出する．コミット時には，シグネチャの全てのビットを 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．

競合検出にシグネチャを用いるトランザクショナル・メモリには LogTM-SE[12] などがある．

2.3 競合解決

競合が検出されたとき，その競合状態を解決するために行う動作を競合解決という．競合解決方法は大きく分けて 2 種類ある．

- 競合検出されたトランザクションのどちらか一方を，競合状態が解消されるまでストールさせる．
- 競合検出されたトランザクションのどちらか一方をアボートさせて，トランザクションの最初から再実行する．

LogTM では前者を採用し，さらにデッドロック状態を回避するための機構として，各プロセッサごとに possible_cycle と呼ばれるフラグを用意している．

図 2.2 に，possible_cycle を用いてデッドロックを解消する仕組みを示す．図 2.2(a) では，スレッド T_1 とスレッド T_2 でそれぞれトランザクション TX_1 と TX_2 を実行している．TX_1 が B のロードを行おうとすると，TX_2 は既に B をストアしているため，競合の発生を検出して，TX_1 に対して NACK を返す．これにより，ロード・リクエストを行った TX_1 はストールする．続いて，TX_2 が A のロードを行おうとすると，TX_1 は既に A をストアしているため，競合の発生を検出して，TX_2 に対して NACK を返す．これにより，ロード・リクエストを行った TX_2 はストールし，TX_1 と TX_2 はデッドロック状態となる．

これを解消するために，図 2.2(b) では possible_cycle フラグを用いている．possible_cycle フラグは以下のように利用する．

- 自身より早くトランザクションを開始したスレッドからの要求によって競合が発生した場合，possible_cycle フラグをセットする．
- possible_cycle フラグがセットされている状態で，自身よりも早くトランザクションを開始したスレッドに対して要求を送ったことにより競合を検出すると，デッドロックを回避するためにアボートする．

(a) デッドロックが起こる場合　(b) possible_cycle を利用する場合

図 2.2: possible_cycle によるデッドロックの解消

図 2.2(b) では，図 2.2(a) と同様に TX_1 が B のロードを行おうとして競合が発生し，TX_2 から NACK が返される．このとき，TX_1 は TX_2 よりトランザクション開始時刻が早いため，TX_2 には possible_cycle フラグがセットされる（図 2.2(b) では $p_c = 1$ と表記している）．続いて，TX_2 が A のロードを行おうとすると，TX_1 は既に A をストアしているため，競合の発生を検出して，TX_2 に対して NACK を返す．このとき，TX_2 は possible_cycle フラグがセットされている状態で，自身よりも早くトランザクションを開始した TX_1 から NACK を受け取ることになるため，デッドロックを回避するために自身をアボートする．これにより，図 2.2(b) ではデッドロックを生じることなく TX_1 と TX_2 を実行できる．

2.4　バージョン管理

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ため，トランザクション内で更新した値と更新前の値とを併存させる必要がある．そこで，ハードウェア・トランザクショナル・メモリではトランザクション内で発生したライト・アクセスにより更新した値，あるいは更新される前の

古い値をそのアドレスとともに別の領域に保存する．このようなデータの管理をバージョン管理という．

トランザクショナル・メモリのバージョン管理は，以下に述べる 2 つの方式に分類される．

lazy 方式 ライト・アクセスにより更新された値をメモリ内の別領域に保存し，更新前の古い値をメモリ内に残しておく．アボートは別領域に保存していたデータを無効にするだけでよいので高速に行うことができるが，コミットは別領域に保存していたデータをメモリ内にコピーしなければならないので時間がかかる．これを採用したトランザクショナル・メモリには，Rock[5]，TCC[7]，VTM[11] などがある．

eager 方式 ライト・アクセスにより値が更新されるとき，メモリ内に新しい値を書き込み，古い値とそのアドレスとその時点でのレジスタ状態とをソフトウェア・ログに書き込む．コミット時には，すでにメモリの更新が完了しているため，古い値を破棄するだけでよい．一方，アボート時にはソフトウェアハンドラによってトランザクション開始前の状態に復帰しなければならないので，時間がかかる．これを採用したトランザクショナル・メモリには UTM[2]，LogTM，LogTM-SE などがある．

どちらの方式が適しているかは実行するワークロードのアボートの頻度によって異なる．

トランザクショナル・メモリはロックに代わる同期機構として研究されてきたという背景があるため，使用されるベンチマークもロック向けに最適化されたものが多かった．よって，トランザクションは粒度が小さく，アボートはほとんど起こらないものと考え，近年では eager 方式のバージョン管理を採用するトランザクショナル・メモリが多く研究される傾向にあった．しかし，実用上の観点では，粒度の大きなトランザクションを扱ったり，多スレッドでの実行を検討すべきである．このとき，アボートそのものやアボート時のオーバヘッドが増加すると考えられるため，eager 方式のバージョン管理のみに頼るのは性能の低下をもたらす可能性がある．そこで，最近ではバージョン管理について eager 方式と lazy 方式の両方の利点を活かせる設計が研究されている [9, 13]．

2.5 従来の研究の問題点

2.4 節でも指摘したように，トランザクショナル・メモリはロックに代わる同期機構として研究されてきたという背景があるため，ロック向けに最適化さ

れたベンチマークを利用する研究も多かった．そのようなベンチマークではトランザクションの粒度が小さく，実用的なトランザクションには即していないといえる．プログラマの負担を考えると，トランザクションの粒度が大きい方が書きやすいため，粒度の大きなトランザクションでも性能が低下しないトランザクショナル・メモリが求められている．

一方で，従来のトランザクショナル・メモリの研究では，2.2 節や 2.3 節，2.4 節の各手法を組み合わせ，特定のベンチマークに対する性能向上を図る傾向があった．このようなやり方では，実行するアプリケーションによって効率よく実行できる場合とそうでない場合とのばらつきが大きくなる．実用的なトランザクショナル・メモリでよく生じる問題を明らかにし，それに対する改善手法を検討することが求められる．

第3章　STAMP

STAMP[3] とは，トランザクショナル・メモリ用のベンチマークである．トランザクションの粒度や競合度の異なる 8 つのアプリケーションで構成されている．各アプリケーションは microbenchmark のように特定の条件に特化したベンチマークと異なり，より実用的な状況を想定したものである．

本章では，トランザクション中のデータ構造やメモリ・アクセスに着目して STAMP のソースコードを解析し，各アプリケーションの特徴やトランザクショナル・メモリのアプリケーション固有の問題点を指摘する．

3.1　STAMP の特徴

配列へのアクセスが中心となるアプリケーションでは，トランザクションの粒度は小さくなりやすい．一方，リストやハッシュテーブルなどのデータ構造を持つアプリケーションでは，トランザクションの粒度は大きくなりやすい．これは，先頭ノードから対象ノードまで辿るようなデータ・アクセス時にリード・アクセスを繰り返すことなどが原因である．

また，ライト・アクセスはトランザクションの後半に集中しやすい傾向がある．トランザクションの後半からアボートした場合，トランザクション開始時点まで戻ってから再実行すると，再実行ペナルティが大きくなる．

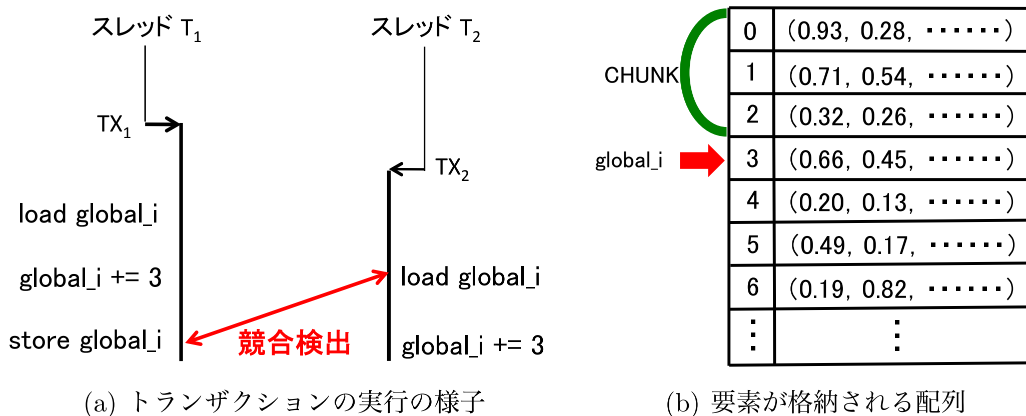
3.2　kmeans

3.2.1　kmeans の概要

kmeans は，クラスタリング手法の 1 つである K-means を実行するアプリケーションである．実行の手順は以下のとおりである．

1. 各要素に対してランダムにクラスタを割り振る．

2. 割り振ったデータを元に各クラスタの重心を計算する．



(a) トランザクションの実行の様子　　(b) 要素が格納される配列

図 3.1: kmeans のあるトランザクションの実行の様子

3. 各要素と各重心との距離を求め，その距離が最短となるクラスタに要素を割り当て直す．

4. 2, 3 の処理ですべての要素のクラスタ割り当てが変化しなかった場合は処理を終了し，それ以外の場合は再び 2 に戻る．

上記の手順のうち，トランザクションを用いて実行するのは 2 に該当する部分である．

図 3.1 に kmeans 内のあるトランザクションの実行の様子を示す．このトランザクションは，各クラスタの重心を計算するために，スレッドごとに計算する要素を割り当てるものである．各要素は図 3.1(b) に示すような配列で提供され，各スレッドが計算する対象要素は配列の先頭から順に CHUNK という変数の表す個数ずつ取得される．現在何番目の配列要素まで計算したかを示すために，global_i という変数を用いている．図 3.1(a) では，スレッド T_1 上で動作しているトランザクション TX_1 と，スレッド T_2 上で動作しているトランザクション TX_2 がそれぞれ global_i にアクセスして，競合が発生している．

3.2.2 競合の頻発

図 3.2 に，kmeans 内でのトランザクションで競合が頻発する様子を示す．なお，2.3 節で説明した possible_cycle フラグを用いた方法で競合解決を行うものとする．図 3.2 では，スレッド T_1 から T_3 でトランザクション TX_1 から TX_3 がそれぞれ実行されている様子を表している．トランザクションの内容は図 3.1(a) と同様，global_i の操作である．まず，TX_1 から TX_3 までが global_i のロード

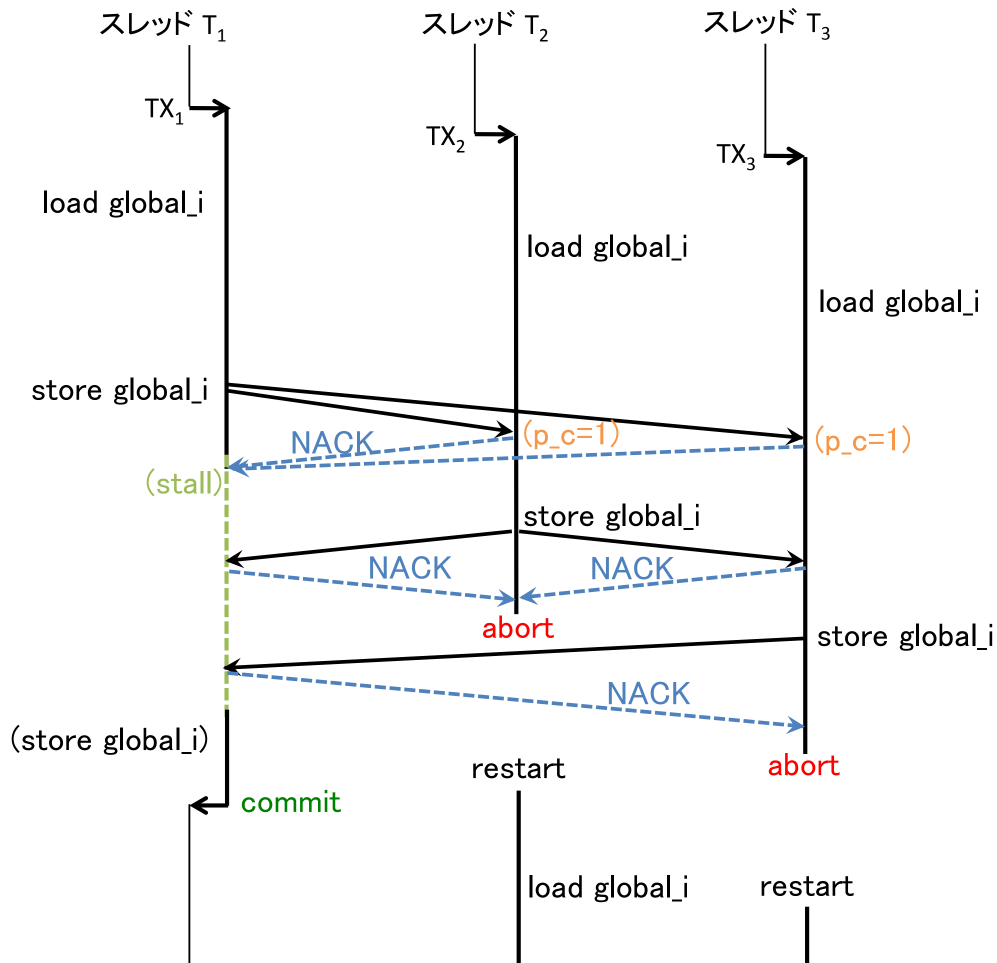


図 3.2: 競合が頻発するトランザクション

を実行する．続いて，TX_1 が global_i のストアを行おうとすると，TX_2 と TX_3 は既にこれをロードしているため，TX_1 に対して NACK を返し，TX_1 はストールする．このとき，TX_1 は TX_2 と TX_3 よりもトランザクション実行開始時刻が早いため，TX_2 と TX_3 は possible_cycle フラグをセットする．

続いて，TX_2 が global_i のストアを行おうとすると，TX_1 と TX_3 は既にこれをロードしているため，TX_2 に対して NACK を返す．このとき，TX_2 よりもトランザクション実行開始時刻が早い TX_1 から NACK が返されるため，possible_cycle フラグがセットされている TX_2 はアボートされる．

同様に，TX_3 が global_i のストアを行おうとすると，TX_1 は既にこれをロードしているため，TX_3 に対して NACK を返す．このとき，TX_3 よりもトランザ

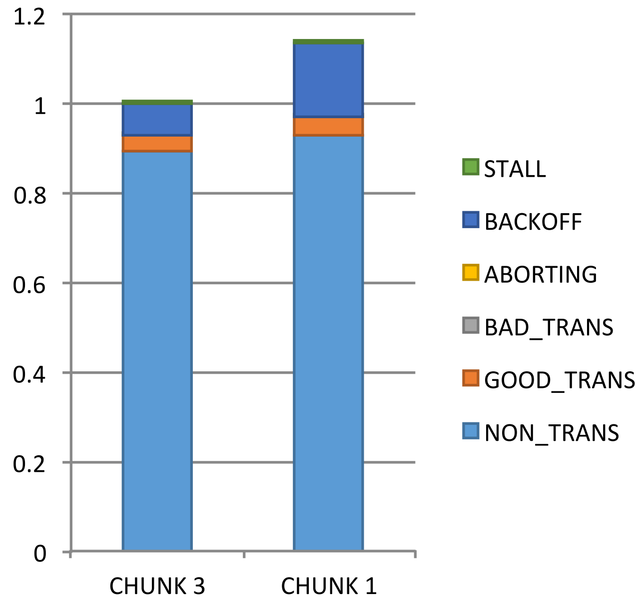


図 3.3: CHUNK を 3 または 1 としたときの kmeans での実行サイクル数の比率

クション実行開始時刻が早い TX_1 から NACK が返されるため，possible_cycle フラグがセットされている TX_3 はアボートされる．

以上の動作により競合が解決された TX_1 は，再びストアを行い，その後トランザクションをコミットする．アボートされた TX_2 と TX_3 は，ランダム時間待機した後で再実行され，再び先程のような競合を起こすことになる．結果，global_i を操作するこのトランザクションでは，並列性を引き出すためにトランザクションを利用したにもかかわらず，逐次実行よりも実行時間が増えることとなる．図 3.2 では 3 スレッドでの実行であったが，スレッド数が増えるほど，実行に時間がかかることがわかる．

3.2.3 実装の問題点

本アプリケーションでは，図 3.1(b) で示したように，CHUNK の初期値は 3 と定義されている．CHUNK の値が小さいほど，CHUNK に対する各スレッドのアクセスは増加し，競合が起きやすくなると考えられる．

図 3.3 に CHUNK を 3 または 1 としたときの kmeans での実行サイクルの比率を示す．評価環境については 5.1 節で説明するものと同様で，手法は LogTM-SE を用いており，スレッド数は 16 である．グラフの縦軸は CHUNK を 3 と

したときにかかった総サイクル数を1としたときの，総実行サイクル数の比率を表す．凡例は5.2節で説明している．ここで注目するのはBACKOFFである．これは，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を示している．図3.3ではCHUNKが1の方がBACKOFFが大きい．よって，このグラフからはCHUNKが小さいほど競合が起きやすくなっていることがわかる．

実際にK-meansのプログラムを作成する場合には，CHUNKを用意するような実装は不自然である．計算する要素数は確定しているので，スレッド数であらかじめ分割しておくことで変数global_iはそもそも必要なくなる．また，各クラスタの重心を計算する部分と計算する要素を割り当てる部分とは1つのトランザクションにまとめる実装にすることも考えられる．このときにglobal_iを利用したとしても，global_iに対するアクセスは分散され，競合の発生が減少すると考えられる．

以上より，本アプリケーションではベンチマークとしての性質上，あえて競合を起こりやすくした不自然な実装であるといえる．

3.3 vacation

vacationは，オンライン予約システムを想定したアプリケーションである．データベースのテーブル操作にトランザクションを使用している．

本アプリケーション内のトランザクションの中で実行される回数が最も多いトランザクションでは，予約対象の空き数とその金額を検索し，その後必要に応じて顧客データを追加し，予約を作成して予約対象の空き数を1減らすという操作を行なっている．予約対象の空き数とその金額の検索ではリード・アクセスのみが行われ，ライト・アクセスはトランザクションの後に集中している．

本アプリケーションでは，使用するテーブルを赤黒木で実装している．赤黒木では，データ挿入時に木を回転させるため，ライト・アクセスが頻発し，競合が多発する原因となる．通常のデータベースシステムでよく用いられるB木等で実装すれば，ライト・アクセスの頻度は減少すると考えられるので，kmeansと同様，あえて競合を発生しやすくさせるための不自然な実装であるといえる．

3.4 genome

genomeは，長いDNA配列から短い文字列segmentをランダムにコピーし，segmentをすべて含んだ上で長さが最短となるDNA配列を推定するアプリケー

ションである．生成した segment の重複を取り除くためにハッシュテーブルを用いる部分などをトランザクション中で実行する．各 segment は，スレッドごとに均等に割り当てられるが，ハッシュテーブルへの挿入時には segment を 12 個ずつに分割して 1 つのトランザクションとしてまとめている．これは，3.2.3 項で説明した kmeans における CHUNK と同様，意図的にトランザクションの競合を発生しやすくした実装である．

以上より，STAMP の各アプリケーションでは競合をあえて発生させたり，プログラマの負担を増やすような実装がされており，実用的なベンチマークという前提に則さない点があるといえる．

第4章　提案手法

3.2.2 項で指摘したように，トランザクション中では短期的に同一アドレスへリード・ライト・アクセスを行うケースがあり，そのような箇所では競合が頻発しやすくなる．同時多発的に複数のスレッドが変数にアクセスしたとしても，リード・ライト・アクセスを完了できるのは一スレッドに限られ，他のスレッドは全てストールまたはアボートされる．これを繰り返すことにより，逐次実行した場合よりアクセスの効率が悪くなると考えられる．

本章では，キューを用いて競合頻発トランザクションを逐次実行する手法を提案する．

4.1　アプローチ

共有メモリ型の並列プログラミングでは，同時多発的に行われる共有メモリへのアクセスを制御するために，ミューテックス，セマフォなどといった同期プリミティブが存在する．だが，同期プリミティブはオーバヘッドを引き起こす原因となる．このようなオーバヘッドを削減し，効率的にロックを獲得し制御する手法として，QOLB(Queue on LockBit)[1][1] がある．QOLB では，ロックの獲得にキューを用いる．

本手法では，QOLB と同様の発想でトランザクション中での競合頻発箇所へのアクセスをキューを用いて逐次実行させることで，アクセスの効率を向上させる．対象とする競合頻発アドレスは，短期的にリード・ライト・アクセスを行うものとする．

[1]初出時は QOSB（Queue on SyncBit）[6].

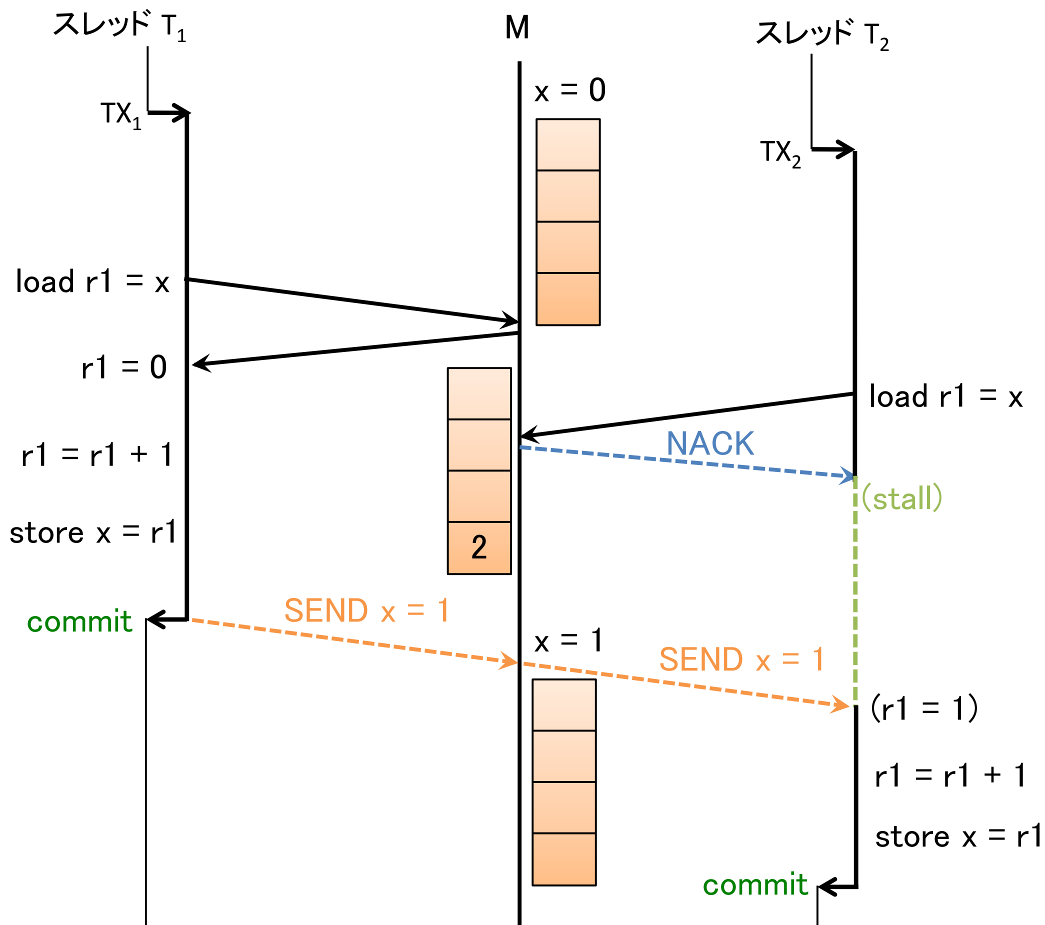


図 4.1: コアとメモリ間の通信

4.2　キューを用いたアクセス管理

4.2.1　コアとメモリ間の通信

提案手法では，共有メモリ上にキューを用意する．このキューは，競合が頻発するアドレス1つに対して1つずつ用意され，そのアドレスへのアクセスを待機しているスレッドを持つコアの番号を記録する．

該当アドレスの値を現在保持しているスレッドがなければ，リード・アクセスしたスレッドは，値を得ることができる．該当アドレスの値を保持しているスレッドがある場合，共有メモリはリード・リクエストを送信してきたスレッドを持つコアの番号をキューに追加し，そのスレッドに対して NACK を返す．NACK を受け取ったスレッドはストールすることになる．通常，ストール時に

は競合が解決されるまでリクエストを送信し続けるが，提案手法では，キューに追加されたスレッドはリクエストの再送を行わない．該当アドレスの値を保持しているスレッド上のトランザクションをコミットするとき，そのアドレスの値を共有メモリおよびキューの先頭に記録されているコア上のスレッドに送信し，キューの先頭要素を削除する．

コア数がnの場合，キューに追加されるコアの数は最大で$n-1$であるため，キューのエントリ数は$n-1$あればよい．

キューに追加されているコアでは，そのコアで実行されているトランザクションはその時点でストールしている．つまり，キューに追加されている状態でコミットされることはない．したがって，コミット時に該当変数以外のキューを参照する必要はない．

図 4.1 に，提案手法におけるコアとメモリ間の通信の様子を示す．図 4.1 では，コア 1 で動作しているスレッドT_1，コア 2 で動作しているスレッドT_2と共有メモリの間での通信を矢印で図示している．縦軸方向が時間の進行を表す．TX_1で変数xに対するリード・アクセスを行った後に，TX_2もリード・アクセスを行なう．ここで，変数xは現在TX_1が保持しているので，TX_2にはNACK が返され，変数xのキューにはコア番号 2 が記録される．

TX_1がコミットされると，共有メモリにxの値が送信されるとともに，キューの先頭に追加されているコア番号を見て，TX_2にもxの値が送信され，TX_2の実行が再開される．このとき，キューの先頭要素は削除される．

以降の項では，簡略化のため，共有メモリとの通信は図から省略する．

4.2.2 競合頻発箇所での実行例

本項では，競合が頻発するアドレスを含むトランザクションを，2.3 節で説明した possible_cycle フラグを用いて競合を行うトランザクショナル・メモリと，提案手法とで実行した場合の例を比較する．

図 4.2 に，possible_cycle フラグを用いて競合解決を行うトランザクショナル・メモリを利用したときの競合頻発アドレスへのアクセスの様子を示す．この図では，変数 A に対してのアクセスで競合が頻発している．スレッドT_1からT_3でトランザクションTX_1からTX_3がそれぞれ実行されている．まず，TX_1からTX_3までが変数 A のロードを実行する．続いて，TX_1が A のストアを行おうとすると，TX_2とTX_3は既にこれをロードしているため，TX_1に対してNACK を返し，TX_1はストールする．このとき，TX_1はTX_2とTX_3よりもトランザクション実行開始時刻が早いため，TX_2とTX_3は possible_cycle フラグをセットする．

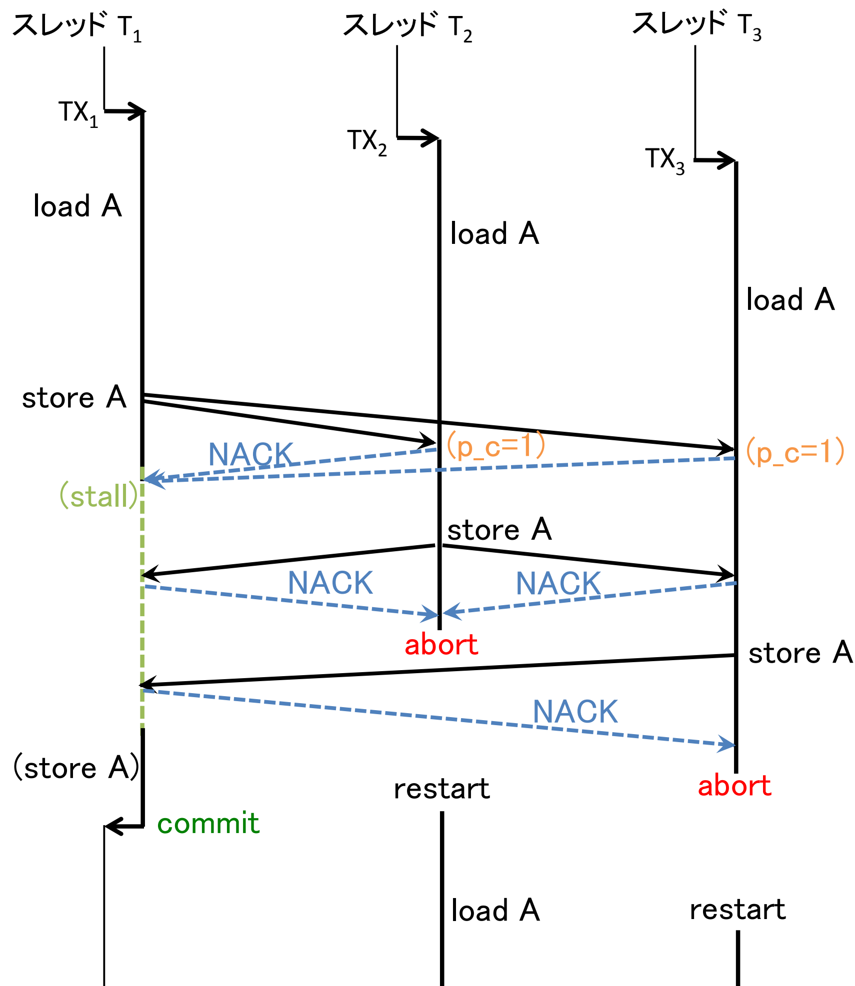


図 4.2: 競合が頻発する例

続いて，TX_2 が A のストアを行おうとすると，TX_1 と TX_3 は既にこれをロードしているため，TX_2 に対して NACK を返す．このとき，TX_2 よりもトランザクション実行開始時刻が早い TX_1 から NACK が返されるため，possible_cycle フラグがセットされている TX_2 はアボートされる．

同様に，TX_3 が A のストアを行おうとすると，TX_1 は既にこれをロードしているため，TX_3 に対して NACK を返す．このとき，TX_3 よりもトランザクション実行開始時刻が早い TX_1 から NACK が返されるため，possible_cycle フラグがセットされている TX_3 はアボートされる．

以上の動作により競合が解決された TX_1 は，再びストアを行い，その後トランザクションをコミットする．アボートされた TX_2 と TX_3 は，ランダム時

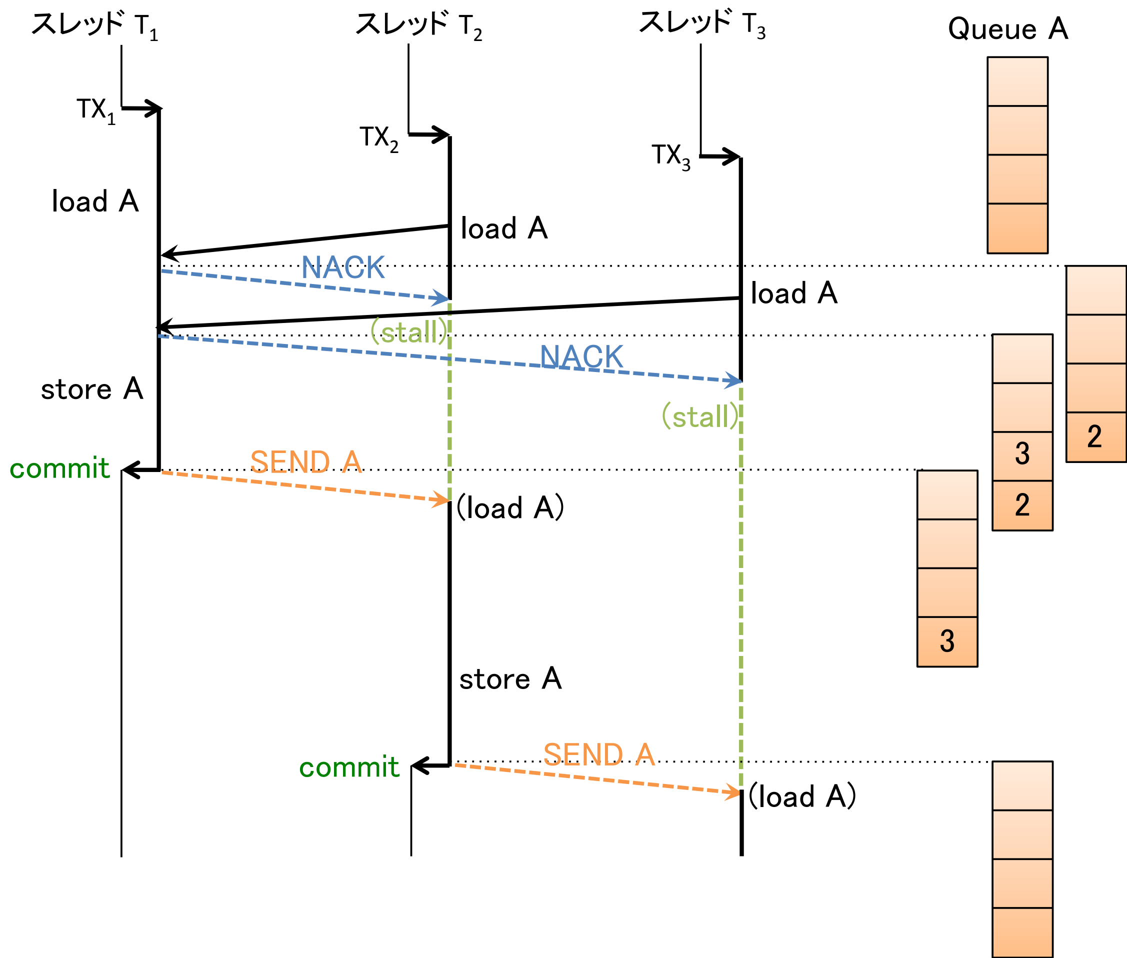


図 4.3: 提案手法でのトランザクションの実行の様子

間待機した後で再実行され，再び先程のような競合を起こすことになる．

図 4.3 に，提案手法での競合頻発アドレスへのアクセスの様子を示す．図の右側のキューは，点線部の時点での状態をそれぞれ表している．まず，TX_1 が A をロードする．続いて，TX_2 が A のロードを行おうとすると，A は現在 TX_1 がアクセスしているため，NACK が返されて TX_2 はストールし，コア番号 2 がキューに追加される．TX_3 でも同様のことが起こり，キューにコア番号 3 が追加される．

TX_1 が A のストアを終え，コミットするときに，キューの先頭に追加されている番号を見て，TX_2 に対して A の値を送信する．これにより，TX_2 はストール状態から復帰し，キューの先頭要素は削除される．TX_2 が A のストアを終え，

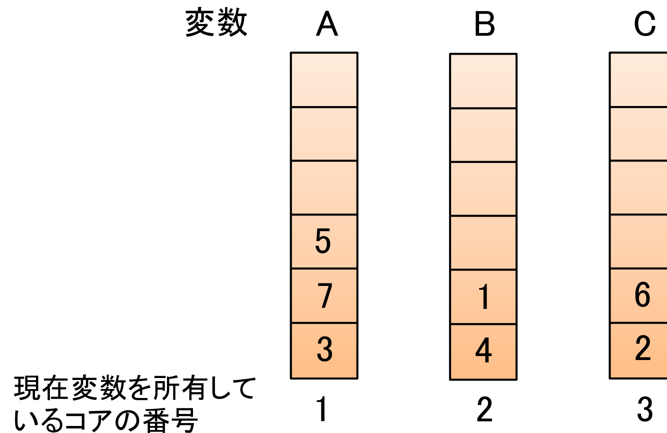


図 4.4: キューで生じるデッドロックの例

コミットするときに，キューの先頭に追加されている番号を見て，TX_3 に対してAの値を送信する．これにより，TX_3 はストール状態から復帰し，キューの先頭要素は削除される．

以上のように，提案手法では，キューを利用することで競合トランザクションを逐次実行し，競合頻発アドレスへのアクセスの効率を向上させる．

4.2.3 キューのデッドロック

1つの変数に対してキューが1つ用意されるため，競合頻発アドレスが複数ある場合には複数のキューが存在することになる．2.3 節でトランザクションがデッドロックを起こす例を説明したが，キューにも同様のデッドロックが生じる．

図 4.4 に，キューのデッドロックが生じる例を示す．キューの上に表記されているのは，そのキューを持つ変数である．この図では A, B, C の3つの変数がキューを持つ．キューの下に表記されているのは，現在その変数を所有しているコアの番号である．この番号は，トランザクションがコミットされるときに，キューの先頭要素に置き換えられる．

コア1からコア3の上で実行中のスレッドはそれぞれ変数 A, B, C を現在所有している．この状態でトランザクションの実行を続けたところ，コア1上のスレッドが変数Bにアクセスしようとしたところで，変数Bのキューに追加され，トランザクションはストールする．同様に，コア2上のスレッドは変数Cに，コア1上のスレッドは変数Aにアクセスしようとしたところで，キューに

追加され，トランザクションはストールする．コア1，2，3のいずれもストールすることになるが，変数A, B, Cのキューは，これらのコア上で実行されているトランザクションがコミットされない限り，キューの状態は変わらない．こうしてデッドロック状態に陥る．

このデッドロックを回避するために，タイムアウトを設ける必要がある．

第5章　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，SLICC(Specification Language including Cache Coherence) を用いて，メモリの階層構造とキャッシュ・コヒーレンス・プロトコルを記述している，また，Ruby モジュールを用いて LogTM-SE のメモリ・シミュレーションを実行する．各パラメータは表 5.1 の通りである．

表 5.1: 評価パラメータ

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

評価対象として，STAMP の kmeans を用い，8 スレッドと 16 スレッドで実行し評価を行う．提案手法と LogTM-SE についてベンチマーク実行時のサイクル数を計測し，比較を行う．

3.2.1 項で説明したように，重心を計算するときに，各スレッドに対して計算する要素を一定の個数ずつ割り当てる操作があり，その個数を CHUNK としている．今回は CHUNK の値を 1 または 3 として評価を行う．

5.2 評価結果

STAMP の kmeans を 8 スレッドと 16 スレッドで実行したときの評価結果のグラフを図 5.1 と図 5.2 に示す．図 5.1 は CHUNK が 3 のとき，図 5.2 は CHUNK が 1 のときのグラフである．図のグラフの縦軸は LogTM-SE で評価対象を実行したときにかかった総サイクル数を 1 としたときの，提案手法での総実行サイクル数の比率を表す．

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．

- NON_TRANS: トランザクション外で要した実行サイクル数
- 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
- 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
- STALL: ストールに要したサイクル数

CHUNK が 3 の場合，提案手法では LogTM-SE の場合と比較して，8 スレッドでは 6.2％，16 スレッドでは 3.5％，総実行サイクル数が削減されている．BACKOFF に着目すると，提案手法では LogTM-SE の場合と比較して，8 スレッドでは 1000 分の 1 以下にまで減少しており，16 スレッドでは 59.5％のサイクル数が削減されている．CHUNK が 1 の場合，提案手法では LogTM-SE の場合と比較して，8 スレッドでは 3.0％，16 スレッドでは 13.6％，総実行サイクル数が削減されている．BACKOFF に着目すると，提案手法では LogTM-SE の場合と比較して，8 スレッドでは 99.2％，16 スレッドでは 86.3％のサイクル数が削減されている．

5.3 考察

CHUNK が 3 の場合も 1 の場合も，いずれのスレッド数でも，LogTM-SE と比較したときの提案手法での BACKOFF の割合が減少している．これは，提案手法を用いることにより，競合にともなうアボートの数が減少したためであるといえる．

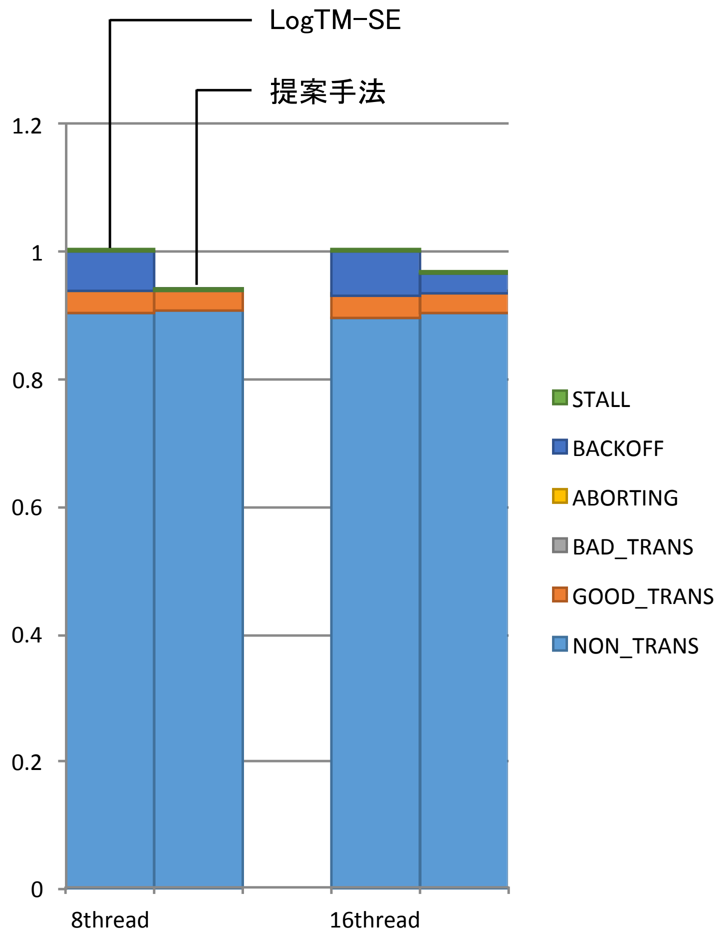


図 5.1: CHUNK が 3 のときの評価結果

CHUNK が 3 の場合では，16 スレッドのときより 8 スレッドの方が，LogTM-SE と比較したときの提案手法での総サイクル数削減率が大きい．スレッド数が増えることにより，キューを用いてアクセスを行う競合頻発アドレスを含まないトランザクションでの競合およびアボートが増える．このため，16 スレッドでは提案手法の適用がされていないトランザクションでの BACKOFF が増加し，8 スレッドのときほど性能が良くならなかったと考えられる．実際，kmeans 内のトランザクションでは，キューを用いてアクセスを行う競合頻発アドレスを含まないトランザクションの方が，キューを用いてアクセスを行うトランザクションよりもトランザクション長が大きい．スレッド数が増えることで，前者のトランザクションの影響が大きくなったと考えられる．

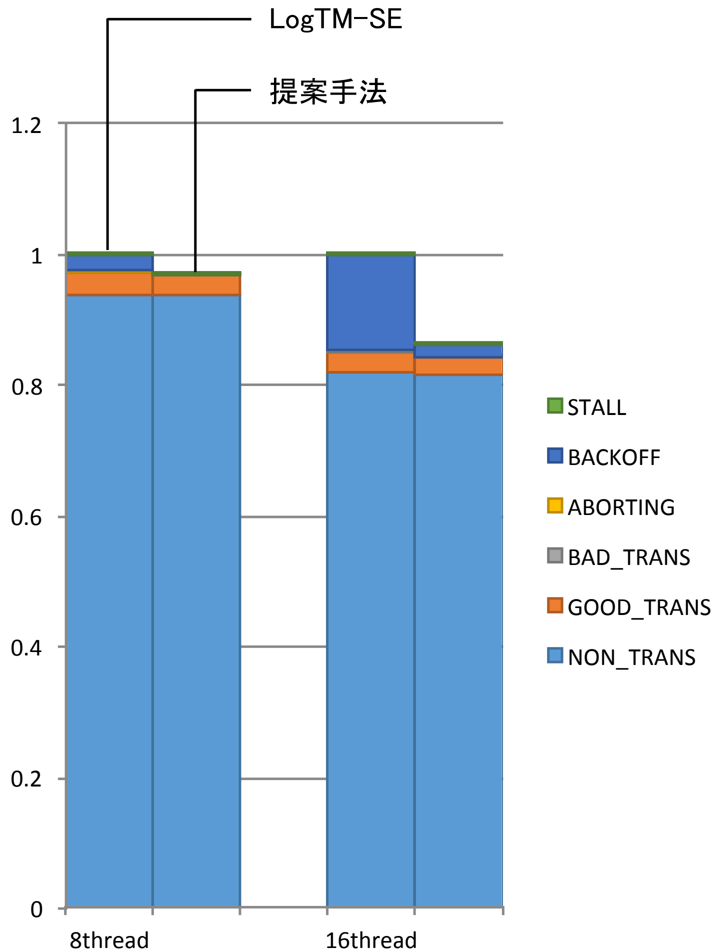


図 5.2: CHUNK が 1 のときの評価結果

CHUNK が 1 の場合では，8 スレッドのときより 16 スレッドの方が，LogTM-SE と比較したときの提案手法での総サイクル数削減率が大きい．CHUNK が 1 になることで，競合頻発アドレスへのアクセス回数が増え，アクセスの間隔も短くなる．これにより，スレッド数が増えるほど競合が頻発し，アボート回数の増加にともない BACKOFF が増加したと考えられる．

続いて，CHUNK が 1 の場合と 3 の場合との比較を行う．CHUNK が 1 の場合，3.2.3 項で説明したように，CHUNK が 3 のときより競合が増えると考えられる．8 スレッドの場合では，特に LogTM-SE について，CHUNK が 3 のときよりも BACKOFF の割合が小さくなっている．これは，CHUNK 数を小さくしたことで各スレッドでのトランザクションの実行のタイミングがずれ，う

まい具合にアボートが起きにくくなった結果であると考えられる．一方，16 スレッドの場合は，CHUNK が 3 のときよりも BACKOFF の割合が大きくなっている．スレッド数が増えることで競合が起きやすくなり，8 スレッドのときのようにうまい具合にトランザクションの実行のタイミングがずれることもなく，アボートが増加したためであると考えられる．

第6章　おわりに

本論文では，従来のトランザクショナル・メモリの研究での問題点とトランザクショナル・メモリの実用的なアプリケーションでの問題点を指摘し，複数のスレッドがトランザクション中で短期的に同一アドレスへのリード・ライト・アクセスが繰り返されることが原因で競合が頻発する部分に対して，キューを用意することでアクセスを逐次的に実行させる手法を提案した．

今後の課題として，短期的にリード・ライト・アクセスが繰り返されるアドレスを検出する方法の検討があげられる．また，短期的にリード・ライト・アクセスが繰り返されるアドレスへのアクセスが粒度の大きなトランザクションの前半に存在するような場合に，本論文での提案手法ではトランザクショナル・メモリの恩恵が得られにくいと考えられる．そのようなケースに対して，トランザクションの並列度をできるだけ下げない手法の検討も今後の課題である．

参考文献

[1] Nagi M. Aboulenein, Stein Gjessing, James R. Goodman, and Philip J. Woest. Hardware support for synchronization in the scalable coherent interface (sci). In *Proc. of the Eighth International Parallel Processing Symposium*, 1992.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In *Proc. of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[3] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of the IEEE International Symposium on Workload Characterization*, 2008.

[4] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. of the 33rd Annual International Symposium on Computer Architecture*, 2006.

[5] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proc. of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[6] James R. Goodman, Mary K. Vernon, and Philip J. Wwst. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proc. of the Third International Conference on Architectual Support for Programming Language and Operating Systems*, 1989.

[7] Lance Hammond, Peter G. Gyarmati, Christos Kozyrakis, Kunle Olukotun, Brian D. Carlstrom, Ben Hertzberg, Vicky Wong, Mike Chen, John D. Davis, Manohar K. Prabhu, and Honggo Wijaya. Transactional

memory coherence and consistency. In *Proc. of the 11th International Symposium on Computer Architecture*, 2004.

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

[9] Marc Lupon, Grigorios Magklis, and Antonio González. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proc. of the 18th International Conference on ParallelArchitectures and Compilation Techniques*, 2009.

[10] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based Transactional Memory. In *Proc. of the IEEE 12th International Symposium on High-Performance Computer Architecture*, 2006.

[11] Rajwar Ravi, Herlihy Maurice, and Lai Konrad. Virtualizing Transactional Memory. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, 2005.

[12] 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 *Proc. of the IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[13] Lihang Zhao, Woojin Choi, and Jeff Draper. SEL-TM: Selective eager-lazy management for improved concurrency in transactional memory. In *Proc. of the 26th International Parallel and Distributed Processing Symposium*, 2012.

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をいただきました．また，五島正裕准教授には，研究テーマの決定や本研究の進行など幅広いご指導をいただきました．

事務補佐員の八木原春水さん，長谷部環さんには，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。