Vol.2015-ARC-215 No.18
Vol.2015-OS-133 No.18
2015/5/27

P-WAL: 並列ログ先行書き込みの提案

神谷 孝明[1,a)]　川島 英之[2,b)]　建部 修見[2,c)]

概要：本研究では ioDrive をストレージデバイスとする時にふさわしい WAL プロトコルとして *P*-WAL を提案する．まず，ioDrive においては並列ランダムライト時の I/O アクセス性能がシーケンシャルライトよりも高速になることを示す．そのような状況において，I/O アクセスと排他制御処理による性能劣化の問題に対処するために，*P*-WAL はそれぞれのログライタが専用の領域にログを書き込む，並列ログ書き込み方式を採用する．*P*-WAL の評価を行うため，数種類のモジュールからなるプロトタイプのトランザクションマネージャを設計する．これを用いて，スレッド数やグループコミットのパラメータを変化させ，従来方式の WAL と *P*-WAL の性能比較を行う．スレッド数を 16，グループコミットのパラメータを 16 にした時，*P*-WAL は 172,018 tps の性能を発揮し，従来方式と比べて，3.23 倍の性能向上を達成することを示す．

1. はじめに

証券取引所における株価の変更 [1] やクレジットカードの決済 [2] などは，処理の途中で障害が起こったとしても，それまでに完了された操作の情報は確実に保存され，システムのリスタート時に適切に障害回復処理を行う必要がある．このような不可分な一連の操作群はトランザクション [3] と呼ばれる．同時実行制御を行いながら複数のトランザクションを管理する機構はトランザクションマネージャと呼ばれる．トランザクション処理の高性能化は様々な研究で行われている [4][5][6][7]．トランザクションの実行中に障害が起こった場合は，トランザクションマネージャはデータベースの状態をトランザクション前の状態に戻さなければならない．トランザクションマネージャが保証しなければならないトランザクションの特性は，原子性（Atomicity），一貫性（Consistency），独立性（Isolation），永続性（Durability）であり，この中で障害復旧に関する特性は AD である．

AD 特性を保証するため，トランザクションマネージャはデータベース操作（例：SQL 問合せ，SQL 更新処理）に加えて特別な処理を行う必要がある．それはログ先行書き込み（WAL: Write Ahead Logging）[8] である．WAL とは，ストレージ中のデータを書き換える前にその更新内容をストレージにログとして記録する処理である．WAL はストレージアクセスを要するためにトランザクション処理の性能を劣化させる．現在の WAL アルゴリズムはストレージデバイスとして HDD を前提としている．そのため，現在の WAL は高い I/O アクセスコストを極小化することを目的関数としており，複数のトランザクションログをメモリ中でまとめてストレージデバイスに一括して書き込む方式を採用している．この方式は PostgreSQL[9] や Oracle[10] などのデータベースシステムで用いられているに留まらず，現在でも技術革新が行われている [11]．

高性能ストレージとして知られる ioDrive[12] は HDD とは I/O アクセスコストが異なるため，現在の WAL アルゴリズムがそのストレージデバイスの性能を極大化できるか明らかではない．そこで本研究では ioDrive をストレージデバイスとする時にふさわしい WAL プロトコルとして *P*-WAL を提案する．*P*-WAL はそれぞれのワーカースレッドに専用の WAL バッファを持たせることで，ロックを必要とせずに，ログレコードを WAL バッファに挿入できる．また，不揮発ストレージ上に WAL バッファと同じ数の WAL ファイルを作成し，各 WAL ファイルを WAL バッファと一対一で対応させる．これにより，ログレコードの挿入処理と同様に，それぞれのワーカースレッドはロックを必要とせずに，WAL バッファの内容を WAL ファイルへ書き込むことができる．ioDrive における並列ランダムライトの高性能性を活用することで，*P*-WAL は高い性能を達成する．

本研究は ARIES[13] スキームに基づき，WAL バッファへのログ挿入処理の競合緩和と WAL バッファの WAL ファ

[1] 筑波大学大学院 システム情報工学研究科 コンピュータサイエンス専攻
[2] 筑波大学 システム情報系
[a)] kamiya@hpcs.cs.tsukuba.ac.jp
[b)] kawasima@cs.tsukuba.ac.jp
[c)] tatebe@cs.tsukuba.ac.jp

Vol.2015-ARC-215 No.18
Vol.2015-OS-133 No.18
2015/5/27

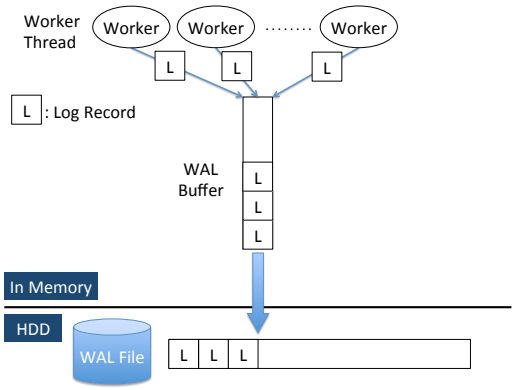

図 1 WAL のアーキテクチャ

イルへの移送の並列化を行う．ARIES スキームに基づき WAL を高速化する既存研究には Aether[11]，Deuteronomy[14][15]，そして分散ロギング [16] がある．これらの研究は本研究とは異なり WAL バッファの WAL ファイルへの移送を並列化しない．一方，シーケンス番号を使わずに WAL を再設計する研究には Silo[17][18] と FOEDUS[19] がある．これらの研究はいずれも ARIES スキームを大幅に修正する必要があるため，本研究とは異なり，成果を既存システムへ導入することは困難だと考えられる．

本稿の構成は以下の通りである．2 節ではトランザクション処理で用いられる WAL について述べる．3 節ではストレージデバイスの種類毎に特徴を説明し，実際に I/O 性能を測定した結果を示す．4 節では高性能ランダムライトを活用した高性能並列 WAL として *P*-WAL を提案し，そのプロトコルを述べる．5 節では *P*-WAL を実装したプロトタイプのトランザクションマネージャの設計と実装について述べる．6 節ではスレッド数とグループコミットのパラメータを変えてトランザクション処理性能を計測した実験の結果を示す．7 節ではロギングにまつわるその他の関連研究を述べる．8 節では結論と今後の課題を述べる．

2. WAL

ログ先行書き込み（Write-Ahead Logging），通称 WAL とはシステム障害に備えてデータの更新前にログを書き込む手法である．トランザクションによって作成されたログはメモリ中の WAL バッファに溜められる．トランザクションのコミット処理で，それまでに溜められた WAL バッファ中のログとコミットログをまとめ，ストレージの WAL ファイルに一括で書き込む（**図 1**）．トランザクションマネージャはコミットログの有無によってトランザクションが成功したか失敗したかを判断する．コミットされたトランザクションによる更新はデータベースに適用し，コミットされていないトランザクションによる更新を書き戻す．このようにして，WAL はトランザクションの ACID 特性の Atomicity と Durability を保証する．WAL はログを永続化するため，ストレージへの書き込みを必要とする．書き込み待ち時間や WAL バッファへログを挿入する際の排他制御処理にかかる時間が大きいと，WAL がトランザクション処理のボトルネックになる可能性がある．

Algorithm 1 log_insert_with_lock(log)

```
1: WALbuffer.lock() #WAL バッファをロック
2: LSN ← WALbuffer.insert(log)
3: if log.Type == 'COMMIT' then
4:     #コミットログが挿入された場合，カウントアップ
5:     WALbuffer.ncommit ← WALbuffer.ncommit +1
6:     if WALbuffer.ncommit == NGROUP or WALbuffer.full
       () then
7:         WALbuffer.flush() #WAL バッファの内容を WAL ファ
           イルに書き込む
8:     end if
9: end if
10: WALbuffer.unlock() #WAL バッファをアンロックする
11: return LSN
```

2.1 グループコミット

WAL の基本プロトコルはトランザクションがコミットする度に WAL バッファの中のログを WAL ファイルに追記させる．トランザクションが頻繁にコミット処理を行うと，コミットによる I/O が多発し，性能劣化が起こる．そこで，複数のトランザクションをまとめてコミットさせることで，トランザクション全体の性能を上げるべく考えられたのがグループコミットである．

グループコミットでは，指定されたグループ数のコミットログが挿入された時，または，トランザクションがコミットログを挿入してから一定時間が経過した時に，WAL バッファのログを WAL ファイルに書き出す．このグループコミットによって I/O の回数がグループ数分の一になる．

2.2 ログレコードの挿入アルゴリズム

トランザクションのログレコードを WAL バッファに挿入する手続き log_insert_with_lock を Algorithm 1 に示す．まず，WAL バッファにロックをかける（1 行目），引数として渡されたログレコードを WAL バッファに挿入する（2 行目）．その際にはログレコードの ID となる LSN (Log Sequence Number）が決定される．挿入処理が終わると WAL バッファをアンロックする（10 行目）．WAL バッファがグループコミットのグループ数に達したか，あるいは WAL バッファが一杯になったかをチェックして（6 行目），そうであれば WAL バッファのログレコード群を WAL ファイルに書き込む（7 行目）．ログレコードの WAL バッファへの挿入，及び WAL ファイルへの書き込みは，WAL バッファのロックを保持した状態で行われる．そのため，あるスレッドが WAL バッファへ挿入処理をしている間は，他のスレッドはログレコードを挿入することがで

表 1 実験環境 (1)

CPU	Intel（R） Xeon（R） CPU E5-2665 × 2
コア数	8 × 2
メモリ	64GB
ioDrive	SLC, 160GB, VRG5T VSL v3.3.3, Low-Level Formatting

きない．特に WAL ファイルへの書き込みが発生すると，I/O 待ちのために遅延時間が大きくなる．

3. ストレージデバイスの性能特性

従来の DBMS では HDD を前提として WAL プロトコルが設計されている．一方，普及しつつある SSD や ioDrive においては性能特性が HDD とは異なるために，従来のプロトコル設計が効率的であるかは不明である．そこで ioDrive を前提とした WAL プロトコルを設計する準備として，ioDrive の性能特性を示す．

3.1 ストレージデバイス

HDD（Hard Disk Drive）は一般的な記録デバイスである．HDD の中には磁性体を塗布したプラッタと呼ばれる円盤が回転軸に数枚取り付けられており，磁気ヘッドによってデータの読み書きを行う．読み書きを行う前には，磁気ヘッドを目的のデータが存在するトラック上にシークさせ，ヘッドと読み書き開始位置が重なるまでディスクが回転するのを待つ必要がある．この機械的な動作のため，HDD の I/O レイテンシは大きい．さらにランダムアクセスが起こると，より多くのシーク時間や回転待ち時間を必要とするため性能が極端に劣化する．

ioDrive は PCI Express 接続型のフラッシュストレージである．通常ストレージデバイスは SATA などのサウスブリッジに接続するものが多いが，PCI Express はノースブリッジに接続するため，CPU に近いところに位置する．また Nehalem 以降の Intel CPU では，PCI Express コントローラが CPU に内蔵されており，CPU が直接 ioDrive にアクセスできる．またフラッシュメモリ上の論理空間と物理空間のマッピングを，Virtual Storage Layer（VSL）と呼ばれるドライバによって，ホスト上の CPU を使って変換を行うことにより，低いレイテンシを実現している．ioDrive へ書き込みを行うスレッド数が 1 の時，ランダムライトがシーケンシャルライトと同等の性能であることが [20] に示されている．

3.2 ioDrive の基本性能評価

シーケンシャルライトとランダムライト（スレッド数＝1 and スレッド数＝ 16）を行い，ioDrive の基本性能を評価する．実験環境を**表 1**，実験結果を**図 2** に示す．ブロックサイズが大きくなるにつれ，スループット性能は向上している．シーケンシャルライトとランダムライト（スレッド数＝ 1）の性能差はほとんどない．ブロックサイズが小さい場合，ランダムライト（スレッド数＝ 1）と比べて，ランダムライト（スレッド数＝ 16）は最大 6.13 倍のスループット性能を発揮する．このように，ioDrive においては並列ランダムライトがシーケンシャルライトよりも高いスループット性能を持つ．

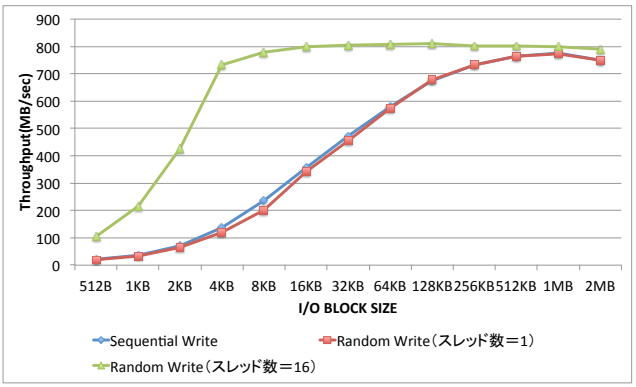

図 2 ioDrive の書き込み性能評価

4. *P*-WAL：高性能ランダムライトを活用した高性能並列 WAL

4.1 従来方式の WAL の問題点

従来方式の WAL プロトコルで問題となるのがログレコードを WAL バッファに挿入する際のロック（Algorithm 1 - 1 行目）と WAL ファイルへの I/O（Algorithm 1 - 7 行目）である．HDD などの I/O が低速なデバイスでは，トランザクション処理の最大のボトルネックは，デバイスの I/O 性能だと考えられていた．高い I/O 性能を持つ ioDrive を使用することによって，I/O のボトルネックは解消されるが，ロックの競合という新しいボトルネックが顕在化する．また，ログのシーケンシャル追記方式は I/O がシリアライズされていることを前提とするため，マルチコアのワーカースレッドモデルを活用することが難しい．

4.2 *P*-WAL

ioDrive の並列ランダムライトの高性能性を活用すべく，ワーカースレッド毎に WAL バッファと WAL ファイルを割り当てる WAL プロトコル-*P*-WAL を提案する．**図 3** に *P*-WAL 方式の全体像を示す．図 1 と異なる点は，従来一つだけであった WAL バッファと WAL ファイルをワーカースレッドの数だけ分割した点である．

4.2.1 WAL バッファの分割

従来方式における WAL バッファは一つである．そのため，ログレコードを WAL バッファに挿入する際に衝突が頻発する．そこで衝突を緩和させるべく，ワーカースレッド毎に専用の WAL バッファを用意する．各 WAL バッファ

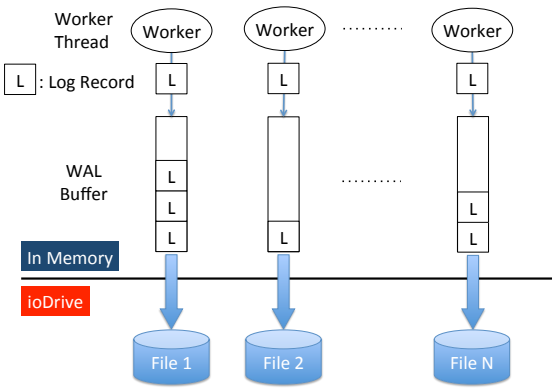

図 3 *P*-WAL のアーキテクチャ

Algorithm 2 log_insert(log,buffer_id)

```
1: #buffer_id で挿入する WAL バッファを指定
2: LSN ← WALbuffer[buffer_id].insert(log)
3: if log.Type == 'COMMIT' then
4:     #コミットログが挿入された場合，カウントアップ
5:     WALbuffer[buffer_id].ncommit ← WALbuffer[buffer_id].
       ncommit +1
6:     if WALbuffer[buffer_id].ncommit == NGROUP or WAL-
       buffer[buffer_id].full() then
7:         WALbuffer[buffer_id].flush() #WAL バッファの内容を
           WAL ファイルに書き込む
8:     end if
9: end if
10: return LSN
```

への挿入は対応するワーカースレッドのみが行うので，それぞれのワーカースレッドはロックを必要とせずに，ログレコードを WAL バッファに挿入できる．

4.2.2 WAL ファイルの分割

ログレコードの書き込み先である WAL ファイルは WAL バッファと一対一で対応させる．そのため，*P*-WAL における WAL ファイルの分割数は WAL バッファの数と等しくなる．これにより，ログレコードの挿入処理と同様に，それぞれのワーカースレッドはロックを必要とせずに，WAL バッファの内容を WAL ファイルへ書き込むことができる．

4.2.3 ログレコードの挿入アルゴリズム (log_insert)

P-WAL における，トランザクションのログレコードを WAL バッファに挿入する手続き log_insert を Algorithm 2 に示す．従来の log_insert_with_lock（Algorithm 1）と異なる点は，ロックを必要とせずにログレコードを並列に WAL バッファに挿入する点である．新たな引数 buffer_id で，ログの挿入先の WAL バッファの番号を指定する．buffer_id で指定された WAL バッファにログを挿入する（2 行目）．グループコミットのグループ数に達したか，あるいは WAL バッファが一杯になったかをチェックして（6 行目），もしそうであれば，WAL バッファのログレコード群を WAL

Algorithm 3 next_lsn()

```
1: repeat
2:     old ← global_LSN
3:     new ← old+1
4: until CAS（global_LSN,old,new） is returned with success
5: return old
```

ファイルに書き込む（7 行目）．

4.3 提案するリカバリプロトコル

P-WAL は WAL ファイルを分割する．これにより，従来のリカバリプロトコルが使用不能になる．ナイーブな解決策は全ファイル中の N 件のログレコードの整列だが，これには O（NlogN）の高いコストを要する．そこでマージ方式を提案する．

4.3.1 各ログの順序の決定

従来の WAL 方式では共通の WAL バッファにログレコードを挿入し，それをそのまま WAL ファイルにシーケンシャルに追記する．そのため，リカバリ時には WAL ファイルの先頭からログを読んでいけば，時系列順にログを処理できる．一方，*P*-WAL 方式では，ワーカースレッドの数だけ WAL ファイルが生成される．各 WAL ファイル内のログレコードの順序関係は，末尾に近い方が新しい一方，複数の WAL ファイル間でのログレコードの順序関係は不明である．そこで LSN（Log Sequence Number）により，WAL ファイル間でのログの順序関係を解決する．LSN とは単調増加するログの ID[3] である．

リカバリの際は，LSN の値が小さい順にログを処理する．従来方式の WAL では，この LSN の値が WAL ファイル上の位置を示す役割も兼ねる．*P*-WAL で発行される LSN はログのファイル上の位置を示すものではないため，ワーカースレッドは格納先の WAL ファイルの番号とファイル中のオフセットを LSN の付属情報としてログに記録する．

4.3.2 LSN アクセスの高性能化

ログレコードの LSN は，WAL バッファへの挿入時に，共有変数 global_LSN にアクセスして取得する．この global_LSN へのアクセスのため，スレッド間で衝突が発生する．global_LSN への最も単純なアクセス方法は，ロックの利用だが，この方法だとロックの競合による性能劣化が大きい．そこで，ロックを使わずに CAS（Compare And Swap）[21] を使用する．

CAS を使って LSN の取得と値のインクリメントを行う処理 next_lsn を Algorithm 3 に示す．まず，現在の global_LSN を変数 old に読み込む（2 行目）．先ほど読み込んだ global_LSN の値にプラス 1 した値を変数 new に読み込む（3 行目）．「CAS(global_LSN,old,new)（4 行目）」は，global_LSN と old の値を比較して，値が同じであれば global_LSN の値を new に変えた後に success を返す．この

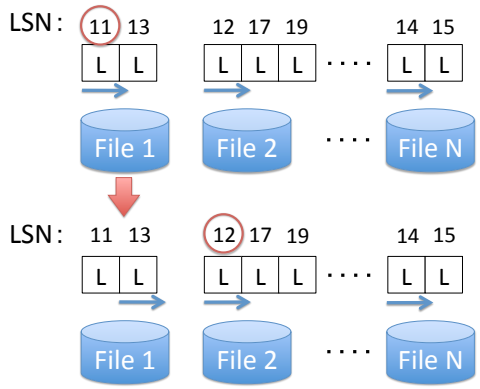

図 4 マージ処理によるリカバリ

Algorithm 4 redo()

```
1: while 未処理のログが残っている do
2:     min_LSN ← ∞
3:     selected ← -1
4:     for i = 1 to N do
5:         log ← WAL_file[i].head() #先頭ログを取り出す（WAL
           ファイル上のポインタは進めない）
6:         if log.LSN < min_LSN then
7:             min_LSN ← log.LSN
8:             selected ← i
9:         end if
10:    end for
11:    if selected ≠ -1 then
12:        log ← WAL_file[selected].next() #先頭ログを取り出す
           （WAL ファイル上のポインタを一つ進める）
13:        process(log)
14:    end if
15: end while
```

時，global_LSN と old の値が違っていれば failure を返す．failure の場合には 2 行目に戻って再度 global_LSN を読み込んで，成功するまで CAS を試みる．最後に，取得した LSN の値を呼び出し元に返す（5 行目）．

4.3.3 マージ方式

この方式の前提となる条件は，各 WAL ファイル内のログレコードが時系列順になっていることである．各 WAL ファイルの先頭ログレコードの LSN を見て，LSN の値が最も小さいものから処理する．

図 4 にマージ処理の流れを示す．まず，各 WAL バッファの先頭にある LSN=11, LSN=12, LSN=14 のログレコードの中で，最も値の小さい LSN=11 のログレコードを選択して，処理する．処理した LSN=11 のログレコードをバッファから取り出すと，LSN=13 のログレコードが File 1 の次の先頭のログレコードとなる．このように各 WAL ファイルの先頭のログレコード同士を比較し，LSN の小さい順にログを処理することで障害発生前までの歴史を繰り返す事ができる．

マージ方式によってログを時系列順に適用する redo 処理を Algorithm 4 に示す．for 文（4-10 行目）の中で，各 WAL ファイルの先頭ログレコードの LSN を比較して（6 行目），LSN の最小値を更新したら，そのログレコードを持つ WAL ファイルの番号を変数 selected に読み込む（8 行目）．次に処理するログレコードが決定すると，WAL ファイル上のポインタを一つ先に進めて（12 行目）．取り出したログレコードを処理する（13 行目）．これを繰り返し，処理すべきログが無くなるまでマージ処理を繰り返す．

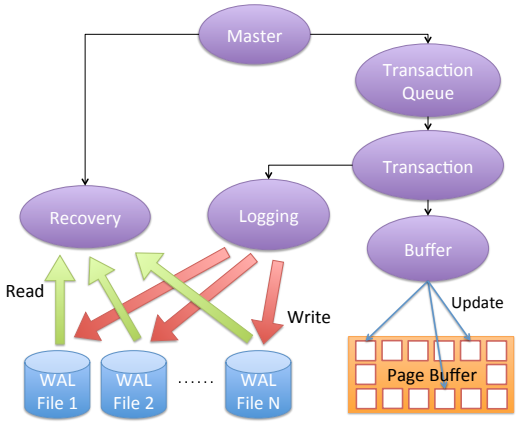

図 5 モジュール構成図

5. 設計と実装

この節ではプロトタイプのトランザクションマネージャの設計と実装を説明する．プロトタイプのトランザクションマネージャは機能毎に master，buffer，transaction queue，transaction，logging，recovery のモジュールから構成される．モジュールの構成を**図 5** に示す．楕円が各モジュールを表し，立方体が WAL ファイルを表す．黄緑の矢印は WAL ファイルの読み込み，赤の矢印は WAL ファイルへの書き込みを表す．黒の矢印はモジュール間の作用，青の矢印はメモリ上のページバッファへの更新を表す．

5.1 Master モジュール

本モジュールは，プログラムの起動時に前回の終了時の状態を確認し，正常終了だった場合は，transaction queue モジュールに実行するトランザクションの件数を渡す．異常終了だった場合は，最初にリカバリを行ってからトランザクション処理を開始する．

プログラム起動時に，トランザクションの数やスレッド数（WAL バッファ数），グループコミットに使うグループ数を指定して，トランザクション処理を開始する．前回終了時の状態はマスターレコードという特別なレコードの last_exit というフィールドに記録する．プログラムの起動処理でマスターレコードの last_exit フィールドを false にして書き込む．正常に終了されるときは必ずこのレコードの last_exit フィールドを true に設定する．

5.2 Transaction Queue モジュール

本モジュールは処理すべきトランザクションをキューに入れて管理する．キューにトランザクションを追加するクリエイターと，キューからトランザクションを取り出し，処理を行うワーカーがある．ワーカー毎にキューは存在する．クリエイターは各キューに空きがあれば，そのキューにトランザクションを追加し，ワーカーはキューにトランザクションがあれば，それを取り出して実行する．

本システムは起動時にトランザクションを，まとめて各キューにセットする．そのため，クリエイターがキューに書き込むのは初期化時のみである．このように実装することにより，初期化時以降クリエイターはこのキューを覗くことは無いので，クリエイター・ワーカー間の衝突コストは存在しない．

5.3 Transaction モジュール

本モジュールは，XID（トランザクション ID）の発行と，トランザクションの各オペレーションを実行する．また，トランザクションテーブル [22] を管理する．

トランザクションテーブルは，各トランザクションが途中でアボートされた場合に次に Undo するログレコードの LSN や，そのトランザクションが最後に発行したログレコードの LSN などのトランザクションの状態を保持する．

5.4 Buffer モジュール

本モジュールは，更新対象となるページをバッファ内で管理する機能を提供する．排他制御を行って同じページに対する異なるトランザクションからの更新を制御している．並行実行制御方式として strict 2-phase lock（S2PL）方式を採用している．トランザクションによって更新されたページはダーティページテーブル [22] のエントリに追加される．

全ページを主記憶上に載せている．ページバッファはページの配列として実装されている．ページはページヘッダとページボディからなる．ページヘッダは PageID と PageLSN から構成される．PageID はページを一意に特定する ID である．PageLSN は最後にこのページに更新を行ったログの LSN である．また，ページボディに一つの整数オブジェクトを書き込める．ダーティページテーブルは，チェイン法のハッシュテーブルで実装されている．ダーティページテーブルへのロックの粒度はバケット単位で行われる．S2PL の実装には pthread ライブラリの read-write ロックを使用している．

5.5 Logging モジュール

本モジュールは，ロギングに関する機能を提供する．設計の詳細は 4.2 節で述べた．

ログレコードは自身の LSN の他に，更新前のページの値や更新後のページの値，ログレコードの位置を示す WAL ファイルの ID とファイル中のオフセット，ロールバック時にこのログレコードを Undo した後で次に Undo するログレコードの LSN を含む．

5.6 Recovery モジュール

本モジュールでは，リカバリに関する機能を提供する．リカバリは ARIES[13] に基づき，analysis パス，redo パス，undo パスの三つのパスが順に行われる．ログの読み込みに関しては，4.3.3 節で述べたマージ方式に従って，各 WAL ファイルからログをブロックで読み込み，各 WAL ファイル間で先頭のログレコードの LSN を比較し，LSN が小さいログレコードを順に取り出す．このマージ方式によって，WAL ファイルが分割されていてもリカバリが可能である．

本システムは，チェックポイントを実装していない．そのため，Analysis pass は必ずログの先頭から開始される．

5.7 WAL ファイル

本実装では，WAL ファイルとしてデバイスファイルを用いている．デバイスファイルは Linux では「/dev/」以下に存在している．デバイスファイルへの書き込みは，ファイルシステムへの書き込みとは異なり，どこからどこまでを 1 つのファイルの範囲として，自分で決めたルールに従って管理しければならない．ファイルシステムを介した書き込みに比べて，デバイスファイルへの直接書き込みは，ファイルシステム上のメタデータ更新やキャッシュをバイパスできるので，デバイスの性能上限に近い基本 I/O 性能を期待できる．

今回，一つのデバイスファイルを複数の WAL ファイルとして扱うために，一定のオフセット毎をファイルの区切りとし，区切りの先頭を各ファイルの先頭と捉える．例えば 10GB をファイルの区切りとすると，WAL ファイル 1 はデバイスファイルのオフセット位置 0 から書き込み，WAL ファイル 2 はデバイスファイルのオフセット位置 10GB から書き込むというように，それ以降も 10GB 単位でファイルの先頭位置を決めていく．そのため，書き込み時は一つのファイルが 10GB を超えないよう注意が必要である．

デバイスファイルへログを書き込む場合，ファイルシステム上のファイルへの書き込みとは異なり，ファイルの長さに関する情報を明示的に書き込む必要がある．そのため，ログを書き込む際は常に，ログ本体だけでなくログヘッダへの書き込みも発生する．

また，ログヘッダだけを更新すると本来存在しない不正な領域を読み込む可能性があるため，ログ本体を書き込んだ後にログヘッダを書き込まなければならない．ログヘッダの書き込みに成功して初めてクライアントにコミットの成功を伝えるので，ログヘッダの書き込みに失敗すると，

表 2　実験環境 (2)

CPU	Intel（R） Xeon（R） CPU E5620 × 2
コア数	4 ×2
メモリ	24GB
ioDrive	SLC, 160GB, VRG5T VSL v3.3.3, Low-Level Formatting

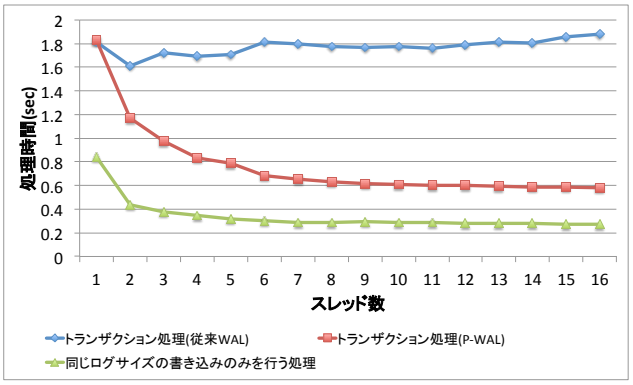

図 6　WAL と *P*-WAL の比較: 処理時間

トランザクションはアボートされたものとして，書かれたログ本体は捨てられるが，このログは無かった物として扱われるので不整合は起こらない．

一方，ファイルシステム上のファイルへログを書き込む場合，ファイルの書き込みに伴って自動的にファイル長の情報も更新されるので，実装がシンプルになるという利点がある．

6. 評価実験

実験環境を**表 2** に示す．

6.1 WAL と *P*-WAL の比較実験

6.1.1 実験内容

この実験ではトランザクション十万件の処理性能を計測して，従来 WAL（従来方式の WAL），*P*-WAL，同じサイズのログを書き込むだけのプログラムを比較する．ここで一つのトランザクションは UPDATE を一回行うものとする．全てのトランザクションは（BEGIN, UPDATE, END）の三つのログを生成する．各ログの大きさは固定長で 512（Bytes）である．1 トランザクションにつき 512×3（Bytes）のログを生成する．十万件のトランザクションで合計 512×3×100,000（Bytes）=約 146MB のサイズのログが書き込まれる．グループコミットのグループ数は 16 とした．すなわち 16 トランザクションのログを一括してストレージに書き込む．

6.1.2 実験結果

処理時間を比較した実験結果（**図 6**）と，スループットを比較した実験結果（**図 7**）を示す．図 6 において，トランザクション処理を行わずにログの書き込みのみを行うプログラムが ioDrive 一台の性能の上限と考えられる．従来

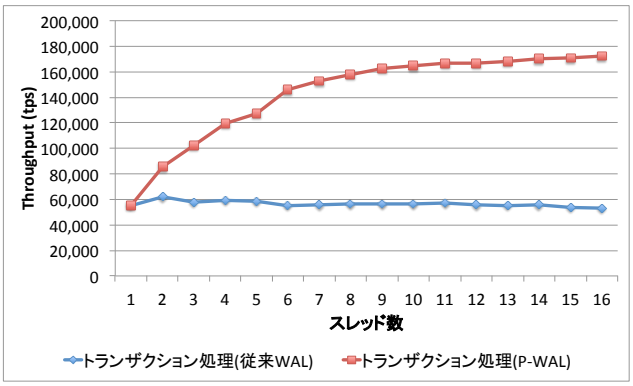

図 7　WAL と *P*-WAL の比較: tps

WAL ではスレッド数が 1 から 2 になった時に，僅かな性能向上が見られるが，全体的な傾向としてはスレッド数が多くなるほど性能が低下している．一方，*P*-WAL では，スレッド数の増加に伴って性能向上が見られる．スレッド数が 16 の時のスループットは，従来 WAL は 53,116（tps）で，*P*-WAL は 172,018（tps）となり，3.23 倍の性能向上が達成された．

従来 WAL のスループットがスレッド数の増加に伴って向上しない原因として，ログの挿入がトランザクションのボトルネックになっていると考えられる．一方，*P*-WAL はログの挿入がワーカースレッド間で衝突せず，ログの書き込みによる I/O を並列化しているため，スレッド数の増加に伴って性能が向上していると考えられる．同じログサイズの書き込みのみを行うプログラムと *P*-WAL の性能差は，ログ本体以外のログヘッダ（512 バイト）の書き込みやバッファの更新などのトランザクション処理が原因であると考えられる．一般的にはスレッド数が実コアの数よりも多くなると，スレッドの切り替えのオーバーヘッドによって性能が低下する傾向にあるが，使用したマシンのコア数 8 を超えても性能が向上している．これは I/O 待ち時間を他のスレッドがバッファの更新処理やログレコードの作成・挿入処理で CPU を利用できているためだと考えられる．

6.2 グループコミットのグループ数を変える実験

6.2.1 実験内容

この実験では *P*-WAL 方式において，グループコミットのグループ数を変えると性能にどのような影響を与えるのかを調査する．6.1 節と同じく，トランザクション十万件を処理し，一つのトランザクションは UPDATE を一回行い，（BEGIN,UPDATE,END）の三つのログを作成する．

6.2.2 実験結果

図 8 にスループット性能を表した実験結果を示す．スレッド数に関わらず，グループ数の増加に伴って性能が向上している．

グループ数が大きくなると，一度に WAL ファイルに書

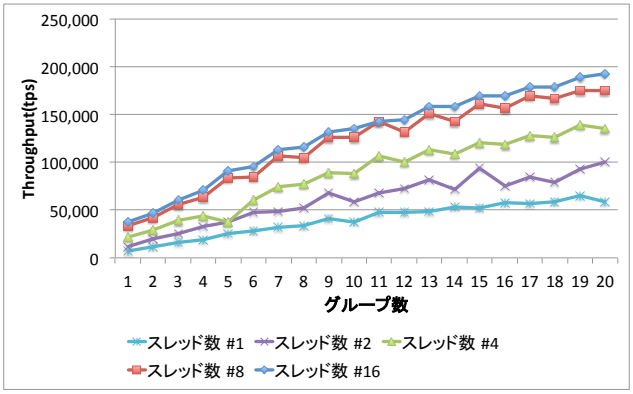

図 8 グループ数の比較: tps

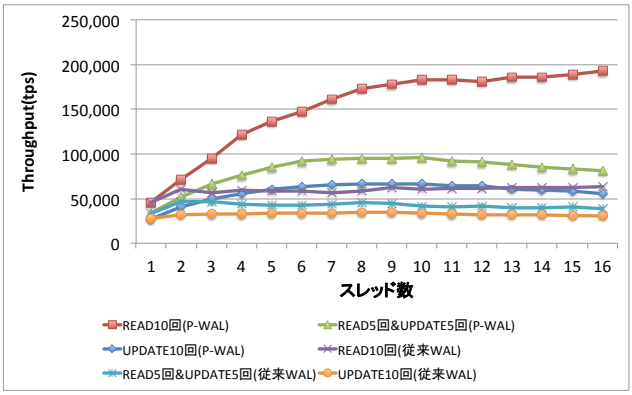

図 9 3 種類のトランザクションの比較: tps

き込むログサイズが大きくなり，書き込みの回数が減る．しかし，END ログがストレージに書き込まれるまでクライアントにコミットの完了を通知できないという制約があるため，グループ数を大きくした分，他のトランザクションのコミットを待たなければいけない．そのため，トランザクションによっては，グループコミットによってコミットの時の遅延が大きくなるものもあると考えられる．

クライアントサーバモデルのトランザクションマネージャを設計する際には，グループ数による性能向上とトランザクションのコミットのレイテンシでうまく兼ね合いをとってパラメータを決める必要がある．

6.3 トランザクション中の READ と UPDATE の比率を変える実験

6.3.1 実験内容

この実験では，トランザクションの READ オペレーションと UPDATE オペレーションの比率を変えると性能にどのような影響を与えるのかを調査する．READ はメモリ中のページバッファから 1 ページを読み込む処理で，UPDATE はメモリ中のページバッファの 1 ページを更新する処理である．またページバッファの排他制御に READ は読み込みロック，UPDATE は書き込みロックを用いている．トランザクションのオペレーション数を 10 として，トランザクション十万件を処理した際の処理性能を以下の 3 種類のトランザクションについて評価した．グループコミットのグループ数は全て 16 であり，() 内に各トランザクションが生成するログの種類と数を示す．READ10 回のトランザクション（BEGIN,END），READ5 回&UPDATE5 回のトランザクション（BEGIN,UPDATE×5,END），UPDATE10 回のトランザクション（BEGIN,UPDATE×10,END）．

6.3.2 実験結果

P-WAL と従来 WAL でトランザクションの種類とスレッド数を変えた時の実験結果を**図 9** に示す．スレッド数 8 の時の結果について注目して，トランザクションの種類毎に *P*-WAL と従来 WAL を比較したものを**図 10** に示す．

スレッド数が 8 の時（図 10），*P*-WAL は従来 WAL と

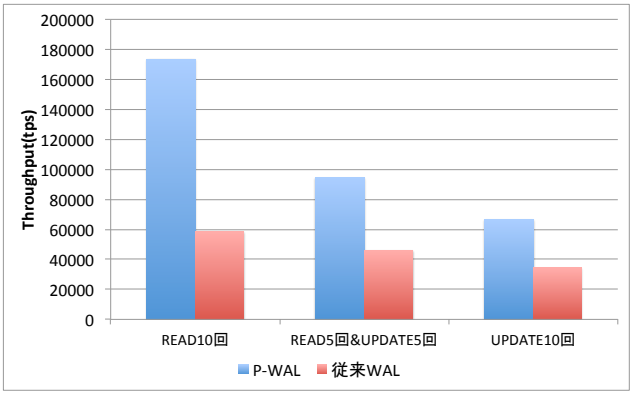

図 10 8 スレッド時の 3 種類のトランザクションの比較: tps

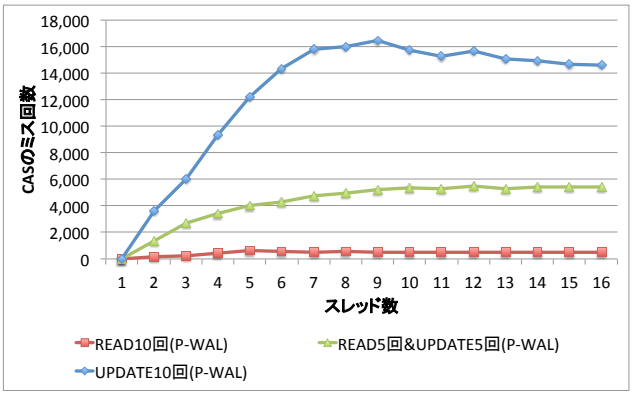

図 11 3 種類のトランザクションの比較: CAS のミス回数

比較して，READ10 回のトランザクションでは 2.95 倍，READ5 回&UPDATE5 回のトランザクションでは 2.06 倍，UPDATE10 回のトランザクションでは 1.93 倍性能が向上した．UPDATE の比率が多くなるにつれ，性能向上率が小さくなっているが，これは CAS のミス回数の増加が関係している可能性がある．

P-WAL 方式において，この時の CAS のミス回数を計測した結果を**図 11** に示す．UPDATE の比率が多いトランザクションほど，スレッド数を増加させたときに CAS のミス回数の増加率が高い．CAS に失敗すると，CAS を再実行しなければいけないため，トランザクション処理において性能劣化の要因となりうる．

7. 関連研究

WAL を高性能化する研究は 2 種類に分類される．ARIES スキームに基づきシーケンス番号を使いながら高性能化を図る研究と，シーケンス番号を使わずに WAL を再設計する研究である．

ARIES スキームに基づく研究には Aether[11] , Deuteronomy[14][15]，そして分散ロギング [16] がある．Aether はロック解放タイミングの早期化，グループコミット待ち処理の非同期化，そして WAL バッファへのログ挿入処理の競合緩和を行う．Deuteronomy はトランザクションコンポーネントとデータコンポーネントを分離して非モノリシックな設計を用いる．この柔軟な設計によりトランザクショナル KVS，アトミック KVS，ページストレージエンジン等の様々なシステムが容易に構成される．分散ロギングは論理クロックを用いながらページ ID とトランザクションの ID を組み合わせてグローバルシーケンス番号（GSN）として管理することで，分散ロギングを実現する．これらの研究は ARIES の WAL スキーム [13] に基づいて行われており，いずれもシーケンス番号を利用する．多くの既存システムは ARIES を採用しているため，これらの成果を既存システムへ導入することは本研究成果同様に容易だと考えられる．これらの既存研究と本研究との違いは並列 I/O の活用にある．既存研究は本研究とは異なり WAL バッファの WAL ファイルへの移送を並列化しない．本研究では各ワーカースレッドに WAL バッファと WAL ファイルを専用的に持たせる設計により，ログ書込み処理の並列性を高めている．

WAL を再設計する研究には Silo[17][18] と FOEDUS[19] がある．Silo と FOEDUS はシーケンス番号を使わない．その代わりに時区間 (epoch) と楽観的実行制御 [23] を組み合わせる．この方式は同一時区間におけるログの順序を非決定とする．この問題はページアクセス衝突に伴うアボートにより防がれている．ただし，それにより平均遅延が悪化する可能性がある．これらの研究はいずれも ARIES スキームを大幅に修正する必要がある．従って本研究成果とは異なり，これらの成果を既存システムへ導入することは困難だと考えられる．

8. 結論と今後の課題

本研究では，ioDrive の高並列度におけるランダムライトの高性能性を示し，ログ先行書き込みを並列化した *P*-WAL のプロトコルを提案した．*P*-WAL と従来の WAL を使ったプロトタイプのトランザクションマネージャを設計・実装し，評価を行ったところ，スレッド数が 16 の時に *P*-WAL は約 17 万 tps の性能を発揮し，従来の WAL に比べて 3.23 倍のスループット性能を達成した．

今後の課題は LSN アクセスの効率化，S2PL よりも効率的な並行実行制御プロトコルの導入，現実的なベンチマークの利用，そしてストレージデバイスの更なる活用である．

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 析による統計計算宇宙物理学」，科研費「#25280043HA」による．

参考文献

[1] NYSE. NYSE, New York Stock Exchange > About Us > News & Events > News Releases > Press Release 06-03-2009:. http://www1.nyse.com/press/1244024115279.html.（アクセス日: 2015-04-15）.

[2] MasterCard. Processing: Brilliance Behind the Scenes of Commerce — MasterCard. http://www.mastercard.com/us/company/en/whatwedo/processing_brilliance_behind_commerce.html.（アクセス日: 2015-04-15）.

[3] Jim Gray. The Transaction Concept: Virtues and Limitations. VLDB, pp. 144–154, 1981.

[4] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory OLTP recovery. ICDE, pp. 604–615, 2014.

[5] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery Write-Ahead System for In-Memory Non-Volatile Data-Structures. PVLDB, Vol. 8, No. 5, pp. 497–508, 2015.

[6] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. PVLDB, Vol. 8, No. 4, pp. 389–400, 2014.

[7] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: transaction support for next-generation, solid-state drives. SOSP, pp. 197–212, 2013.

[8] D. Gawlick, J.N. Gray, W.M. Iimura, and R.L. Obermarck. Method and apparatus for logging journal data using a log write ahead data set, March 26 1985. US Patent 4,507,751.

[9] PostgreSQL. PostgreSQL: The world's most advanced open source database:. http://www.postgresql.org/.（アクセス日: 2015-05-01）.

[10] Oracle. Oracle — Hardware and Software, Engineered to Work Together:. http://www.oracle.com/index.html.（アクセス日: 2015-05-01）.

[11] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: A Scalable Approach to Logging. PVLDB, Vol. 3, No. 1, pp. 681–692, 2010.

[12] Fusion-io. Application Acceleration Enterprise Flash Memory Platform — Fusion-io:. http://www.fusionio.com/.（アクセス日: 2015-04-15）.

[13] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. ACM Trans. Database Syst, Vol. 17, No. 1, pp. 94–162, 1992.

[14] Justin J. Levandoski, David Lomet, Mohamed F. Mokbel, and Kevin Zhao. Deuteronomy: Transaction Support for Cloud Data. CIDR, pp. 123–133, 2011.

[15] Justin Levandoski, David Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. High Performance Transactions in Deuteronomy. CIDR, 2015.
[16] Tianzheng Wang and Ryan Johnson. Scalable Logging through Emerging Non-Volatile Memory. PVLDB, Vol. 7, No. 10, pp. 865–876, 2014.
[17] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. SOSP, pp. 18–32, 2013.
[18] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. OSDI, pp. 465–477, 2014.
[19] Hideaki Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. SIGMOD Conference, 2015.
[20] Xiangyong Ouyang, David W. Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. Beyond Block I/O: Rethinking Traditional Storage Primitives. HPCA, pp. 301–311, 2011.
[21] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., Vol. 13, No. 1, pp. 124–149, 1991.
[22] Raghu Ramakrishnan and Johannes Gehrke. Database Management Systems. McGraw Hill Higher Education, 2002.
[23] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. ACM Trans. Database Syst., Vol. 6, No. 2, pp. 213–226, 1981.