アボートの抑制を目的とするスレッド制御を用いたトランザクションへの資源割り当て

葛木　優太[†]　伊藤　竜一[††]　中園　翔[†††]　佐々木勇和[††]　鬼塚　真[††]

† 大阪大学工学部　〒565-–0871 大阪府吹田市山田丘 1-5
†† 大阪大学情報科学研究科　〒565-–0871 大阪府吹田市山田丘 1-5
††† NTT ソフトウェアイノベーションセンタ　〒180–8585 東京都武蔵野市緑町 3-9-11
E-mail: †{katsuragi.yuta,ito.ryuichi,sasaki,onizuka}@ist.osaka-u.ac.jp, ††syou.nakazono.nu@hco.ntt.co.jp

あらまし　インメモリデータベースにおいて，集中的なアクセスはトランザクションのアボートを多発させ，スループットの向上を妨げていることが近年の研究により示されている．このような背景を受け，アボートの削減を目的として機械学習を用いてアボート予測を行いトランザクションを処理するスレッドを動的に決定する手法が提案されている．この既存手法を使用することでアボートの削減を達成することが可能であるが，アクセスの集中度合いによらず一定の数のトランザクション処理スレッドを使用するため，集中的なアクセスが起こる状況においてはアボートが多発しスループットが悪化する場合が存在するという課題や，トランザクションを各スレッドに割り当てる際の計算コストが高いという課題が存在している．そこで，本稿では既存手法でのアボート予測を用いるコストを削減し，さらにアクセスの集中度合いを考慮するトランザクションのスレッドへの割り当て手法を提案する．提案手法では，アボート予測を適用するコストの削減のために共有データへのアクセス回数を削減し，アクセスの集中度合いを考慮するためにデータベース内のトランザクション処理スレッドの数を動的に制御する．TPC-C ベンチマークを用いた評価により提案手法がワークロードのアクセスの集中度合いによらず，アボート予測のみを行う既存手法および現在広く使用されているランダムスケジューリングに対して同等以上の性能を示すことを確認した．

キーワード　トランザクション処理，スケジューリング

1　は じ め に

近年のハードウェア技術の進歩により，メモリは大容量化が進み，CPU はより高い並列度を提供している．メモリの大容量化が進むことでデータベース全体をメモリ上に保持することが可能になり，インメモリデータベースの研究が盛んに行われている．インメモリデータベースはデータベース全体がメモリ上に存在することが前提とされ，ディスクベースのデータベースでは高コストとされているバッファプールやロックテーブルといった機能を排除 [1,2] することでトランザクション処理において高速な動作を実現する．

高速な処理を実現するインメモリデータベースであるが，近年の研究によって多数のトランザクションが同一のレコードへアクセスするといった集中的なアクセスが起こる場合にはアボートが多発して性能が上昇しない，もしくは性能が低下していくことがあると指摘されている [3]．ハードウェアの並列度が向上することは，データベース中の多数のスレッドが並列に同一のデータにアクセスしやすくなることを意味し，集中的なアクセスを引き起こす．これからも並列度が上がることが期待されるハードウェアを有効に活用するためには集中的なアクセスを引き起こす場合のトランザクション処理の性能を高めることが重要な課題である．

トランザクション処理の性能向上に関する研究は数多く存在する．[4–7] は並行実行制御を改良して，より多くの並行実行パターンを処理可能にし，アボートを削減することを目指している．一方，[8] では，並行実行制御法ではなく，トランザクションのスレッドへの割り当てについて注目している．[8] では多くのトランザクション処理システムがトランザクションをランダムにスレッドに割り当ているが，アボートを削減してスループットを向上させるためにはトランザクションのスレッドへの割り当て方が重要と指摘し，機械学習を用いたスケジューラでアボート予測を行いトランザクションをスレッドに割り当てる手法を提案している．この手法では，アクセスが集中的でない状況においてはアボートを削減しスループットの向上がみられる場合がある．しかし，高並列な環境でアクセスが集中的な状況においてはアボートが多発しスループットが低下する問題が依然として生じる．また，トランザクションをスレッドに割り当てるための計算コストが高いことも問題である．

そこで本稿では，アボート予測を行い，かつアクセスの集中度合いを考慮する効率的なトランザクションのスレッドへの割り当て手法を提案する．提案手法は 2 つの特徴を有する．1 つめの特徴はアボート予測を行う計算コストを削減することを目的として，アボート予測を行いトランザクションをスレッドに割り当てる際の共有データへのアクセス回数を削減することである．2 つめの特徴はアクセスの集中度を考慮して適切な並列度を保つために，データベース中のトランザクション処理スレッドの数を動的に制御することである．このとき，スルー

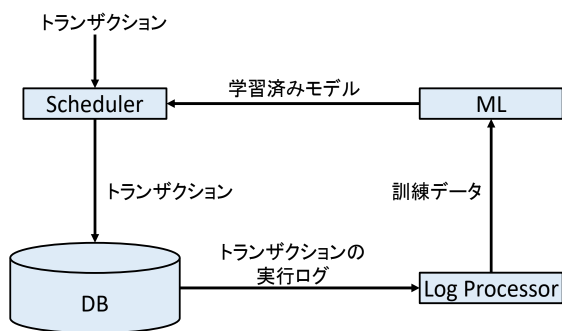

図 1　システム図 (図中の矢印に処理されるデータの名称を示す)

プットが最大となるスレッド数を探索することで，どのようなアクセスの集中度においても高いスループットが得られるように制御を行う．

本稿の構成は次の通りである．2 章で事前知識について説明し，3 章で提案手法の詳細を示し，4 章で評価実験及びその結果に関して述べる．5 章では関連研究について述べ，6 章でまとめと今後の課題について述べる．

2　事 前 知 識

この章では，2.1 節で本稿のシステムアーキテクチャ，2.2 節で本稿で使用するデータベースのアーキテクチャ，2.3 節 でデータベースが採用する並行実行制御法，2.4 節 で機械学習を用いたトランザクションのスレッドへの割り当てについて述べる．

2.1　システムのアーキテクチャ

図 1 に本稿のシステム図を示す．本システムは大きく次の 4 つの要素からなる．

（1）　トランザクションを処理するデータベース (図中 DB)

（2）　トランザクションの実行結果を収集し処理する機能 (図中 DB，Log Processor)

（3）　実行ログからアボート予測のためのモデルを学習する機能 (図中 ML)

（4）　学習済みモデルを用いてトランザクションをスレッドへ割り当てる機能 (図中 Scheduler)

(1) に関しては次の 2.2 節，2.3 節で，(2)，(3)，(4) に関しては 2.4 節で詳細に述べる．

2.2　使用するデータベースのアーキテクチャ (図中 DB)

本節では，本稿で使用するデータベースのアーキテクチャについて述べる．本稿で使用するデータベースはインメモリのデータベースであり，マルチスレッドで動作し，ストアドプロシージャのみサポートする．SQL はサポートしていない．データベースは Shared Everything [9] アーキテクチャを採用しており，トランザクションは任意のスレッドで実行され，全てのデータベースのデータにアクセスすることが可能である．加えて，各トランザクション処理スレッドは 決められた物理コア上で動作する．データベース中のトランザクション処理スレッドには各スレッドごとに固有のキューが存在しており，トランザクション処理スレッドはそのキューからトランザクションを取りだして実行する．本稿ではランキューの実装として，Boost [10] の boost::lockfree::queue を使用する．以下，本稿ではデータベース内のトランザクション処理スレッドの数を n とする．

2.3　使用する並行実行制御法 (図中 DB)

本節では，本稿で使用するデータベースが採用する並行実行制御法について述べる．データベースシステムにおいて並行実行制御法とは，複数のトランザクションがデータベース内で並行に実行された際に，全てのトランザクションが ACID 特性を満たすことを保証するための制御方法である．本稿ではアボートを削減することが性能に与える影響を調べることを目的とするため，アボートが多く発生する並行実行制御法である TimestampOrdering [11] を採用する．TimestampOrdering ではデータベースに投入された順に従ってトランザクションに一意で単調に増加する timestamp が付与され，トランザクションの実行順序がこの timestamp の順序に一致するように制御される．各レコードでは，そのレコードを最後に読んだトランザクションの timestamp ($maxRTS$)，そのレコードに対して最後に書き込みを行ったトランザクションの timestamp ($maxWTS$)，及び，現在あるトランザクションがそのレコードに対して書き込みを試みていることを示す write lock を管理する．

2.4　機械学習を用いたトランザクションのスレッド割り当て

本節では，2.4.1 項で機械学習を用いたアボート予測モデルを構築するために必要なシステムの機能，2.4.2 項で [8] でのロジスティック回帰によるアボート予測モデルを用いたトランザクションのスレッドへの割り当て方法，2.4.3 項で学習に用いる特徴ベクトル，2.4.4 項で学習のための訓練データの作成方法ついて述べる．

2.4.1　機械学習を行うために必要なシステムの機能 (図中 DB，Log Processor)

本項では，機械学習に用いる訓練データを作成するために必要となるシステムの機能について述べる．アボート予測を行うモデルを学習するために用いる訓練データはトランザクションのコミットを表すデータと，アボートを表すデータからなる．コミットを表すデータは並行に実行されて共にコミットされたトランザクションのペアの情報，アボートを表すデータはアボートされたトランザクションとその原因となったトランザクションのペアの情報から作成する．本システムではコミットされたトランザクションに関して，そのトランザクションと並行に実行されてコミットされたトランザクションが存在していればそのうちの一つを特定する機能を実装した．また，アボートされたトランザクションに対して，その原因となったトランザクションを特定する機能を実装した．アボートの原因となるトランザクションの特定方法はデーベースが採用する並行実行制御法に依存する．本稿で使用した TimestampOrdering ではト

ランザクション Tx がレコードへの読み書きを行う場合，次のうちいずれかの条件によりアボートされる．

- Tx の timestamp $< maxWTS$ であるレコードに対して読み込みを行う
- Tx の timestamp $< maxRTS$ であるレコードに対して書き込みを行う
- write lock を保持しているレコードに対して書き込みを行う

本システムではアボートされたときの，$maxRTS$ や $maxWTS$ の timestamp を持つトランザクションをアボート原因のトランザクションとして扱う．write lock によりアボートされた場合は，その時点での $maxWTS$ の timestamp を持つトランザクションをアボートの原因のトランザクションとして扱う．

2.4.2 機械学習を用いたスレッド割り当て (図中 Scheduler)

本項では [8] での機械学習を用いたトランザクションのスレッドへの割り当て手法である Balanced Vector Scheduling について述べる．Balanced Vector Scheduling ではロジスティック回帰でトランザクションのスレッドへの割り当てを行う．学習済みのモデルを M とする． 2 つのトランザクション Tx_1，Tx_2 が与えられたとき，$M(Tx_1, Tx_2)$ は 0 以上かつ 1 以下の実数値を返す．以降ではこの値を 「Tx_1 と Tx_2 が並行に処理されるときにどちらかがアボートされる確率」を表すものと考える．トランザクション T_{new} を 1 つのスレッドに割り当てることを行う．割り当ての方針は T_{new} とアボートする確率が高いトランザクションが存在しているランキューに T_{new} をプッシュすることである．このようにして，アボートを起こす確率が高いトランザクション同士が並行に動作することを防ぐ．具体的には，各ランキューを代表する特徴ベクトル $R_{avg}[i]$ を用意し，すべてのランキューを探索して $M(T_{new}, R_{avg}[i])$ の値が最大となるキュー i に T_{new} をプッシュする．$R_{avg}[i]$ はランキュー i にプッシュされたトランザクションの特徴ベクトルの平均である．この各キューを代表する特徴ベクトルは複数スレッドから参照，更新される共有データであるため，アクセスには排他制御が必要となる．本システムではこれらのデータを reader-writer lock を用いて排他制御を行い管理する．

2.4.3 特徴ベクトル

本稿のシステムにおいてトランザクションから特徴ベクトルを構成する方法について述べる．まず，トランザクションの特徴について述べる．トランザクション Tx の特徴は Tx の SQL 表現における，WHERE 句内の条件を文字列とみなしたもの全体からなるとする．例えば，Tx の SQL 表現が

```
SELECT *
FROM DISTRICT
WHERE D_W_ID = 1 AND D_ID = 9
```

であるとき，Tx の特徴は {"D_W_ID = 1","D_ID = 9"} となる．トランザクションの特徴ベクトルは次のように構成される．なお，以下で n_{Tx} はデータベースで実行されるトランザクションの種類の総数とし，トランザクションの種類は 1 から n_{Tx} までの通し番号により識別されるものとする (TPC-C において neworder と payment のみを実行する設定の場合には $n_{Tx} = 2$ となる)．

（1） 特徴ベクトルの長さ $feature_length$ を決定する

（2） 全ての特徴に対して長さ $feature_length - n_{Tx}$ のベクトルを得るために feature hashing を適用する

（3） トランザクションが i 番目の 種類のトランザクションであるとき，特徴ベクトルの $feature_length - i$ 番目の要素を 1 とする

こうして構成される特徴ベクトルは長さ $feature_length$ で各要素が 0 もしくは 1 である．以上を用いて，ロジスティック回帰で使用する特徴ベクトル V をトランザクション Tx_1，Tx_2 から次のようにして構成する．

（1） Tx_1，Tx_2 からそれぞれ長さ $feature_length$ の特徴ベクトル V_1，V_2 を構成する

（2） $V = V_1 \mid V_2 \mid V_1 \& V_2$ (ただし，"|" は特徴ベクトルの連結を表すものとする)

こうして構成された特徴ベクトル V の長さは $3 * feature_length$ となる．

2.4.4 訓練データの作成方法 (図中 Log Processor)

学習を行うための訓練データの作成方法について述べる．V_a をアボートされたトランザクションの特徴ベクトル，$V_{causing}$ をアボートの原因となったトランザクションの特徴ベクトル，V_c をコミットされたトランザクションの特徴ベクトル，$V_{concurrent}$ をコミットされたトランザクションと並行に動作しコミットされたトランザクションの特徴ベクトルであるとする．訓練に用いる特徴ベクトルは次の 2 種類である．

（1） $V = V_a \mid V_{causing} \mid V_a \& V_{causing} \mid 1$ (1 はアボートを示すラベル)

（2） $V = V_c \mid V_{concurrent} \mid V_c \& V_{concurrent} \mid 0$ (0 はコミットを示すラベル)

訓練データを作成する際には，アボートのラベルを持つ特徴ベクトル，コミットのラベルを持つ特徴ベクトルを同数使用する．

3 提案手法

この章では提案する手法について述べる．提案手法は [8] での Balanced Vector Scheduling を改良した手法であり，3.1 節で述べる探索打ち切り，3.2 節で述べるスレッド数制御の 2 つの要素から構成される．提案手法はアクセスの集中度によらず低い abort rate と高いスループットを達成することを目標とする．3.1 節では，2.4.2 項でのトランザクションのスレッド割り当てアルゴリズム (Balanced Vector Scheduling) に対してキューの探索を高速化した手法を提案する．3.2 節では，アクセスが集中する下では使用するトランザクション処理スレッド数を増やすことがスループットの低下につながるという観察に基づき，スループットの低下を防止することを目的としてトランザクション処理スレッド数の制御を行うことを提案する．

3.1 キューの探索の高速化 (探索打ち切り)

本節では，提案手法における探索打ち切りに関して述べる．まず，探索打ち切りの概観を示した後，探索打ち切りのアルゴリズムについて詳細に述べる．

Balanced Vector Scheduling でのスレッド割り当てにおいてはキューの探索にかかるオーバーヘッドがアボートの減少による高速化への影響より大きくなっており，結果としてスループットがランダムスケジューリングと比較して低下する問題がある．このことに注目し，キューの探索にかかるコストの削減を目的としたアルゴリズムを提案する．Balanced Vector Scheduling ではトランザクションを割り当てるために全てのランキューに対して各ランキューを代表する特徴ベクトルにアクセスし，モデルによる評価を行う．しかし，各ランキューを代表する特徴ベクトルは全スレッドの共有データであるためそのアクセスには排他制御が必要であり，アクセスのコストが高い．そこで，提案手法ではトランザクションをスレッドに割り当てる際の共有データへのアクセス回数を削減して，トランザクションの割り当てにかかるコストを削減することでスループットの向上を狙う．2.4.2 項で述べたように，モデルによる評価値は割り当てるトランザクションがキュー内のトランザクションと並行に実行されるときにアボートされる確率を表している．提案手法では，探索時にモデルによる評価値があるしきい値以上であればその時点でアボートが発生すると判断して探索を打ち切る．本稿でのシステムにおいては，しきい値を 0.5 とした．探索打ち切りを行うことにより，探索すべきキューの数および共有データへのアクセス回数を削減することができるため，アボート予測を行うコストが低くなりスループットの向上が期待できる．一方で，探索打ち切りによりアボート予測の精度が低下し，探索打ち切りを行わない場合と比較して高い abort rate を示すことが想定される．

Algorithm 1 に提案する探索アルゴリズムを示す．*idx* は新たに割り当てるトランザクションをプッシュするランキューのインデックスを表す (2 行目)．新たに割り当てるトランザクション T_{new} から特徴ベクトル V_{new} を作成する (3 行目)．V_{new} を用いて各ランキューを探索しモデル M による評価を行う (6 行目)．このとき，モデルによる評価値がしきい値以上である場合 (8 行目) その時点で探索を打ち切り，現在探索しているランキュー i を T_{new} をプッシュするべきランキューとする (9 行目)．モデルによる評価値がしきい値を超えない場合は探索を打ち切らずに続行する．

3.2 スレッド数の制御

この節では提案手法におけるスレッド数の制御に関して述べる．最初に，スレッド数制御の必要性を Balanced Vector Scheduling を用いた事前実験の結果を用いて示す．次に，提案手法でのスレッド数制御の概観について示した後，スレッド数制御のアルゴリズムについて詳細に述べる．

以下でアクセスが集中する状況ではスレッド数を制御する必要があることを示す．アクセスが集中する状況として TPC-C ベンチマーク [12] において warehoue 数を 2 とした場合を採

Algorithm 1 提案するスレッド割り当てアルゴリズム

```
1:  max_prob ← 0
2:  idx ← −1
3:  V_new ← Hash(T_new)
4:  //全てのキューを探索
5:  for i := 0 to n − 1 do
6:      p ← M(V_new, R_avg[i])
7:      //探索打ち切り
8:      if p > threshold then
9:          idx ← i
10:         break
11:     end if
12:     if p ≧ max_prob then
13:         max_prob ← p
14:         idx ← i
15:     end if
16: end for
```

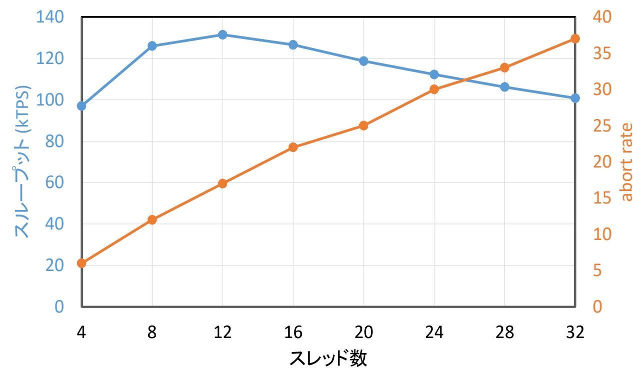

図 2 Balanced Vector Scheduling を用いた場合のスループットと abort rate (warehouse = 2)

用する．この下で Balanced Vector Scheduling を用いてトランザクションのスレッドへの割り当てを行った場合のスループット， abort rate をトランザクション処理スレッドの数を横軸として図 2 に示す．ただし，abort rate はアボート数をコミット数で割ったものとする．図 2 から，12 スレッド以上ではスレッドの数が増えるにつれてスループットが低下していること，abort rate がスレッド数の増加に対して単調に増加していることが分かる．これはスレッドの数が増加するにつれてより多数のトランザクションが同一のレコードへアクセスするようになり，その結果として多くのスレッドでトランザクションのアボートとその再実行のために多くの処理時間が使用されていることによると考えられる．この結果から，アクセスが集中する環境において高いスループットを達成するためには, スレッド数を制御して適切な並列度を保つことが必要であることが分かる．

提案手法でのスレッド数制御の概観について述べる．図 2 から分かるように，アクセスが集中する場合においてはある程度のスレッド数まではスループットが上昇するが，過度なスレッド数はスループットを悪化させる．提案手法でのスレッド数制御のアルゴリズムはこのことを前提として，ワークロードに対

して最適なスレッド数を制御をすること，特にアクセスが集中するワークロードにおいて過度なスレッド数によりスループットが低下することを防ぐことを目的として設計される．提案するスレッド数制御アルゴリズムはスループットの計測 (step 1)，スレッド数を変更 (step 2) の 2 つのステップからなり，この 2 つのステップを繰り返し実行しながらスレッド数をスループットが高くなる方向へ動的に制御する．大まかには，最初のステップではあるスレッド数の下でのスループットを計測し，次のステップではその計測結果に基づいてスレッドの数を変化させる，という動作を行う．このようにスレッド数を制御することで，ワークロードのアクセスの集中度合いに対して最適なスレッド数を保つことを目指す．本稿でのシステムにおいてはスレッド数制御用のスレッドをデータベース内に用意する．

次に提案するスレッド数制御アルゴリズムについて詳細に述べる．2.2 項で述べたように，各トランザクション処理スレッドは固有のランキューを持つ．また，ここでは各トランザクション処理スレッドには固有の id ($0 \leqq \mathrm{id} < n$) が与えられているものとし，各トランザクション処理スレッドはトランザクション処理を行っている状態をアクティブであると表現する．スレッド数制御アルゴリズムを Algorithm 2 に，Algorithm 2 中で使用されている主な記号の定義を表 1 に示す．step 1 ではあるスレッド数の下で，*duration* (msec) 間のスループットを計測する (9 〜 18 行目，本システムでは *duration* を 2000 とした)．そして， step 2 では step 1 で計測したスループットをもとにスレッド数を制御する．具体的には，前のループでのスレッド数の変更によりスループットが増加していれば 更にスレッド数を増加させ (23 行目)，スループットが低下していれば スレッド数を減少させる (27 行目)．ただし，スレッド数を増加させることでスループットが低下した回数 *mistake* に応じてスループットを計測する間隔を長くし，頻繁にスレッド数を変化させないようにする (12 行目)．また，少しのスループットの低下により頻繁にスレッド数が変化することを防ぐことを目的として，スループットの低下率が *error* 以内であればスレッド数を変化させないようにする (20 行目)．本システムでは *error* は 0.02 とした．step 2 において非アクティブとなったスレッドが存在すれば，そのスレッドのランキューが空になるまで待機し (32 行目) 次のループでのスループットの計測に非アクティブとなったスレッドの影響が及ばないようにする．

表 1　Algorithm 2 で使用されている主な記号の定義

active_thread	現在アクティブなトランザクション処理スレッドの数
commit[*i*]	*i* 番目のスレッドのコミット数
duration	スレッドが sleep する単位期間
isactive[*i*]	*i* 番目のスレッドがアクティブかどうか
mistake	スレッド数を増やして TPS が減少した回数
error	step 2 で許容するスループットの低下率の下限
to_clear	直前のステップで非アクティブとなったスレッドが存在

Algorithm 2 スレッド数制御アルゴリズム

```
1: active_thread ← 4 //探索は 4 スレッドから開始
2: TPS_prev ← 0
3: TPS_now ← 0
4: to_clear ← false
5: mistake ← 1
6: while experiment do
7:   //step1
8:   begin ← current_time
9:   for i := 0 to active_thread − 1 do
10:    commit_prev ← commit_prev + commit[i]
11:  end for
12:  sleep_for(mistake ∗ duration)
13:  for i := 0 to active_thread − 1 do
14:    commit_now ← commit_now + commit[i]
15:  end for
16:  end ← current_time
17:  time_difference ← end − begin
18:  TPS_now ← (commit_now−commit_prev)/time_difference
19:  //step2
20:  if 1 − error < TPS_now/TPS_prev < 1 then
21:    do nothing
22:  else if TPS_now ≧ TPS_prev && active_thread < n
     then
23:    isactive[active_thread] ← true
24:    active_thread ← active_thread + 1
25:  else
26:    active_thread ← active_thread − 1
27:    isactive[active_thread] ← false
28:    mistake ← mistake + 1
29:    to_clear ← true
30:  end if
31:  TPS_prev ← TPS_now
32:  while to_clear is true do
33:    do nothing
34:  end while
35: end while
```

4 評価実験

既存手法である Balanced Vector Scheduling，提案手法を実装し，提案手法の有効性を スループット， abort rate により評価する．スループットは計測期間中のコミット数を計測期間 (sec) で割ったものとし，abort rate は計測期間中のアボート数をコミット数で割ったものとする．実験には TPC-C [12] ベンチマークを使用する．実験環境として， 16 個の物理コアをもつ Intel(R) Xeon(R) Gold 6130 が 2 つ搭載され，メモリが計 1.5 TB の環境を使用した．全ての実験において，データベース内の最大総スレッド数は 32 とする．アボートが多く発生するワークロードを再現するために TPC-C では neworder と paymeny トランザクションのみを 1 : 1 の割合で実行する．比較手法として，2. 4. 2 項で示した Balanced Vector Scheduling，現在広く使用されているランダムスケジューリングを採用する．

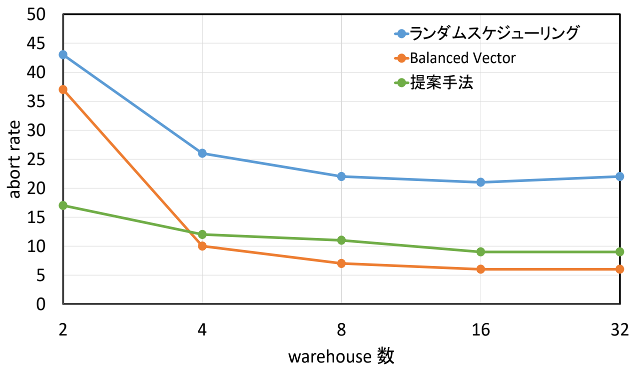

図 3　abort rate

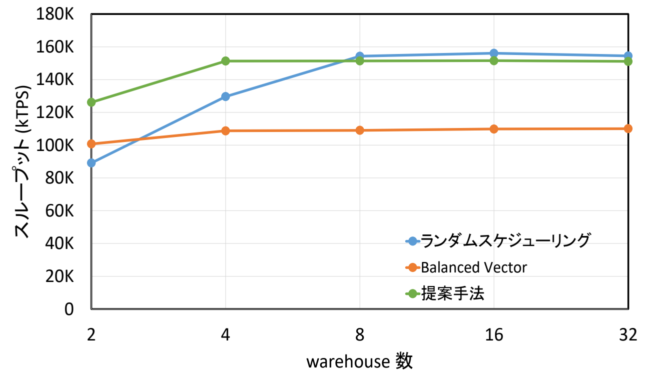

図 4　スループット

この章ではまず，4.1 節において提案手法の効果を評価し，4.2 節において提案手法の構成要素である探索打ち切り，スレッド数制御のそれぞれについてその効果を評価したマイクロベンチマークの結果を示す．4.3 節で提案手法でのスレッド数とスループットの変化を示す．

4.1　提案手法の評価

この節ではアボート予測を用いたスレッド割り当てに探索打ち切りおよびスレッド数制御を適用する場合 (提案手法) の性能を評価する．図 3 に TPC-C での warehouse 数を横軸として，縦軸に abort rate をとったグラフを示す．図 4 に TPC-C での warehouse 数を横軸として，縦軸にスループットをとったグラフを示す．TPC-C において warehouse 数はアクセスの集中度合いを表すパラメータとみなすことができ，小さいほどアクセスが集中するワークロードである．図 3 から提案手法はどのようなアクセスの集中度合いにおいても低い abort rate を保っていることが分かる．特に提案手法はアクセスが集中する場合 (warehouse = 2) において最も低い abort rate を示している．これはスレッド数を制御することで，過度なアボートが発生することを抑止していることによる．提案手法は Balanced Vector Scheduling に対して warehouse 数が 4 以上において，高い abort rate を示していることが確認できるが，これは探索打ち切りを行っていることによる．

図 4 から，提案手法は Balanced Vector Scheduling に対して常に高いスループットを示していることが確認できる．提案手法はアクセスが集中する場合 (warehouse = 2) では ランダムスケジューリングに対して 0.4 倍の abort rate，1.41 倍のスループット，Balanced Vector Scheduling に対して 0.46 倍の abort rate ，1.25 倍のスループットであった．アクセスが集中しない場合 (warehouse = 32) ではランダムスケジューリングに対して 0.41 倍の abort rate，0.98 倍のスループット，Balanced Vector Scheduling に対して 1.5 倍の abort rate，1.37 倍のスループットであった．

4.2　マイクロベンチマークの結果

この節ではスレッド数制御と探索打ち切りのそれぞれのみを適用した場合の結果について示す．図 5 に warehouse 数を横軸として，提案手法，Balanced Vector + スレッド数制御，

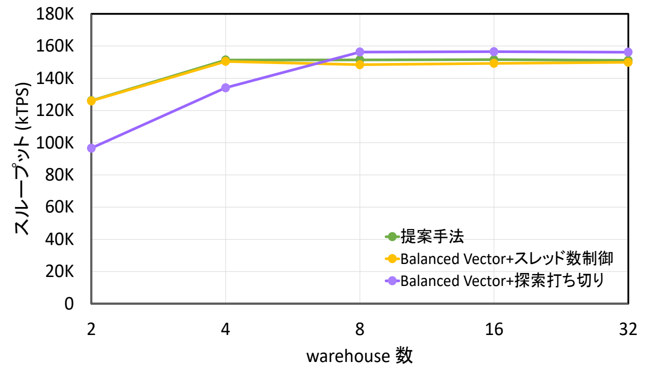

図 5　マイクロベンチマーク

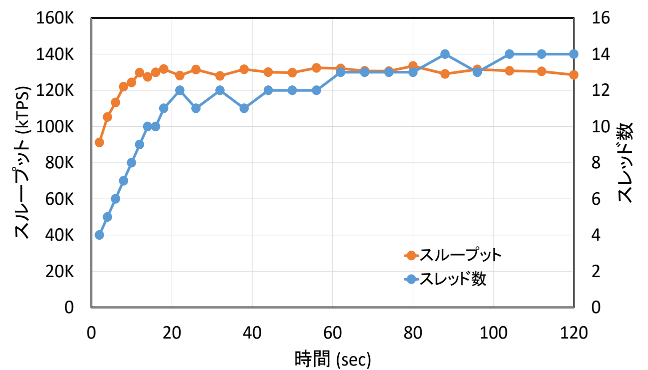

図 6　スレッド数とスループット

Balanced Vector + 探索打ち切り のスループットを示す．提案手法は Balanced Vecotr + スレッド数制御と同程度のスループットを示していることがわかる．これはスレッド数を制御することで，トランザクションのスレッド割り当て時に探索するべきスレッド数が少なくなり，探索のオーバーヘッドが小さくなっていることによると考えられる．スレッド数制御のみを行う場合と比較して，探索打ち切りを同時に行う提案手法のスループットが低下する場合は確認されなかった．しかし，スレッド数の制御に加えて探索打ち切りを合わせて行うことによるスループットの向上は最大で 2 % であった．

4.3　スレッド数制御の様子

この節では提案手法におけるスレッド数とスループットの変

化を示し，スレッド数制御が機能していることを示す．図 6 に warehouse 数を 2 とした場合において，実験開始からの経過時間 (sec) を横軸に，スレッド数およびスループットを示す．この図から，高いスループットを保つようにスレッド数が制御されていることが確認できる．

5 関 連 研 究

トランザクションをどのようにしてスレッドに割り当てるかに関しては複数のアーキテクチャが存在している [9]．本稿で対象としたアーキテクチャは Shared Everything である．Shared Everything ではデータベース全体が共有メモリ上に置かれ，任意のトランザクションはデータベース全体にアクセスが可能で，どのコア上でも動作することができる．Partitioning and Serial Execution (PSE) はデータベースを重複なく物理的にパーティショニングし，各パーティションに関してはシングルスレッドで処理される．各トランザクションは実行前に各パーティションのロックを取得する必要がある．DORA [13] はトランザクションをスレッドに割り当てる方式を採用しない．データベースを論理的にパーティショニングし，トランザクション処理スレッドは各論理パーティションの処理を担当する．トランザクションはアクセスするデータに基づいて，そのデータを担当するスレッドを移動していく．PSE と異なり，データベースを物理的にパーティショニングしないため，スレッドが担当するパーティションを動的に変更することが可能である．[9] では集中的なアクセスが起こる状況で，様々な並行実行制御法と各アーキテクチャの組み合わせの特性を比較している．全ての場合において最も優れた性能を示すアーキテクチャは無いとしながらも，Shared Everything が多くの場合で最適な選択であることが結論付けられている．

6 ま と め

本稿では，アクセスの集中度を考慮する効率的なトランザクションのスレッドへの割り当て手法を提案した．提案手法の有効性を TPC-C ベンチマークを用いて評価し，既存の手法と同等程度以上の性能を示すことを確認した．

今後の課題として，複数の並行実行制御法およびワークロードに対しての有効性を検証することがあげられる．

謝 辞

この成果は, 国立研究開発法人新エネルギー・産業技術総合開発機構（ＮＥＤＯ）の委託業務の結果得られたものです．

文 献

[1] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’08, page 981–992, New York, NY, USA, 2008. Association for Computing Machinery.

[2] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server’s Memory-Optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’13, page 1243–1254, New York, NY, USA, 2013. Association for Computing Machinery.

[3] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[4] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for in-Memory Databases. *Proc. VLDB Endow.*, 9(6):504–515, January 2016.

[5] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’15, page 677–689, New York, NY, USA, 2015. Association for Computing Machinery.

[6] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD ’17, page 21–35, New York, NY, USA, 2017. Association for Computing Machinery.

[7] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD ’16, page 1629–1642, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Yangjun Sheng, Anthony Tomasic, Tieying Sheng, and Andrew Pavlo. Scheduling OLTP Transactions via Machine Learning. http://arxiv.org/abs/1903.02990, 2019.

[9] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-contention Workloads. *Proc. VLDB Endow.*, 11(2):121–134, October 2017.

[10] Boost c++ libraries. https://www.boost.org/.

[11] Robert H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[12] Transaction Processing Performance Council. TPC benchmark C, 2010.

[13] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. Data-oriented Transaction Execution. *Proc. VLDB Endow.*, 3(1-2):928–939, September 2010.