Vol.2017-OS-139 No.16
2017/3/2

# 並行実行制御手法TicTocと並列ログ先行書き込み手法P-WALの結合

中村 泰大[1,a)]　川島 英之[2,b)]　神谷 孝明[3,c)]　建部 修見[2,d)]

**概要**：本論文はタイムスタンプベースの並行実行制御手法 TicToc に着目する．トランザクション処理システムは並行実行制御とログ先行書き込みから構成される．TicToc の原論文では Silo 等の並列ログ書き込み方式が実装可能だと簡単に述べられているのみであり，その詳細は明らかにされていない．そこで並列ログ先行書き込み手法 P-WAL と TicToc を結合し，性能を実験的に評価した．評価実験の結果，ログ書き込み先に不揮発性メモリを想定した環境では，データベースオブジェクトへのロックを早期解放する ELR と，ログレコードをストレージへ一括転送するグループコミットが性能劣化を招くという興味深い知見を得た．

## 1. はじめに

### 1.1 背景

トランザクション処理を高性能化する研究は急速に進展している．FOEDUS/MOCC [1] は 1 億 transaction/sec (tps) をシングルノードで示し，FaRM [2] は 1 億 4000 万 tps を 90 台のクラスタ環境で示している．これ程の高性能はトランザクション処理には必要ではなく，トランザクション処理性能に関する研究は終わったと考える研究者も存在する [3]．データベースを高性能化する研究者は同様の批判をされるようである [4]．

我々は正反対の考えをもっている．かつて人工衛星制御に使われるジャイロセンサーの精度が要求を超える程に高まったとき，それはワインパーティが開催される豪華客船の姿勢制御に使われるようになった [5]．同様に，これまでは想定されていなかった新市場に高性能トランザクション処理が使われるようになりつつあると我々は考える．その兆しの一つに，FOEDUS のグラフ処理への適用がある [6]．トランザクション処理システムの高性能化が新市場を拓くと我々は信じている．

### 1.2 研究課題

トランザクション処理システムは並行実行制御とログ先行書き込みから構成される．並行実行制御手法については Silo [7] をはじめとして，近年多くの斬新な手法 [8] が次々に提案されている．それらの手法の一つに TicToc [9] がある．TicToc はタイムスタンプをベースとしたスケーラブルな並行実行制御手法である．既存のタイムスタンプベースの並行実行制御手法はタイムスタンプ発行処理に排他制御が必要なため，マルチコア環境では高い性能を示せない．TicToc はタイムスタンプ発行処理における排他制御を排除することにより，マルチコア環境で性能がスケールアップすることが報告されている [9]．この TicToc について我々は次のひとつの疑問を持った．

- TicToc にログ先行書き込みの常識は成り立つのか?
  トランザクション処理システムは並行実行制御とログ先行書き込みから構成される．TicToc の原論文 [9] では Silo のような並列ログ先行書き込み手法が容易に実装可能だと述べられている．並列ログ先行書き込みにおける常識の一つに，データベースへのロックを早期に解放する Early Lock Release(ELR)，および複数のログレコードを一括して DRAM から永続的ストレージへ転送するグループコミットが性能向上に効果的な点がある．この常識は成り立つのだろうか?

### 1.3 貢献

上記の疑問を解消するために，我々はまず TicToc を再実装した．TicToc をメニーコアプロセッサ Xeon Phi (Knights Landing) 上で評価した結果，ログ先行書き込みが無い場合，TicToc の性能は 256 スレッドまでスケールアップした (2.1.3 節を参照)．

[1] 筑波大学 情報学群情報科学類
[2] 筑波大学 計算科学研究センター
[3] 筑波大学大学院 システム情報工学研究科
[a)] nakamura@hpcs.cs.tsukuba.ac.jp
[b)] kawasima@cs.tsukuba.ac.jp
[c)] kamiya@hpcs.cs.tsukuba.ac.jp
[d)] tatebe@cs.tsukuba.ac.jp

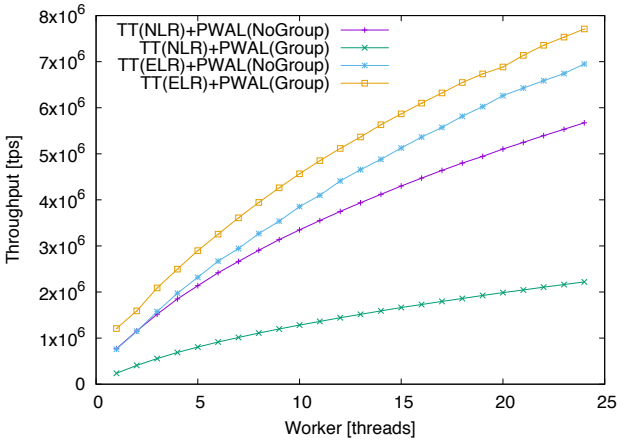


**図 1** マルチコア環境での実験結果

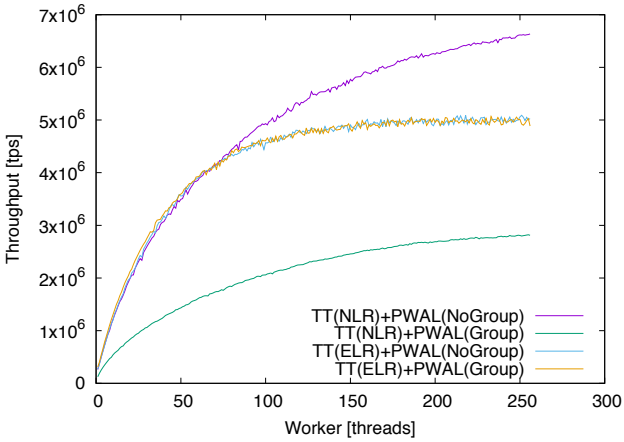


**図 2** メニーコア環境での実験結果

そして並列ログ先行書き込み P-WAL [10] を TicToc へ適用した．ストレージデバイスとして不揮発性メモリを想定した．マルチコア環境においては ELR とグループコミットが TicToc の性能を改善するという常識的な結果が得られた．これを図 1[*1]に示す．図中で最高性能を示しているのは ELR とグループコミットを適用した TT (ELR) + PWAL (Group) である．一方，メニーコア環境においては驚くべきことに，ELR とグループコミットの適用が TicToc の性能を劣化させるという，常識を覆す結果が得られた．これを図 2[*2]に示す．図中で最高性能を示しているのは ELR とグループコミットを適用しない TT (NLR) + PWAL (NoGroup) である．

### 1.4 論文構成

本論文の構成は以下の通りである．2 章では背景として TicToc と P-WAL を述べる．3 章では TicToc と P-WAL を結合し，TicToc に永続化処理を施す手法を提案する．4 章では提案手法の評価を行う．5 章では関連研究を述べ，6 章では結論を述べる．

[*1] 図 7 と同一である．実験環境を含めた詳細は 4.1.3 節を参照．
[*2] 図 11 と同一である．実験環境を含めた詳細は 4.2.3 節を参照．

## 2. 準備

### 2.1 並行実行制御手法 TicToc

楽観的並行実行制御 (OCC) [11] では，トランザクションのタイムスタンプを得るために共有メモリ上のカウンタを用い，それに排他制御や fetch-and-add [12] オペレーションを発行している．OCC は共有カウンタを用いたタイムスタンプの生成が性能のボトルネック [13] になっている．TicToc ではそのボトルネックをなくすために，トランザクションのタイムスタンプは全てワーカーが計算によって求める．

#### 2.1.1 データ構造

TicToc のタプルは Write Timestamp(wts) と Read Timestamp(rts) を持つ．これらはそれぞれ，そのタプルの値が書き込まれたタイムスタンプと最後に読まれたタイムスタンプである．タプルが wts と rts を持つのは，タプルの値が有効な範囲を示すためである．あるタプル A を読み込んだ時に，タイムスタンプがそれぞれ wts = 3，rts = 6 だったとする．タプル A を読み込んだトランザクションのタイムスタンプが 5 だと算出された場合，タプル A の値はタイムスタンプ 3 から 6 の間で有効であるから，タイムスタンプが 5 の時もタプル A の値は有効だと判断できる．従って，作業用にコピーしてきたタプルを確認するだけでよく，データベース中のタプル A に再度アクセスする必要がなくなる．

wts と rts をアトミックに読み込むために，TicToc ではそれらを 1 つの 64 bit word に格納して保管する．これを Timestamp word(TS_word) と呼ぶ．Timestamp word の内訳は Lock bit(1 bit)，delta = rts - wts(15 bits)，wts(48 bits) である．

#### 2.1.2 プロトコル

プロトコルは Read フェーズ, Validation フェーズ, Write フェーズの 3 つに分割される．トランザクション処理の流れを Algorithm 1 に示す．はじめに Read フェーズを実行し操作の対象となるタプルをワーカーにコピーする．コピーしたタプルに対してオペレーションを実行する．この時，読み込みを行ったタプルを Read セットに，書き込みを行ったタプルを Write セットに保持する．オペレーションの結果が他のワーカーと整合性を保っているかを Validation フェーズで検証する．整合性を保っていることが確認された場合，Write フェーズでデータの変更をデータベースに反映する．不整合が検知された場合は Read フェーズから再度実行する．

#### 2.1.3 メニーコア環境での性能評価

TicToc を C++を用いて作成し，メニーコア環境での性能評価を行った．具体的な設計と実装については 3.5 節にて述べる．実験環境を表 1 に示す．

評価にはマイクロベンチマーク R-10 と U-5/R-5 を用い

**Algorithm 1** execute transaction

```
    Retry point:
1:  readPhase()
2:  doOperation()
3:  if validationPhase() is fail then
4:      abort()
5:      retry()
6:  end if
7:  writePhase()
```

**表 1** メニーコアの評価環境

| | |
|---|---|
| プロセッサ | Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz |
| コア | 64 (256 スレッド) |
| メモリ | MCDRAM: 16 GB<br>DRAM: 96 GB |
| OS | CentOS Linux release 7.2.1511 (Core) |

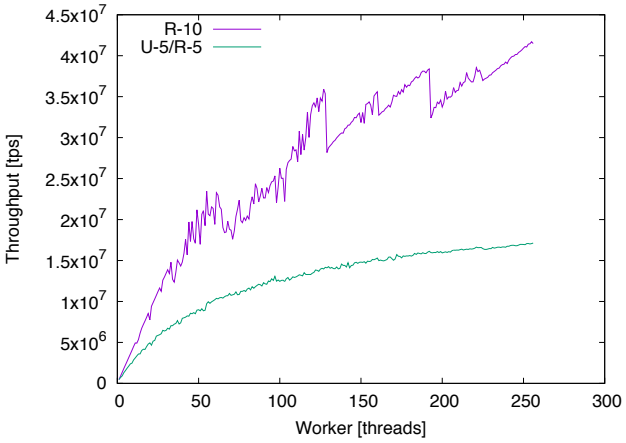


**図 3** メニーコア環境での TicToc のスループット

た．R-10 は各トランザクションが READ を 10 回行うもの，U-5/R-5 は各トランザクションが UPDATE と READ をそれぞれ 5 回行うものである．

実験結果を図 3 に示す．TicToc がメニーコア環境でスケールアップすることを観測した．R-10 ワークロードにおいてワーカー数が 64，128，192 を超えると急激に性能が低下していることがわかる．評価環境は実コア数が 64 であるから，ワーカースレッド数が 64，128，192 の場合には各コアに対してワーカースレッドが均一に割り当てられる．ワーカースレッド数が 64，128，192 を超えると，各コアに割り当てられるワーカースレッド数が不均一になる．コアとメモリ間のバンド幅は限られているから，コアに割り当てられたワーカースレッドが多くなるほど，ワーカースレッドあたりが使えるバンド幅が小さくなり，メモリのロードに時間がかかってしまう．従って，他のコアよりも多くのワーカースレッドを処理しているコアが他のコアに比べて遅くなり，全体のスループットが低下していると考えられる．

### 2.2 ログ先行書き込み手法 P-WAL

P-WAL はフラッシュストレージをストレージデバイスとするときにふさわしい Write-Ahead Logging (WAL) プロトコルである．P-WAL のアーキテクチャを図 4 に示す．HDD へのアクセスはランダムアクセスよりもシーケンシャルアクセスが I/O アクセス性能が高い．しかし，フラッシュストレージでは，並列ランダムアクセスの方が I/O アクセス性能が高くなる．P-WAL は，フラッシュストレージの並列ランダムアクセスが高性能であることを活用し，並列に動作する WAL プロトコルを提案している．

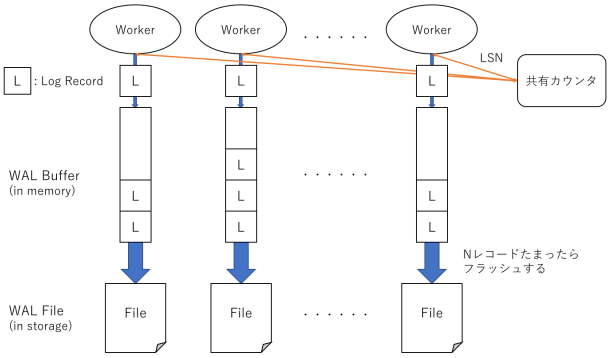


**図 4** P-WAL のアーキテクチャ

#### 2.2.1 WAL バッファの分割

従来 WAL バッファは HDD へのシーケンシャルアクセスを実現するため，システムで 1 つであった．そのため，ログレコードを WAL バッファへ挿入する際に衝突が発生し，性能劣化の原因になっていた．ランダムアクセスの方がシーケンシャルアクセスよりも高性能であるようなデバイスでは，並列にログを記述した方が高速になる．そこで，各ワーカースレッドにそれぞれ WAL バッファを持たせ，それぞれが対応する WAL ファイルに書き込みを行うのがフラッシュストレージにふさわしい WAL の構造となる．

#### 2.2.2 ログの順序

P-WAL では，ワーカースレッドごとに WAL バッファを持つため，ログの順序が不明瞭になる．そこで，P-WAL では共有カウンタを用いて Log Sequence Number(LSN) を各ログレコードに発行する．LSN は発行される毎にインクリメンタルに増加する．この共有カウンタはアトミックに操作する必要があるため，fetch-and-add もしくは compare-and-swap オペレーションを用いて実現する．

#### 2.2.3 通知制御

ログレコードを永続化したとしても，そのトランザクションがコミット可能とは限らない．なぜならば，トランザクションが他のワーカースレッドが生成したログレコードに依存している場合，そのログレコードが永続化されていないとトランザクションのコミット条件を満たしていないためである．コミットの通知制御を行うために，各 WAL バッファは自身が最後に永続化したログレコード

の LSN(flushedLSN) を保持する．あるトランザクションのコミット条件は，全ての WAL バッファが保持している flushedLSN の中で最小のものが自分の LSN 以上の場合である．ワーカースレッドはコミット待ちのトランザクションをキューに保持し，トランザクションが終わるたびに他の WAL バッファの flushedLSN を確認する．そしてコミット条件を満たしたトランザクションのコミット通知を行い，トランザクションをキューから除外する．

## 3. 提案：TicToc と P-WAL の結合

トランザクション処理システムは並行実行制御とログ先行書き込みから構成される．TicToc の原論文 [9] では Silo のような並列ログ先行書き込みが容易に実装可能だと記述されているが，具体的な検討はされていない．従って，TicToc にはどのようなログ先行書き込み手法が適切なのかを検討する必要がある．本章では，TicToc と P-WAL を結合し，TicToc に永続化処理を施す手法を提案する．

### 3.1 永続化プロトコル

我々は TicToc に永続化処理を施すプロトコルを提案する．P-WAL と同様に各ワーカースレッドに専用の WAL バッファと WAL ファイルを割り当てる．各ワーカースレッドは，実行中のトランザクションが Validation フェーズを通過しコミット可能と判断したら，WAL バッファ内のログレコードを WAL ファイルに書き込み永続化する．永続化が完了したら，Write フェーズを実行し，データベースへの書き込みとタプルのロックの解放を行う．

P-WAL では各ログレコードに共有カウンタを用いて LSN を発行するが，本プロトコルでは不要である．データベース上の値は全て永続化していることが保証されるためである．

### 3.2 ロック制御とグループコミット

#### 3.2.1 ロック解放時期

ロック解放時期には Nomal Lock Release(NLR) と Early Lock Release(ELR) [14] がある．

NLR はログレコードを永続化をしてからデータベースオブジェクトのロックを解放する手法である．TicToc ではロックされているタプルを Read フェーズで読み込まない．したがって，ワーカースレッドが Read フェーズで読み込むデータベースオブジェクトの値は，永続化されていることを保証できる．

ELR は，ログレコードを永続化する前にデータベースオブジェクトのロックを解放する手法である．これにより，ロックの保有期間をログレコード永続化のレイテンシだけ短くすることができる．ロックを早期に解放することで，他のワーカースレッドとの競合を緩和し，並行性を高めることが目的である．

ELR では，ワーカースレッドが Read フェーズで読み込むデータベースオブジェクトの値は永続化されている保証はない．トランザクションのコミット条件は，依存する全てのデータベースオブジェクトが永続化されていて，障害後もリカバリ可能なことである．従って，依存するデータベースオブジェクトが永続化されるまでコミット待機し，コミットの通知制御を行う必要がある．P-WAL ではこの問題を解消するために，各ログレコードに共有カウンタを用いて LSN を発行し全順序をつける．各ワーカースレッドは最後に永続化したログレコードの LSN(flushedLSN) を保持する．トランザクションは flushedLSN の最小値を見て，それがログレコードに割り当てられた LSN よりも大きければ自分よりも前に実行されたトランザクションのログレコードが全て永続化されたことがわかり，コミットを通知することができる．本プロトコルでは ELR を用いる場合にはトランザクションごとに LSN を発行して，それを通知制御及び後述するリカバリに用いる．しかし，共有カウンタを利用することは，共有領域を徹底的に排除した TicToc の設計に反する設計である．従って，LSN の利用は TicToc の高い並行性を犠牲にする可能性がある．

#### 3.2.2 グループコミット

グループコミット [14] はトランザクションをまたがって，複数のログレコードをまとめて永続化する手法である．ログレコードをまとめて永続化することにより，トランザクション処理の性能を向上させる．グループコミットは NLR と ELR の両方に適用することができる．

NLR にグループコミットを適用する場合，ログレコードを永続化してからロックを解放する必要があるため，ロックを複数のトランザクションにまたがって保持する必要がある．従って，ロックの保有期間が長くなってしまう．ロックの獲得を全順序に従って行うとデッドロックは発生しない．NLR にグループコミットを適用すると，トランザクションをまたがってロックの獲得が行われるため，全順序に従ってロックを獲得することができない．従って，容易にデッドロックしてしまう．デッドロックを回避するために，デッドロックが予想される挙動を示す場合にはグループコミット数に達していなくても WAL バッファを永続化する必要がある．永続化する必要がある挙動は以下の 2 点である．一つ目は，ロックの獲得に失敗したときである．この時に WAL バッファの永続化を行いワーカーの持つ全てのロックを解放すれば，次のトランザクションは全順序に従ってロックを獲得することができる．二つ目は，Read フェーズでタプルのコピーをしている時にロックの解放待ちをしている時である．2 つのワーカースレッドにおいて，データベースからコピーしたい値が，互いに相手がロックしているとデッドロックになってしまう．そこで，Read フェーズ時にロックの解放待ちが発生した場合には WAL バッファの永続化とロックの解放を行い，前述

した循環待機が発生しないようにする．

### 3.3 手法を適用したプロトコル

提案手法に ELR とグループコミットの適用・非適用を考えると以下の 4 つのパターンがプロトコルとして考えられる．

(1) TT(NLR) + PWAL(NoGroup)
(2) TT(NLR) + PWAL(Group)
(3) TT(ELR) + PWAL(NoGroup)
(4) TT(ELR) + PWAL(Group)

これらを Algorithm 2，Algorithm 3，Algorithm 4，Algorithm 5 に示す．

TT (NLR) + PWAL (NoGroup)(Algorithm 2) は NLR を行い，グループコミットは行わないプロトコルである．トランザクション毎に WAL バッファの永続化を行い，その後タプルのロックを解放する．

TT (NLR) + PWAL (Group)(Algorithm 3) は NLR とグループコミットを行うプロトコルである．グループコミット数のトランザクションを実行したのちに WAL バッファの永続化を行い，その後タプルのロックを解放する．3.2.2 節で述べたように，デッドロックする可能性があるため，グループコミット数に達していなくても WAL バッファを永続化しタプルのロックを解放する．

TT (ELR) + PWAL (NoGroup)(Algorithm 4) は ELR を行い，グループコミットは行わないプロトコルである．WAL バッファを永続化する前にタプルのロックを解放する．LSN をトランザクション毎に取得し，通知制御に用いる．WAL バッファの永続化はトランザクション毎に行う．

TT (ELR) + PWAL (Group)(Algorithm 5) は ELR とグループコミットを行うプロトコルである．WAL バッファを永続化する前にタプルのロックを解放する．LSN をトランザクション毎に取得し，通知制御に用いる．グループコミット数のトランザクションを実行したのちに WAL バッファの永続化を行う．

**Algorithm 2** TT(NLR) + PWAL(NoGroup)

```
Require: WAL Buffer walBuffer
 1: while runable() do
 2:     tx ←fetchTransaction()
    Retry point:
 3:     readPhase()
 4:     doOperation()
 5:     if validationPhase() is fail then
 6:         abort()
 7:         retry()
 8:     end if
 9:     walBuffer.flush()
10:     writePhase()
11:     releaseLocks()
12:     reply(tx)
13: end while
```

**Algorithm 3** TT(NLR) + PWAL(Group)

```
Require: WAL Buffer walBuffer, Group Commit Number N
 1: ntx ← 0
 2: txQueue ←<>                          ▷ <> is empty queue
 3: while runable() do
 4:     tx ←fetchTransaction()
    Retry point:
 5:     readPhase()
 6:     doOperation()
 7:     if validationPhase() is fail then  ▷ validationPhase 内でロックの獲得に失敗した場合も WAL の永続化処理を行う
 8:         abort()
 9:         walBuffer.flush()
10:         releaseLocks()
11:         reply(txQueue)
12:         txQueue ←<>
13:         retry()
14:     end if
15:     writePhase()
16:     txQueue.push(tx)
17:     ntx ← ntx + 1
18:     if ntx == N then
19:         walBuffer.flush()
20:         releaseLocks()
21:         reply(txQueue)
22:         txQueue ←<>
23:     end if
24: end while
```

**Algorithm 4** TT(ELR) + PWAL(NoGroup)

```
Require: WAL Buffer walBuffer
 1: flushedLSN ← 0
 2: commitQueue ←<>                      ▷ <> is empty queue
 3: while runable() do
 4:     tx ←fetchTransaction()
    Retry point:
 5:     readPhase()
 6:     doOperation()
 7:     if validationPhase() is fail then
 8:         abort()
 9:         retry()
10:     end if
11:     tx.lsn ←fetchLSN()
12:     writePhase()
13:     releaseLocks()
14:     commitQueue.push(tx)
15:     flushedLSN ← walBuffer.flush()
16:     minFlushedLSN ← min(worker.flushedLSN | worker ∈ workers)
17:     while commitQueue.notEmpty() do
18:         t ← commitQueue.front
19:         if minFlushedLSN ≥ t.lsn then
20:             repty(t)
21:             commitQueue.pop()
22:         else
23:             break
24:         end if
25:     end while
26: end while
```

**Algorithm 5** TT(ELR) + PWAL(Group)

**Require:** WAL Buffer $walBuffer$, Group Commput Number $N$

```
1: ntx ← 0
2: flushedLSN ← 0
3: commitQueue ←<>                          ▷ <> is empty queue
4: while runable() do
5:     tx ←fetchTransaction()
   Retry point:
6:     readPhase()
7:     doOperation()
8:     if validationPhase() is fail then
9:         abort()
10:        retry()
11:    end if
12:    tx.lsn ←fetchLSN()
13:    writePhase()
14:    releaseLocks()
15:    commitQueue.push(tx)
16:    ntx ← ntx + 1
17:    if ntx == N then
18:        flushedLSN ← walBuffer.flush()
19:        ntx ← 0
20:    end if
21:    minFlushedLSN ← min(worker.flushedLSN | worker ∈ workers)
22:    while commitQueue.notEmpty() do
23:        t ← commitQueue.front
24:        if minFlushedLSN ≥ t.lsn then
25:            repty(t)
26:            commitQueue.pop()
27:        else
28:            break
29:        end if
30:    end while
31: end while
```

### 3.4 リカバリ

リカバリプロトコルは NLR を採用した場合と ELR を採用した場合とで異なる．ELR を採用した場合には P-WAL のリカバリプトロコルを用いる．各 WAL ファイルの先頭のログレコードの LSN を比較し，LSN の小さい順にログレコードを適用する．

NLR を採用した場合のリカバリには，TicToc のタイムスタンプの性質を用いる．TicToc の方式を用いて計算したトランザクションのタイムスタンプは，依存するトランザクションのタイムスタンプよりも必ず大きくなるという性質を持つ．また，データベースオブジェクトの値は永続化されてから更新されるため，依存するトランザクションのログレコードは全て永続化されていることが保証される．従って，あるログレコードを適用するための条件は，自分が持つタイムスタンプよりも小さいタイムスタンプを持つログレコードが全て適用されていることである．

具体的なプロトコルを述べる．はじめに，各 WAL ファイルのログレコードをタイムスタンプの昇順にソートする．その後，ソートされた WAL ファイルの先頭のログレコードのタイムスタンプを比較し，最も小さいログレコードから順に適用する．

### 3.5 設計と実装

提案手法を評価するために，プロトタイプシステムの設計と実装を行った．実装には C++を用いた．設計したプロトタイプシステムのモジュール構成を図 5 に示す．Transaction はトランザクション処理全体の制御を行うモジュールで，WalBuffer モジュールおよび Database と通信する．Database はオブジェクトを保持する．Transaction モジュールと Database が持つオブジェクトを区別するために，Transaction モジュールは LocalTuple，Database は Tuple という形式でオブジェクトを保持する．WalBuffer モジュールは Transaction モジュールからログレコードを受け取って蓄積し，Transaction モジュールからの制御命令によって蓄積したログレコードを WAL File に書き込んでログレコードを永続化する．

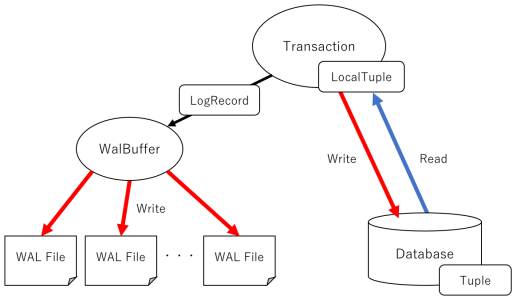


**図 5** プロトタイプシステムのモジュール構成

## 4. 評価

設計したトランザクション処理手法の評価を行う．評価はマルチコア環境とメニーコア環境で行う．NLR/ELR とグループコミットをそれぞれ組み合わせた TT (NLR) + PWAL (NoGroup)， TT (NLR) + PWAL (Group)， TT (ELR) + PWAL (NoGroup)， TT (ELR) + PWAL (Group) の 4 手法と，トランザクションシステム全体で単一の WAL ファイルを用いる TT (ELR) + SWAL の手法を実験により性能の測定を行い，評価する．この実験では，各ワーカースレッドがそれぞれ事前に指定された回数だけトランザクションを実行する．WAL ファイルの書き込み先は不揮発性メモリを想定し，WAL ファイルへの書き込みが行われるタイミングで固定長のレイテンシを挿入する．プロトタイプシステムでは固定長のレイテンシを 50ns，グループコミット数を 16 とした．実行するワークロードは U-1, U-10，U-5/R-5 の 3 種類である．

表 2　マルチコアの評価環境

| | |
|---|---|
| プロセッサ | Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz |
| コア | 24 |
| メモリ | 64GB |
| OS | CentOS release 6.8 (Final) |

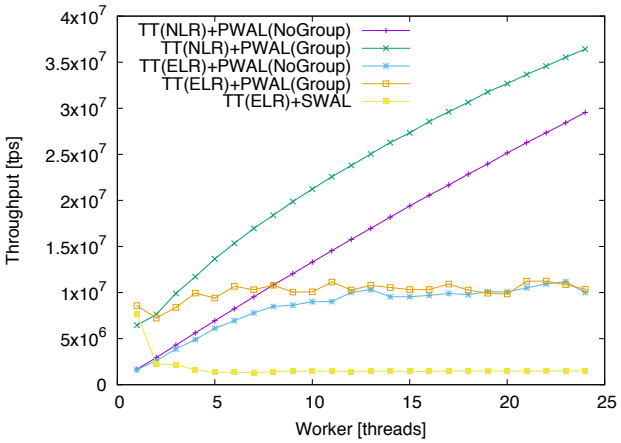


図 6　マルチコア環境での U-1 ワークロードのスループット

### 4.1　マルチコア環境

マルチコア環境の実験では，各ワーカースレッドに 200 万件のトランザクションを処理させ，そのときのスループットを測定した．

#### 4.1.1　実験環境

実験環境を表 2 に示す．

#### 4.1.2　U-1

U-1 は各トランザクションが UPDATE を 1 回行うものである．実験結果を図 6 に示す．

実験結果より，ELR は約 1000 万 tps を上限に，ワーカースレッドを増加させてもスループットが上昇しないことが観察される．この現象を発現させた原因の一つの可能性として，共有カウンタへのアクセス過多があると考える．このワークロードでは各トランザクションはタプル 1 つに関してのみロックを獲得するため，競合する可能性が少ない，従って共有カウンタへのアクセスが集中しそのレイテンシが増加し，スループットが上昇しなかった可能性がある．

NLR はスケールしていることが観察される．これはトランザクション間の競合が少ないことで，ワーカースレッドが他のワーカースレッドに阻害されることなく動作しているためだと考えられる．グループコミットを行ってロックの保有期間が延びても競合の可能性が低いため，他のワーカースレッドがロックを獲得している可能性が低い．また，WAL ファイルへの書き込みをまとめて行うため，グループコミットを行わない場合に比べてスループットが高まっている．

TT (ELR) + SWAL は他の提案手法に比べて性能が悪いことが観察される．今後の実験では，TT (ELR) + SWAL の性能評価は省略する．

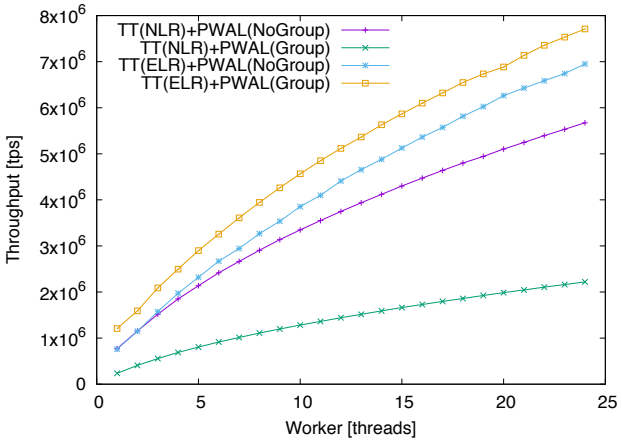


図 7　マルチコア環境での U-10 ワークロードのスループット

#### 4.1.3　U-10

U-10 は各トランザクションが UPDATE を 10 回行うものである．実験結果を図 7 に示す．

ELR，NLR ともにスレッド数増加と共にスケールしている．このワークロードはトランザクションが競合する可能性が U-1 と比べて多い．したがって，ロック保有期間が長いと，他のトランザクションを長時間待機させ，並行性が低くなることになる．

TT (NLR) + PWAL (Group) の性能が最も悪いことが観察される．この原因は，この手法がデータベースオブジェクトのロック保有期間が 4 手法中で最長であることだと考える．

TT (ELR) + PWAL (Group) の性能が最も良いことが観察される．この原因は，この手法がデータベースオブジェクトのロック保有期間が 4 手法中で最短であることだと考える．

TT (ELR) + PWAL (Group) は TT (ELR) + PWAL (NoGroup) よりも高いスループットを示したことが実験から観察される．この原因はグループコミットが WAL ファイルへの書き込みコストを削減したことだと考える．

#### 4.1.4　U-5/R-5

U-5/R-5 は各トランザクションが UPDATE と READ をそれぞれ 5 回行うものである．UPDATE と READ の対象となるタプルは同一のタプルである．実験結果を図 8 に示す．アボート率を図 9 に示す．

はじめは TT (ELR) + PWAL (Group) が他の手法に比べて性能が高いが，24 スレッドでは TT (NLR) + PWAL (NoGroup) が他の手法と比べて最も性能が高くなっている．性能が逆転している原因は ELR において，ワーカースレッド数の増加により共有カウンタへの競合が増えたことが要因であると考えられる．図 6 からも ELR の性能限界は約 1000 万 tps であると考えられる．このワークロードでも 1000 万 tps に近づくにつれて徐々に性能が劣化して

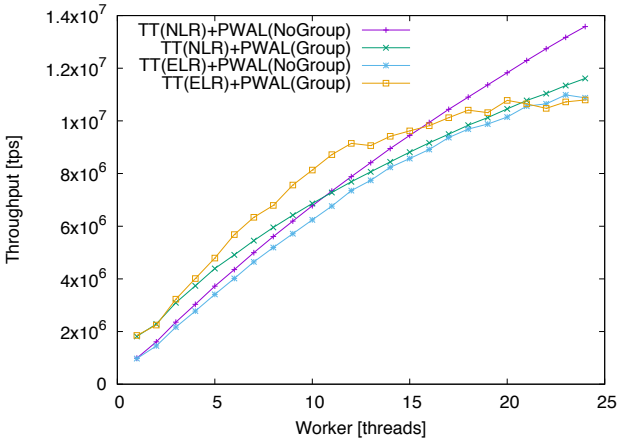


**図 8** マルチコア環境での U-5/R-5 ワークロードのスループット

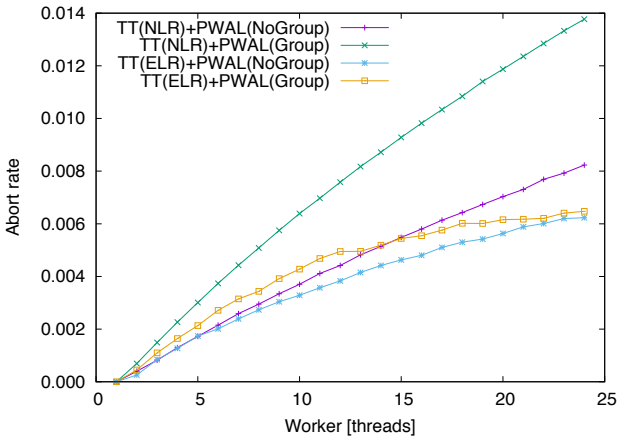


**図 9** マルチコア環境での U-5/R-5 ワークロードのアボート率

いる．NLR は共有カウンタの競合は存在しないため，ワーカースレッド数増加と共にスケールし，ELR のスループットを上回ったのだと考える．

TT (NLR) + PWAL (Group) は TT (NLR) + PWAL (NoGroup) に比べてスループットが低い．これは TT (NLR) + PWAL (Group) のアボート率が TT (NLR) + PWAL (Group) のアボート率に比べて高いことが原因だと考える．

24 スレッド時に，ELR が NLR に対してアボート率が低くなっている．しかし，スループットは NLR の方が高い．これは，共有カウンタアクセスのレイテンシが原因であると考えられる．LSN を発行する時には，そのトランザクションはすでに Validation フェーズを通過し，不整合がない状態だと確認されている．もうアボートしえない状態で LSN を発行するレイテンシが長いためにアボートレートは上昇しないが，トランザクションにかかる時間が増加しているためスループットは低下すると考える．

### 4.2 メニーコア環境

メニーコア環境の実験では，各ワーカースレッドに 100 万件のトランザクションを処理させ，そのときのスループットを測定した．

#### 4.2.1 実験環境

実験環境を表 1 に示す．この実行環境は 2.1.3 節で評価を行った環境と同一である．

#### 4.2.2 U-1

実験結果を図 10 に示す．TT (NLR) + PWAL (NoGroup) の性能が最も良いことが観測される．TT (NLR) + PWAL (NoGroup) のピーク性能は TT (ELR) + PWAL (Group) と比べて 9.0 倍である．

TT (ELR) + PWAL (NoGroup) および TT (ELR) + PWAL (Group) が最も低い性能であることが観察される．また，10 スレッド以降でスループットが横ばいになっていることが観察される．この原因は，共有カウンタへのアクセスがボトルネックになっていると考える．U-1 は短いトランザクションであるから，ELR を用いると共有カウンタへのアクセスが頻繁に発生し，衝突が増える．10 スレッド付近で LSN 発行のスループットが限界性能に達したのだと考える．

TT (NLR) + PWAL (NoGroup) は TT (NLR) + PWAL (Group) に比べて 1.8 倍のスループットを示している．TT (NLR) + PWAL (Group) の方が性能が低いのは，ロック獲得の競合が原因だと考えられる．NLR でグループコミットを使うとログレコードの永続化が完了するまでロックを保持しなければならない．従ってロックの保有期間が長くなる．マルチコア環境ではロックの保有期間が長くなってもワーカースレッドの数が少ないことからロックの競合が起こることは少ない．しかしメニーコア環境ではロック獲得で競合する可能性がワーカースレッド数の上昇に伴って高まった．トランザクション毎にログを永続化するコストよりも，ロックを獲得するコストの方が高くなってしまったと考える．

ここで注意すべきことは，この結果が常識に反している点である．商用システムはもちろんのこと，研究レベルの最先端システムにおいてさえ，ELR とグループコミットは性能向上に必須だと考えられている．しかしこの結果は，それらがいずれも不要であることを示している．

#### 4.2.3 U-10

実験結果を図 11 に示す．TT (NLR) + PWAL (Group) の性能が最も悪いことが観測される．この原因は，この手法のロック保有期間が最も長くなることだと考える．ロック保有期間が長いと，他のトランザクションを長時間待機させ，並行性が低くなる．加えてワーカー数が増大することにより，ロックの競合率が増加し，並行性が低くなったのだと思われる．

TT (ELR) + PWAL (NoGroup) および TT (ELR) + PWAL (Group) は 150 スレッド以降でスケールアップが観測されない．この原因は，U-1 と同様に，LSN の発行がボトルネックになっているのだと考える．U-10 は U-1 に比べてトランザクションが長いため，スレッド数が少ない場合に

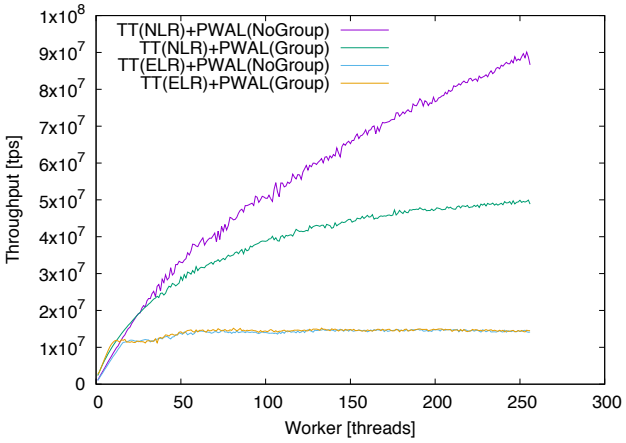


**図 10** メニーコア環境での U-1 ワークロードのスループット

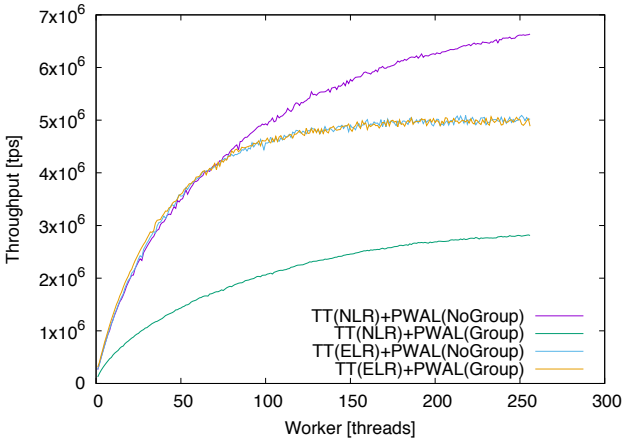


**図 11** メニーコア環境での U-10 ワークロードのスループット

は共有カウンタの競合は少ないためスケールが観測されているが，スレッド数増加によって共有カウンタアクセスの競合が起こることでそれ以上スケールアップしなくなったのだと考える．

TT (NLR) + PWAL (NoGroup) の性能が最も良いことが観測される．この原因は，ロックの保有期間が TT (NLR) + PWAL (Group) よりも短く，ELR のような共有カウンタのボトルネックがないことだと考える．

#### 4.2.4 U-5/R-5

実験結果を図 12 に示す．アボート率を図 13 に示す．マルチコア環境での U-5/R-5 と同様に，スレッド数が増加するにしたがって TT (NLR) + PWAL (NoGroup) の性能が他の手法を上回ることが観測された．TT (NLR) + PWAL (NoGroup) は TT (ELR) + PWAL (Group) に比べて 1.3 倍のスループットになっている．

TT (NLR) + PWAL (Group) の性能が最も悪いことが観測される．この原因は，他の手法に比べてこの手法のアボート率が高いことだと考える．

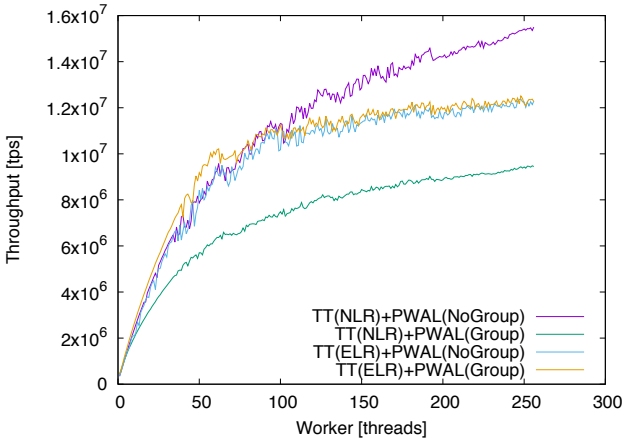


**図 12** メニーコア環境での U-5/R-5 ワークロードのスループット

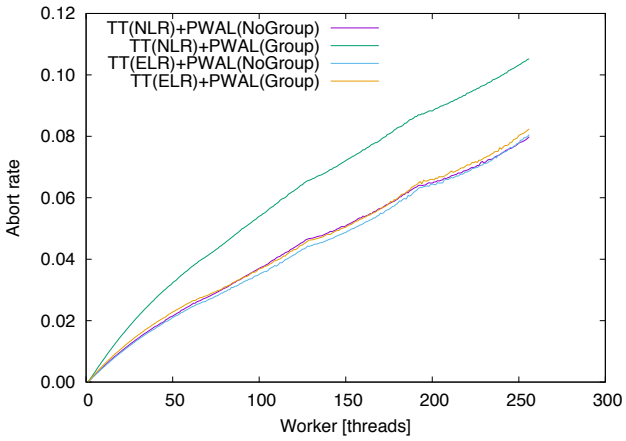


**図 13** メニーコア環境での U-5/R-5 ワークロードのアボート率

## 5. 関連研究

### 5.1 共有メモリを想定した研究

共有メモリ構造，すなわちシングルノードを想定したトランザクション処理研究において現代の基礎技術と考えられる技術は Silo [7] である．Silo はマルチコアマシンで優れたパフォーマンスとスケーラビリティを達成するために作られたインメモリデータベースシステムである．Silo は，タイムスタンプの算出手法とコミットプロトコルを提案している．Epoch という単位ごとにコミット完了の通知制御を行うことでレイテンシを犠牲にする代わりにスループットを高めている．Epoch の識別には Global epoch counter という共有カウンタを用いられる．このカウンタは専用のスレッドが定期的にインクリメントし，各ワーカースレッドがこの値を読み込むことで同期処理が行われる．この定期処理は 40 ms ごとに行われる．

FOEDUS [15] は不揮発性メモリを前提として設計されたトランザクション処理システムである．範囲検索を高性能化するために Masstree [16] を不揮発性メモリに対応させた MasterTree が新規に提案されており，Silo [7] に基づ

くコミットプロトコルならびに並行実行制御プロトコルが提案されている．FOEDUS の応用としてグラフ処理システム Janus [6] が考案されている．また，温度制御などを用いて楽観的実行制御と悲観的実行制御を組み合わせた高性能並行実行制御方式 MOCC [1] は FOEDUS 上で実装されている．

本研究で取り上げた TicToc [9] はタイムスタンプをベースにした並行実行制御であり，Silo に比べて高い性能を示すことが報告されている．一方，本研究で述べたようにメニーコア環境での性能が不明であり，さらにログ先行書き込みとの結合方式については検討が未熟だった．

### 5.2 ログ先行書き込みに関する研究

Wang ら [17] は複数の WAL バッファを不揮発性メモリ内に用意し，その中へログレコードを並列的に転送する手法である Passive group commit を提案している．Passive group commit は不揮発性メモリ上に WAL バッファを置き，ログを DRAM を経由せずに直接不揮発性メモリに書き込む．Passive group commit は ELR であり，各トランザクションがコミット条件を満たしたかどうかをまとめて確認する専用のデーモンを用意する．また，Passive group commit には GSN(Global Sequence Number) と呼ばれるシーケンス番号が必要である．

P-WAL [10] は並列にログ先行書き込みを実行するプロトコルであり，本論文でも詳述した．P-WAL は ioDrive のようにストレージデバイスが並列性を内包する際に優れたスループットを示す．

## 6. 結論

本論文はタイムスタンプベースの並行実行制御手法 TicToc に着目した．タイムスタンプ発行処理における排他制御を排除することにより，TicToc はマルチコア環境で性能がスケールアップすることが原論文で報告されている [9]．

我々は TicToc についてひとつの興味を持った．それは，TicToc にふさわしいログ先行書き込み手法である．トランザクション処理システムは並行実行制御とログ先行書き込みから構成される．TicToc の原論文 [9] では Silo [7] 等の並列ログ書き込み方式が実装可能だと簡単に述べられているのみであり，その詳細は明らかにされていなかった．そこで並列ログ先行書き込み手法 P-WAL [10] と TicToc を結合し，実験的に評価した．

評価実験の結果，興味深い現象が観察された．それはトランザクション処理性能を向上させる標準的な手法が性能劣化を招くことである．データベースオブジェクトへのロックを早期解放する ELR と，WAL ログをストレージへ一括転送するグループコミットが性能向上をもたらすことは常識となっている．ログ書き込み先に不揮発性メモリを想定した環境では，この常識が覆されるという結果を得た．Xeon Phi を用いた実験において，3 種類のワークロードすべてにおいて，ELR とグループコミットを用いない手法が最高性能を示した．

TicToc と P-WAL を結合した場合，不揮発性メモリ・メニーコア環境では ELR とグループコミットが性能劣化を招く，と本論文は結論する．

**謝辞** 本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」，JST CREST「EBD:次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算物理学」，科研費「#16K00150」による．

### 参考文献

[1] Wang, T. and Kimura, H.: Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores, *Proc. VLDB Endow.*, Vol. 10, No. 2, pp. 49–60 (2016).
[2] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A. and Castro, M.: No Compromises: Distributed Transactions with Consistency, Availability, and Performance, *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 54–70 (2015).
[3] Pavlo, A.: Chasing the Dragon of OLTP Databases, *http://www.cs.cmu.edu/ ./pavlo/blog/2016/08/chasing-the-dragon-of-oltp-databases.html*. 2017/2/7 アクセス.
[4] 鬼塚真：喜連川先生インタビュー，*http://www.ieice.org/iss/de/old/newsletter/letter1.pdf*. 2017/2/7 アクセス.
[5] 三宅秀道：新しい市場のつくりかた，東洋経済新報社 (2012).
[6] Kimura, H., Simitsis, A. and Wilkinson, K.: Janus: Transaction Processing of Navigation and Analytic Graph Queries on Many-core Servers, *Biennial Conference on Innovative Data Systems Research* (2017).
[7] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy Transactions in Multicore In-memory Databases, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 18–32 (2013).
[8] Kim, K., Wang, T., Johnson, R. and Pandis, I.: ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads, *Proceedings of the 2016 International Conference on Management of Data*, pp. 1675–1687 (2016).
[9] Yu, X., Pavlo, A., Sanchez, D. and Devadas, S.: TicToc: Time Traveling Optimistic Concurrency Control, *Proceedings of the 2016 International Conference on Management of Data*, pp. 1629–1642 (2016).
[10] 神谷孝明，川島英之，星野喬，建部修見：並列ログ先行書き込み手法 P-WAL，情報処理学会論文誌データベース，Vol. 10, No. 1. 採録決定.
[11] Kung, H. T. and Robinson, J. T.: On Optimistic Methods for Concurrency Control, *ACM Trans. Database Syst.*, Vol. 6, No. 2, pp. 213–226 (1981).
[12] Herlihy, M.: Wait-free Synchronization, *ACM Trans. Program. Lang. Syst.*, Vol. 13, No. 1, pp. 124–149 (1991).
[13] Yu, X., Bezerra, G., Pavlo, A., Devadas, S. and Stone-

braker, M.: Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores, *Proc. VLDB Endow.*, Vol. 8, No. 3, pp. 209–220 (2014).

[14] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M. and Ailamaki, A.: Aether: a scalable approach to logging, *Proceedings of the VLDB Endowment*, Vol. 3, No. 1-2, pp. 681–692 (2010).

[15] Kimura, H.: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pp. 691–706 (2015).

[16] Mao, Y., Kohler, E. and Morris, R. T.: Cache craftiness for fast multicore key-value storage, *Proceedings of the 7th ACM european conference on Computer Systems*, ACM, pp. 183–196 (2012).

[17] Wang, T. and Johnson, R.: Scalable Logging Through Emerging Non-volatile Memory, *Proc. VLDB Endow.*, Vol. 7, No. 10, pp. 865–876 (2014).