並列WALにおける共有カウンタの競合緩和化

神谷 孝明[1,a)]　星野 喬[2]　川島 英之[3]　建部 修見[3]

概要：
トランザクション処理において，Atomicity と Durability を保証する代表的な方法に WAL（Write Ahead Logging: ログ先行書き込み）がある．従来の WAL はストレージデバイスとして HDD を使うことを前提としており，ランダムライトを起こさないように，単一の集中型 WAL バッファを用いてシーケンシャルにログを追記書き込みしていた．しかし，CPU コア数の増加に伴いトランザクション処理の並列度が増すにつれ，この方式では，WAL バッファへのログレコード挿入時の競合や永続化のための書き込みの競合が性能劣化の要因となっていた．我々は以前，フラッシュストレージの書き込みの内部並列性を活用した並列 WAL プロトコルとして P-WAL を提案した．P-WAL ではログレコードの順序番号である LSN を，ログレコードの論理アドレスではなく，共有カウンタを用いて単調増加に割り当てることで，複数の WAL バッファを用いてログレコード挿入の並列化とストレージ書き込みの並列化を行った．しかし，メニーコア，NVRAM を想定した環境においては，トランザクション処理時間の中で WAL の永続化のための flush の時間が相対的に小さくなり，共有カウンタのインクリメントにおける競合がより問題になってくると考えられる．本稿は，一つのログレコード作成毎ではなく，WAL の永続化単位毎に共有カウンタにアクセスし，複数のログレコードに一つの順序番号を割り当てることで，競合を緩和する方式を提案する．本方式がクラッシュリカバリ可能であることを示し，Xeon Phi（60 cores，240 threads）を用いて，予備評価を行う．

1. はじめに

WAL（ログ先行書き込み）はトランザクションの原子性と永続性を保証するために使われる代表的な手法である．従来の WAL はストレージデバイスとして HDD を使うことを前提としており，ランダムアクセスを起こさないように，単一の集中型 WAL バッファを用いてログを一括してストレージデバイスに追記する方式を採用している．WAL はトランザクション処理性能を劣化させる要因の一つであるため，WAL の高性能化に関する研究は様々行われている．

WAL の高速化に関する研究には次のようなものがある．Aether[2] は Flush pipelining や Early Lock Release などのテクニックを提案している．これらのテクニックの一部は本研究にも適用可能である．Silo[9] や FOEDUS[3] はメニーコア環境においてはログレコード毎に一意に順序番号を割り当てるための共有カウンタはスケーラビリティを妨げるとして，Epoch と呼ばれる期間毎にまとめてトランザクションをコミットすることでスケーラビリティの高いロギング手法を提案している．しかし，Silo は高いスループットを得る代わりに，レイテンシを犠牲にしている．本研究は，Epoch を導入せずに，共有カウンタへのアクセス頻度を減らすことで，低レイテンシでかつ現実に必要となるワークロードにおいて十分なスループット性能を達成する．

我々は，以前フラッシュストレージの書き込みの内部並列性を活用した並列 WAL プロトコルとして P-WAL の提案・評価を行った [14] [15]．P-WAL ではログレコードの順序番号である LSN（Log Sequence Number）を，ログレコードの論理アドレスではなく，共有カウンタを用いて単調増加に割り当てることで，複数の WAL バッファを用いてログレコード挿入の並列化とストレージ書き込みの並列化を行った．これを GLSN（Global LSN）方式と呼ぶ．しかし，メニーコア環境においては共有カウンタのインクリメントにおける競合がより問題になってくると考えられる．そこで我々はログレコード毎に割り当てていた GLSN の代わりに，WAL 永続化時の書き込みのタイミングで割り当てる一つの GWSN（Global WAL-flush Sequence Number）という順序番号を複数のログレコードに割り当てることで，共有カウンタの競合を緩和する方式を提案する．

本稿の構成は以下の通りである．2 章では，以前提案し

[1] 筑波大学大学院 システム情報工学研究科
[2] サイボウズ・ラボ株式会社
[3] 筑波大学 計算科学研究センター
[a)] kamiya@hpcs.cs.tsukuba.ac.jp

© 2016 Information Processing Society of Japan

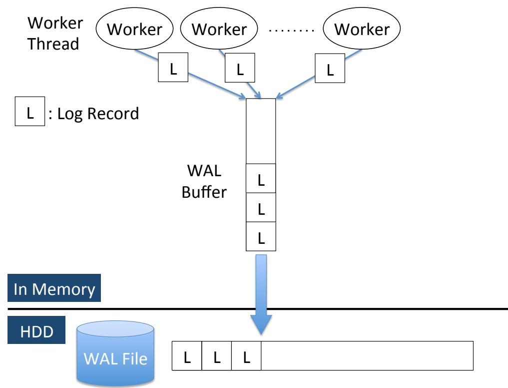

図 1 直列 WAL のアーキテクチャ

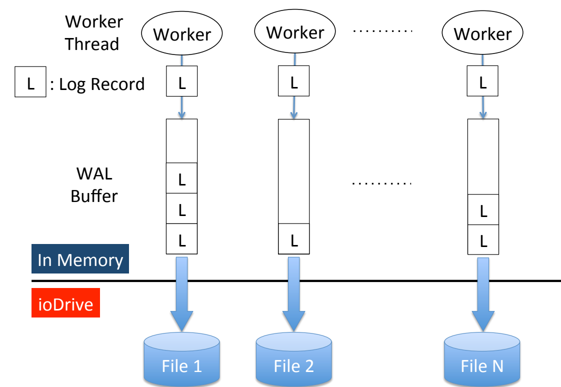

図 2 P-WAL のアーキテクチャ

た並列 WAL プロトコルである P-WAL について簡単に説明する．3 章では，GLSN の競合を緩和する GWSN 方式の概要を説明する．4 章では，トランザクションシステムを定義し，そこから GWSN 方式が WAL の目的とするリカバリ操作が可能であることを示す．5 章では，本方式の予備評価を行う．6 章では WAL にまつわるその他の関連研究について述べる．7 章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, WAL）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．WAL の目的は，トランザクションの原子性と永続性を保証して，システムの障害時にはログを用いてシステムを整合性のある状態にリカバリすることである．トランザクションシステム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．

2.1 直列 WAL

従来の WAL はストレージデバイスとして HDD を使うことを前提としており，ランダムアクセスを起こさないように，単一の WAL バッファを用いてログを一括してストレージデバイスに追記する方式を採用している．次の節で紹介する並列 WAL と比較して，以降この方式を直列 WAL と呼ぶ．直列 WAL のアーキテクチャを**図 1**（[14] より引用）に示す．直列 WAL の問題点は，WAL バッファがひとつしかないため，CPU コア数の増加に伴い処理するトランザクションの並列度が増すにつれ，ログレコードの WAL バッファ挿入時の競合やトランザクションのコミット処理による WAL バッファの書き出し時の競合が深刻になることである．

2.2 P-WAL

我々は以前，フラッシュストレージの書き込みの内部並列性を活用した並列 WAL プロトコルとして P-WAL を提案した．P-WAL ではログレコードの順序番号である LSN）をログレコードの論理アドレスではなく，共有カウンタを用いてログレコードの WAL バッファ挿入時に単調増加に割り当てる GLSN 方式を用いることで，複数の WAL バッファを用いてログレコード挿入の並列化とストレージ書き込みの並列化を行った．これにより，メモリ上の WAL バッファ挿入時の競合緩和, 及びストレージ上の WAL ファイル書き込み時の競合緩和と，並列書き込みによるフラッシュストレージの帯域の活用を行った．P-WAL のアーキテクチャを**図 2**（[14] より引用）に示す．ワーカースレッドと WAL バッファと WAL ファイルが一対一対一で対応している．P-WAL における制約の一つは，各ワーカーが一つのトランザクションにおいて最初から最後まで責任を持つことである．すなわち，同一トランザクションのログは全て同一 WAL バッファに挿入され，一対一で対応する WAL ファイルに書き込まれる．P-WAL における制約のもう一つが，トランザクションログを永続化してコミットするまでデータベースリソースへのロックを解放しないようにする S2PL（Strict 2-Phase Lock）のプロトコルを採用することである．S2PL により，WAL バッファにログが存在しているトランザクション間にはレコードアクセスの依存関係（read-after-write（w-r）, write-after-read（r-w）, write-after-write（w-w））がない．上記の二つの制約を組み合わせることで，複数存在する WAL バッファの一部が障害で消失し，永続化したログレコードの GLSN に歯抜けが生じたとしてても，リカバリが可能である．16 cores のマシンにおいて，P-WAL は 更新オペレーションが一回の One-shot transaction で，直列 WAL の 2.42 倍の性能を達成した [15]．

2.3 LSN と GLSN

直列 WAL では，LSN は以下の二つの役割を持つ．
(1) ログレコードの論理アドレス（ログレコードの格納先

アドレスと相互変換可能なもの）

(2) 順序番号

一方，並列 WAL においては，WAL バッファ，及び WAL ファイルが複数存在するため，役割毎に値を分割する必要がある．並列 WAL では，ログレコードが含まれる WAL ファイルの番号と，一つの WAL ファイル内で単調増加するログレコードの論理アドレスを記録することで，(1) と同等になるので，これをログレコードの格納先を示す値として使用する．(2) の順序番号はログレコードの WAL バッファ挿入時に単調増加する共有カウンタにより作成する．この順序番号はシステム障害の発生後，リスタートリカバリを行う時に，ログレコードを適用する順番になる．以降，(1) と (2) を明確に区別する時，(1) の役割を持つ値を LSN，全 WAL ファイルを含むシステム内で単調増加する (2) の役割を持つ値を GLSN（Global Log Sequenece Number）と呼び区別する．

共有カウンタである GLSN 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，GLSN のアクセス方法はアトミックに値のインクリメントを行う fetch-and-add 命令を利用する．

3. GLSN アクセスの競合緩和

3.1 概要

従来の P-WAL では，ログレコード毎に一意な GLSN を振っていた．メニーコア，NVRAM を想定した環境においては，トランザクション処理時間の中で WAL バッファデータの永続化処理（WAL-flush）時間が相対的に小さくなり，共有カウンタである GLSN の競合がより問題になってくると考えられる．そこで我々はログレコード毎に割り当てていた GLSN の代わりに，WAL-flush 毎に発行する GWSN（Global Write Sequence Number）という値を永続化対象の全ログレコードに割り当てる．GWSN は GLSN と同様に，単調増加する共有カウンタとして実現され，アトミックな fetch-and-add 命令でアクセスされる．一般に，一つのトランザクション（Tx）は複数のログレコードを作成し，WAL バッファのログはコミット時にストレージに書き出されるので，WAL-flush 回数 = Tx 数 <= ログレコード数 となる．また，複数の Tx をまとめてコミットする group commit[13] を使うと，WAL-flush 回数 ¡= Tx 数 ¡= ログレコード数 となるため，例えば 1 Tx が 10 log records を作成し，group commit によって 10 Txs がまとめてコミットされる場合，GLSN 方式では 100 回の共有カウンタのアクセスが必要になるが，GWSN 方式では 1 回のアクセスで済むようになり，従来の GLSN アクセスで引き起こされていた競合を大幅に緩和することができる．

3.2 GLSN から GWSN へ

一般に一つのログレコードに一つのオペレーション（データレコードのアップデートなど）が記録される．この時，ログレコード毎に GLSN を振ると，ログに記載される全てのオペレーション間で全順序が決定される．しかし，実際はオペレーション間に存在するのは半順序であり，全順序である必要はない．例えば同じページリソースを更新する二つのトランザクション A と B があり，A が先に更新した値を，B がその後に更新する時，A と B の該当するログレコードの GLSN をそれぞれ GLSN1，GLSN2 とすると，GLSN1 < GLSN2 であり，GLSN1，GLSN2 の順で更新を適用しなければならない．一方，A と B が異なるページリソースを更新する時，同様に A と B の該当ログレコードの LSN をそれぞれ GLSN1，GLSN2 とすると，GLSN1 と GLSN2 のログレコードの間に依存関係は存在しないため，GLSN（A）< GLSN（B）または GLSN（A）> GLSN（B）のどちらであっても問題はなく，GLSN がたまたま小さい値のログレコードから処理されることになる．すなわち依存関係（w-r, r-w, w-w）がないオペレーションについては，その適用順序を考慮する必要はない．このような依存関係がない GLSN1，GLSN2 のログレコードに個別にユニークな順序番号を振るのではなく，まとめて一つの順序番号を割り当てるというのが GWSN 方式の基本的なアイデアである．

各 WAL バッファはログレコードの挿入（領域確保）要求が来る度に，一つの WAL バッファ内における論理アドレスとしての LSN を割り当てる．コミット要求が来ると，GWSN の現在の値を取得し，インクリメントする操作をアトミックに行う．

取得した GWSN は WAL-flush 対象の先頭ログレコードに含めることで対象のログレコード列がひとつの WAL-flush に含まれ，その WAL-flush にひとつの GWSN を割り当てる操作となる．

4. リカバリ可能であることの証明

この章では GWSN 方式が WAL の目的であるリカバリ操作が可能なことを証明する．証明に先立って，まずはトランザクションシステムの用語の定義を行う．

4.1 定義

リソース

アクセスの最小単位とする．リソースに対するオペレーションには read/write がある．(insert/update/delete が write に含まれる．) ロックはリソース単位で取る．あるリソースは atomic に操作されるものとする．つまり，リソースひとつを見たとき，操作は直列化される．簡単のため，リソースのことを data record と呼ぶ．

© 2016 Information Processing Society of Japan

Tx（トランザクション）

複数リソースへのアクセス列．完了時に，commit か abort のどちらかになる．isolation level によって同時実行が制御される．分離性（Isolation）制約を満たすためにリソース単位でのアクセス順序が S2PL などを用いて制御される．

ログ

ログはリソースへの write アクセス履歴を記録したデータである．一般に一つの log record は，一つの Tx の一つ以上のアクセス履歴データを含む．Tx は一般に複数の log record を出力し，最後に commit log か abort log を出力する．log record には，複数の data record アクセスを含むが，同一 Tx のものである必要がある．log record は（Tx, LSN, GWSN, Data record contents）ら構成される（LSN, GWSN は後述）．Data record contents は before/after イメージとして，undo/redo 両方のログを含むものとする．before/after image は差分ログとは異なり，同一ログを複数回適用しても冪等性が保たれるものとする．

WAL バッファ，WAL ファイル

WAL バッファはメモリ上にあり，WAL ファイルは永続ストレージ上にある．WAL ファイルは管理の都合上複数のファイルに別れていることもあるが，単調増加する一つのログの列からなるため，ここではひとつの WAL バッファにひとつの WAL ファイルが対応しているものとし，まとめて WAL と呼ぶ．WAL バッファと WAL ファイルは一対一で対応しているため，まとめて WAL と呼ぶ．ここでは，並列 WAL を想定しているので，複数の WAL が存在する．各 WAL において，log record には単調増加する LSN が割り当てられる．LSN は各 WAL 内での log record の論理アドレスになっている．つまり，システムにおいてログは（WAL, LSN）で一意となる．Commit 要求時に WAL バッファに存在するログは WAL ファイルに書き出され，fsync で永続化される．これを WAL flush と呼ぶ．WAL flush により永続化されるログ列にグローバルに単調増加する GWSN を割り当てる．すなわち，ひとつの GWSN に対して，(WAL, [LSN1, LSN2]) が対応する．ただし，LSN1 < LSN2．WAL flush を W と置いたとき，その対象ログに commit log が含まれる Tx について，Tx in W と書く．W の内容は log record 単位で atomic に書き込まれるものとする（実装上はチェックサムなどで対応される）．

各 WAL について WAL flush 時の GWSN と，それに対応する LSN を保持する．全 WAL の中で flush 済みの GWSN が最小のもの min (GWSNs) を考え，`GWSN <= min (GWSNs)` となる（GWSN, LSN）のデータをひとつ以上保持し，古いものは捨てる．ある WAL において，与えられた GWSN を越えない最大の GWSN に対応する flush 済みの LSN は getPermanentLSN（WAL, GWSN）で取得できるものとする．

データベース本体

Data record 集合が，メモリ上にキャッシュとして，ファイル上に永続データとして存在する．Data record 複数をまとめて page に格納し，キャッシュと永続データのやりとりは page 単位で行う．Page の更新は適切に排他され atomic に行われるものとする．Page は，最後に更新されたアクセスに対応するログ情報（WAL, LSN）を保持する．更新は，メモリ上の page でまず実施し，その後ファイル上に書き出し，永続化する（write down）．Write down 時に page は atomic に書き込まれるものとする (double write[6] や undo page log などを用いる)．Background writer が存在し，page の write down を担う．

アクセス操作

Read 操作はデータベース本体の該当 page 上の data record 読む．Isolation level が serializable のときは s-lock（shared-lock）を伴いアクセスし，commit/abort 後に lock を開放する（S2PL）．Read committed や snapshot isolation のときは適切な version を読むものとする．Write 操作はまず WAL バッファにログレコードを書き，その後，メモリ上の page に書く．commit 時に，使用した WAL の flush を行う．Page の write down は background writer に任せる．メモリ上の page 上で，data record write 操作をするとき，x-lock（eXclusive-lock）を伴いアクセスし，commit/abort 後に lock 開放する（S2PL）．Isolation level によっては適切な version が付けられ，古い version の record が残され，適切に消されるものとする．それを実現する具体的な実装について，ここでは議論しない．デッドロックは適切に回避されるものとする（timeout，ユーザが lock order に気をつける，deadlock detection など）．

チェックポイント

minGWSN = min (GWSNs) を求め，条件 (後述) を満たす全ての page を write down し，その旨を checkpoint log として WAL に記録する．checkpoint log は WAL = 0 (最初のもの) を用いる．checkpoint log は，(LSN, minGWSN) を含む．

クラッシュとリカバリ

ある瞬間に，突然システムがダウンすること（クラッシュ）が想定され，そのとき，永続データ以外は失われる．書き込み操作中のデータは一部または全部が失われる．ただし，必要な単位での atomic 性は別途担保されていることに注意．クラッシュリカバリは，WAL に記録されたログを一部適用してデータベース本体を正常な状態に復帰することである．データベース本体が正常とは，(過去に遡って）有効な Tx のログを依存関係の順に全て適用した状態である．リカバリ対象のログ範囲は，ログの GWSN が最後のチェックポイントログに記録されている minGWSN < GWSN のものとなる．

有効なログとは，各 WAL において，過去から連続して完全に永続化されており，クラッシュ後も読めるものとする．WAL 内での歯抜けは許されない．有効な Tx とは，Tx の記録したログが全て有効で commit log または abort log を含む Tx を指す．有効か commit log を含む Tx は単に committed Tx と呼ぶ．それ以外の Tx は無効である．無効な Tx は，有効なログをひとつも含まないものと，有効なログを一つ以上含むが commit log も abort log も含まないものがある．有効な Tx だが abort log を含むものと，無効な Tx だが有効なログを一つ以上含む Tx は undo-required Tx と呼ぶ．Undo-required Tx は commit 前の Tx の更新をデータベースに適用した可能性のある Tx である．ログの適用とは，ログに含まれる更新内容をデータベース本体に対して redo または undo することである．Redo 時は，after image を用いて，データベース本体の data record を更新する．Undo 時は，before image を用いて，同様に data record を更新する．Commited Tx について LSN 順に各 log record の redo を実行し，undo-required Tx について LSN の逆順に各 log records の undo を実行する．

Tx の依存関係

Tx1, Tx2（Tx1 != Tx2）に対して，それぞれがアクセスした read と write の data record 集合を read-set, write-set と呼び，rs1, ws1, rs2, ws2 と書く．ある data record を Tx1 が read/write し，その後 Tx2 が read/write し，その関係が r-w, w-r, w-w のどれかだったとき（すなわち r-r は除く），当該 data record において Tx2 は Tx1 に x-x（r-w/w-r/w-w のどれか）依存していると言い，`Tx1 --> Tx2 @ (rec, x-x)` と書く．

Data record 集合 recs の各 rec に対して全て `Tx1 --> Tx2 @ (rec, x-x)` が成立するとき，`Tx1 --> Tx2 @ (recs, x-x)` と書くこととする．ただし，空集合{}について，`Tx1 --> Tx2 @ ({}, x-x)` は常に False とする．

```
rw-set := rs1 and ws2
wr-set := ws1 and rs2
ww-set := ws1 and ws1
```

とする．

Serializable isolation level を用いるときは，以下が成立する場合，Tx2 は Tx1 に依存していると言い，`Tx1 --> Tx2` と書く．

```
Tx1 --> Tx2 :=
 (rw-set or wr-set or ww-set) is not empty
  and (rw-set is empty or
       Tx1 --> Tx2 @ (rw-set, r-w))
  and (wr-set is empty or
       Tx1 --> Tx2 @ (wr-set, w-r))
  and (ww-set is empty or
       Tx1 --> Tx2 @ (ww-set, w-w))
```

また，data record 依存がまったくない場合，Tx1 – Tx2 と書く．

```
Tx1 -- Tx2 :=
(rw-set or wr-set or ww-set) is empty.
```

Read committed や snapshot isolation を用いるときは，r-w, w-r 依存を無視し，w-w 依存のみを考えて，Tx 依存と定義する．すなわち，

```
Tx1 --> Tx2 := Tx1 --> Tx2 @ (ww-set, w-w)
Tx1 -- Tx2 := ww-set is empty
```

W の依存関係

ある W1, W2（W1 != W2）について，以下のように `W1 -- W2` および `W1 --> W2` を定義する．任意の Tx1, Tx2（ただし，Tx1 in W1, Tx2 in W2）について `Tx1 -- Tx2` が成立するならば，`W1 -- W2` と書く．`Tx1 --> Tx2` もしくは `Tx1 -- Tx2` が成立するならば，`W1 --> W2` と書く．

並列 WAL における追加制約

上記のシステムにおいて，以下の制約を加える．

（C1）同一 Tx のログは同一 WAL を使う．

（C2）page に記録されている（WAL, LSN）に対して，`LSN <= getPermanentLSN(WAL, min(GWSNs))` を満たす場合は write down して良い．

（C3）クラッシュリカバリ時のログ適用は，GWSN が小さいものから順に行う．

リカバリ可能

リカバリ可能なトランザクションシステムとは，以下の 2 条件を全て満たすリカバリ操作を実行可能なシステムのことをいう．

- 有効な Tx が依存している Tx は有効．
- リカバリ完了後，データベース本体は committed Tx のみが依存関係の順に適用された状態．

4.2 証明

11 の補助命題から，GWSN 方式のトランザクションシステムがリカバリ可能であるという定理を導く．証明のリストは次からなる．

Lemma 1. 任意の Tx1, Tx2（Tx1 != Tx2）に対し，Tx1 --> Tx2，Tx2 --> Tx1，Tx1 -- Tx2 のうちひとつのみが成り立つ．

Lemma 2. 任意の W1, W2（W1 != W2）について，W1 --> W2, W2 --> W1, W1 -- W2 のうちひとつのみが成り立つ．

Lemma 3. commit log もしくは abort log が有効な Tx は有効．

Lemma 4. W1 --> W2 である W1, W2 について，W2 に含まれる Tx が 1 つでも有効ならば，W1 に含まれる全ての Tx が有効．

Lemma 5. 有効な Tx が依存している Tx は有効．

Lemma 6. Wi の GWSN を GWSNi と置く．任意の W1,

W2（W1 != W2）について，GWSN1 < GWSN2 ならば，W1 --> W2 or W1 -- W2.

Lemma 7. GWSN <= minGWSN を満たす全ての GWSN について，対応するログは永続化されている．

Lemma 8. データベース本体への変更を write down する前に該当ログは WAL ファイルに永続化される．

Lemma 9. 正常終了し，データベース本体への永続化も完了した Tx について，Commit した Tx は依存関係順に更新が反映されている．Abort した Tx の更新は反映されていない．

Lemma 10. 正常終了しなかった，もしくはデータベース本体への永続化が完了しなかった Tx について，クラッシュリカバリ時に，committed Tx は全てデータベース本体に依存関係順に反映される．それ以外の Tx はデータベース本体には全く適用されなかったことになる．

Lemma 11. リカバリ完了後，データベース本体は committed Tx が依存関係の順に適用された状態となる．

Theorem. 制約（C1）（C2）（C3）の制約を満たすトランザクションシステムはリカバリ可能である．

Lemma 1. 任意の Tx1，Tx2（Tx1 != Tx2）に対し，Tx1 --> Tx2，Tx2 --> Tx1，Tx1 -- Tx2 のうちひとつのみが成り立つ．

Isolation level が serializable の場合，S2PL のロックプロトコルにより，（rw-set or wr-set or ww-set）is not empty のとき，rec in（rw-set or wr-set or ww-set）を満たす rec について，Tx1 が先に lock を取った場合，Tx2 は Tx1 の commit 後しか lock を取れない．すなわち，

Tx1 --> Tx2 @（rec，r-w）if rec in rw-set
Tx1 --> Tx2 @（rec，w-r）if rec in wr-set
Tx1 --> Tx2 @（rec，w-w）if rec is ww-set

が成立する．また，dead-lock は避けられるものとしたので，Tx1 と Tx2 が実行できるためには，任意の rec in（rw-set or wr-set or ww-set）について，上記が成立する．よって，定義により，Tx2 --> Tx1 が成り立つ．（rw-set or wr-set or ww-set）is not empty かつ，ある 1 つの rec について Tx2 が先に lock を取った場合について，同様に，Tx1 --> Tx2 が成り立つ．（rw-set or wr-set or ww-set）is empty のとき，定義により Tx1 -- Tx2 である．上記の 3 つの条件は重複せず，条件の和が全集合である．よって，serializable の場合において，示された．

Isolation level が read committed or snapshot isolation の場合，ロックプロトコルにより，ww-set is not empty かつ rec in ww-set を満たす rec について，Tx1 が先に lock を取った場合，

Tx1 --> Tx2 @（rec，w-w）

が成立し，dead-lock が避けられた場合，任意の rec in ww-set について上記が成立するため，定義から Tx1 --> Tx2 が成立する．Tx2 が先に lock を取った場合は，Tx2 --> Tx1 が成立する．ww-set is empty のとき，定義より Tx1 -- Tx2 が成立する．上記 3 つの条件は serializable と同様に，重複せず，条件の和が全集合．よって，read committed or snapshot isolation の場合において，示された．

Lemma 2. 任意の W1，W2（W1 != W2）について，W1 --> W2，W2 --> W1，W1 -- W2 のうちひとつのみが成り立つ

Tx1 in W1, Tx2 in W2 となるような Tx1, Tx2 について，Tx1 --> Tx2 ならば，以下の操作は直列化される．

(1) Tx1 の commit/abort log 準備
(2) W1 flush
(3) Tx2 の commit/abort log 準備
(4) W2 flush

Tx3 in W1，Tx4 in W2，Tx4 --> Tx3 となるような Tx3, Tx4 が存在したとする．すると，同様に直列化される．

(1) Tx4 の commit/abort log 準備
(2) W2 flush
(3) Tx3 の commit/abort log 準備
(4) W1 flush

しかし，W1, W2 の順が逆になるため，矛盾する．よって，そのような Tx3, Tx4 は存在しない．上記から，以下が成り立つ．

(1) Tx1 in W1，Tx2 in W2，Tx1 --> Tx2 なる Tx1, Tx2 が存在した場合，W1 --> W2 となる．
(2) Tx1 in W1，Tx2 in W2，Tx2 --> Tx1 なる Tx1, Tx2 が存在した場合，W2 --> W1 となる．
(3) Tx1 in W1，Tx2 in W2, である全ての Tx1, Tx2 について，Tx1 -- Tx2 であった場合，W1 -- W2 となる．

Lemma 1. により，(1)（2）（3）の条件は重複せず，和が全集合となる．よって，示された．

Lemma 3. commit/abort log が有効な Tx は有効

（C1）により，commit/abort log が記録されている WAL に Tx のログは記録されている．commit/abort log は Tx ログの最後に出力される．各 WAL において，過去から連続して永続化されているログは有効である．すなわち，commit/abort log が有効であれば，それより過去の同一 WAL の全ログは有効．よって Tx の全ログは有効．すなわち，Tx は有効．

Lemma 4. W1 --> W2 である W1，W2 について，W2 に含まれる Tx が 1 つでも有効ならば，W1 に含まれる全ての Tx が有効

W1 --> W2 から，

W1 の flush 完了時間 < W2 の flush 開始時間

W2 に含まれる Tx がひとつでも含まれるため，クラッシュ時刻 ct は，

W1 の flush 完了時間 < W2 の flush 開始時間 < ct

© 2016 Information Processing Society of Japan

よって，W1 の永続化は完了している．すなわち，W1 に含まれる全ての commit log は有効．Lemma 3. により，W1 に含まれる全ての Tx は有効．

Lemma 5．　有効な Tx が依存している Tx は有効

任意の Tx1, W1（ただし Tx1 in W1，Tx1 は有効）に対して，定義より，任意の Tx2, W2（ただし T2 in W2）について Tx2 --> Tx1 ならば，W2 --> W1．Lemma 4. より，Tx1 が有効ならば，W2 に含まれる Tx2 は全て有効．よって，示された．

Lemma 6．　GWSN <= minGWSN を満たす全ての GWSN について，対応するログは永続化されている

i 番目の WAL で flash 済みの GWSN を GWSNi とする．すなわち，i 番目の WAL において GWSN <= GWSNi である任意の GWSN に対応するログは永続化されている．定義より minGWSN = min（GWSNi）for all i すなわち，GWSN <= minGWSN <= GWSNi for all i である任意の GWSN について，対応するログは永続化されている．よって，示された．

Lemma 7．　データベース本体への変更を write down する前に該当ログは WAL ファイルに永続化される

ある page に記録されている最新更新情報を（WAL1, LSN1）とする．また，LSN2 = getPermanentLSN（WAL1, minGWSN）と置く．（C2）より，各 page を write down して良い条件は，LSN1 <= LSN2．Lemma 6.　より，GWSN <= minGWSN を満たす GWSN は全て永続化されている．getPermanetLSN（）の定義により，LSN2 のログを含む GWSN を GWSN2 とすると，GWSN2 <= minGWSN である．すなわち，LSN <= LSN2 である LSN を持つログは全て永続化されている．すなわち，ログが永続化されていることが，各 page を writedown して良い条件に一致する．よって，示された．

Lemma 8．　データベース本体への永続化が全て完了した Tx について，Commit した Tx の更新はデータベース本体に依存関係順に反映されており，Abort した Tx の更新はデータベース本体に反映されていない

S2PL プロトコルにより，serializable, read-commited, snapshot-isolation のどの isolation level においても少なくとも write する data record には lock をかけてから write し，commit または abort まで lock を開放しない．このため，メモリ上の page には Tx の依存関係順に更新が反映されている．よって，background writer がメモリ上の page を write down した時，commit した Tx の更新においては，依存関係順にデータベース本体に適用したのと同じ結果になる．

Abort した Tx に関しては，background writer により，データベース本体を更新した write が存在する場合でも，Abort が完了するまで，その対象 records の lock を保持しており，undo を実行してから，abort log を書き，その後 lock を開放する．このため，abort Tx が依存する Tx は完了しており，abort Tx に依存する Tx はまだ lock 開放を待っている．Abort が完了した時点で，メモリ上の該当 page には abort した Tx の更新は反映されていない．Abort 完了後，background writer がメモリ上の該当 page を write down した時，Abort した Tx の更新はデータベース本体に反映されていない．

よって，示された．

Lemma 9．　Wi の GWSN を GWSNi と置く．任意の W1, W2（W1 != W2）について，GWSN1 < GWSN2 ならば，W1 --> W2 or W1 -- W2.

GWSN を割り当てるのは，WAL flush の実行直前．すなわち，W1 --> W2 ならば，以下の操作が直列化される．

(1) GWSN1 の割り当て
(2) W1 flush
(3) GWSN2 の割り当て
(4) W2 flush

よって，W1 --> W2 ならば GWSN1 < GWSN2．添字を入れ替えて，W2 --> W1 ならば，GWSN2 < GWSN1 対偶を取って，GWSN1 <= GWSN2 ならば not(W2 --> W1) GWSN1 != GWSN2 であるのと，Lemma 2. より，GWSN1 < GWSN2 ならば W1 --> W2 or W1 -- W2．よって示された．

Lemma 10．　データベース本体への永続化が完了しなかった Tx について，クラッシュリカバリ時に，committed な Tx は全てデータベース本体に依存関係順に反映される．それ以外の Tx はデータベース本体には全く適用されなかったことになる．

Lemma 7. より，Tx はデータベース本体への変更を write down する前に該当ログを WAL ファイルに永続化するため，以下の 4 つに場合に分けられる．

(1) WAL ファイルに Tx ログが全て永続化されており，commit log が含まれる．
(2) WAL ファイルに Tx ログが全て永続化されており，abort log が含まれる．
(3) WAL ファイルに Tx ログが一部のみ永続化されており，commit log も abort log も含まれない．
(4) WAL ファイルに Tx ログがひとつも存在しない．

定義から，(1) の場合のみ Tx は committed となる．定義より，Tx1 in W1，Tx2 in W2，W1 != W2 であるような任意の Tx1，Tx2，W1，W2 に対して，W1 --> W2 ならば，Tx1 --> Tx2 or Tx1 -- Tx2. Lemma 9. より，GWSN1 < GWSN2 ならば，W1 --> W2 or W1 -- W2．すなわち，GWSN1 < GWSN2 ならば，Tx1 --> Tx2 or Tx1 -- Tx2．（C3）により，GWSN の小さい順に適用を行うと，全ての Tx1 --> Tx2 の関係について，Tx1，Tx2 の順に適用される．

(2) の場合は，abort log が永続化しているため，undo 処理はクラッシュの前に完了している．すなわち，データ

ベース本体キャッシュには Tx の更新内容はなかったことになっている．しかし，既に変更がデータベース本体ファイルに永続化されており，undo 操作は永続化される前にクラッシュする可能性がある．この場合，undo-required Tx となるが，依存関係は有効な Tx と同様に存在する．よって，(1) と同じ順序で redo ではなく undo を実行することで，クラッシュ前の実行と同じ結果を得られる．

(3) の場合 Tx は完了しておらず，undo-required Tx となる．すなわち，ログとして永続化されている record はクラッシュするまで lock されていたことになるし，`Tx --> Tx1` となるような `Tx1` は存在しない．この場合，当該 Tx の最初のログを含む W かそれ以降に undo を実行すれば，リカバリにおいてその record を更新する他の Tx はないため，安全である．どちらにせよ，commit/abort log がないことを知るためには，有効なログを最後まで走査しないといけないため，(1) (2) の適用が全て終わってから，(3) の undo をしても依存関係を破ったことにはならない．各 log record は，同一 Tx の ひとつ前の log record の LSN を記録する prevLSN フィールドを含んでいる．ある log record の prevLSN が 0 の時，その log record は 当該 Tx の最初の log record である．すなわち，最後に適用した log record の LSN さえ覚えておけば，commit/abort がないと発覚してから，prevLSN を辿って 当該 Tx の全ての log record を undo することができる．

(4) の場合，WAL に Tx ログがひとつも永続化されていないので，データベース本体には何も適用されていないため，リカバリ時に考慮する必要はない．

(1) (2) (3) (4) により，リカバリ処理対象のログについて，committed な Tx は全てデータベース本体上に依存関係順に反映され，それ以外の Tx の更新は全く反映されないことが示された．

`Lemma 11.` **リカバリ完了後，データベース本体は `committed` な `Tx` が依存関係の順に適用された状態となる**

最終 checkpoint に記録されている GWSNc に対して，`GWSN <= GWSNc` である GWSN に含まれる Tx に関しては，Lemma 8. より，クラッシュ前にデータベース本体に有効な Tx が依存関係順に適用済みである．

`GWSNc < GWSN` となる GWSN を持つ W に含まれるログがリカバリの対象ログとなる．それについては，Lemma 10. により，クラッシュ後のリカバリ操作によって，データベース本体に有効な Tx 依存関係順に適用される．よって，示された．

`Theorem.` **上記の制約を満たすトランザクションシステムはリカバリ可能である**

Lemma 5. より，有効な Tx が依存している Tx は有効．Lemma 11. より，リカバリ完了後，データベース本体は committed な Tx が依存関係の順に適用された状態．定義より，そのようなシステムはリカバリ可能．よって示された．

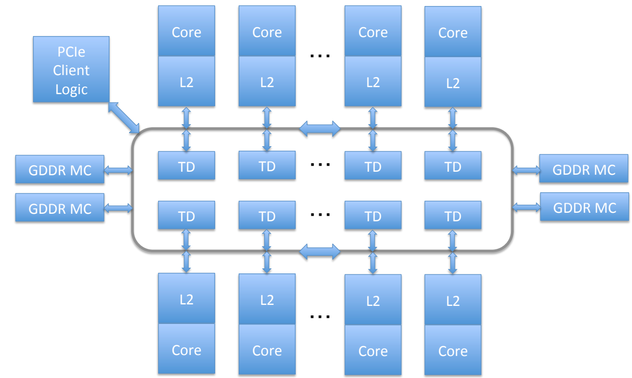

図 3 MIC のアーキテクチャ

5. 予備評価

本章では，トランザクションシステム上に GWSN 方式を実装して総合的な評価を行う前の予備評価として，GLSN 方式および GWSN 方式を模擬する共有カウンタの fetch_and_add および WAL flush を模擬する遅延操作を複数スレッドから繰り返す実験を行い，競合緩和の効果を調べる．

5.1 評価環境

筑波大学計算科学研究センターの COMA を利用した．COMA の 1 ノードは CPU (2 基)，MIC (Many Integrated Core) (2 基) からなる．実験に使用したはプログラムは MIC 上で native 実行したため，この MIC について説明する．使用した MIC は Intel Xeon Phi 7110P, 61 core/MIC, 1.1 GHz である．MIC のアーキテクチャを**図 3** に示す．1 つの core はカーネルプロセス専用のアシスタントコアになっているため，計算に使うのは実質 60 cores である．また，1 つの core につき 4 つのハードウェアスレッドを持つ．一つの巨大なリングバスによって 60 個のコアを接続している．メモリは GPU でも使われる GDR メモリを使用している．L1，L2 キャッシュが各コアに配置されている．コア間で L2 キャッシュのコヒーレンス制御がなされる．キャッシュ制御用の TD (Tag Directory) が各コアに付属している．

5.2 評価実験

共有カウンタのインクリメントを繰り返すプログラム上で競合緩和がどのように影響するのか，書き込みレイテンシを変えることで各ストレージデバイスをシミュレートし，評価する．測定値は 10 回の平均を取っている．NVRAM は PCRAM を想定し，書き込みレイテンシは次の通りとする．レイテンシなし (0 sec)，NVRAM (150ns[12])，PCIe

図 4 永続化レイテンシ: なし - fetch_add（1）

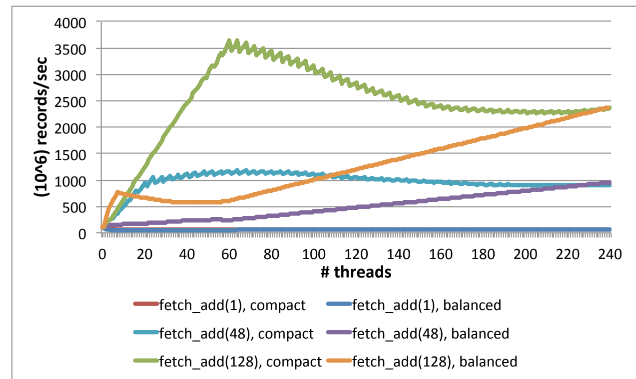

図 5 永続化レイテンシ: なし - fetch_add（N）

SSD（30μs[16]）.

5.3 ログの永続化レイテンシ: なし

実験内容

この実験ではログの書き込みレイテンシが 0 の状況をシミュレーションする．すなわち，共有カウンタのインクリメント以外のデータベースアクセスやログの永続化による書き込みのレイテンシは一切含めない．共有カウンタへのアクセス競合は実際のトランザクションシステムより大きい設定と言える．

またスレッドの割り当て方式として，同じ物理コアの隣り合うハードウェアスレッドに順に割り当てる方式（compact）と別の物理コアにラウンドロビンで順に割り当てる方式（balanced）の性能の違いも調べる．

共有カウンタへの fetch-and-add は，C++の std::atomic::fetch_add を用いる（省略して fetch_add と呼ぶ）．グラフの fetch_add（N）は N 回のループ毎に共有カウンタに対して fetch_add を行うということを意味している．すなわち fetch_add（1）は 1 個のログレコード毎に LSN を割り当てる GLSN 方式，fetch_add（128）は 128 個のログレコード毎に GWSN を割り当てる方式をシミュレートしている．

実験結果

fetch_add（1）の場合について，compact と balanced を比較する．この時の実験結果を**図 4** に示す．縦軸は一秒間に回すことができたループの回数であり，これをログレコード数として換算する．横軸はスレッド数で，x 軸が対数の片対数グラフになっている．balanced は，2 並列で性能が劣化している．これは異なる物理コア間においてインターコネクトのリングバスを通じてキャッシュのコヒーレンシ制御が行われているからだと考えられる．一方，compact では 2 並列にすると性能が向上し，3〜4 並列では性能はほとんど変わらず，5 並列で大きく性能が劣化した．これは同じ物理コア内でハードウェアスレッドを割り当てた場合，L1, L2 キャッシュを共有しているため 4 並列までは他コアとの通信が発生しないが，5 並列で他の物理コアを跨った時にキャッシュのコヒーレンシ制御が必要になったためだと考えられる．3〜4 並列の時，2 並列よりも並列度が高くなったにも関わらず，ほとんど性能が変らなかったのは，単独コアで実行される fetch_add の性能が 2 並列で既に頭打ちになったからだと考えられる．また，balanced は 60 並列を過ぎたところから，compact に近づくように性能が向上している．これは一つの物理コアに複数のハードウェアスレッドが割り当てられるようになったためである．

次に，fetch_add（48），fetch_add（128）の性能を加えたものを**図 5** に示す．fetch_add（48），fetch_add（128）は fetch_add（1）と比べて，同じ fetch_add の発行回数でシミュレーション上は，より多くのログレコードを作成したことになる．また，fetch_add の間隔が広くなり競合が緩和されるため，並列に fetch_add 発行を行うことで性能がスケールするようになった．グラフの傾向としては，並列度の増加による性能向上と，キャッシュのコヒーレンシー制御に伴うコア間のインターコネクトの通信量の増加による性能劣化の二つの要因が関係していると考えられる．compact において，一部グラフの線がノコギリ状となっているのは，4 スレッド毎に新たな物理コアにスレッドが割り当てられるためである．balanced においては，60 並列以降は一つの物理コアに複数のハードウェアスレッドが割り当てられるようになり，インターコネクトのリングバスの帯域を圧迫せずに，並列度が増加したことによる性能向上が効いていると考えられる．一般に compact の方が，ピーク性能は高いが，fetch_add（128）の 2〜11 並列では balanced の方が compact よりも良い性能を示した．これは，balanced だと 1 スレッドが物理コアのリソースを占有できるため，fetch_add の競合が少ない場合は，balanced の方が fetch_add 以外の処理（ループ用のローカル変数のインクリメントと条件分岐）が高速に行われたためだと考えられる．7 並列で一旦性能が頭打ちになるのは，キャッシュのコヒーレンシ制御にかかるコストが上回ったためだと考えられる．

以降の実験では compact に固定して評価を行う．

© 2016 Information Processing Society of Japan

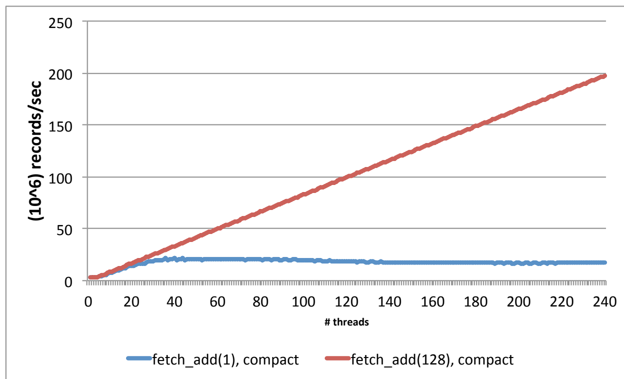

図 6 永続化レイテンシ: PCRAM（150ns）

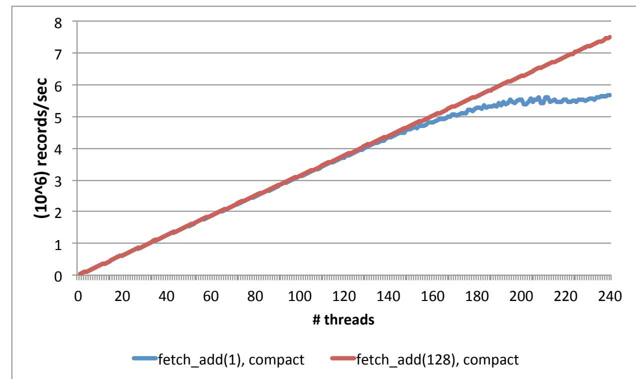

図 7 永続化レイテンシ: PCIe SSD（30μs）

5.4 ログの永続化レイテンシ: PCRAM（150ns）

実験内容

この実験では書き込みに 150 ns かかる PCRAM をログストレージに用いた状況をシミュレートする．fetch_add（1）と，fetch_add（128）の二つについて比較する．ログレコードは共に 128 個毎に書き込まれるものと想定する．すなわち fetch_add（1）は毎回のループ毎に fetch_add し，128 回のループ毎に書き込みレイテンシとして 150ns の人工遅延を入れる．fetch_add（128）も同様に，128 回のループ毎に fetch_add し，150ns の人工遅延を入れる．人工遅延の方法は rdtsc を用いた busy loop で実装した．

実験結果

実験結果を**図 6** に示す．fetch_add（1）では 30 並列ほどで性能が頭打ちになっているのに対し，fetch_add（128）では並列数に比例して性能が向上している．これは，NVRAM（PCRAM）を用いた場合，IO はボトルネックでなく，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られる．NVRAM には他にも MRAM や Memorister などがあり，それらは一般的に PCRAM よりも更にレイテンシが短いとされていることから，それらのデバイスを用いた時にもこのような傾向になると考えられる．

5.5 ログの永続化レイテンシ: PCIe SSD（30μs）

実験内容

この実験では書き込みに 30 μs かかる PCI Express 接続（PCIe）SSD をログストレージに用いた状況をシミュレートする．PCRAM を用いた実験と同様に，fetch_add（1）は毎回のループ毎に fetch_add し，128 回のループ毎に 30μs の人工遅延を入れる．fetch_add（128）は 128 回のループ毎に fetch_add し，30μs 人工遅延を入れるする．

実験結果

実験結果を**図 7** に示す．PCIe SSD では，140 並列程度までは fetch_add（1）と fetch_add（128）の性能はほとんど変わらない．この時，ボトルネックは共有カウンタアクセスではなく，IO であると考えられる．fetch_add（1）で 140 並列を超えると，インターコネクトのリンクバスの帯域が圧迫されて性能が下がったと考えられる．

6. 関連研究

WAL の高速化に関する研究には次のようなものがある．Aether[2] は直列 WAL を高速化するテクニックを紹介している．Flush pipelining は，コミット時の WAL flush を非同期に行い，flush 完了後のコールバック処理でクライアントに結果を返してコミットを完了することで，WAL flush にかかる遅延を隠蔽する．Flush pipelining は本研究にも適用可能である．Early Lock Release（ELR）は，Tx が pre-commited な状態（commit log を含む全ての log records の flush 以外の処理を終えた状態）になった時点で，リソースの lock を開放する．クライアントに対しては commit log が書かれるまでトランザクションの結果を返さないようにする．本研究に ELR を適用する場合，lock 解放のタイミングは GWSN 取得後とし，クライアントに結果を返すのは，GWSN が歯抜けなく永続化されている minGWSN に対して，当該 GWSN が `GWSN <= minGWSN` 満たす場合とする．上記の制約を加える必要があるが，ELR も本研究に適用可能である．

Distributed Logging[10] は，グローバルシーケンス番号（GSN）という論理クロックを用いて，依存関係のあるオペレーションが発生する場合にのみ同期を取ることで，スケーラビリティの高い分散ロギングを実現していると述べている．論理クロックを使うため，log record だけでなく page や Tx にも対応する GSN を記録する必要があるが，本研究は log record に GWSN を記録すればよいため，設計はより単純になる．

Silo[9] や FOEDUS[3] はメニーコア環境においてはログレコード毎に一意に順序番号を割り当てるための共有カウンタはスケーラビリティを妨げるとして，epoch と呼ばれる期間毎にまとめてトランザクションをコミットすることでスケーラビリティの高いロギング手法を提案している．これらは高いスループットを得る代わりに，レイテンシを犠牲にしている．クライアントにトランザクションの結果を返すことができるのは epoch の区切りになるため，特に

© 2016 Information Processing Society of Japan

ある Tx の結果を元に，次の Tx の内容を決定するような場合においては，レイテンシが積み重なってしまう．本研究は，Epoch を導入せずに，共有カウンタへのアクセス頻度を減らすことで，低レイテンシでかつ現実に必要となるワークロードにおいて十分なスループット性能の達成を目指す．

7. 結論と今後の課題

本稿では，並列 WAL において共有カウンタの競合を緩和する GWSN 方式を提案した．GWSN 方式は，一つのログレコード作成毎に共有カウンタにアクセスする GLSN 方式とは異なり，WAL の永続化単位毎に共有カウンタにアクセスし，複数のログレコードに一つの順序番号を割り当てる．Xeon Phi（60 cores, 240 threads）で GLSN 方式および GWSN 方式を模擬する予備評価を行ったところ，書き込みレイテンシが短い NVRAM においては，ボトルネックは IO ではなく共有カウンタへのアクセスになっていた．GLSN 方式はメニーコアの領域において，スレッド数に応じて性能がスケールしないのに対し，GWSN 方式はメニーコアの領域において，スレッド数に応じて性能がスケールした．

今後の課題は，この GWSN 方式をデータベースシステム上に実装し，データベースアクセスやログの作成などを含めた総合的な評価を行うことである．

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」による．

参考文献

[1] Gray, J.: The Transaction Concept: Virtues and Limitations, VLDB, pp. 144–154 (1981).
[2] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M. and Ailamaki, A.: Aether: A Scalable Approach to Logging, PVLDB, Vol. 3, No. 1, pp. 681–692 (2010).
[3] Kimura, H.: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM, SIGMOD Conference (2015).
[4] Levandoski, J., Lomet, D., Sengupta, S., Stutsman, R. and Wang, R.: High Performance Transactions in Deuteronomy, CIDR (2015).
[5] 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）.
[6] MySQL: MySQL 5.7 Reference Manual :: 15.10.1 InnoDB Disk I/O, https://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html,（アクセス日: 2016-05-07）.
[7] 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）.
[8] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. PVLDB, Vol. 8, No. 4, pp. 389–400, 2014.
[9] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy transactions in multicore in-memory databases, SOSP, pp. 18–32 (2013).
[10] Wang, T. and Johnson, R.: Scalable Logging through Emerging Non-Volatile Memory, PVLDB, Vol. 7, No. 10, pp. 865–876 (2014).
[11] Zheng, W., Tu, S., Kohler, E. and Liskov, B.: Fast Databases with Fast Durability and Recovery Through Multicore Parallelism, OSDI, pp. 465–477 (2014).
[12] S. J. Ahn, Y. J. Song, C. W. Jeong, J. M. Shin, Y. Fai, et al: Highly Manufacturable High Density Phase Change Memory of 64Mb and Beyond, IEDM, pp. 907910 (2004).
[13] Helland Pat, Sammer Harald, Lyon Jim, Carr Richard, Garrett Phil and ReuterAndreas: Group Commit Timers and High Volume Transaction Systems, Proceedings of the 2Nd International Workshop on High Performance Transaction Systems, pp. 301–329 (1989).
[14] 神谷孝明 川島英之 建部修見: P-WAL: 並列ログ先行書き込みの提案, 研究報告システムソフトウェアとオペレーティング・システム (OS) Vol. 2015-OS-133, No. 18, pp. 1–10 (2015).
[15] 神谷孝明 川島英之 建部修見: 並列ログ先行書き込みの評価, 研究報告ハイパフォーマンスコンピューティング (HPC) Vol. 2015-HPC-150, No. 37, pp. 1–6 (2015).
[16] Micron: Micron 9100 PCIe NVMe SSD, https://www.micron.com/~/media/documents/products/product-flyer/9100_ssd_product_brief.pdf.（アクセス日: 2016-05-09）.
[17] HGST: HGST Ultrastar He10, https://www.hgst.com/sites/default/files/resources/Ultrastar-He10-DS.pdf,（アクセス日: 2016-05-09）.

© 2016 Information Processing Society of Japan