Vol.2015-OS-134 No.18
2015/8/5

NVDIMM の活用によるファイルシステムの信頼性向上

石田紳太朗[1]　青田直大[1]　河野健二[1]

概要：ファイルシステムでは, システム障害時にデータ構造の一貫性を保証する必要がある. 一貫性が損なわれるのは, ディスクのデータ構造が部分的に書き換えられてしまうためである. ファイルシステムの一貫性を保証する代表的な手法であるジャーナリングは, 一連の更新をジャーナルと呼ばれるディスク領域に保存してから, 実際のディスクデータを書き換える. 障害発生時もジャーナルの内容から一貫性を保証できる. 性能低下を避けるために, ファイルシステムにおけるメタデータのみをジャーナリングする方式が主流となっている. メタデータ以外の更新でジャーナリングの必要がなくなるため, 性能向上が図れる反面, 障害発生時にユーザデータの内容は保証されないため, 信頼性が犠牲になっている. 本研究では, 近年登場している NVDIMM を用いることでデータジャーナリングのオーバーヘッドを削減し, 性能を犠牲にすることなく信頼性を向上させる. NVDIMM は DRAM としてアクセスが可能であり, かつ, 不意な電源遮断時にデータを NAND Flash に退避する. NVDIMM をジャーナル領域として用いることでジャーナルへの書き込みを高速化する. また, NVDIMM の使用により生じる, 書き込みのアトミシティの問題と CPU キャッシュによる問題を解決する. Linux の ext4 において提案手法を実装し, 提案手法のデータジャーナリングにおいて, 既存手法のメタデータジャーナリングの性能を上回る結果を得た.

キーワード：ファイルシステム, ジャーナリング, NVDIMM

1. はじめに

ファイルシステムでは, システム障害時にデータ構造の一貫性を保証する必要がある. 一貫性が損なわれるのは, ディスクのデータ構造が部分的に書き換えられてしまうためである. これはディスクがセクタ単位での書き込みのアトミシティしか保証しないために起こる.

ファイルシステムにおいて, 一貫性を保証する代表的な手法に, ジャーナリング (journaling)[1], [2] がある. ジャーナリングは ext4, NTFS, XFS などで採用されている. ジャーナリングでは, まず更新内容をディスク上のジャーナルと呼ばれる領域に保存する. この操作をコミットと呼ぶ. この時点ではディスク上の実データは一切更新されていないため, 一貫性は保たれている. コミットが完了すると, ディスクの実データを書き換える. この操作をチェックポイントと呼ぶ. チェックポイントの際に障害が発生しても, すでにジャーナルに完全な更新内容が保存されているため一貫性を回復することができる [3]. 以上の手順によって, どの時点で障害が発生しても一貫性を保証することが可能となる.

しかし, ジャーナリングは性能の低下を招く. 書き込みの度に同じ内容を 2 回ディスクに書き込むためである. 性能低下を避けるために, 現在はメタデータのみのジャーナリングが主流となっている. メタデータの更新以外ではジャーナリングの必要がなくなるため, 性能向上が図れる反面, 障害発生時にユーザデータの内容は保証されないため, 信頼性が犠牲になっている.

SQLite [4] のようなデータベースや, LevelDB [5] のような key-value ストアは, ファイルシステム上で実装されている. ファイルシステム上で実装されたアプリケーションにおいて, ユーザレベルのデータ構造の一貫性を保証するのは簡単ではない. なぜならユーザデータを永続化する際に保証される振る舞いが, ファイルシステムによって異なるからである. また, 同じファイルシステムであってもオプションが異なることでデータを保存する振る舞いが変わってしまう [6].

ファイルシステムが異なることで, データ永続化の振る舞いの保証が変わる一例として次のようなケースがある. あるファイルに対してアペンドを行った後に, そのファイルにリネームを行うという場合である. ext3 の ordered モードでは, ファイルに対するアペンドが, リネームよりも先に永続化されることが保証される. しかしながら, 同じモードの ext4 ではそのような更新は保証されない.

[1] 慶應義塾大学
Keio University

© 2015 Information Processing Society of Japan

もしも, すべての更新が同期的にディスクに反映されるとしたら, 比較的単純に一貫性を保証することが可能になる. しかしながら, そのような更新は性能を低下させる. そのため, ほとんどのアプリケーションでは一貫性を保ちながらも高い性能を維持するために, 複雑な更新方式を用いている. そのような更新方式においては, 広範囲に渡るほとんど実行されないようなコーナーケースも考慮する必要がある. したがって, 初期のファイルシステムやデータベースの手法と同様に, 正しく一貫性が回復されることを保証するのは難しい.

同期的な更新がどの程度性能を低下させるかというのを確かめる調査を行った. ext4 の ordered モードを対象として, まったく同期を取らずに更新を行う場合と, 逐一同期を取りながら更新を行う場合とを比較した. ワークロードとして簡易的にログを用いた更新を再現したものを実行した. まずログファイルを作成し, そこに対してデータを書き込んだ後に, ファイルに対してデータの書き込みを行い, 最後にログファイルを削除するというようなものである. この処理を 1 万回実行した際の実行時間を比較した. 同期を取らない場合は 0.25 秒程度で実行が終了するが, 逐一同期的な更新を行った場合には 7 分以上かかるという結果になった.

本研究では, 近年登場している不揮発性 DIMM (NVDIMM) [7], [8], [9] を使って, ジャーナリングのオーバーヘッドを削減する. 同期的な更新による性能低下を抑え, 性能を犠牲にすることなく信頼性を向上させる. NVDIMM は DRAM のアクセス速度と NAND フラッシュの不揮発性を両立するデバイスである. DRAM としてアクセスができ, 不意な電源遮断時に NAND フラッシュに内容を書き出す設計となっている. 近年製品化が始まっており, 現在の最大容量は 8 GB である [7]. また, 16 GB の製品のサンプル出荷が行われている [8]. 本提案手法では, ジャーナルを NVDIMM に配置することでジャーナルへの書き込みを高速化する. fsync はジャーナルに内容を書き込んだ時点で完了する. したがって, ジャーナルへの書き込みを高速化することで, 同期的更新によるオーバーヘッドの削減が見込める. コミット時のディスクジャーナルへの書き込みを NVDIMM のジャーナルへのメモリコピーに置換することで, ディスクアクセスを減らすことに加えて, カーネルのブロックレイヤを回避する.

Linux の ext4 を対象とし, Linux 3.17.3 カーネル上で実装を行った. ext4 におけるジャーナリングは Journaling Block Device 2 (JBD2) を用いて実装されている. JBD2 におけるコミットの関数に変更を加えた. ディスクに対してコミットを行っている関数を NVDIMM への memcpy() に置き換えた. また, ジャーナル領域を NVDIMM に配置することで生じる 2 つの問題を解決した. 1 つは, CPU キャッシュによるジャーナリング処理の順序変更の問題で, もう 1 つは, コミットレコード書き込みのアトミシティの問題である. 現状は NVDIMM の入手が難しいため DRAM を NVDIMM と仮定してジャーナル領域を確保している.

実験によって, 既存のジャーナリングと提案手法の性能を比較した. Filebench [10] によるワークロードを実行し, スループットを計測した. また, 同期的な更新を擬似的に再現するために, 無限ループで sync を行うプロセスをバックグラウンドで実行している場合の性能を評価した. 本来であれば, fsync によって同期的に更新を行わせる必要があるが, ワークロードの修正が必要となるため, 今回は簡易的にこのような方法を採用した. 比較したのは, 既存手法のメタデータジャーナリング, 既存手法のデータジャーナリング, 提案手法のメタデータジャーナリング, 提案手法のデータジャーナリングである. 提案手法のデータジャーナリングにおいて, sync を行っている場合でも, sync を行っていない既存手法のメタデータジャーナリングの性能を上回る結果となった.

本論文の構成を以下に示す. 2 章ではファイルシステムにおける一貫性および, アプリケーションレベルの一貫性について説明する. 3 章では本研究が提案する手法の概要及び設計を示す. 4 章では提案手法を ext4 において実装した詳細を述べる. 5 章では提案手法と既存の手法の性能を比較した実験について述べる. 6 章では関連研究について紹介する. 7 章ではまとめと今後の課題を述べる.

2. 障害発生時におけるデータ構造の一貫性

ファイルシステム, データベース, key-value ストア, バージョン管理システムなどのデータを保存・管理するシステムではシステム障害時にデータ構造の一貫性を保証する必要がある. この章では, ファイルシステムおよびアプリケーションにおける障害発生時のデータ構造の一貫性について述べる.

2.1 ファイルシステムにおける一貫性の問題

ファイルシステムは, 電源遮断時やシステム障害時にファイルの一貫性を保証する必要がある. 一貫性が損なわれるのは, ディスクデータが部分的に書き換えられるためである. ファイルシステムは 1 つの操作で複数の構造を書き換える必要があるが, ディスクはセクタ単位での書き込みのアトミシティのみを保証するため, このような問題が起こる.

例えばファイル削除において, ディレクトリエントリの削除と inode の解放をアトミックに行う必要がある. もしも, inode の解放のみが行われた場合には, ディレクトリエントリは解放済みの inode を参照することになってしまう. 逆にディレクトリエントリの削除のみが行われた場合には, inode は割り当て済みのままどこからも参照されず,

© 2015 Information Processing Society of Japan

解放もできないというような状態になってしまう．inode ブロックの更新と, ディレクトリのブロックの更新が発行されると, ディスクは 1 つずつリクエストを処理する．そのため, どちらかの書き込みがディスクに対して先に行われる (スケジューリング次第でどちらが先かは変わる)．一方の書き込みのみが行われた状態で障害が起きてしまうと, 先に述べたような一貫性のない状態に陥る．

2.2 ジャーナリング

ファイルシステムにおいて一貫性を保証するための代表的な手法にジャーナリング [1] がある．ext3, ext4, NTFS, XFS などのファイルシステムで採用されている．ジャーナリングはディスクデータ更新の前に, その内容をジャーナルと呼ばれるディスク領域に保存しておくことでアトミックな更新を可能にする．アトミックに行われる更新をトランザクションと呼ぶ．

ジャーナリングはディスクデータの更新を始める前に, 更新の内容をジャーナルと呼ばれるディスク領域に書き込む．この処理をコミットと呼ぶ．先ほどの例で言えば inode とディレクトリエントリのブロックをジャーナルに保存する．この際, 実データは一切書き換えられておらず, 一貫性は保たれている．ジャーナルへの書き込みが完了すると, コミットレコードと呼ばれる構造がジャーナルに書き込まれる．このコミットレコードの書き込みによってコミットの完了を示す．コミットレコードは 1 つのセクタにおさまる大きさになっており, 書き込みのアトミシティが保証されている．コミットレコードの書き込みが完了すると, ジャーナルに保存された内容をディスクの実際の位置に書き出す．この操作をチェックポイントと呼ぶ．ジャーナルに保存されていた inode ブロックとディレクトリのブロックがファイルシステムにおける実際の位置に書きだされる．この時障害が起きたとしても, すでにジャーナルには完全な更新の内容が保存されているため, 一貫性を回復することが可能である．以上の手順により, どのタイミングで障害が発生しても一貫性は保証される．

しかし, ジャーナリングではディスクに対するアクセスが書き込みの度に 2 回発生するため, 性能が低下する．そのため現状はメタデータのみをジャーナリングするという方式が主流となっている．メタデータのみをジャーナルに書き込むことで, 性能低下を防いでいる．しかし, 障害発生時のユーザーデータの内容については一切保証されないため, 信頼性は犠牲になっている．

2.3 アプリケーションレベルの一貫性の問題

データベースや key-value ストアのようなアプリケーションでは, ユーザーデータレベルで一貫性を保証する必要がある．現状のファイルシステムレベルの一貫性保証手法は, 通常メタデータのみの一貫性を保証する．そのため

表 1 ファイルシステム毎のデータ永続化の振る舞い [6]

	ext2	ext2-sync	ext3-writeback	ext3-ordered	ext3-datajournal	ext4-writeback	ext4-ordered	ext4-nodealloc	ext4-datajournal	btrfs	xfs	xfs-wsync	reiserfs-nolog	reiserfs-writeback	reiserfs-ordered	reiserfs-datajournal
Atomicity																
Single sector overwrite																
Single sector append	×		×			×								×		
Single block overwrite	×	×	×	×		×	×	×			×	×	×	×	×	
Single block append	×	×	×			×							×	×		
Multi-block append/writes	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
Multi-block prefix append	×	×	×			×							×	×		
Directory op	×	×											×			
Ordering																
Overwrite->Any op	×		×	×		×	×	×			×	×	×	×	×	
[Append, rename]->Any op	×		×			×							×	×		
O_Trunc Append->Any op	×		×			×							×	×		
Append->Append (same file)	×		×			×							×	×		
Append->Any op	×		×			×	×			×	×		×	×		
Dir op->Any op	×									×			×			

アプリケーションは独自にデータ構造の一貫性を保証する手法を実装している．例えば多くのデータベースではログ先行書き込みを行う．ログ先行書き込みでは, 新しいデータをまずログに対して書き込む．その後でファイルにデータを書き込み, それが終わるとログを削除する．このような操作によりユーザーデータレベルでの一貫性の保証が可能になる．

現在, SQLite [4] や LevelDB [5] のようなアプリケーションは, raw ディスクではなく, ファイルシステム上で実装されている．ファイルシステム上で実装されたアプリケーションにおいて一貫性を保証することは簡単ではない．なぜならファイルシステムによってデータを永続化する際の振る舞いが異なっているからである．Pillai らは, ファイルシステム毎のデータ永続化の振る舞いを調査した [6]．その結果を表 1 にまとめる．ファイルシステムとそのオプションの違いによる, 動作の特性をまとめている．表中の × は, その列のファイルシステムにおいて, その行の動作が保証されないことを示している．X -> Y は, X が Y よりも前に永続化されることを示している．ファイルシステムのオプションの違いによっても, データ永続化の方式というのは変わってくる．ファイルシステム上のアプリケーションにおいて一貫性を保証するためには適切な箇所で fsync 等を用いて同期的に更新をディスクに反映させる必要がある．このことについて, 簡単な例を用いて以下で説明する．

例として, ログを使った一貫性保証手法を簡易的に再現する．以下に擬似コードを示す．

```
create(/log);
```

© 2015 Information Processing Society of Japan

```
   write(/log, "new data");
   write(/file, "new data");
   unlink(/log);
```

まずログを作り, そこに対して書き込むべきデータをあらかじめ書き込んでおく. そのあとでファイルに対してデータの書き込みを行い, 最後にログファイルを削除する. この例のような更新方式は ext3 のどのジャーナリングモードにおいても正しく動作しない. メタデータとデータの両方をジャーナリングするとしても, 親ディレクトリに対する更新がディスクに反映されることが保証されないからである.

```
   create(/log);
   write(/log, "new data");
-> fsync(/)
   write(/file, "new data");
   unlink(/log);
```

この場合は, ext3 の journal モードであれば一貫性を損なわずに動作することが可能である. journal モードはメタデータとデータの両方をジャーナルに書き込むオプションである. しかしながら, ext3 の ordered モードにおいては一貫性が保証されない可能性がある. ordered モードはメタデータのみをジャーナリングするオプションである. ext3 の ordered モードで正しく動作させるためには次のようにコードを修正する必要がある.

```
   create(/log);
   write(/log, "new data");
-> fsync(/log)
   fsync(/)
   write(/file, "new data");
-> fsync(/file)
   unlink(/log);
```

こうすれば journal モードと ordered モードで正しく動作することが可能になる. しかしながら, もう 1 つのジャーナルオプションである, writeback モードにおいてはこれでも正しく動作しない. writeback モードもメタデータのみをジャーナリングするオプションである. ordered モードでは, メタデータよりも先にデータが書き込まれることが保証されているのに対して, writeback モードではメタデータとデータの書き込み順序は保証されない. wirteback モードで正しく動作させるためには, 次のようにコードを修正すればよい.

```
   create(/log);
-> write(/log, "new data & checksum");
   fsync(/log)
   fsync(/)
   write(/file, "new data");
   fsync(/file)
   unlink(/log);
```

writeback モードにおいては, ログが部分的に書き込まれた状態で残ってしまう可能性がある. なぜならメタデータが先に書き込まれる可能性があるため, ログファイルは存在するが, データの内容が不完全に残ることがあるからである. そのため, 変更したコードではログへの書き込みの際にチェックサムを計算して格納している. これで ext3 におけるすべてのオプションに対して正確に動作させることが可能となる.

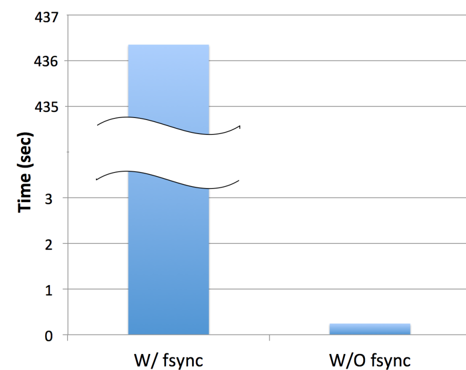

図 1　同期的な更新による性能低下の調査

2.4　同期的な更新によるオーバーヘッドの調査

fsync を用いた同期的な更新がどの程度性能を低下させるかを確認する. 擬似コードで示したワークロードを実際に作成し, まったく fsync を用いない場合と, 逐一同期を取った場合を比較する. ログ作成, ログへの書き込み, ファイルへの書き込み, ログの削除, といった一連の更新を 1 万回実行した際の実行時間を比較する.

実験に使用したマシンは, CPU Intel Xeon X5675 3.07GHz, メモリ 16 GB, ハードディスク 600 GB 10K RPM SAS 2.5 である. Linux 3.17.3 カーネル上で, ファイルシステムは ext4 の ordered モードを用いた.

結果を図 1 に示す. 同期をまったく取らない場合には, 0.25 秒程度で終了するワークロードであるが, 逐一 fsync を発行した場合, 実行時間は 7 分以上になった. 同期的な更新をすることでアプリケーションは大幅な性能低下を強いられることが分かる.

3.　提案

3.1　概要

本研究では, NVDIMM を利用してデータジャーナリングのオーバーヘッドを削減することで, ファイルシステムの信頼性を向上させる. また, 同期的な更新による性能低下を抑える. NVDIMM は普段 DRAM として動作し, 不意な電源遮断時に内容を NAND フラッシュに書き出す. NVDIMM は近年開発が進んでいる NVRAM の前身として注目されている. 近年 NVRAM を用いたファイルシステムの研究も行われている [11], [12], [13], [14], [15].

本研究では, NVDIMM 上にジャーナルを作成することでジャーナルへの書き込みを高速化させる. NVDIMM へ

© 2015 Information Processing Society of Japan

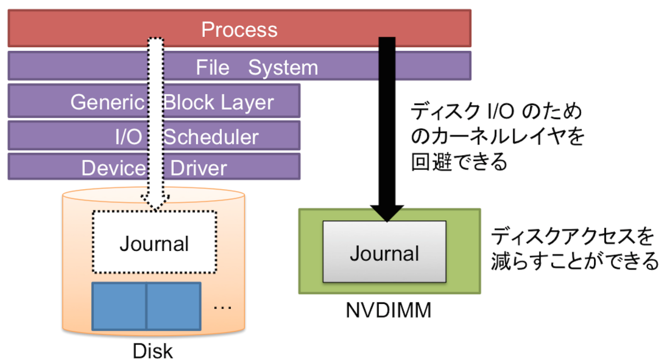

図 2 提案手法の概要

アクセスする方法には, 次の 2 つが考えられる. 1 つ目は, NVDIMM を RAM ディスクとして扱う方法である. この方法のメリットとして, 実装の変更が必要ないということが上げられる. NVDIMM をブロックデバイスとしてアクセスすることが可能になるためである. NVDIMM はディスクよりもアクセス速度が高速であるため, その分の性能向上は見込める. しかしこの方法は, 本来 DRAM としてアクセスできる NVDIMM に対して, ディスクと同じようにアクセスすることになる. すなわち, カーネルの汎用ブロック層や, I/O スケジューラ, デバイスドライバといったレイヤを経由する必要がある.

もう 1 つの方法として考えられるのは, NVDIMM に対してメモリコピーを行う方法である. この方法は, 既存の実装を一部変更する必要がある. しかし, ディスクへのアクセスを減らせることに加えて, カーネルのブロックレイヤを回避することができるため, RAM ディスクの方法よりも性能向上が見込める. そこで, 本提案手法では NVDIMM に対してメモリコピーを行う方法を採用する. 図 2 にこの方法のイメージを示す.

ジャーナルがディスクから NVDIMM へと移ることによって考慮すべき問題が 2 つある. 1 つはコミットレコード書き込みのアトミシティが保証できないということである. ディスクにおける書き込みはセクタ単位でのアトミシティが保証されているため, コミットレコードの書き込みをアトミックに行うことができる. しかし NVDIMM にはそのような保証がない. 本提案手法ではコミットレコードに専用のフラグを設けることでこの問題を解決する. コミットレコードの書き込みが完了した段階で 1 ビットのフラグの更新により, アトミシティを保証する.

もう 1 つの問題は CPU キャッシュによってジャーナリングの処理の順序が変更されてしまうことである. NVDIMM は DRAM としてアクセス可能であるため CPU キャッシュにデータがキャッシュされる. 通常キャッシュポリシーはライトバックであるため, キャッシュにのみ書き込みが行われており, NVDIMM にはその書き込みが反映されていないということが起こる. そうなると, 例えばコミットレコードがキャッシュにのみある状態で, チェックポイントの操作が始まってしまうようなことが起こる. そのような場合, コミットが完了する前にディスクの実データが書き換えられてしまい, 一貫性の保証が崩れてしまう. そのため本研究では NVDIMM のジャーナル領域のキャッシュポリシーをライトスルーにする. ライトスルーであれば, キャッシュに書き込むと同時に NVDIMM に対しても書き込みを行うため, 処理順序変更の問題を解決できる.

3.2 NVDIMM 上のジャーナル領域

NVDIMM 上にジャーナルを配置し, ブロックレイヤを通さずにアクセスする上で, 既存手法に対して変更を加える必要が生じるのはコミットとリカバリの部分である. まずはコミットの変更について説明する. 既存手法においては, コミット時にディスクへの書き込みリクエストを生成している. 提案手法ではこの部分を NVDIMM へのメモリコピーに置き換える必要がある. この際, バッファの内容を NVDIMM 上にそのままコピーする.

リカバリの変更も必要である. 既存手法ではディスクのジャーナルのブロックを読み込んでいるが, 提案手法では NVDIMM からメモリコピーしてくるという変更が必要である. 以上のような変更によって NVDIMM のジャーナルに対する読み書きが可能となる. しかし, ジャーナル領域をディスクから NVDIMM に移動させることで, 解決すべき問題が生じる. その問題について次節から述べる.

3.3 CPU キャッシュによるジャーナリング処理の順序変更

NVDIMM 上のデータは CPU キャッシュにも格納される. キャッシュから NVDIMM への書き込みにおいて順序変更が起きる場合があり, 一貫性の保証が崩れてしまうケースが考えられる. 例えば, コミットレコードの書き込みを行った際に, キャッシュ上にのみそれが反映されて NVDIMM には書き込みが到達していない状態がある. この時点で, コミットレコードの書き込みは完了されたものとみなされてしまいチェックポイントの処理が開始されてしまう可能性がある. そうなると, ジャーナル上でコミットの完了が示されないまま, 実データが更新されてしまう. このタイミングで障害が発生すると, リカバリ時にコミットレコードがジャーナル上に確認できないため, ジャーナルの内容は削除されてリカバリのプロセスが終了する. しかし, ディスクの実データは部分的に書き換えられた状態になっており, 一貫性が損なわれてしまう.

この問題を解決する方法はいくつか考えられる. まず, 既存の CPU 命令を組み合わせることでキャッシュの内容を強制的に NVDIMM に書き出す方法がある. メモリバリアとキャッシュラインフラッシュを組み合わせることで, CPU キャッシュの内容をメモリに対して書き出すことを

© 2015 Information Processing Society of Japan

保証することが可能である [13], [16]．しかし，コミット部分のさらなる実装の変更が必要となる．

また，BPFS [11] のように Epoch と呼ばれる書き込みをまとめた単位を導入し，Epoch における書き込み順序を強制するという方法もある．しかしこれには CPU のキャッシュタグの拡張と，メモリコントローラの拡張が必要になる．

本研究ではジャーナル領域のキャッシュポリシーをライトスルーにするという方針をとることで，ジャーナリングの実装の変更や，ハードウェアの拡張を行わずに問題を解決する．ライトスルーキャッシュであれば常に CPU キャッシュと NVDIMM の両方に書き込みを行うので一貫性が損なわれることはない．

3.4 コミットレコード書き込みのアトミシティ

NVDIMM のジャーナルにおいて，コミットレコードの書き込みのアトミシティを保証する必要がある．既存のジャーナリングにおいてはディスクがセクタ単位の書き込みのアトミシティを保証しているため，コミットレコードはアトミックに書き込まれる．しかし，NVDIMM にそういった保証がないため，コミットレコード書き込み中に障害が発生した場合には不完全なコミットレコードがジャーナルに書き込まれてしまう可能性がある．

本提案手法では専用のフラグをコミットレコードに設けることで，1 ビットの書き込みによってコミットレコード書き込みのアトミシティを保証する．

4. 実装

ext4 を対象とし，Linux 3.17.3 カーネルにおいて提案手法を実装した．ext4 では Journaling Block Device2 (JBD2) によってジャーナリングを実装している．ジャーナリングへのコミットとチェックポイントを担当するのは kjounrald2 というデーモンである．コミットの実装を変えるために，kjournald2 に手を加えた．現状は NVDIMM の入手が困難であるため，DRAM 上にジャーナル領域を確保しその領域に対してコミットを行うように実装している．DRAM のため，電源が落ちた時点でデータはなくなる．そのためリカバリは未実装である．

ジャーナル領域のキャッシュポリシーをライトスルーにする実装を行った．指定したメモリ領域のキャッシュポリシーをライトスルーにする API を追加し，ジャーナル領域に対して適用した．コミットレコードの修正は現在実装中である．

4.1 NVDIMM のジャーナル領域

NVDIMM におけるジャーナル領域は 2 次元配列により確保している．ブロックサイズ (4096 バイト) の char 型配列の配列である．ディスク上のジャーナル領域は，ブロックのリングバッファとして使われているが，その構造と対応を取るためである．ジャーナルサイズは NVDIMM の最大容量である 16 GB を確保する設定となっている．実際の DRAM 上でのジャーナルサイズは 128 MB となっており，その領域を 16 GB と仮定して上書きを繰り返している．

4.2 ジャーナルへのコミット

kjournald2 においてジャーナルへのコミットを行っている関数は jbd2_journal_commit_transaction() である．その中でディスクに対するブロック I/O のリクエストを行っている submit_bh() を DRAM 上に作成したジャーナル領域に対する memcpy() に置き換えている．submit_bh() が実行されると I/O が完了した際にコールバック関数が呼び出されるようになっている．その処理を memcpy() の直後に呼び出している．

jbd2_journal_commit_transaction() 以外にも，ジャーナルスーパーブロックやコミットレコード，廃棄レコード，ディスクリプタブロックなどの構造がジャーナルに書きだされるため，それぞれ対応する部分を同様に変更している．

4.3 リカバリ

現在 NVDIMM の実物を入手することが難しく，DRAM 上で実装を行っているため，リカバリは未実装となっている．JBD2 のリカバリの実装において jread() という関数で，ディスクのジャーナルからデータを読み込む．jread() の中では __getblk() によってディスクブロックを読み込む．したがってこの部分を NVDIMM のジャーナル領域からメモリコピーする実装に置き換える．

4.4 ライトスルーキャッシュの実装

ジャーナル領域のキャッシュポリシーをライトスルーとする実装は以下のように行った．まず，既存の API に set_memory_*() というものがある．これはページ単位で CPU のキャッシュポリシーを変更することができる API である．しかし，現状はライトスルーをサポートしていないため，この API にまず set_memory_wt() を加える実装を行った．そして，これを使ってジャーナル領域のキャッシュポリシーをライトスルーにした．

5. 実験

5.1 目的

本実験では既存のジャーナリングと比較して提案手法においてどれだけ性能が向上したのかを検証する．また，同期的な更新を行った場合にどの程度性能低下を抑えることができるかという点を確認する．

5.2 方法

DRAM を NVDIMM と仮定して実験を行う．

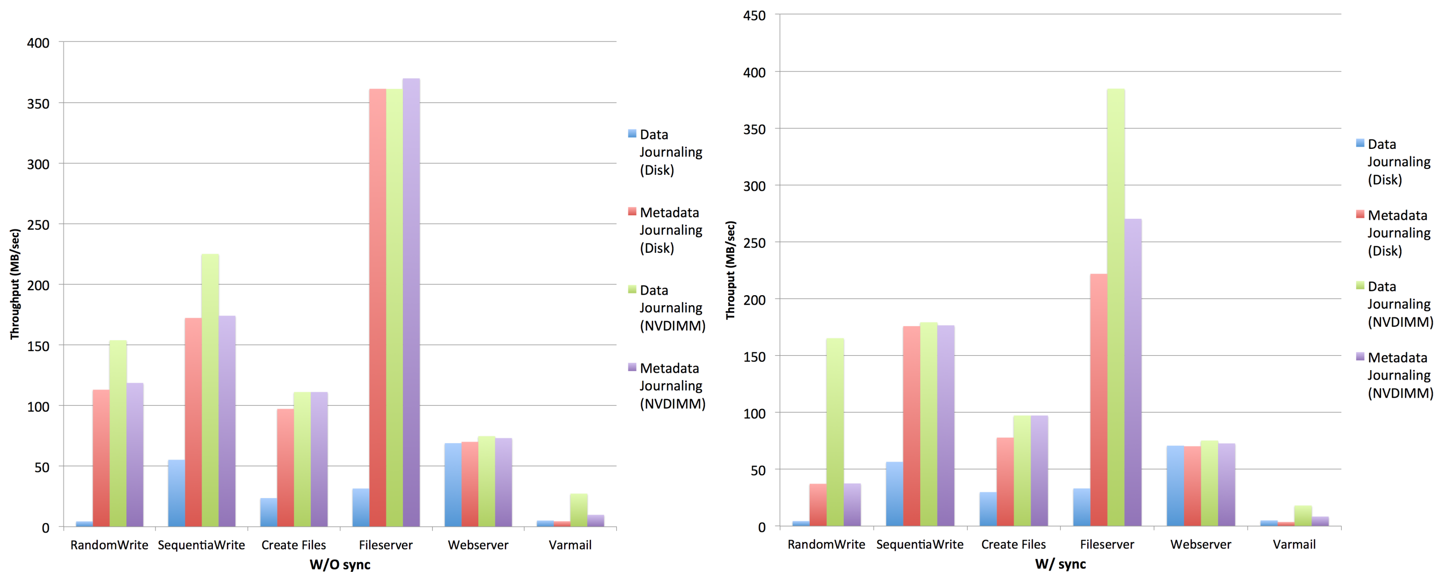

図 3 性能評価の結果

- 既存手法におけるメタデータジャーナリング (ordered モード)
- 既存手法におけるデータジャーナリング (journal モード)
- 提案手法におけるメタデータジャーナリング
- 提案手法におけるデータジャーナリング

ベンチマークツールの Filebench [10] を利用し, 6 つのワークロードを実行する. また, ワークロードと同時に, 無限ループで sync を行うプロセスを実行した状態での性能も評価する. これは, 同期的な更新による性能への影響を簡易的に評価するためである. 実際は fsync を逐一実行するべきであるが, それにはワークロードの修正が必要になるため, 今回は簡易的な方法を用いた.

5.3 ワークロード

Random Write は 5 GB のファイルに対して 8 KB のランダムな書き込みを 1 スレッドで行う. Sequential Write は 1 つのファイルに対して 1 MB の書き込みをシーケンシャルに 1 スレッドで行う. Create Files は 16 KB のファイルを 50,000 個作成する. 16 スレッドで行う. Fileserver はファイルサーバの I/O を模している. 作成, 削除, アペンド, 読み出し, 書き込みを 50 スレッドで行う. Webserver は, ウェブサーバの I/O を模している. 複数のファイルへのオープン - 読み出し - クローズと, ログファイルへのアペンドを 100 スレッドで行う. Varmail はメールサーバの I/O を模している. 作成 - アペンド - fsync, 読み出し - アペンド - fsync, 読み出し, 削除を 16 スレッドで行う.

5.4 環境

実験に使用したマシンは, CPU Intel Xeon X5675 3.07GHz, メモリ 16 GB, ハードディスク 600 GB 10K RPM SAS 2.5 である.

5.5 結果

実験結果を図 3 に示す. まず sync するプロセスを実行しない場合の結果について説明する. 提案手法のデータジャーナリングがほとんどのワークロードで最も高い性能を示した. メタデータジャーナリングでは, メタデータのみをジャーナルに書き込むため, NVDIMM による性能向上がほとんど見られない. データジャーナリングは, すべてのデータをジャーナルに書き込むため, NVDIMM による性能向上が大幅に達成された. ディスクへのデータジャーナリングと比較すると, Random Write においてスループットが 約 35 倍になっている. 元々 fsync を多用している Varmail でも, 提案手法のデータジャーナリングにおいて性能が大きく向上している. 提案手法のデータジャーナリングは, 既存手法のデータジャーナリングと比べて 5 倍以上のスループットを示している. この Varmail の結果から, 同期的な更新による性能低下を抑えていることが確認できる.

無限ループで sync を行うプロセスを実行している状態では, 既存手法のメタデータジャーナリングにおいて性能低下が見られる. 特に Random Write の性能は 3 分の 1 程度になっている. 提案手法のメタデータジャーナリングにおいても, 同程度の性能低下が見られる. これは ext4 のメタデータジャーナリング (ordered モード) では, メタデータがデータよりも後に書き込まれることを保証するためである. sync によってダーティなブロックが書き出される際に, データブロックが書きだされるのを待ってから, 対応するメタデータのブロックを書き出す. このために既存手法でも提案手法でもメタデータジャーナリングの性能が低下していると考えられる. しかしながら, データジャーナリ

© 2015 Information Processing Society of Japan

ングではバックグラウンドで sync を行うプロセスの影響がほとんど見られない．今回は簡易的にこのような方法を取ったため, 逐一 fsync を使った場合のように, 毎回の書き込みを同期的に行っているわけではない．したがって, 実際に同期的に更新を行う場合のような性能低下は再現し切れていない．しかしながら, 提案手法のデータジャーナリングにおいて, 書き込みの多いワークロードである Random Write と Fileserver の性能は向上している．これは, 元々バッファリングされていてジャーナリングされていなかったようなトランザクションが sync によってコミットまで至ったためだと考えられる．

以上の結果から, まず提案手法によってデータジャーナリングの性能が大幅に改善されるということが分かる．メタデータジャーナリングのように, sync するプロセスによって性能が大きく悪化するということもない．また Varmail の結果から, fsync による同期的な書き込みによる性能低下を改善できることも分かった．

6. 関連研究

ファイルシステムおよび, アプリケーションレベルの一貫性について調査する研究がなされている [6]. Pillai らはファイルシステムにおけるデータ永続化の振る舞いの保証と, アプリケーションレベルの一貫性について調査を行った．6 つの Linux ファイルシステムについて調査を行い, データを永続化する際に保証される振る舞いがそれぞれ異なっていることを明らかにした．また, それらのファイルシステム上で実装された 11 のアプリケーションにおいて 60 以上の脆弱性を発見した．

ファイルシステムにおける一貫性保証手法の研究は近年もなされている [17], [18]. OptFS [17] はジャーナリングにおけるフラッシュに着目し, チェックサムの拡張等によりフラッシュを不要にすることで性能を向上させている．NoFS [18] では, 新しい一貫性保証手法を提案している．バックポインタというテクニックを提案し, 一貫性保証手法における書き込みの順序強制を不要にしている．これらの提案は, ファイルシステムにおけるメタデータの一貫性のみを対象としている．

近年登場している, Phanse-Change Memory (PCM) などの不揮発性メモリ (NVRAM) を用いたファイルシステムの研究がなされている [11], [12], [13]. これらの提案はいずれもファイルシステム全体を NVRAM 上に配置するという前提でフルスクラッチから設計されている．BPFS [11] は一貫性の保証に, シャドウページング (shadow paging) に基づいた手法を提案している．シャドウページングはコピーオンライトによる更新を行うことで一貫性を保証する．CPU キャッシュからの書き込みの順序変更を防ぐために, CPU タグやメモリコントローラの拡張を必要とする．PMFS [12] は, CPU の命令, キャッシュラインの粒度のジャーナリング, BPFS のようなコピーオンライトによる更新を組み合わせて一貫性の保証を行っている．永続化の保証のために CPU 命令の拡張を提案している．Aerie [13] は, ユーザライブラリとして NVRAM のためのファイルシステムを提供する設計を提案している．一貫性の保証にはログ先行書き込みを用いている．これらの研究は, フルスクラッチからファイルシステムを設計しているのに対して, 本研究では既存のコードベースを再利用し, 実装の変更を最小限に留めている．

7. まとめ

ファイルシステムにおいて, ユーザデータ永続化の際に保証される振る舞いが異なるために, アプリケーションレベルの一貫性を保証することは簡単ではない．同期的に更新をディスクに反映させることで, 一貫性の保証を比較的単純にできる反面, そのような更新は著しく性能を低下させる．本研究では NVDIMM を用いることで, 同期的な更新による性能低下を抑え, 信頼性を向上させる手法を提案した．NVDIMM にジャーナルを配置することで, ジャーナリングを高速化し, 同期的な更新による性能低下を抑えた．データを含めたジャーナリングにおいて, 簡易的に同期的な更新を再現した環境で, 性能を大幅に向上することに成功した．既存手法のメタデータジャーナリングの性能を超える結果を得た．また, fsync を多用するワークロードにおいても, 既存手法と比べてスループットが向上することが確認できた．

参考文献

[1] Hagmann, R.: Reimplementing the Cedar file system using logging and group commit, *Proceedings of the eleventh ACM Symposium on Operating systems principles(SOSP'87)*, pp. 155–162 (1987).
[2] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A. and Sidebotham, R. N.: The Episode File System, *Proceedings of the USENIX Winter 1992 Technical Conference*, pp. 43–60 (1992).
[3] Prabhakaran, V., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Analysis and Evolution of Journaling File Systems, *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pp. 105–120 (2005).
[4] SQLite: SQLite transactional SQL database engine. http://www.sqlite.org/.
[5] Google: LevelDB. http://leveldb.org/.
[6] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications, *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation(OSDI '14)*, pp. 432–448 (2014).
[7] V.Technology: ArxCis-NV (TM): Non-Volatile DIMM. http://www.vikingtechnology.com/arxcis-nv.
[8] SKHynix: SK Hynix Developed the World's Highest Density 16GB NVDIMM. http://www.hynix.com/en/pr_room/news-data-view.

jsp?search.seq=2383&search.gubun=0014.
[9] JEDEC: JEDEC Announces Support for NVDIMM Hybrid Memory Modules. http://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules.
[10] McDougall, R. and Mauro, J.: Filebench. http://sourceforge.net/projects/filebench/.
[11] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D. and Coetzee, D.: Better I/O Through Byte-Addressable, Persistent Memory, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles(SOSP '09)*, pp. 133–146 (2009).
[12] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R. and Jackson, J.: System Software for Persistent Memory, *Proceedings of the Ninth European Conference on Computer Systems(EuroSys'14)* (2014).
[13] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P. and Swift, M. M.: Aerie: Flexible File-System Interfaces to Storage-Class Memory, *Proceedings of the Ninth European Conference on Computer Systems(EuroSys'14)* (2014).
[14] Wu, X. and Reddy, A. L. N.: SCMFS : A File System for Storage Class Memory, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis(SC '11)* (2011).
[15] 追川修一: ブロックストレージとの組合せによるメモリストレージ容量拡張手法, **情報処理学会論文誌 : コンピューティングシステム**, pp. 15–24 (2015).
[16] Volos, H., Tack, A. J. and Swift, M. M.: Mnemosyne: Lightweight Persistent Memory, *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems(ASPLOS '11)*, pp. 91–104 (2011).
[17] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Optimistic Crash Consistency, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles(SOSP '13)*, pp. 228–243 (2013).
[18] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Consistency Without Ordering, *Proceedings of the 10th USENIX conference on File and Storage Technologies(FAST'12)* (2012).

© 2015 Information Processing Society of Japan