修士論文

AMFS: アトミックなデータ永続化を備えた次世代不揮発性メモリ向けファイルシステム

AMFS: A File System for Emerging Persistent Memory Supporting Atomic Data Durability

平成 28 年 2 月 3 日提出

指導教員　　田浦 健次朗 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-146421 島津 真人

概 要

HDD や SSD に代わり, 次世代不揮発性メモリ (Persistent Memory, PM) が SSD と比較し 1000 倍程度低いンテンシを持ち, SSD よりも大きな容量を持つ次世代のストレージシステムとして期待されている. しかし, この低レイテンシなデバイスを SSD の代替として既存のファイルシステム上で利用しようとした場合, 現在のファイルシステムの仕組み上高いパフォーマンスを活かすことができない. 特に, 耐障害性を持つプログラムを記述しようとするためには, データを随時永続化してしまうという現在のファイルシステムのセマンティクスにより, Write Ahead Logging を始めとする複雑な処理を行う必要がある.

そこで, 本論文ではデータ永続化をアトミックに行うための API をもつファイルシステム, AMFS を提案する. AMFS では fbegin という API を導入し, fbegin が呼び出されてから fsync が呼ばれるまでの間に行われたファイルへの書き出しは, fsync が呼ばれた際にアトミックに行われる. 更に, ファイルシステムがこのようなアトミックなデータ永続化をサポートすることにより, ユーザープログラムがデータの書き出しを独自に管理する必要がなくなり, 高速に行うことができる.

AMFS を, read-through/write-back というキャッシュの手法を用いることで実装を行った. AMFS 上で mmap した領域を std::unordered_map として KVS のように利用した場合, 既存の KVS である leveldb を DRAM 上で実行した場合と比較して 2.4 倍の高速化を図ることができた. さらに, API が簡潔であり, ユーザープログラムが容易かつ安全に記述できることも示した.

目 次

図 目 次

表 目 次

第1章　序論

1.1　背景

既存のオペレーティングシステム上で動くソフトウェアでは, 一般的に永続化されたデータはファイルという単位でストレージに保管されている. このファイルという単位のデータは, 仮想ファイルシステム (Virtual File System, VFS) の提供する API を通じて, 様々なデバイスとデータの授受を行うことができる.

この永続化されたファイルが保管されるストレージとして現在利用されている HDD や SSD は, DRAM に比較しレイテンシが非常に大きい. たとえば, 広く知られているように HDD はシーク時間によりレイテンシは数 ms 程度となる [36]. また, SSD も書き込みに関しての構造上の理由から一旦ページ単位で削除をしなければならず, またウェアレベリングを行う必要もあるため, 書き込みのレイテンシは 0.1-1ms 程度 [28] となる. これは DRAM の 10ns 程度という値に比較して 10^5 倍も大きい.

このように DRAM に比較し非常に大きなレイテンシを持つものとして考えられてきたブロックデバイスであるが, HDD や SSD に変わるデータの永続化のためのストレージとして, 次世代不揮発性メモリ (Persistent Memory, PM) と呼ばれるものの実用化が近いと言われており, この前提が変わろうとしている [38]. たとえば, PCM では読み取りで 40ns 程度 [39], 書き込み時は 150ns 程度 [17] のレイテンシでデータのアクセスが可能であるという報告がある. 従来の SSD や HDD のレイテンシが ms オーダーであったことを鑑みると, PM は DRAM と同様に高速にデータにアクセスが可能であると言える.

また, PM のもうひとつの特徴として, 1 バイトごとの読み書きが可能であるという利点がある. 従来の HDD や SSD はその構造上, ブロック単位でのアクセスしかできなかった. しかし, PM は直接アドレスを指定することで, 1 バイトずつ更新が可能である. そこで, PM 向けのファイルシステムとして研究・開発されてきた PMFS[9] を始めとし, PM に保存されているファイルのデータをユーザープログラムから直接, load/store 命令を用いて読み書きが行えるようにしようという動きがある.

従来レイテンシの大きいブロックデバイスを前提としていたため, ファイルシステムは読み書きを高速化するために, ページキャッシュと呼ばれる機構を用いて, データを DRAM に一度キャッシュしている. しかし, PM ではレイテンシが DRAM と比較して遥かに大きいという前提が崩れ, さらには直接 load/store 命令でデータを触れるようにするため, ページキャッシュは排除していくという方向性で開発が進められているようである.

一方で, 直接データを読み書きしてしまうと, 電源喪失等の障害が発生した場合に不都合が生じてしまう. たとえば, 100byte の配列を PM 上の領域にコピーしていた場合を考えると, そのうち 50byte が書き込まれた時点で障害が発生してしまうと, 中途半端な状態となってデータが意味のないものとなってしまう. そこで, データの永続化をアトミックに行う操作が重要になってくる.

従来のファイルシステムにおいて, アトミックなデータ永続化を可能にするアルゴリズムは高信頼が求められるアプリケーションにおいて実装されてきた. 簡単な方法としては, 別のファイルに一旦書き出しておき, 後で `rename` によって変更を不可分に反映するという操作がある. 他にも, 従来 DBMS といった高信頼なアプリケーションにおいて用いられてきた Write Ahead Logging (WAL) [24] や Copy on Write (CoW)[26] などがある. これらは本質的には `rename` と同様の, データを編集する前に一旦そのデータのコピーを取っておき, 編集している際に障害が発生した場合にはコピーからデータを復旧する, というようなものである.

これらのアルゴリズムは, 従来のファイルシステムのセマンティクスではデータが永続化されているかいないかわからないシチュエーションがあるため, ユーザーが実装する必要があった. しかし, 障害発生時にも壊れないデータの永続化というのは本来であればファイルシステムによって保障されるのが好ましい.

1.2　本研究の目的と貢献

これらを踏まえ, PM に適したファイルシステム, AMFS (AtoMic File System) を提案する. AMFS は, read-through/write-backing というページキャッシング手法を用いることで, PM 領域への直接のアクセスとアトミックなデータ永続化を両立させることができる. また, ただ性能を向上させるだけではなく, アトミックなデータ永続化を可能にするファイルシステムのセマンティクスを導入することで, プログラマが高信頼なプログラムを容易に記述することができるようになった.

AMFS の実装は PMFS を元にして行った. AMFS 上に mmap した領域を C++の STL である `std::unordered_map` で利用し, KVS のように評価した場合, 軽量 KVS ライブラリである leveldb よりも 2.4 倍高速に書き込みが行えることが確認できた. また, AMFS の API を利用することで, 高信頼なユーザープログラムを実現するためのプログラムの記述量が削減できていることも確認できた.

1.3　本稿の構成

2 章では, アトミックなデータ永続化の重要性と現在のファイルシステムにおける状況, アトミックな永続化を実現するアルゴリズム, そのサポートをファイルシステムレベルで行う試みなどを紹介する. また, PM を用いた場合, 現状のセマンティクスではパフォーマンス上不利であるということを示す. つぎに 3 章では, アトミックなデータ永続化を備えたファイルシステム, および PM 向けのデータの保存に関する研究動向について確認し, ページキャッシュが不要となった場合にどのようなアクセス方法が考えられるかを考察する. それらを踏まえた上で 4 章で提案するファイルシステム, AMFS におけるセマンティクスや実装を説明し, 5 章で実験評価をする. 最後に, 6 でまとめと今後の PM に関わる研究課題をまとめる.

第2章　PMを用いた際のファイルシステムによるアトミックなデータ永続化のサポート

本章では, 現状のファイルシステムの設計について軽く触れた後, 現状での `write` や `mmap` によるデータの編集においてデータの永続化が中途半端に行われてしまうということを説明する. さらに, アトミックなデータの永続化が必要であるということを, 現在のファイルシステムのセマンティクスにおいてアトミック性に関する脆弱性が存在していることを明らかにした研究, およびアトミックなデータ永続化のための API を備えたファイルシステムに関する研究を参照しつつ説明する.

2.1　現在のファイルシステムの設計

図 2.1 に示すように, 現状の Linux におけるファイルシステムは仮想ファイルシステム (VFS) と呼ばれるインターフェースに相当するレイヤと, そのインターフェースの実装にあたる Local File Systems (LFS) により構成されている. VFS はユーザープログラムが永続化されたデータを扱うための API を定義しており, 抽象的な” ファイル” や” ディレクトリ” といった構造に関しての一般的な操作を行う. また, それらの実際のデータ構造の操作の実装としては, ext4, xfs, btrfs, nfs, fuse といった異なる特徴を持つ様々な LFS が存在する. この VFS とユーザープログラムがやりとりをするインターフェースとしては, `open` や `close`, ファイル入出力のための `read`, `write` のようなユーザーバッファをファイルにコピーさせる API, `mmap` というファイル自体をアドレス空間にマップし, load/store 命令であたかもファイルのデータを操作できるように見せる API などを備える. これらを利用すると, VFS はその実装にあたる LFS のコールバックを呼び出す. LFS

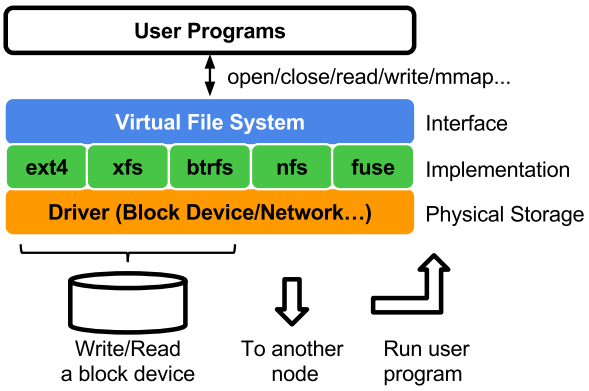

図 2.1. 現行のファイルシステムの階層構造

では, 対応するブロックデバイスやネットワーク上のデータ, ユーザープログラムなどに格納したデータ構造を用いて, API の呼び出し側から要求されたファイル操作を実行する.

現在のファイルシステムの仕組みでは, ブロックデバイスを DRAM と比較し圧倒的にレイテンシの大きいものとして考えて設計されている. このレイテンシを隠蔽するための仕組みとして, VFS のレイヤでページキャッシュという機構が実装されている. ページキャッシュとは, ファイルごとに管理されるキャッシュ機構であり, ファイル内オフセットをキーとする基数木によって保持されている.

このページキャッシュ機構の動作の手順について次に説明する. VFS は, 初めてユーザープログラムからファイル IO を要求された際には LFS へ問い合わせを行う. この際に, LFS はブロックデバイスなどのバッキングストアとのやりとりを行い, ファイル内オフセットに対応したデータを返す. そして, VFS はこの受け渡されたブロックを一旦 VFS 内のページキャッシュに格納しつつ, read や write の操作を行う. こうすることで, 次回以降の read/write の操作は LFS へ問い合わせることなくページキャッシュとのやりとりでよくなり, DRAM のコピーだけで済む. しかし, ここでページキャッシュを実際にディスクに書き戻すタイミングというのが問題になる.

2.2　現在のファイルシステムにおけるセマンティクス

現在のファイルシステムでは前述したページキャッシュにより速度の向上を図っている. しかし, ユーザープログラムがデータ永続化のための write や mmap した領域に対する store 命令の発行といった処理を行ったにもかかわらず, 実際には永続化が遅延されるという場合が存在する, ということになってしまう. この状況で電源喪失といった障害が発生すると, 永続化されたデータが失われることになる. そこで, 現在のファイルシステムはどのようなセマンティクスでデータの永続化を行っているのかを理解する必要がある.

現在のファイルシステム上では, データの永続化はページキャッシュからブロックデバイスへと追いだされたタイミングで行われる. この追い出しの操作は, フラッシャープロセスと呼ばれるカーネルがブロックデバイスごとに保持するプロセスが行う. このプロセスが動作するタイミングは通常, プログラマが指定するものではなく, ファイルシステムが定期的に実行したり, ダーティーページの割合が高くなったり, メモリが不足してキャッシュを解放するときに実行される.

このように, 通常ではデータの永続化のタイミングに関してはユーザープログラムは関与しない. しかし, VFS はデータの永続化に関わる API として, `sync`, `fsync`, `fdatasync` や `msync`, Linux であれば `sync_file_range` などを提供している [16]. 機能に多少の差こそあるものの, いずれもその API 呼び出しが完了した時点でデータが永続化されたことを保障するものである. この API の枠組みの中でユーザープログラムを記述すると, 永続化に関して (1) なにも書かれていない状態, (2) 永続化されているかもしれないがされていないかもしれない状態, (3) 確実に永続化されている状態, という 3 つの状態が生じる. この例として List 2.1 を用意した.

この 3 つの状態のうち, 特に (2) が永続化の保障の上で問題となる. この例では `some_func` を利用すると `some_data` を write の API を利用してファイルに書き込むが, 仮に `some_data` が巨大な配列だった場合, (2) の状態ではどこまで永続化されたのか不明である. 仮に (2) の部分で電源喪失といった障害が発生した場合, `some_data` のどこまで永続化がおわったのかもわからない上に, `some_data_t` の構造体のうち, 一部のメンバだけが永続化されている可能性すらある. この例

から, write に渡したデータが正しく永続化されていることを保障するためには, 現状の `fsync` を始めとする, 永続化が完了していることを保障する API を用いるだけでは不十分であるということがわかる.

List 2.1. Three state of data write-backing

```
1  void some_func(some_data_t *some_data, size_t some_data_size) {
2    int fd = open("somefile", ...);
3    /* (1) Clean state */
4    write(fd, some_data, some_data_size);
5    /* (2) Ambiguous state */
6    fsync(fd);
7    /* (3) Persisted state */
8  }
```

2.3　アトミックなデータ永続化

現在のファイルシステムのセマンティクスでは中途半端にデータが永続化されてしまう可能性があるという問題は, データ永続化の API の仕様によるものであると考えられる. 現状ではファイルシステムが, ユーザープログラムにとって意味のある書き込み単位 (以降書き込みセクションと呼ぶ) を把握する手段が存在しない. そこで, 書き込みセクションを指定するための API を用意し, 書き込みセクション中のデータの更新はすべて書き込みセクションの終了時にアトミックに永続化されるという考え方を導入することで問題を回避することができると考えられる.

この書き込みセクションの開始, 終了を知らせるような API を導入したコードの例を List 2.2 に示す. この例では, fbegin という API を導入することで, 書き込みセクションが開始されたことをファイルシステムに知らせる. このあとの Write 1, Write 2 はページキャッシュに反映はされるが実際には永続化されず, バッファリングされる. 最後に fsync を呼び出すと, これが書き込みセクションの終了を宣言することになり, 従来通りページキャッシュからストレージへとデータを追い出す. このセマンティクスでは前項の例における状態 (2) をなくした, アトミックな永続化が行われる必要があるため, fsync は次項で紹介するようなアルゴリズムを用いて実装されている必要がある.

List 2.2. Two state of data update with fbegin

```
1  void some_func(some_data_t *some_data_a, size_t some_data_size_a,
2                 some_data_t *some_data_b, size_t some_data_size_b) {
3    int fd = open("somefile", ...);
4    /* (1) Clean state */
5    fbegin(fd); /* Declare the beginning of write section */
6    /* (1) Clean state */
7    write(fd, some_data_a, some_data_size_a); /* Write 1 */
8    write(fd, some_data_b, some_data_size_b); /* Write 2 */
9    /* (1) Clean state */
10   fsync(fd);
11   /* (3) Persisted state */
12
13   /* State (2) is gone. */
14 }
```

なお, 従来のファイルシステムにおいても, 例えば ext4 であれば data-journal というモードで利用すればユーザー領域の更新も一貫性を保った状態で行えるという機能があるが, これは今回ここで説明しているアトミックなデータ永続化のセマンティクスとは異なる. data-journal はあくまでも write を一つの単位として更新を反映するだけであり, 複数の write をまとめてアトミックに反映したり, mmap した領域の更新に対して明示的に書き込みセクションを指定したりすることができない. このように, 逐次データの永続化を行うというセマンティクスに基づいて設計が行われている従来のファイルシステムでは, 汎用性の高いアトミックなデータ永続化をサポートすることはできない.

このアトミックなデータ永続化をサポートするセマンティクスを導入することで, 複雑な一貫性を保つための復旧処理や更新処理をシステム側に移譲することができる. そのため, 従来のファイルシステムでは一貫性を保つような書き方がなされていなかったようなプログラムも容易に一貫性を保った状態にすることができるようになる. また, OS レベルでのサポートを行うことで, 従来ページキャッシュという OS レイヤに保持されていたデータ構造を用いて, 効率よく永続化のためのアルゴリズムを実装することが可能になる. 本論文の提案するファイルシステムである AMFS がどのようにこの機能を実装しているかについては, 4 章で詳しく説明を行う.

2.4　現在のファイルシステム上での安全なデータ永続化のためのアルゴリズム

DBMS のようなシステムでは, 任意のタイミングでのデータの一貫性 (integrity) を保障するため, VFS の API を利用しながら安全な書き出しを行えるようなアルゴリズムを利用している. ここでは, Write Ahead Logging (WAL) と Copy on Write[1] (CoW) [20] の 2 つの基本的なアルゴリズムを紹介する.

2.4.1　Write Ahead Logging

WAL はジャーナリングとも呼ばれ, 従来広く持ちいられてきたアルゴリズムである. このアルゴリズムの基本の動作は, Write Ahead という名からもわかる通り, 実際にファイルに対して変更を施す前にログという変更履歴のようなデータを書き込むというものである. これによって, ログを書いている最中に障害が発生した場合には実際のファイルには全く影響がないために復旧する必要すらなく, その後ファイルへと書き込んでいる途中に障害が発生した場合にはログが正しく書かれている保障があるため, ログを用いてファイルを復旧することが可能である.

WAL には主に Redo ログと Undo ログという 2 つのアルゴリズムが存在する. Redo ログでは新しい内容のログを取り, Undo ログではもともと書かれていた内容をログに書くという違いがある. いずれのログにしろ, データを書き込んだ場所と変更した領域のサイズ, ログが有効なものかどうか判断するためのチェックサム等の仕組みは備えてある必要がある.

また, 複数のログがまとめて有効になったことを示すため, 一般的にコミットという特殊なログ用意する. コミットログが書かれた状態の場合は, そのコミットログに関連するログがすべて有効であることがわかる. 逆に, コミットログがなかった場合にはそれに関連する処理が中断されたも

[1] shadow paging とも呼ばれる.

のとみなす. この方法により, ログが許す限り任意の数の任意のサイズの操作をアトミックに反映させることが可能になる.

WAL の特徴としては, アトミックにデータの変更を反映できることに加え, ログ領域はほぼシーケンシャルアクセスになるという点が挙げられる. 基本的にコストが大きいのはログへの書き出しであり, ログにコミットログまで書き込むことができれば, ファイル本体への書き出しはその後どのタイミングで行ってもよい. つまり, ファイル本体に関してはページキャッシュのような仕組みを利用してもよいということになり, ランダムアクセスが生じにくくなっている.

一方で, 常にログの管理の問題が生じる. 以前の終了時に障害が発生したかどうかをアプリケーションの起動時などに確認し, 仮に障害が発生していたとしたら, ログを用いた復旧を行わねばならない. これは複雑な処理となるため, 一般の開発者が耐障害性を考慮したユーザープログラムを記述するコストが高いと言える.

2.4.2　Copy on Write

CoW は, 一度データをページ単位でコピーしておき, そのコピーしたものに編集を加えた後にそのページを参照するようにする, という方式である. これにより, コピーしたページに対して編集を行っている間も元のデータはそのまま残っているため, 障害が発生したとしてもそのまま元のデータを参照することができ, コピーしたページを破棄するだけで障害復旧が可能である. この手法のアトミック性はコピーしたページへ参照を切り替えるところによって保障される. 木構造を用いて各ページを保持した場合, その木構造の根の参照を切り替えるという操作がそれに当たる. この根の参照をアトミックに行うことが可能であれば, データ永続化をアトミックに行うことができる.

CoW の特徴として, アルゴリズム中にデータ構造が壊れる状況というのが存在しないちう点が挙げられる. WAL ではログが破壊されているかもしれないため, 障害発生を検出する機構, 及びログからの復旧機構が必要となる. 一方の CoW では, 根の差し替えを行う前後ではいずれも一貫性の保たれた状態になっており, 根の差し替えという操作がアトミックに行うことができれば, 復旧処理を行う必要がない.

しかし, 従来のブロックデバイスではブロック単位でしか読み書きが行えない上, ランダムアクセス性能が低かったためにこのようなアルゴリズムを用いることが難しかったため, あまり利用されてこなかったという背景がある. SSD のようなランダムアクセスが比較的早いブロックデバイスの出現, またバイト単位での書き換えも可能な PM などを前提とするならば, CoW は有力なデータ永続化を行うアルゴリズムの一つとなる.

2.5　Persistent Memory の特性とストレージとしての利用

PM には Phase Changed Memory (PCM) [17] と呼ばれる手法を始め, Resistance RAM (ReRAM) [11], Spin Transfer Torque RAM (STT-RAM)[32, 29], Memoristor[31, 3] といった多様な手法が提案され, 盛んに開発が行われている. この PM の特徴としては, DRAM に匹敵する低レイテンシや, SSD で用いられている NAND Flash が $10^4 - 10^5$ 回程度の書き換え回数 [14] なのに比較して PCM であれば 10^8 回 [15] などと非常に多くの書き換え回数を誇る点が挙げられる.

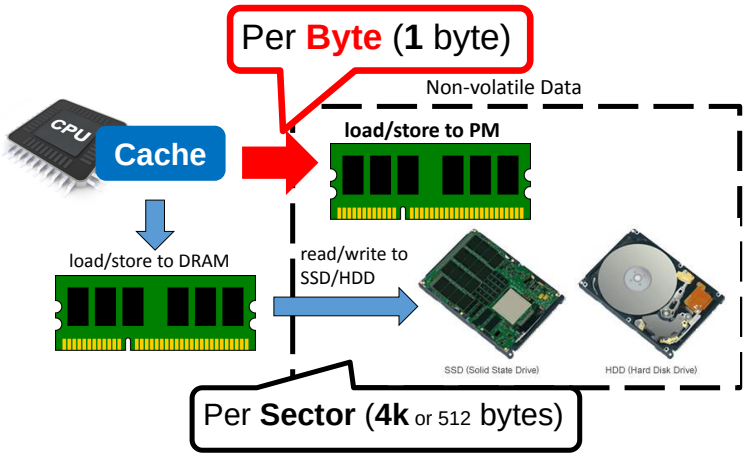

図 2.2. Difference between PM and block devices

また, 最大の特徴としては Byte Addressability と呼ばれる, 1 バイトずつ読み書きが可能であるという点が挙げられる. 図 2.2 のように, 従来のストレージではブロックという単位でしか読み書きできなかったため, それに合わせて現在のファイルシステムはインターフェースが設計されている. そのため, 永続化するための API に関して, 大幅な再設計が必要となると考えられる.

また, 書き出し回数に制限があるというのは依然として利用に大きな制約が課されると考えられる. NAND Flash と比較して書き換え回数が 1000 倍になったとはいえ, 10^9 というのは 1GHz で書き換えれば 1 秒で書き潰してしまう程度の耐久性しか無いと言える. また, PM ではバイト単位の書き換えが可能であるため, ブロック単位で管理すればよかった NAND Flash に比較し, 書き換え回数制限の回避策であるウェアレベリングをより細粒度かつ高速に行う必要があるという技術的難易度の高い問題を抱えており, その解決を図る Start-gap Wear Leveling[23] といった技術も研究開発されている.

2.6　PM 上で高信頼なデータ永続化をサポートする際のパフォーマンス要求

このような低レンテンシであるという特徴を持つ PM を現在のシステムで利用することを考えると, 従来の HDD や SSD の代替として PM を利用することになる. しかし, その場合にはアトミックなデータ永続化を行うために必要なログの構築や CoW を, データの更新の度に行わなければならず, パフォーマンス向上の足かせとなると考えられる.

例えば PCM では書き込みレイテンシが 150ns 程度 [17] と言われているが, これは 2GHz の CPU では 300 クロックである. ランダムアクセスでさえも 300 クロックでできてしまうため, 十分な性能を活かすためにはログの構築などが 300 クロック以内で行えないと, ストレージの速度を十分に活かすことができていないことになる.

そこで, PM を用いた場合には, データ永続化を高速に行うことができるような永続化手法が求められる. 毎回の書き込みに対して余分な計算を行うのではなく, 書き出す際にまとめてアトミックな操作を行うことで, 書き出し操作を高速に行えることが期待できる.

第3章　関連研究

3.1　アトミックなデータ永続化に関する研究

3.1.1　安全性の評価

T. S. Pillai ら [22] は, データベースやバージョン管理システムといったデータを管理するアプリケーションに着目し, このような不可分操作が正しくアプリケーション上に実装されているか調査するためのツールを実装した. その結果, アプリケーションにより想定しているファイルシステムの性質が異なっており, 時には一貫性が保たれない場合があることが示された. この論文ではそのようなファイルシステムの性質のことを Persistent Property と呼んでおり, これは操作の不可分性と順序の 2 つから構成される. 不可分性では 1 セクタの上書き・追記や複数ブロックの読み書きといった操作が, 順序ではデータの上書き後に他の操作を行った場合に順序が保持されるか, といったことが挙げられている. 特に, 不可分操作を実装するために用いる操作である `rename` は, ファイルシステムによっては不可分操作ではなく, データの一貫性を確保する上での脆弱性となる.

3.1.2　アトミック操作を備えたファイルシステム

不可分操作に関してファイルシステムがユーザーに対して明示的に保証するための API が不可欠であると言える. そのような API を提供したファイルシステムとして, Valor[30], AdvFS[33] などが提案されている. 前者では, DBMS と同様の ACID 特性を備えたログ書き出しをサポートするための 7 つのシステムコールを提案しており, その有用性について議論を行っている.

一方の AdvFS では, 論文中で *consistent modification of application durable data* (CMADD) という特性についての議論を行っている. これは, 永続化されたデータの編集中に障害が発生しても, データが復旧不可能な状態にならない, という性質のことである. これを保証するため, AdcFS では `open` に `O_ATOMIC` というオプションを追加した. このオプションをつけると, ファイルシステムは `fsync` ないしは `msync` が呼び出されるまでデータの変更を反映しない. これにより, セマンティクスをデータの変更が反映されていない状態, もしくはデータの変更が不可分に反映された状態のいずれかに限定することができ, ユーザーは一貫性のあるプログラムを記述することが可能になる.

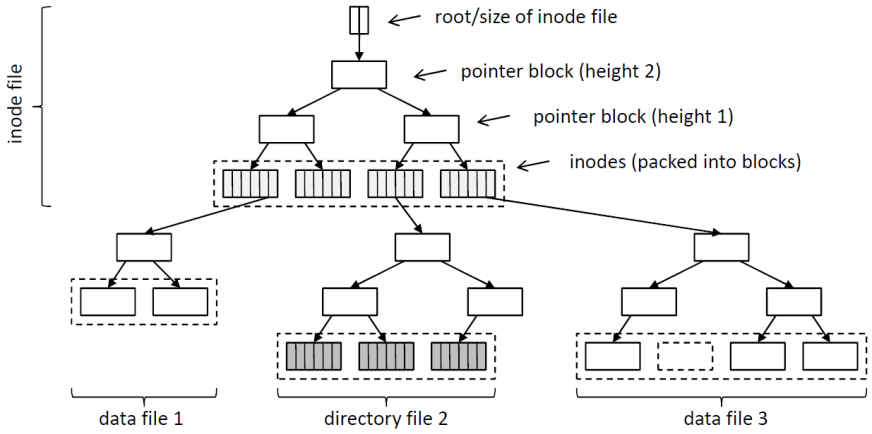

図 3.1. Data placement of BPFS[13]

3.2　従来のファイルシステムと互換性を持ったPM向けのデータ管理手法

3.2.1　BPFS

J. Condit らは, BPFS (Byte-addressable Persistent File System) [13] と呼ばれる, より安全で一貫性の保たれたファイルシステムを提案している. また, この論文ではただファイルシステムの設計を変更したのみではなく, 2 つのハードウェア的な工夫が PM の効率的な利用には必要であるという点についても述べているという点で, それ以降の PM のアルゴリズムを提案する際にはほぼ必ず引用されている.

ファイルシステムの説明に入る前に, 簡単なファイルシステムの構成について紹介する. 現在広く利用されているファイルシステムとしては ext4 や NTFS といったものがある. これらのファイルシステムは, ブロックなどと呼ばれる一定サイズ (4kB など) の領域に区切られて管理されており, ファイルやディレクトリの情報をもつ inode やユーザーが実際に生成したファイルを保存するデータブロック, ディレクトリの情報を持つディレクトリブロックなどが, ポインタのような参照関係を持ちながらツリー状に保持される. NTFS や ext4 のようなジャーナリングファイルシステムと呼ばれるファイルシステムは, ファイル生成や削除といった操作を行う際, このツリー構造を操作する前にログを残すことでファイルシステムの破壊を防ぐ工夫がなされている. しかし, ジャーナリングファイルシステムではファイルシステム自体の一貫性の保証しか行っていないため, データブロックへの書き込み中に電源喪失が発生するとデータは壊れてしまう.

データの故障を防ぐための工夫としては, 前章で紹介した WAL (Write Ahead Logging) や Copy on Write (もしくは Shadow Paging) と呼ばれる手法が用いられる [20]. このうち, Persistent Memory の登場により, データの読み書き, 特にランダムアクセスが高速に行えるという特徴を活かすことで CoW が利用しやすくなった. そこで BPFS では, CoW に in-place な変更を追加することで効率的な実装をした, *Short-circuit Shadow Paging* を提案し, 利用している.

BPFS における全体のデータ構造を図 3.1 に示す. root の inode ファイルの位置は PM 上の予め決められた位置に置く. また, ポインタブロックは図中では子ノードが 2 つだが, 実際には 512 ノード持っている. 一つのブロックは 4KB, ポインタは 64bit であり, ポインタブロックにはポイ

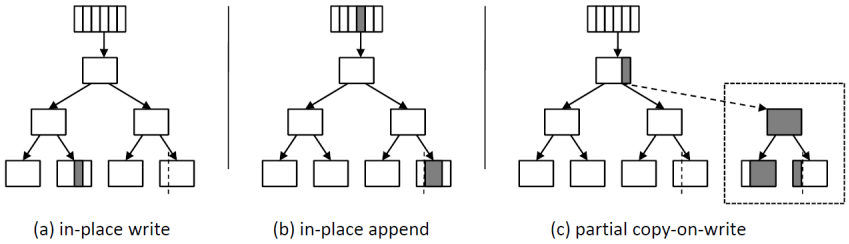

図 3.2. Short-circuit Shadow Paging[13]

ンタ以外保持されてない. また, 図中の点線の部分はそこに実際のデータやディレクトリの情報が保持されていることを示している.

このようなデータ構造を持つデータに対し, 常に一貫性を持った状態でアトミックにデータを書き込む手法が Short-circuit Shadow Paging である. BPFS においてデータを書き込む際の模式図が図 3.2 である. 図 3.2(a) が示すのは, 書き込むデータサイズがアトミックに書き込める程度の大きさであれば直接データブロックに書き込むという操作である. 後述するが, BPFS では 8byte のアトミックな書き込みをハードウェアで保証するという前提があるため, この場合は 8byte までのデータは直接データブロックに書き込むことになる. 図 3.2(b) は, データブロックに追記する際の操作である. データブロックに追記する際には, データを始めに追記し, それが終わってから inode に記述されているファイルサイズを変更することでデータにアトミックな追記を行うことができる. 図 3.2(c) は, 実際に Shadow Paging を行う操作である. 始めに変更を加える箇所を全て含む, 最も葉ノードに近いポインタを探す. 次にそのポインタが指すノード以下をすべてバッファへコピーし, データに変更を加える. データへの変更が終了したら, 最後にコピー元ノードと編集されたノードをポインタを差し替えることで変更する.

ここまでで既に触れたように, BPFS では一つ目のハードウェアに対する要求として, 8byte のアトミック性を保った書き込みを挙げている. これにより広く持ちいられる最も基本的なデータ構造であるポインタやカウンタといったデータを常に正しく書き込むことができるようになる. また, もうひとつの要求として, Epoch Barrier と呼ばれるバリアを提案している. これは, キャッシュが揮発性であるため, 正しくデータの前後関係を保つために必要となる機能である.

今回は PM を DRAM と同一アドレス空間にマッピングする前提で考えられているため, CPU から見ると DRAM と区別がつかず, キャッシュの機能も同様に働くと考えられる. しかし, 通常キャッシュは揮発性であるため, PM に書き込んだデータがキャッシュに残ってしまうと正しく操作が行えたとは言えない. また, より悪いことに実際に PM にストア命令を発行した順序とキャッシュから追い出される順序が逆になってしまうことでデータの一貫性が保たれない可能性もある. 例えば, データの追記を行う際に, 本来であれば追記後にデータサイズを書き換えているが, データサイズの含まれるキャッシュラインのほうがデータ自体よりも先に追い出されてしまう, というようなシーンが考えられる. このような問題を防ぐ単純な方針としては, キャッシュのポリシーを Write Through にするといったことや, mfence を行った際にキャッシュを全て追い出すようにするといった操作が考えられる. しかし, キャッシュが効かないということはそれだけ速度が落ちるということでもあり, これは PM を受け入れがたいものとしてしまう. そこで, BPFS では Epoch Barrier と呼ばれる機能をキャッシュに導入することで, このようなデータの順序に関する問題を

解決した.

Epoch Barrier というのは, 小さい Epoch を持つものが必ず先にメモリに追い出されることを保証するものである. 似たようなものとしてはメモリバリアがあるが, これはある時点の前後でアウトオブオーダー処理が行われないことを保証する一方, Epoch Barrier は単純にキャッシュからメモリに追い出される順序のみを保証しているという点で異なっている.

手法としては, まずスレッドを独立して実行できる単位, ハードウェアコンテクストごとに Epoch カウンタを用意し, 現在実行中の Epoch を記憶する. さらに, 各キャッシュラインについて 1bit の Persistent ビットと 3bit の Epoch ポインタを用意し, このラインは PM からのものかどうかという情報と, このキャッシュラインの所属する Epoch に関する情報が記録されているテーブルの番号を保持する. Epoch ポインタが 3bit なのは, 今回想定したハードウェアは 8 個の Epoch がキャッシュ上で共存することを許容しているためである. また, Epoch に関する情報としては, Epoch カウント, 及び残りの書き込み回数が必要となる. この書き込み回数というのはその Epoch の間に何度書き込むかという情報であり, Epoch が完了したかどうかを確認するために必要な情報となる. これを用いることで, Epoch カウンタの若いものが終わっていない場合に他のハードウェアコンテクストから上書きされてしまうことを防ぐことが可能になる.

3.2.2　PMFS

PMFS[9] では, PM は十分に速度が DRAM に近く, 高速であるという立場をとり, 完全にページキャッシュを廃したファイルシステムとして実装された. この PMFS は, 次のような特徴をもつ.

- Linux の XIP (eXecution In Place) というインターフェースを利用し, ページキャッシュを回避する API を提供
- メタデータの更新は Undo ログによって細粒度に管理することで一貫性を確保
- mmap 時に直接 PM 領域をマップすることで, ファイルのデータが直接操作可能
- CPU に備わる SMAP (Supervisor Mode Access Prevention) という機能やコントロールレジスタの書き込み保護ビットなどを利用し, カーネル・ユーザー空間間での安全なデータの更新を実装
- PMEP という専用のエミュレーターにより, PM 領域へのアクセスをトラップし, 適当に PM 領域へのアクセスを遅らせることで正確な評価を実現

特に重要なのが mmap であり, PMFS は huge page なども利用しながら, load/store 命令による直接のファイルデータ操作を実装し, 評価している点が特徴的である. PMFS の論文中では, DRAM にページキャッシュを確保しないことにより, 読み書きの性能が 2.3x - 14.3x 高速になったとしている. 一方で, PMFS ではアトミックな永続化に関してはメタデータ領域にしか実現しておらず, inode の情報更新は Redo ログによって行われる一方でユーザーデータに関してはログを取るなどの操作は行っていない.

本論文の提案する AMFS はこの PMFS をベースに実装が行われており, 具体的な実装については 4 章で述べる.

3.3　全く新しい PM 向けのデータ管理手法

PM では DRAM と同様のアドレス空間を持たせてストア命令やロード命令を直接発行することができるため, 従来のブロックデバイスを前提とした write/read といったシステムコール経由でデータを扱うのとは異なった, DRAM と同様の API を提供するということも考えられる.

DRAM と同様というのはつまり直接変数に代入するといった操作を前提とするということであるが, ここでも前述の通りキャッシュの揮発性と PM の不揮発性という非対称性が問題となる. このような点に注目した研究は NV-Heaps[4] や Mnemosyne[35], また NVM 一般に利用できるより大きな API を提案しようとしている OpenNVM プロジェクト [21] など多数存在する. ここでは, そのうち Aerie[34] というユーザーライブラリとして永続化されたデータを扱うものと, NV-Heap/Mnemosyne という PM 領域を利用するためのアロケータに関する研究を取り上げる.

3.3.1　Aerie

Aerie[34] は, 従来のファイルシステムに相当する, ファイルの概念やデータ構造の管理, パーミッションなどの機能をユーザー空間にあるライブラリを中心にして実装したシステムである. このシステムでは, カーネル空間への問い合わせを可能な限り減らし, 不揮発性メモリをユーザー空間で読み書きすることで直接ファイル操作をすることでパフォーマンスを改善することを目指した. また, このときに Trusted FS Service というデーモンを立ち上げておき, パーミッションの管理やファイルのメタデータの管理などをそのデーモンが司ることで, OS レイヤの責任を最小限にしている.

この手法を用いると, 許可が与えられたファイルへの読み出しリクエストはユーザー空間のライブラリ (libFS) 内でメモリを読み出すだけで済む. ファイルのメタ構造を辿って特定ファイルの操作をする場合でも, inode に相当するオブジェクトを走査するのはライブラリが適当なデータ構造をたどるだけとなる. この読み書きのパーミッションは, TFS にリクエストを投げることによって得られる. TFS は 1 プロセスで集中的にパーミッションなどを管理しているため一見ボトルネックになりそうではあるが, 実際には TFS に投げなければならないリクエストというのはメタ情報の操作やファイルの権限のリクエストなどに限られているため, 実際のファイルの操作はユーザープログラムから直接メモリ操作として行うことができるのが利点である.

また, もうひとつの特徴として, データ構造をユーザーライブラリレベルで自由に決められるという点が挙げられる. Aerie ではメモリ領域の確保こそ TFS などにリクエストしなければならないものの, ファイルの単位やデータ構造, API はすべて libFS の設計に委ねられている. そのため, 使うライブラリによって異なる API を持つというような設計が可能になり, アプリケーションに最適なデータ構造を選ぶことが容易である. 論文中では, POSIX API にほぼ準拠している API をもつ PXFS と, 独自の API をもち, 小さいファイルを高速かつ大量に確保できる KVS のような API をもつ FlatFS の 2 つが実装・評価されていた.

この Aerie は, PM 向けに最適化されたストレージとの API のひとつといえ, 従来のファイルシステムの構成とは全く異なった方向性であるといえる.

3.3.2　NVHeap/Mnemosyne

NV-Heap は, BPFS で紹介された 8Byte のアトミックな書き込みや Epoch Barrier を利用しつつ, PM 上で扱えるヒープ領域を提供し, また揮発性の領域と不揮発性の領域を区別しながらオブジェクトを扱うための基本となる考え方が提案されている. 更に, トランザクションの実装に関しても触れており, 評価においては Memcached[19] や, 同一 API で BerkleyDB[6] を用いて永続化させる機能を追加した Memcachedb, 及びそれを NV-Heap を用いるよう変更したものとの間で性能比較を行っている. また, 論文中で筆者らは BPFS は Epoch の管理やスレッドの競合, アトミック性などのにより一般的なユーザーには優しくないという点を挙げ, NV-Heap の API の簡潔さという観点からの優位性も述べている. list3.1 に, 線形リストの中から値 k を持つものを除去する例を示す. この例では, NVList の宣言と remove 関数による NVHeap の利用法を見て取れる.

List 3.1. NV-Heap example[4]

```
1  class NVList : public NVObject {
2    DECLARE_POINTER_TYPES(NVList);
3  public:
4    DECLARE_MEMBER(int, value);
5    DECLARE_PTR_MEMBER(NVList::NVPtr, next);
6  };
7
8  void remove(int k)
9  {
10   NVHeap *nv = NVHOpen("foo.nvheap");
11   NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
12   AtomicBegin {
13     while(a->get_next() != NULL) {
14       if (a->get_next()->get_value() == k) {
15         a->set_next(a->get_next()->get->next);
16       }
17       a = a->get_next();
18     }
19   } AtomicEnd;
20 }
```

NV-Heap で特に特徴的な部分としては, 不揮発性メモリを利用する際に (1) 永続的にデータが残るため, メモリリークに関する制約が厳しくなる点, (2) 不揮発性の領域に揮発性の領域のポインタを保持することができない点, (3) トランザクションのサポートをヒープのレベルで行う点の 3 つが挙げられる. これらについて簡単に説明する.

まず 1 つめのメモリリークに関する解決であるが, NV-Heap では参照カウントを利用している. 参照カウントを行う場合に問題になる循環参照時に正しくメモリが開放されないという点に関しては C++11 などと同様, 弱参照型を利用することで解決ができると記述されている. しかし, この点に関しては筆者らの述べる簡潔性という点では問題があると思われ, 考慮する余地があると思われる. また, その参照カウントを保持する際にスレッドセーフな変更をするため, 各オブジェクトにはロック機構が含まれる. しかし, 前提として PM を利用しているため, ロックが保持された状態で電源喪失が起こった場合に対処する必要がある. そこで, ロックに通常のロックとは異なる *generational locks* と呼ばれる実装を用いている. これは, ファイルとして保持されているヒープ領域に対し 1 つの *generation number* が保存されており, ファイルを開くたびにインクリメントされる番号を利用する. これにより各オブジェクトに保持されているロックが generation number

より小さい場合は電源喪失以前のものだと判断できるため, ヒープを開く際に全てのロックをリリースすることができる.

次に, 2 つめの不揮発性の領域に揮発性のポインタを保持できない点について説明する. これは, 揮発性の部分に関するポインタはプロセスの起動のたびに変更されるため, 不揮発性領域に保存してしまうと一貫性が確保できないという問題である. そこで, NV-Heap では不揮発性領域に関するポインタを NV-to-NV, V-to-NV, weak NV-to-NV の 3 種類に限定している. また, 弱参照は参照先のオブジェクトが破棄された時点で NULL にする必要があるが, プロキシオブジェクトを経由することにより, プロキシオブジェクトの参照先を変更するだけで全ての弱参照を変更することができるという工夫も行われている.

最後に, 3 つめのトランザクションに関しては, NV-Heap では ACID 特性を保持するためソフトウェアトランザクショナルメモリに似た手法をとっている. 具体的には, データを書き込む際に Write ログにデータをコピーしてから元のオブジェクトを変更するという操作を行う. また, バージョン番号を導入し, トランザクション開始以降読み込むデータに関しては読み込むタイミングでそのオブジェクトのバージョン番号をログに残し, コミット時に Abort するかどうかの判断を行う.

Mnemosyne[35] では, より既存のライブラリに近い API を提供している. こちらでは言語的な拡張を施し, `pstatic` 修飾子や `pmalloc`, `log_append` といった, 一般的な開発者にも馴染みのある機能を PM 向けに拡張することで容易な操作を可能にするものである. PM 領域のポインタをコンパイラのレベルで解釈することにより, NV-Heap とは異なって API 呼び出しではなく, 直接の load/store をおこないつつも NV-V 問題などに関して安全に扱うことができる. このような言語拡張によって, PM 上の領域を DRAM と同様に扱えるようにすることは, Byte-Addressability を十分に活用することができるといえ, 有用である.

このようなオブジェクト単位での永続化という観点からの研究も, 古くから行われているオブジェクトストレージなどの研究も踏まえつつ, PM のランダムアクセスに対する強みを有効に利用するという点で再度検討する余地があると言える.

3.4　ファイルシステム開発の方向性

ここまでで, PM 向けのデータアクセス手法に関するアプローチを多数紹介した. これらをまとめると, PM の Byte Addressability を活かすために抽象化を薄くしつつ, アトミック性を保つ API を持たせたいという全体の方向性が見えてくる. また, これらの研究を永続化したデータの管理方法に着目して大別すると, 従来のファイルシステムとの互換性を保つものと, 全く新しい, ユーザー空間で永続化されたオブジェクトを触ることができるようにするものの 2 つに分類することができそうである.

その中でも, 従来のファイルシステムとの互換性を考慮したものというのが, 従来の膨大なソフトウェア資源を利用できるという意味で, より普及しやすいと考えられる. そこで, 現実的には従来のファイルシステムの API, すなわち read/write といったものをサポートしながらも, レイテンシを削減するために, VFS や LFS, さらにその下のブロックデバイスに相当するレイヤが薄くなるだろう. また, ブロックデバイスに相当するレイヤとしてメモリの抽象化が起こるとも考えられる. 実際に linux-nvdimm が実用環境への実装の始まりであると言え, 今後広まっていくと推測

できる.

すなわち, 現実的な PM の利用の方向性を考えると, ファイルシステムとしても利用が可能であり, アトミックなデータ永続化 API を持ち, さらに抽象化レイヤが薄く, 余計なデータ操作を行わないというシステムが求められていると考えられる.

第4章　AMFS (AtoMic File System)

本稿で提案する AMFS は，2 章に述べたアトミックなデータの永続化をサポートしつつ, 次世代不揮発性メモリに適したデータ操作を行えるファイルシステムとして設計・実装がなされた. 本章では，その 2 点についての詳細を記載する.

4.1　AMFS における永続化されたデータの構造

まずはじめに, AMFS において永続化されたデータがどのように格納されているかを説明する. これにより, CoW がどの部分のポインタをアトミックに更新することで成り立つのか, ということが理解できるようになる.

今回実装した AMFS は PMFS をベースにしているので, 基本的なデータ構造は PMFS と同様である. そのデータ構造の模式図が図 4.1 である. ファイルやディレクトリごとに inode という単位で管理されており、inode は meta inode blocks を用いた B 木によって管理されている. PMFS と同様に, meta inode blocks は 4KB のページとなっており, それぞれのページには 8 バイトのレコードが 512 個保存されている. 各レコードにはブロック番号が保持されており, 実質的にブロック番号は PM 領域の先頭の物理フレーム番号でオフセットされた, Linux におけるページフレーム番号である. このブロック番号をリトルエンディアンで保持している. この B 木のキーとなるのは inode 番号であり, super block からたどることができる.

特定のファイルを open するときには, そのファイルに対応する inode オブジェクトを見つけることでデータブロックにつながる B 木にアクセスすることができる. この B 木を meta data blocks と図中で表記している. この B 木のキーはファイルの位置であり, ファイルの頭からいくつめのブロックにデータがあるかということがわかれば B 木をたどることで目的のブロックにアクセスすることが可能である.

これらより, AMFS 中のすべてのデータは inode と data の 2 つの B 木によってアクセスが可能であることがわかる. AMFS では, この 2 つの B 木のうち, data について CoW を行うことでアトミック操作を可能にする. 次項では, この CoW についての詳細を説明する.

4.2　OS レベルのアトミック操作のサポート

AMFS では，データをストレージへ書き込んでいる途中に障害が発生すると中途半端な状態でデータが永続化されてしまう問題を解消するため，新たに *fbegin* というシステムコールを導入する．このシステムコール以降のデータ操作は `fsync` や `msync` が呼び出されるまで永続化されることはなく，アプリケーションの開発者がデータを永続化させるタイミングを明示的に記述することが可能になる. また, fsync や msync をファイルシステムがアトミックに実装することを保証す

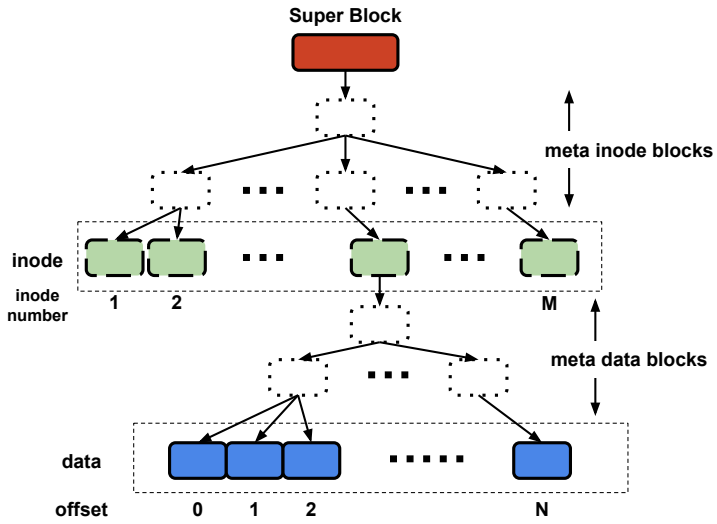

図 4.1. B 木によるブロックの管理構造

ることで, 耐障害性の信頼性の担保をユーザープログラム側からシステム側へと移すことができる. 十分にテストがなされたシステムプログラムによる耐障害性の保障は, 従来プログラムの開発者が信頼性の担保という困難な問題を解消できると考えられる. さらに, fbegin を呼び出さない際には現在のファイルシステムと同様のセマンティクスでストレージへの書き出しが生じるようにすることで, AMFS 上で従来のセマンティクスを前提としたプログラムを実行することも可能になる. 現在世の中に出回っているプログラムは基本的に従来のセマンティクスを前提としているため, fbegin を呼び出さない場合の動きを従来どおりにすることで, 互換性を保つことができると考えられる. ただし, 今回はベンチマークにおいて互換性の問題は生じないため, fbegin を呼び出さない際の処理の実装は行っていない.

図 4.3，図 4.4, 図 4.5 に，mmap を利用してデータを不可分に書き込む例を示す．なお，理解しやすくするため，引数などが一部簡略化されている．これは x と y という 2 つのデータを操作するという例である．WAL を始めとする何らかの工夫を行わない場合には，`data->records[i].x` の書き込みと `data->records[i].y` の書き込みの間で障害が発生すると正しくデータが書き込まれていない状態になってしまう．そこで, 図 4.4 では `data` へ変更を反映する前にログを書いている．これにより，障害発生時にログファイルが残っている場合には，ログファイルの最後から順にたどっていくことでデータの復旧が可能になる． 図 4.4 の 9 行目, `maybe_recover_from_log()` がこの操作に対応している.

AMFS では `fbegin` から `fsync` までの編集を不可分に書き戻すことができる．この API を用いれば，同様の操作が図 4.5 のように簡潔に記述できる．また，このセマンティクスではファイルシステムの動作は，全く変更が永続化されていない状態，もしくは完全に変更が永続化された状態の 2 つが不可分に移行する，という曖昧さがない定義なため, 今後同様のセマンティクスで実装された別のファイルシステムで利用したとしても，確実に動作する.

このように，`fbegin` と `fsync` を利用した不可分なデータ永続化は, プログラム開発者から見ると簡潔に一貫性を保ったデータの更新が行える API であると言える．一方で，そのような不可分

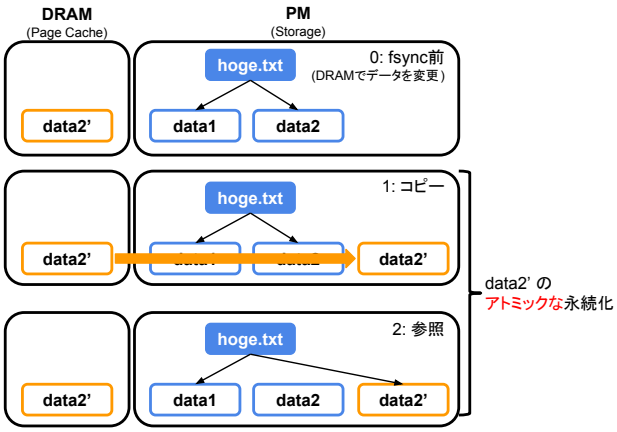

図 4.2. CoW による DRAM 上の変更の反映

```
1  typedef struct {
2    uint64_t x;
3    uint64_t y;
4  } record_t;
5
6  typedef struct {
7    uint64_t size;
8    record_t *records;
9  } data_t;
10
11 typedef struct {
12   uint64_t checksum;
13   uint64_t size;
14   record_t *records;
15   uint64_t *offsets;
16 } log_t;
```

図 4.3. 例で用いる型

データ永続化を提供するためには，ファイルシステム内で何らかの機構を用いてデータを不可分に書き込む必要がある．そのために用いる手法としては WAL や CoW などがある．

AMFS では，8 バイト単位の不可分な書き換えを前提とすることで PM での CoW の有用性を示した BPFS[5] に倣う．つまり, AMFS は PM はデバイスレベルで 8 バイトの不可分な書き換えをサポートしていると仮定しており, その上で CoW を用いて設計されている．

CoW の動きを図 4.2 に示す．`write` や `mmap` で編集されたデータは一旦ページキャッシュにとどまり，`fsync` や `msync` が呼ばれた際に CoW が実行される． 1 ページだけであれば, データをページキャッシュから PM にコピーした後に図 4.1 における meta data blocks の最下層, data block のブロック番号をアトミックに書き換えることで CoW を行うことができる．複数ブロックを書き換える場合には, すべてのブロックを一旦 PM へとコピーした後, それらのブロックをすべて含む最小の meta data blocks もコピーすることにより, CoW を行う．具体的な手順については実装の項に記載する．{ 記載したか最後にチェック }

```
// Open files
creat("log.dat");
int data_fd = open("data.dat");
int log_fd = open("log.dat");
// Map
log_t *logs = mmap(log_fd);
data_t *data = mmap(data_fd);
// Recover from Log if data is crashed
maybe_recover_from_log(data, logs);
int pos = data->size;
for (int i = pos; i < pos + 10; i++) {
  uint64_t x = i;
  uint64_t y = 2*i;
  // Append a log
  logs->records[log->size]->x = x;
  logs->records[log->size]->y = y;
  logs->offsets[log->size] = i;
  logs->size++;
  logs->checksum = calc_checksum(x, y, i);
  fsync(log_fd);
  // Modify data
  data->records[i].x = x;
  data->records[i].y = y;
  data->size++;
}
fsync(data_fd);
// Remove log file
munmap(logs);
close(log_fd);
unlink("log.dat");
close(data_fd);
```

図 4.4. 現行のシステムコールを用いた WAL の例

```
// Open a file
int fd = open("data.dat");
// Map
data_t *data = mmap(fd);
// No recover code!
int pos = data->size;
for (int i = pos; i < pos + 10; i++) {
  // Just modify data
  fbegin(fd);
  data->records[i].x = i;
  data->records[i].y = 2*i;
  data->size++;
  fsync(fd);
}
close(fd);
```

図 4.5. AMFS での fbegin を用いた値の変更例

4.3　次世代不揮発性メモリに適したページキャッシング手法

PMFS のように mmap に対する load/store を直接ファイルに対して行うと, 前項で述べたような CoW を行うことができない. 一方で, PM を用いた場合は DRAM に近いレイテンシで読み書きが行えるため, ページキャッシュを用いることでパフォーマンスの改善を得ることが従来ほど期待できず, 逆にキャッシュすることによるデータ転送量の増大がデメリットとなるとも考えられる. そこで, AMFS では read-through/write-back という戦略を用いることで, 読み出し時にはキャッシュをせずに直接読み出し, 書き出し時にはキャッシュをすることで CoW によるアトミックなデータ操作をサポートすることを可能にした.

この手法の動きを図 4.6 に示す．PM 向けページキャッシュでは, mmap された際には PM のデータ領域を仮想アドレス空間にリードオンリーでマップする．これにより，図中の Read の例のように，DRAM を経由することなく，直接読み出しを行うことができる．また，マップされたページへ書き込みを行うと，ページフォルトが発生し，そのページを DRAM 上にキャッシュし，仮想アドレス空間を DRAM にマップする．DRAM にキャッシュがあるときに `fsync` が呼び出されることで，ファイルシステムはこのダーティーページを前述した CoW を用いてアトミックに

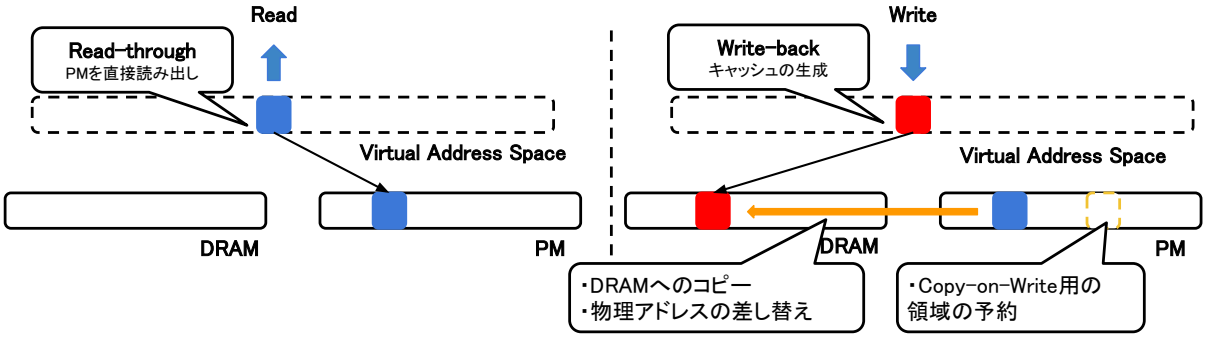

図 4.6. PM 向けページキャッシング手法

書き戻し，データを永続化する．このページキャッシュ手法を用いることにより, CoW によるアトミックなデータ永続化が可能になるだけではなく, 更に 2 つの利点が生じる．

1 つめは, PM 領域を直接マップすることにより, ページフォルトなしでデータを読み出せる点である．従来のファイルシステムでは mmap を実行しただけでは仮想アドレスの確保のみをおこない, 実際のページテーブルにはエントリを確保しない．これにより, DRAM がファイルより小さく十分な領域を確保できない場合でも, 実際に操作をおこなう一部だけ必要なタイミングで DRAM に載せることで操作が行える．これをデマンドページング [1] と呼ぶ．このデマンドページングにおいては, ページテーブルエントリの存在しないアドレスへ load/store を発行すると CPU が例外を発行し, ページフォルトを発生させる．カーネルはこれをハンドラによって処理し, ファイルシステムへと処理を渡す．このように, ページフォルトが生じると OS に処理が渡るため, ユーザープログラムの処理を一旦停止させる必要があり, 重い処理だと言える．しかし, PM を利用した場合には直接 load/store が行える物理アドレスの振られた領域にファイルのデータが存在するため, デマンドページングを行う必要がない．そこで, AMFS では mmap を行った際にページテーブルを確保することで, 読み出し時にページフォルトが生じないようになっている．これにより読み出し時には OS へ処理が渡ることなくプログラムを実行することができるため, 読み出しパフォーマンスの改善が期待できる．

2 つめは, 書き込み回数の低減である．2.5 節で説明したとおり, PM はその特性として書き換え回数に制限が存在する．AMFS では書き出し時には DRAM にキャッシュするため, 一旦 DRAM にキャッシュしてしまえば, それ以降は fsync が呼ばれた際に dirty であったときだけ書き出しが行われる．そのため, パフォーマンスを損なうことなく書き込み回数自体を減らすことができる．

4.4　実装

本項では, AMFS を実装する際に PMFS に施した変更を具体的に説明する．また, 現時点で判明している, 実装上の都合によるパフォーマンス上の問題やわかりにくい仕様についても記載する．

4.4.1　PMFS の実装についての概要

まず, AMFS の実装のベースとなった PMFS の実装について説明する．PMFS では Linux 3.11 をベースに, LFS として実装がなされている．その特徴として, ページキャッシュを使用しないた

め, XIP (eXecution In Place) という ext2 の時代に作られた API を元に実装されている点が挙げられる. なお, 本稿執筆時点の状況としては, この PMFS を元にして新たに設計された, PMEM というブロックデバイスのインターフェースや DAX というファイルシステムの API が Linux カーネルのメインラインへと追加されている [27] が, 実装開始時には存在しなかったために前のバージョンを用いていることに注意が必要である. 現在の実装では PMFS をベースとしているが, 今後の AMFS の実装は PMEM を利用して行われるべきであろう.

PMFS の実装では, mount するときに利用できる物理アドレスをマウントオプションとして指定する. mmap 自体の実装は従来の mmap と対して違いはなく, 仮想アドレス領域を `struct vm_area_struct` として管理しているだけであるが, アドレス空間に対して mixed mapping としてフラグを立てる点が通常の mmap と異なる部分となる. これにより, このアドレス領域に対応するページキャッシュについての処理がスキップされる. また, mmap 後にそのページへアクセスした際にはページフォルトが生じるが, PMFS では PM 領域に存在するページを探し, その物理アドレスをセットするだけである. そのため, 常にマイナーページフォルトとなる.

ファイルの木構造は図 4.1 に示した通りであるが, PMFS では inode より上の木構造の更新を WAL, とくに Redo ログによって管理している. ジャーナル用の inode を用意することでジャーナル領域を確保し, その領域に Redo ログを書いてから inode の内容の更新を行う. 書き換え時には flush のための命令を明示的に発行することで, メモリへのデータ永続化を確実に保障されたものとしている.

また, fsync や msync を呼び出した際には flush するための命令を指定した領域の各キャッシュラインに対して発行することで書き出しを保障している. つまり, fsync 以前の状態では, キャッシュライン上に乗っているが変更がメモリまで反映されていない状況が発生する.

次項以降では, この PMFS に対して行った変更について説明する.

4.4.2　mmap 時のページテーブルの作成

PMFS では mmap 時には `struct vm_area_struct` を利用して仮想アドレスを確保しているだけであったが, AMFS では実際にページテーブルを構築する. mixed mapping の場合にはページテーブルを `vm_insert_mixed()` を利用して作成することができるので, AMFS では以下の手順に基づき, mmap した領域に対応するページを見つけ, `vm_insert_mixed()` を呼び出している.

1. ページキャッシュの検索
   `vm_area_struct` に保持されている `mapping` 構造体がページキャッシュを基数木によって管理している. そのページキャッシュから, ページオフセットを指定してページキャッシュが存在するかどうか確認する. 見つかった場合, そのページキャッシュを利用する.
2. PM 領域の検索
   DRAM にキャッシュされたページが存在しない場合には, PM 領域からページを探す. また, この場合には mmap に `PROT_WRITE` を付けて渡していたとしても, 書き込み権限を禁止にした状態で操作を行う.

厳密には, この時点での書き込み権限の禁止はページテーブルエントリ単位で管理することができないため, Linux の API に完全に則っている仕様ではない. これは, mmap した際に Linux では仮想アドレス領域をひとつの `vm_area_struct` として確保するが, AMFS ではこの例で示した

ようにページごとに読み書きの権限が異なるからである．今回は, Linux が vm_area_struct の vm_flags で権限を管理しつつも vm_page_prot メンバを利用してページテーブルエントリのフラグを決めているという実装を利用した．vm_flags はそのままに vm_page_prot を変更することで, その直後の vm_insert_mixed() において書き込み禁止状態のページテーブルエントリを作成することができたが, これは vm_area_struct の仕様として正しい利用法なのかどうかという点に関しては調査する必要がある．

4.4.3　ページフォルトの実装

ページフォルトの実装は AMFS と大きく異なる．従来は PM に保存されているファイルの対応するページを探し, そのページフレーム番号から物理アドレスを求め, ページテーブルエントリを作成するという操作であったが, AMFS ではページテーブルは mmap の時点で既に作成済みとなっている．そのため, ページフォルトのときに呼び出されるコールバックは通常の vm_operations_struct に保持されている fault メンバではない．

書き込み禁止領域への store を行った際に MMU から生じたページフォルト例外は, Linux では do_wp_page() というハンドラ [1] が処理を行う．通常の仮想アドレス領域においては, 一度 LFS の vm_operations_struct で page_mkwrite というメンバがセットされているか確認し, セットされている場合であればその関数を呼ぶことで該当ページの書き込みを LFS 側で処理することが可能になる．通常はこの処理は fork した際などのメモリの CoW のためのものであり, page_mkwrite はただもうすぐこの領域が書き込めるようになるということを LFS に通知するためだけのコールバックである．このとき, ファイルをロックしたり LFS 側で書き込み許可を禁止することができるようになっている．この page_mkwrite を, AMFS では通常の利用法とは違い, ページキャッシュを確保するために利用した．また, 通常であれば page_mkwrite 中でページテーブルが書き換えられることは想定していないような実装になっていたが, AMFS の page_mkwrite ではページキャッシュを確保したあとにページテーブルを書き換えてしまうため, do_wp_page にも変更を施した. この実装方法では XIP を利用するようなモジュールにおいてバグを生む可能性があるが, 実際には XIP は ext2, もしくは PMFS にしか実装されておらず, ext2 は十分に古いファイルシステムであると言えるため, 実質 XIP を利用しているモジュールは他に存在しない．そのため, 今回はこのような実装方法でも問題はないと判断した．

page_mkwrite の中では, AMFS はページキャッシュの確保とページテーブルエントリの確保を行っている．ページキャッシュの確保に関しては VFS が提供する API を利用しているだけであるが, ページテーブルの操作に関しては直接行っている．これは, LFS でページテーブルを更新するための API が存在しないためであり, 現在の Linux4.2 といった PMEM のようなブロックデバイス上では必要ないことが予想される．

4.4.4　fsync の実装

PMFS の fsync ではキャッシュラインからの追い出しを行うだけであったが, AMFS ではそれに加えて CoW を実行する．その実行手順を実装に即して解説する．

[1] おそらく write-protected page の略であると考えられる．

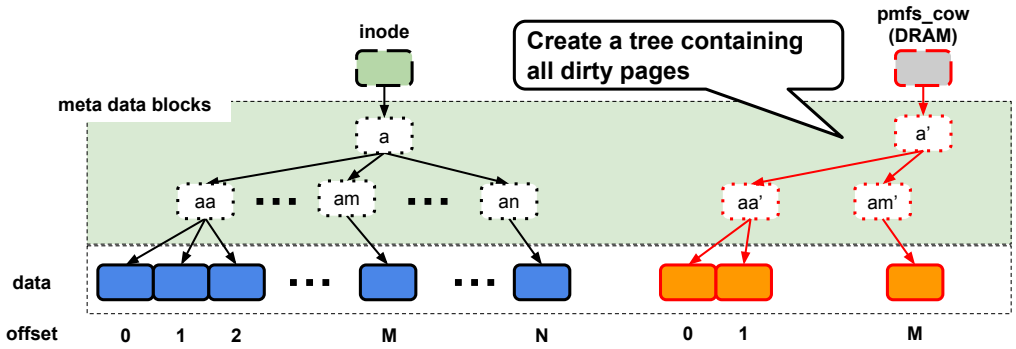

図 4.7. fsync の実行ステップ (1/5) ページキャッシュの走査と CoW 用ツリーの構築

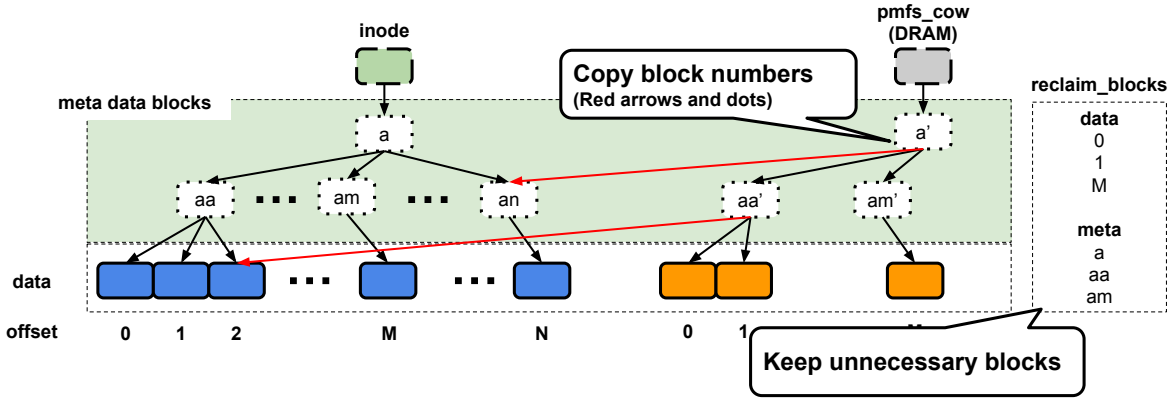

図 4.8. fsync の実行ステップ (2/5) CoW ツリーへのブロック番号のコピー

はじめに, AMFS は `pmfs_cow` 構造体を確保し, dirty なページのみで構成される木構造を構築する. この操作は, fsync を呼び出したプロセスのページテーブルを走査することで dirty なページ探し, PM 領域を確保し, 各ページをコピーすることで行われる. また, dirty ビットの立っているページを cow 用のツリーに追加する際, 追加したデータブロックがすべて収まるように meta blocks も生成する. ブロック生成時にはゼロ初期化を行うことで最終的に dirty ではなかった部分については 0 となるようにしてあり, この 0 となっている部分に関しての処理を次のステップで行っている. このステップを実行後の PM 上の状態が図 4.7 である. この図ではページオフセット, つまりファイル中の先頭からのページ番号が 0, 1, M の 3 つのページが dirty であった例を図示している. この項では今後, この 3 ページが dirty であった場合の動きに関して例として説明を行うこととする. 図の通り, このステップ終了後には 0, 1, M ページ目の data block がコピーされ, さらにそれらをたどるために最小限必要な meta data blocks (図中の a', aa', am') の合計 6 ページが `pmfs_cow` 構造体の下に追加され, ツリー構造が構築される. このツリーは, 最終的に inode が保持するファイル全体のツリーに対する部分木となるものである.

すべての Dirty なページをコピーして `pmfs_cow` 構造体に追加した後, AMFS は `pmfs_cow` 構造体に保持されている meta data blocks の 0 となっている部分を inode 側のノードからコピーする.

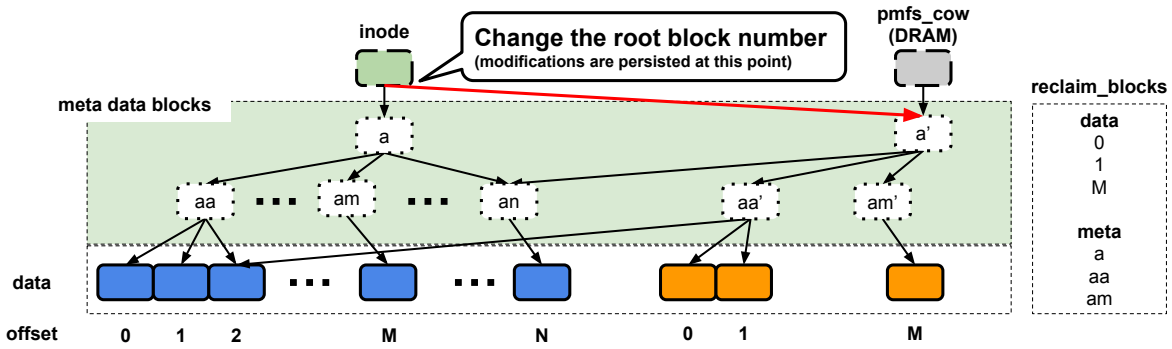

図 4.9. fsync の実行ステップ (3/5) CoW ツリーのルートへの差し替え

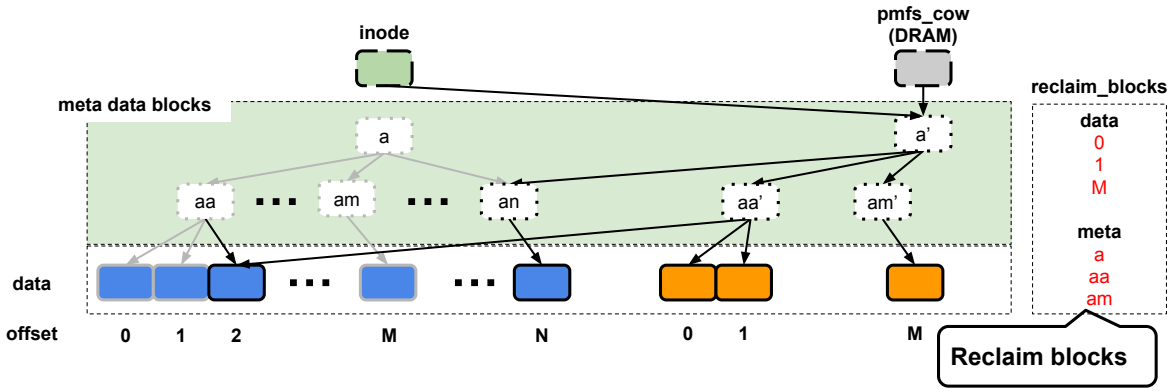

図 4.10. fsync の実行ステップ (4/5) 不要なブロックの削除

図 4.8 がその走査を図示したものである. 具体的な処理としては, 以下の手順に基づきブロック番号をコピー, もしくは削除予定ブロックリストへと追加する. 最初に, 以下のルーチンに `pmfs_cow` が確保している根, 及びそれに対応する inode 側にあるノードを与える.

1. `pmfs_cow` のノード (`dst_node`), 及びそれに対応する inode 側のノード (`src_node`) を受け取る.
2. `dst_node` および `src_node` の子ノードが meta data blocks である場合
   (a) `dst_node` と `src_node` はそれぞれ 512 個ずつ子ノードのブロック番号を持っているので、それを走査する. 以降対象となっているブロック番号を `dst_node[i]` と `src_node[i]` と表記する.
   (b) もし `dst_node[i]` が 0 なら, その葉にあたる data blocks はすべて dirty ではなかったということになる. つまり, `src_node[i]` の葉にあたる data blocks からの変更がないため, `src_node[i]` を `dst_node[i]` にコピーする.
   (c) もし `dst_node[i]` が 0 でないなら, 葉の data blocks に変更が加えられている箇所があると言える. そのため, `dst_node[i]`, `src_node[i]` を与えて再帰的にこのルーチン

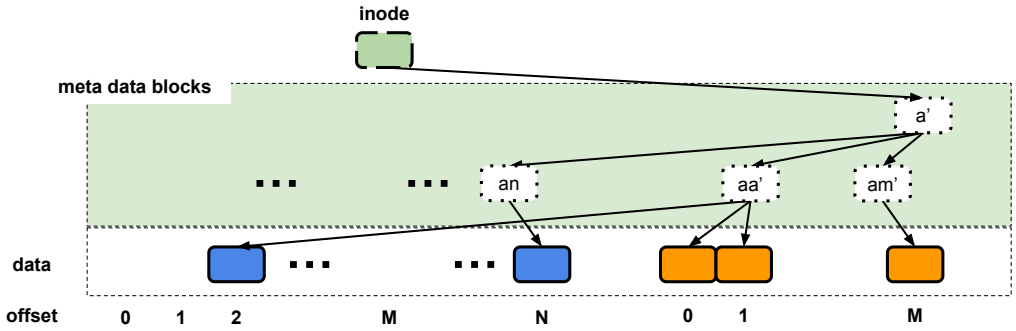

図 4.11. fsync の実行ステップ (5/5) fsync 実行後の AMFS が保持するデータ構造

を呼び出す.

3. 子ノードが data blocks の場合
   (a) `dst_node` と `src_node` はそれぞれ512個ずつ子ノードのブロック番号を持っているので、それを走査する. 以降対象となっているブロック番号を `dst_node[i]` と `src_node[i]` と表記する.
   (b) もし `dst_node[i]` が 0 なら, この data block は dirty ではなかったということになる. つまり, `src_node[i]` からの変更がないため, `src_node[i]` を `dst_node[i]` にコピーする.
   (c) もし `dst_node[i]` が 0 でないなら, `src_node[i]` が更新されて `dst_node[i]` になったということになる. そこで, `src_node[i]` を削除予定ブロックリストに追加する.
4. このルーチンを呼び出しているということは, `src_node` は `dst_node` にコピーされているということなので, `src_node` を削除予定ブロックリストに追加する.

このステップが完了すると, `pmfs_cow` 構造体の保持するツリーは inode が保持するツリーの部分木としての役割を完全に果たすことができるようになる. このタイミングで, 図 4.9 のように, `pmfs_cow` に保持された部分木と inode に保持されている木を差し替える操作を行う. この操作が完了した段階でデータの永続化が完了したことになる.

最後に, 不要なページを回収する. この例では, 図 4.10 でグレーになっている部分が削除されるノードである. AMFS ではスラブアロケータを利用し, 複数ブロック番号を記録できるチャンクを 1 ノードとする線形リストとして削除予定ブロックリストを実装している. この削除予定ブロックリストをたどり, 不要な meta data blocks や data blocks を削除すると, 最終的なデータ構造は図 4.11 のようになる.

AMFS では空き領域の管理は PMFS の実装をそのまま利用しているが, この実装は最も単純なフリーリストであり, フラグメンテーションの影響が非常に大きいことがわかった. この問題については 5 章で触れる.

4.4.5　CoW の際のデータブロック木の更新時におけるフリーリストの更新

前項の fsync の処理において, 図 4.7, 図 4.8, 図 4.9, 図 4.10 の時点で障害が発生すると, 不要ブロックが PM 中に確保されたままになってしまうという問題が生じる. このような問題が生じる理由としては, 空き領域を管理しているフリーリストと, CoW のポインタの差し替え操作をアトミックに実行できていないという点が挙げられる. この部分に関しての解決方法としてはいくつか考えられるが, それらを考慮した上で現在の実装に落ち着いているということをこの項で説明する.

まずひとつ目のアイデアとしては, Redo ログによってフリーリストと部分木の差し替え操作をアトミックに行うことを保障するというものが考えられる. これは確実にアトミックに行えるという特徴があり, 安全性には問題がない. 一方で, ポインタ操作だけでアトミック性が担保され, データが破損している状態というものがそもそも存在しない, という CoW の利点が失われていることになる. また, 小さいデータサイズであれば CoW ではなく Redo ログを直接つかうことで高速を図ることも順当な実装となると予想されるが, これは実装の複雑性を増すものであり, 好ましいものではない.

つぎのアイデアとしては, `pmfs_cow` に管理されている時点ではそのブロックはフリーリストに存在しており, ポインタの差し替えが終わったあとにフリーリストから削除するという手法が考えられる. 障害が発生した際には fsck を走らせ, すべての inode のブロックをツリーを辿って走査することでフリーリストを再構築することが可能である. 一見この実装は良さそうに思えるが, 問題は先述した, “CoW においてはデータが破損している状態というものがそもそも存在しない” という点である. つまり, WAL では commit ログの無いログの検出という簡単な方法で障害発生を検出できるが, CoW ではそもそも破損している状態が存在しないため, 障害の発生を検出することが不可能である. そのため, このような手法を取るのであればマウント時に毎回 fsck を走らせ, フリーリストを再構築するしかないが, ファイルツリーを走査するのは時間のかかる処理であり好ましくない.

最終的に AMFS に実装した手順はどのようになるかというと, 障害発生時には最大でおおよそ更新した領域分の領域が使用済み領域となってしまう. 一方でフリーリストは常に空いている領域を管理することができており, 空き領域の下界が常にわかるということになる. 空き領域が小さくなってきた際には真に空いている領域を探すためにすべてのブロックを辿る必要が生じるが, そもそも障害が頻繁には生じないと仮定するならば, このような状況は多くないと予想される. 当然空き領域が小さくなってきた場合には頻繁に空き領域の回収を行わなければならなくなるが, この問題は CoW によって失われた可能性のある領域の大きさを管理することで回避できる. つまり, fsync で `pmfs_cow` のツリーを構築する際に確保した領域のサイズを `super_block` などに保持するようにすればよい. こうすることで, 空き容量が少なくなってきた場合に, ツリーを走査することで十分な領域を確保できる可能性があるか否かを計算することが可能になる. なお, 今回は AMFS に対してこの実装は行っていない.

4.4.6　fsync をしている際に他のスレッドが mmap 領域に書き込む際の競合

複数スレッドで同じ mmap された領域を編集するような場合には, 片方のスレッドが fsync を実行中にもう片方のスレッドがその領域を編集してしまう可能性がある. このとき, どのように書

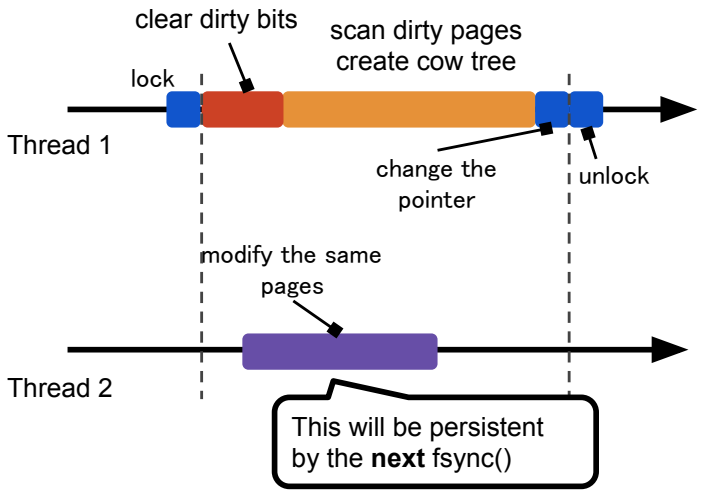

図 4.12. 複数スレッドでの書き込み時の書き出しタイミング

き出しが行われるかというのはきちんと把握しておく必要がある. 4.4.4 で述べたように, fsync の実行にはいくつかのステップがあり, そのステップに応じて永続化される状況が異なる. そこで, この項では, fsync の状況によって他スレッドから編集されるときの振る舞いがどうなるかについて説明する.

この AMFS が前提としている仕様では, ある mmap で確保された領域に関しての編集はすべて一つのトランザクションとして解釈され, fsync ではそのファイルにおけるすべての編集が永続化される. この永続化を行うかどうかが確定するタイミングは, 今回の AMFS の実装では dirty ビットをクリアする時点によって変わる.

図 4.12 はこのような状況での処理の流れを時間軸上に記述したものである. Thread 1 は fsync が呼び出されると inode のロックを取得し, dirty ビットのスキャンを行う. Dirty であるページを見つけたときには, dirty ビットをクリアしてから CoW のツリーにページをコピーする. Thread 2 による編集は, dirty ビットのクリア以前であればこのページコピーで反映される. また, クリア以降に編集が起こった場合には, (1) クリア以降, ページコピー以前であればこの fsync で反映される. さらに, dirty ビットを立てることになるので, 次回の fsync で再度上書きされる. (2) クリア以降, ページコピー以降に編集が起こった場合には, 次回の fsync で編集が反映される. という 2 つの状態が存在しえる. このような状態, 特に (1) のような状態というのは一見すると中途半端な永続化のように感じられるが, そもそも fsync を実行中に行った書き込みは fsync 以前の編集として扱うべきか fsync 以降の編集として扱うべきか不明瞭であり, ユーザープログラムの意図がシステム側ではわからない状況である. つまり, fsync 中の書き込みにおけるアトミック性に関しては考慮する必要がなく, ユーザープログラムによる編集が意図せず消えてしまうという状況だけ避けることができれば良い. その観点から再度振り返ると, (2) の状況では当然正しく書き込まれていると言え, また (1) の状況においても, 2 回同じデータが書き込まれるだけであるので, 問題ないと言える.

第5章　実験評価

本章では, 今回実装した AMFS に関する実験評価を行う. AMFS の基本的な性能を測定するため, mmap した領域に対するアクセスのバンド幅の測定や fsync, mmap 自体のレイテンシの測定を行うためのマイクロベンチマークを用意した. 更に, mmap した領域を STL の `unordered_map` をもちいて KVS として利用するワークロードによって, 実用的なアプリケーションにおける AMFS の優位性を示す.

5.1　実験環境

実験に用いたマシン環境は表 5.1 の通りである. AMFS は Linux 3.11 をベースに実装された PMFS を元にしており, 比較対象としては ext4 を RAM 上で利用した. ext4 のために用いるブロックデバイスとしては linux の brd モジュールで実装された, ramdisk を用いた. この実験設定は, 現在利用されているファイルシステムやブロックデバイスという仕組みをそのまま利用し, HDD や SSD といったデバイスの代替として PM を利用した際の環境として考えることができる. また, PM に関連する実験において PM を模した遅延を入れる場合があるが, 今回は AMFS, DRAM 上での ext4 の両者を比較することでファイルシステム自体の性能を比較できると考えたため, その遅延を導入していない. KVS としての評価では, leveldb は 1.18, SQLite3 は 3.8.1 を利用した.

5.2　mmap した領域に対する連続アクセス・ランダムアクセス

5.2.1　概要

この評価では、mmap した領域に配列を用意し、シーケンシャルな読み書き、及びランダムな読み書きを測定した. 具体的には, fallocate して作成された 250MiB のファイルを mmap し, その領域に対して (1) 連続アクセスの読み書き, (2) ランダムアクセスでの読み書きを行うというマイ

表 5.1. Environment of this evaluation

Characteristics	Value
CPU	Xeon E5-2660 @2.20GHz (8C/16T x 2 sockets)
Cache	L1: 32k, L2: 256k, L3: 20M
DRAM	DDR3-1600 8GB x 8
Kernel	Linux 3.11.0 + AMFS
Distribution	Ubuntu Server 14.04.3 LTS

クロベンチマークを用意した．このワークロードは, AMFS がページフォルト時に PM 領域からページキャッシュへコピーを生成することによるオーバーヘッドが十分に小さいことを確認するためのものである．

比較対象としては ramdisk 上の ext4 を用いる．指定されたサイズに 100 回アクセスし, 毎回計測した時間の平均をプロットしてある．グラフには 95%信頼区間をエラーバーとして記載した．エラーバーは十分に小さいため, 性能測定は安定していると言える．

5.2.2　評価

連続アクセスの性能は図 5.1 のようになった．いずれの結果も 10MB の前後で性能の低下が見られるが, これは L3 キャッシュが 20MB であることによると考えられる．10MB 以下の領域では, はじめにページフォルトしたタイミングで生成されたページキャッシュのすべてが L3 キャッシュに乗り, データアクセスが高速に行われたとすると説明がつく．AMFS と ext4 を比較すると, ほぼ性能は変わらないことがわかる．つまり, AMFS がページキャッシュを作成するコストは ext4 と比較して大きくない．

次に, ランダムアクセスの性能は図 5.2 のようになった．このランダムアクセスの性能に関しても, AMFS と ext4 に性能の差異は認められなかった．ランダムアクセス性能に関しては, プリフェッチミスやキャッシュミスによってスループットが出ないため, 余計に AMFS と ext4 の際が見えづらいと考えられる．

5.2.3　ファイルに対して直接 load/store を行うことによるレイテンシの考察

今回測定した, mmap した領域のランダムアクセスの性能はおおよそ 200MB/s から 300MB/s 程度であった．今回のワークロードは 4Byte ずつのアクセスを行うため, 50×10^6 回のアクセスを行ったと言え, 一回あたりのレイテンシは 20ns 程度であることがわかる．今回用いた計算機は 2.2GHz であるから, このレイテンシはクロック換算だとおおよそ 44 クロックとなる．ユーザーレベルのライブラリによって安全な書き込みを行おうとする場合は, 各書き出しに対してそのライブラリの呼び出しが入る．これにかかる時間を考慮すると, PM を前提にした場合はファイルシステムレベルでアトミックな永続化をサポートすることが有用だと考えられる．

5.3　fsync のレイテンシ

次に, dirty にするデータサイズを可変にして fsync のレイテンシを測定した．fsync では dirty なページを走査しそのコピーを行うため, fsync までに dirty になっているページの数によって速度が低下することが予想される．

ワークロードとしては, 以下のような動作をするプログラムを用意した．

1. 250MB のファイルを fallocate によって確保.
2. mmap によってその全域をアドレス領域に割り当てる．

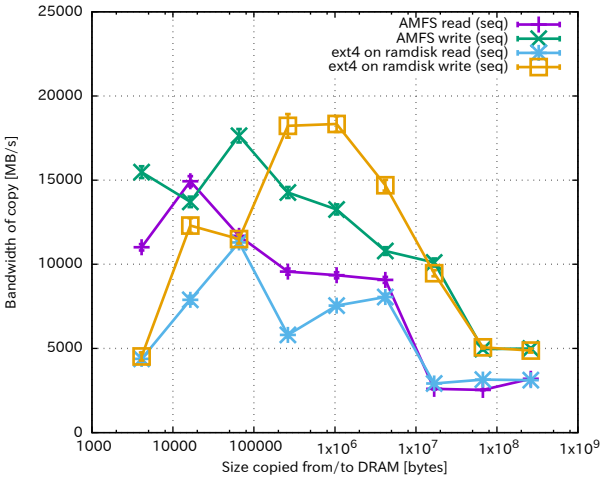

図 5.1. 連続アクセス時のバンド幅

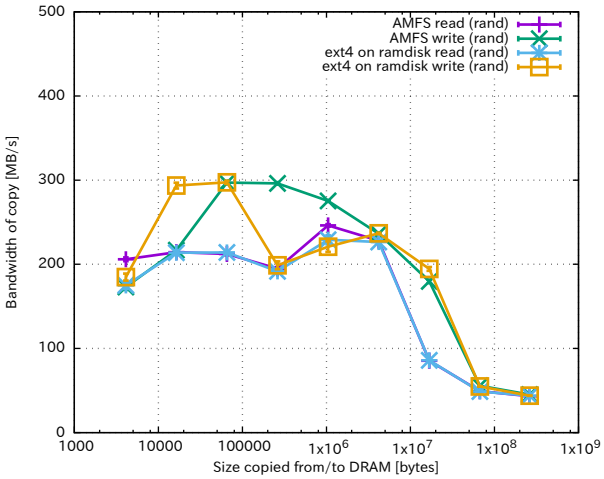

図 5.2. ランダムアクセス時のバンド幅

3. length バイト (変数) の領域だけデータを書き込む. mmap した領域は 100Byte の構造体の配列として扱っているため, 実際には length / 100 レコードの書き込みが生じることになる. 書き込む値はひとつのランダムな値を用意し, これを mmap した領域にコピーすることで書き込みとした.
4. fsync を呼び出し, アトミックにデータを永続化する.
5. 3, 4 を 100 回繰り返す.

また, 3 に関して, (1) ファイルの先頭から length / 100 レコード, (2) ファイル全体からランダムに length / 100 レコード, という 2 種類の書き込みを行った.

このマイクロベンチマークを, 表 5.1 に示した環境で実行した結果が図 5.3 である. 横軸が fsync 前に変更を施した領域のサイズ, 縦軸が 1 回あたりの fsync にかかった時間である. ここからわかるように, 10KB 以下, つまり数ページの fsync であれば安定して 1ms 未満という時間でアトミックな永続化が可能であることがわかる. 数ページの領域に関して 300us 程度の定数とも言える時間がかかっているのは, ファイル全体が 250MB と大きく, ページテーブルを走査して dirty ページを見つけ出す時間がかかっていることが考えられる. 250MB のファイルを mmap すると, 一つのページが 4KB であるため, ページテーブルのエントリを 65536 個走査しなければならない. それだけで単純に 524288 バイトであり, さらにページテーブルは木構造であるため, 一定のランダムアクセスが発生する. 524KB を 300us でアクセスしたと考えれば, 約 170MB/s であるので 5.2 節のランダムアクセスの評価と変わらない. つまり, 300us でページテーブルを走査しているという説明はおおよそ妥当であると考えられる.

また, 変更の範囲が大きくなるにつれて実行時間が延びていることもわかる. とくに連続に 1MB 以上変更した場合には, 変更範囲のサイズに対しておおよそ線形に実行時間が増加していると言える. 連続領域の書き出しでは 100MB で 51.2ms の実行時間がかかっており, これはスループットになおすと約 2GB/s である. fsync では PM 領域の確保とコピーを行っていることを考えると, 5.2 項の結果と比較しても妥当であると言える. また, 400KB 以下ではおおよそ 300us 前後とほぼ一定の実行時間になっていることから, fsync のオーバーヘッドはページの確保とコピーを除くと

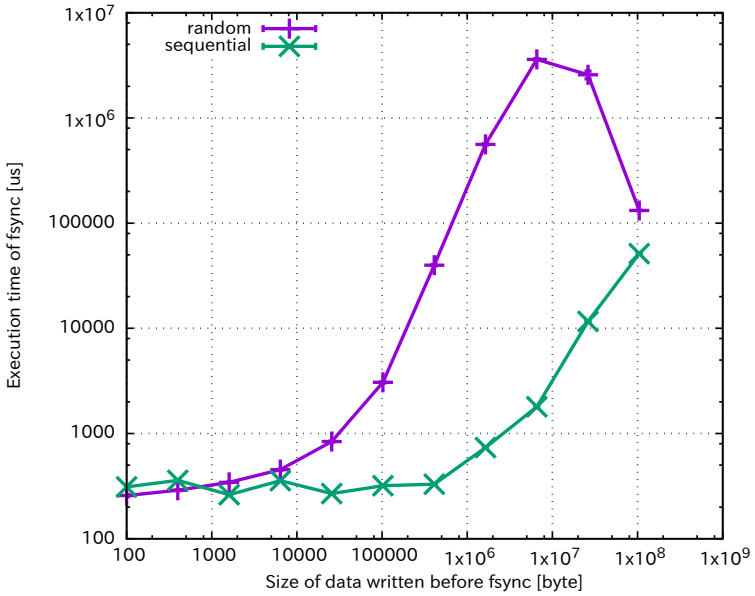

図 5.3. Latency of fsync after modification to the file

300us 程度であることがわかる.

一方, ランダムな場所へ書き込んだ際の fsync の時間が目立つ. fsync の処理はブロックの確保, コピー, ルートの差し替え, 不要ブロックの削除に分けられるが, コピー, ルートの差し替えでは位置に応じた変化は考えにくい. そこで疑われるのは, 空き領域からのアロケーションやブロック削除の処理の実行時間である. PMFS の実装では, 空き領域の管理は Linux における `vm_area_struct` のような, 最もシンプルなフリーリストによって管理されている. このフリーリストの特徴としては, アロケーションに関してはリストの先頭から確保することで O(1) で行える一方, フリーに関してはリストを線形に探索し, フリーする領域の前後の空き領域との併合処理を行うため, O(n) の実行時間がかかる. AMFS ではこの PMFS の実装をそのまま利用したため, 今回のベンチマークにおいては, ランダム領域の更新に関しては実行時間がかかってしまったと考えられる. 連続領域の更新だった場合には, データの更新はつねに頭から行うため, フリーリストの先頭のほうに空き領域が生じる. これによりフリーの処理がほぼ O(1) で完了したためにこの問題は顕在化しなかったと推測される.

また, 100MB よりも 25MB や 8MB のほうが時間がかかっていることもフリーリストの問題から説明ができる. 100MB のアップデートを行うと, ファイル全体のうち 40%の領域が更新されていることになる. それにより, CoW 後の不要ブロックはある程度連続領域となるため, フリーリストのノード数が減少すると考えられる. そのため, フリー操作自体は O(n) であるが, n が減少し, フリーの操作の平均実行時間が削減されたと考えることができる.

5.4　mmap のレイテンシ

AMFS では mmap の実行時に, mmap した領域全体のページテーブルを作成する. つまり, ページフォルトが生じない一方で mmap の実行時間が増加することになるため, mmap の実行時間を

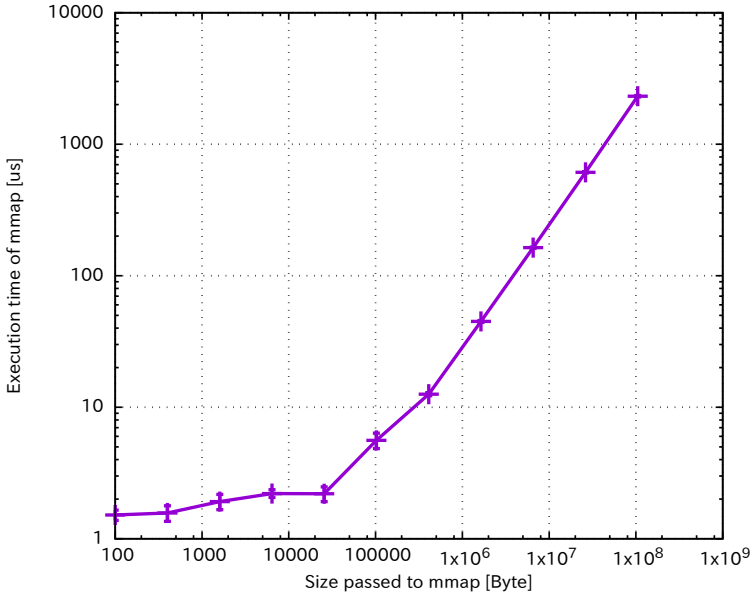

図 5.4. Latency of mmap

測定することで, その増加が十分小さいことを確認する. ワークロードとしては, 一度だけファイルを open し, そのファイルディスクリプタを用いて 100 回 mmap を呼び出し, 実行時間を測定するというものを用意した.

実行結果が図 5.4 になる. それぞれの点には 95%信頼区間がプロットされている. このグラフからわかるように, 20KB 程度まではほぼ一定, 1us 程度の実行時間となっているが, それ以降はサイズに応じてほぼ線形に実行時間は増加していると言える. また, mmap する領域が 100MB であっても 2ms 程度の時間で mmap が実行できており, 十分小さい時間で mmap できていると言える.

5.5　std::unorderd_map による KVS

5.5.1　概要

KVS (Key-Value Store) は memcached[7] や redis[25], leveldb[18, 12] などに代表される, NoSQL と言われるデータベースの種類のひとつである. KVS の特徴としては, 一般的な RDBMS とは異なってスキーマを持たず, キーオブジェクトを与えるとそれに対応した値が返ってくる, という単純な機能を高速に行うことができるという点が挙げられる. つまり, このような KVS は一種の連想配列として捉えることができる.

通常, KVS は並列にデータ操作を行えるよう, アトミック性を保証しているが, AMFS のセマンティクスでもアトミック性を保証しているため, 同等のことができると考えられる. そこで本評価では, AMFS 上にあるファイルを mmap した領域に, std::map のような連想配列オブジェクトを置くことで KVS の代用として利用できることを確認する. この評価により, AMFS を用いることで (1) 安全なデータ更新を行うことができ, (2) 高速に, かつ (3) 容易に不揮発なデータを操作可能であることを示す.

5.5.2　実装

PM のメモリ空間を容易に扱えることを示すために, 今回は出来る限り C++の STL (Standard Template Library) を利用することを考える. まずはじめに, 通常の malloc や new を使ってしまうと DRAM のメモリ空間から OS がメモリをアロケーションしてしまうので、PM 領域からメモリを確保するためのアロケータを実装した.

メモリアロケータとして通常利用されるものには glibc malloc がある. これは ptmalloc[37] というアロケータを linux 向けに改変したものになる. この ptmalloc は Doug Lea が発表した dlmalloc[8] をマルチスレッド向けに改良したものであり, 内部ではほぼ dlmalloc の API を利用しているだけとなっている. 今回の評価では複数のスレッドやプロセスから書き込むことを前提とはしていないため, dlmalloc を利用した.

dlmalloc では, mspace という機能が備わっており, アドレス空間を指定することでそのアドレス空間からメモリ空間をアロケーションすることができる. 本来であれば, そのアドレス空間の中から dlmalloc の内部情報を確保し, 内部情報の永続化を行うべきであるが, 今回はその部分の実装は行えていない. 内部情報の永続化を行っていないため, ベンチマークの起動ごとにそのアドレス空間が初期化される.

次に, このアロケータを STL の std::allocator の代わりに利用した. 標準的なコンテナである std::string や std::map, std::unordered_map のアロケータを今回実装したアロケータに変更することで, これらのコンテナクラスを pm 向けに利用することができる. 実装の一部を List 5.1 に示した. STL を利用したことにより, 利用する方法としては List 5.2 のように非常に簡潔なものとすることができた.

List 5.1. Implementation of my PM library

```
1  namespace pm {
2    using string = std::basic_string<char, std::char_traits<char>, PMAllocator<char>>;
3
4    template<class T>
5    using vector = std::vector<T, PMAllocator<T>>;
6
7    template<class Key, class T, class Compare = std::less<Key>>
8    using map = std::map<Key, T, Compare, PMAllocator<std::pair<const Key, T>>>;
9
10   template<class T>
11   class hash : public std::hash<T> {
12   };
13
14   template<>
15   class hash<string> {
16    public:
17     std::size_t operator () (const string &str) const noexcept {
18       std::string copied = str.c_str();
19       return std::hash<std::string>()(copied);
20     }
21   };
22
23   template<class Key, class T, class Hash = pm::hash<Key>, class Pred = std::equal_to<Key
         >>
24   using unordered_map = std::unordered_map<Key, T, Hash, Pred, PMAllocator<std::pair<const
          Key, T>>>;
```

```
25
26   static inline void *malloc(std::size_t size) {
27     return PMMalloc::malloc(size);
28   }
29 } // namespace pm
```

List 5.2. Sample of my PM library

```
1 #include "pm.h"
2 void pm_sample() {
3   int *persisted_int = pm::malloc(sizeof(int));
4   pm::vector<int> persisted_vector {1, 2, 3};
5   pm::string persisted_str = "hogefuga";
6   pm::map<pm::string, pm::string> persisted_map;
7   persisted_map["key"] = pm::string("value");
8 }
```

本評価では, leveldb に付属する db_bench をこれらのコンテナクラスを用いた実装に移植することで行った. この db_bench は標準で leveldb, sqlite3, Kyoto Cabinet の 3 つのプラットフォーム向けに実装されており, 今回は広く利用されている leveldb, sqlite3 と pm::unordered_map を比較した.

5.5.3　API の簡潔さ

AMFS では,mmap したアドレス領域に対しての load/store に関して fsync 時にアトミックなデータ永続化を実現しているため, mmap した領域にコンテナオブジェクトを確保するだけで容易にデータベースと同等の機能を実現することができる. そのため, API としては非常に簡潔になる. List 5.3, List 5.4 は, db_bench 中のコンテナに対する読み書きを PM 向けコンテナクラスに対して行っている部分を抜粋したものになる. 一方, SQLite3 を用いた実装は List 5.5, List 5.6 のようになっており, 明らかに SQLite3 のほうが複雑になっている. List 5.7, List 5.8 で示した leveldb では STL に近い操作が可能であるが, batch や option といったやや特殊な API を用いる必要があり, STL が利用可能であるという pm::unordered_map の簡潔さには及ばない.

List 5.3. Code of writing by using pm::unordered_map

```
1  void main_write_procedure() {
2    for (int i = 0; i < num_entries; i += entries_per_batch) {
3      if (transaction)
4        pm::begin();
5      for (int j = 0; j < entries_per_batch; j++) {
6        // Create values for key-value pair
7        const char* value = gen_.Generate(value_size).data();
8        const int k = (order == SEQUENTIAL) ? i + j : (rand_.Next() % num_entries);
9        char key[100];
10       snprintf(key, sizeof(key), "%015d", k);
11       (*db_)[key] = pm::string(value, value_size);
12     }
13     bytes_ += value_size + strlen(key);
14     if (transaction)
15       pm::sync();
16   }
17 }
```

List 5.4. Code of reading by using pm::unordered_map

```
void read_sequential_procedure() {
  for (const auto &kv : *db_) {
    bytes_ += kv.first.length() + kv.second.length();
  }
}

void read_random_procedure() {
  for (int i = 0; i < reads_; i += entries_per_batch) {
    for (int j = 0; j < entries_per_batch; j++) {
      // Create key value
      char key[100];
      int k = (order == SEQUENTIAL) ? i + j : (rand_.Next() % reads_);
      snprintf(key, sizeof(key), "%016d", k);
      volatile auto it = db_->find(key);
    }
  }
}
```

List 5.5. Code of writing by using sqlite3

```
void write_sqlite3_procedure() {
  sqlite3_stmt *replace_stmt, *begin_trans_stmt, *end_trans_stmt;
  std::string replace_str = "REPLACE␣INTO␣test␣(key,␣value)␣VALUES␣(?,␣?)";
  std::string begin_trans_str = "BEGIN␣TRANSACTION;";
  std::string end_trans_str = "END␣TRANSACTION;";

  // Check for synchronous flag in options
  std::string sync_stmt = (write_sync) ? "PRAGMA␣synchronous␣=␣FULL" :
    "PRAGMA␣synchronous␣=␣OFF";
  status = sqlite3_exec(db_, sync_stmt.c_str(), NULL, NULL, &err_msg);
  // Preparing sqlite3 statements
  status = sqlite3_prepare_v2(db_, replace_str.c_str(), -1,
                              &replace_stmt, NULL);
  status = sqlite3_prepare_v2(db_, begin_trans_str.c_str(), -1,
                              &begin_trans_stmt, NULL);
  status = sqlite3_prepare_v2(db_, end_trans_str.c_str(), -1,
                              &end_trans_stmt, NULL);

  for (int i = 0; i < num_entries; i += entries_per_batch) {
    // Begin write transaction
    if (transaction) {
      status = sqlite3_step(begin_trans_stmt);
      status = sqlite3_reset(begin_trans_stmt);
    }

    // Create and execute SQL statements
    for (int j = 0; j < entries_per_batch; j++) {
      const char* value = gen_.Generate(value_size).data();

      // Create values for key-value pair
      const int k = (order == SEQUENTIAL) ? i + j :
        (rand_.Next() % num_entries);
      char key[100];
      snprintf(key, sizeof(key), "%016d", k);
      // Bind KV values into replace_stmt
      status = sqlite3_bind_blob(replace_stmt, 1, key, 16, SQLITE_STATIC);
```

```
37        status = sqlite3_bind_blob(replace_stmt, 2, value,
38                                   value_size, SQLITE_STATIC);
39        // Execute replace_stmt
40        bytes_ += value_size + strlen(key);
41        status = sqlite3_step(replace_stmt);
42        // Reset SQLite statement for another use
43        status = sqlite3_clear_bindings(replace_stmt);
44        status = sqlite3_reset(replace_stmt);
45      }
46
47      // End write transaction
48      if (transaction) {
49        status = sqlite3_step(end_trans_stmt);
50        status = sqlite3_reset(end_trans_stmt);
51      }
52    }
53
54    status = sqlite3_finalize(replace_stmt);
55    status = sqlite3_finalize(begin_trans_stmt);
56    status = sqlite3_finalize(end_trans_stmt);
57  }
```

List 5.6. Code of reading by using sqlite3

```
1   void read_sqlite3_procedure() {
2     sqlite3_stmt *read_stmt, *begin_trans_stmt, *end_trans_stmt;
3     std::string read_str = "SELECT␣*␣FROM␣test␣WHERE␣key␣=␣?";
4     std::string begin_trans_str = "BEGIN␣TRANSACTION;";
5     std::string end_trans_str = "END␣TRANSACTION;";
6     // Preparing sqlite3 statements
7     status = sqlite3_prepare_v2(db_, begin_trans_str.c_str(), -1,
8                                 &begin_trans_stmt, NULL);
9     status = sqlite3_prepare_v2(db_, end_trans_str.c_str(), -1,
10                                &end_trans_stmt, NULL);
11    status = sqlite3_prepare_v2(db_, read_str.c_str(), -1, &read_stmt, NULL);
12
13    for (int i = 0; i < reads_; i += entries_per_batch) {
14      // Begin read transaction
15      if (transaction) {
16        status = sqlite3_step(begin_trans_stmt);
17        status = sqlite3_reset(begin_trans_stmt);
18      }
19
20      // Create and execute SQL statements
21      for (int j = 0; j < entries_per_batch; j++) {
22        // Create key value
23        char key[100];
24        int k = (order == SEQUENTIAL) ? i + j : (rand_.Next() % reads_);
25        snprintf(key, sizeof(key), "%016d", k);
26        // Bind key value into read_stmt
27        status = sqlite3_bind_blob(read_stmt, 1, key, 16, SQLITE_STATIC);
28        // Execute read statement
29        while ((status = sqlite3_step(read_stmt)) == SQLITE_ROW) {}
30        // Reset SQLite statement for another use
31        status = sqlite3_clear_bindings(read_stmt);
32        status = sqlite3_reset(read_stmt);
33      }
```

```
34      // End read transaction
35      if (transaction) {
36        status = sqlite3_step(end_trans_stmt);
37        status = sqlite3_reset(end_trans_stmt);
38      }
39    }
40    status = sqlite3_finalize(read_stmt);
41    status = sqlite3_finalize(begin_trans_stmt);
42    status = sqlite3_finalize(end_trans_stmt);
43  }
```

List 5.7. Code of writing by using leveldb

```
1  void DoWrite(ThreadState* thread, bool seq) {
2    RandomGenerator gen;
3    WriteBatch batch;
4    Status s;
5    int64_t bytes = 0;
6    for (int i = 0; i < num_; i += entries_per_batch_) {
7      batch.Clear();
8      for (int j = 0; j < entries_per_batch_; j++) {
9        const int k = seq ? i+j : (thread->rand.Next() % FLAGS_num);
10       char key[100];
11       snprintf(key, sizeof(key), "%016d", k);
12       batch.Put(key, gen.Generate(value_size_));
13       bytes += value_size_ + strlen(key);
14     }
15     s = db_->Write(write_options_, &batch);
16   }
17 }
```

List 5.8. Code of reading by using leveldb

```
1  void ReadRandom(ThreadState* thread) {
2    ReadOptions options;
3    std::string value;
4    int found = 0;
5    for (int i = 0; i < reads_; i++) {
6      char key[100];
7      const int k = thread->rand.Next() % FLAGS_num;
8      snprintf(key, sizeof(key), "%016d", k);
9      if (db_->Get(options, key, &value).ok()) {
10       found++;
11     }
12   }
13 }
```

5.5.4　実行速度

実行速度の比較は図 5.5 のようになった．横軸はワークロードの種別になっており，それぞれの詳細は表 5.2 にまとめたような内容になる．pm::unordered_map, leveldb, sqlite3 に対してこのような更新を行った．いずれもキーは 16 Bytes, ペイロードは 100 Bytes となり，fillsync, fill100K

表 5.2. Workloads

名前	内容
readseq	キーを辞書順で連続に設定し, Key, Value の長さを取得する.
readrandom	キーをランダムに設定し, Key, Value の長さを取得する.
fillseq	キーを辞書順で連続に設定し, レコードを追加する. すべてのレコードを追加した後に 1 度同期処理を行う.
fillrandom	キーをランダムに設定し, レコードを追加する. すべてのレコードを追加した後に 1 度同期処理を行う.
fillsync	キーを連続に設定し, レコードを追加する. レコードを追加するごとに同期処理を行う.
overwrite	キーを連続に設定し, 既にあるレコードを追加する. すべてのレコードを追加した後に 1 度同期処理を行う.
fill100K	キーを辞書順で連続に設定し, 100KB の Value を追加する. すべてのレコードを追加した後に 1 度同期処理を行う.

は 10,000 レコード, それ以外は 1,000,000 レコードの追加や読み出しの操作を行った平均となっている. また, 図 5.5 の縦軸は対数スケールである.

read, fill, overwrite 共に, fillsync を除くと unordered_map が最も高速になっている. これは, unordered_map が load/store によるメモリ操作だけで高速にデータ構造にアクセスできるのに対し, leveldb や sqlite3 では耐障害性の高いアルゴリズムをサポートするため, 複雑な操作を行っていることによると考えられる. また, 毎回の同期処理を行っていない readseq, readrandom, fillseq, fillrandom といったワークロードにおいては HDD 上の ext4 でも比較的高速にデータにアクセスできていることがわかる. これはページキャッシュによる効果であると考えられる.

一方で, fillsync に関しては unordered_map が leveldb, sqlite3 に対して 10 倍弱遅いという結果になっている. これは, leveldb, sqlite3 は毎回の API 呼び出しの際に, 永続化させなければならない範囲を記憶しているためであると考えられる. 一方の unordered_map は 287[us/op] という結果になっている. この理由は, 5.3 節の fsync の測定における触った領域の大きさが小さい場合の性能と変わらないため, ページテーブルの走査に時間がかかっていると考えられる. ファイル全体が大きい一方で編集箇所が小さい場合には, msync を行うことによって一部のページだけを走査させるといったことにより, 高速に実行することが可能であると考えられる.

また, ページテーブルの走査の時間が隠れる程度に大きな領域を触った場合には fsync が高速に行えていることが, fillseq, fillrandom といったワークロードにおいて ext4 での leveldb よりも unordered_map が高速なことから言える. このことから, leveldb などではレコードの追加ごとに API を呼び出してログの構築といった複雑な処理を行わないといけないのに対し, AMFS では OS レベルのアトミック永続化のサポートにより dirty ビットを見るだけで良いことが有利であると言えよう.

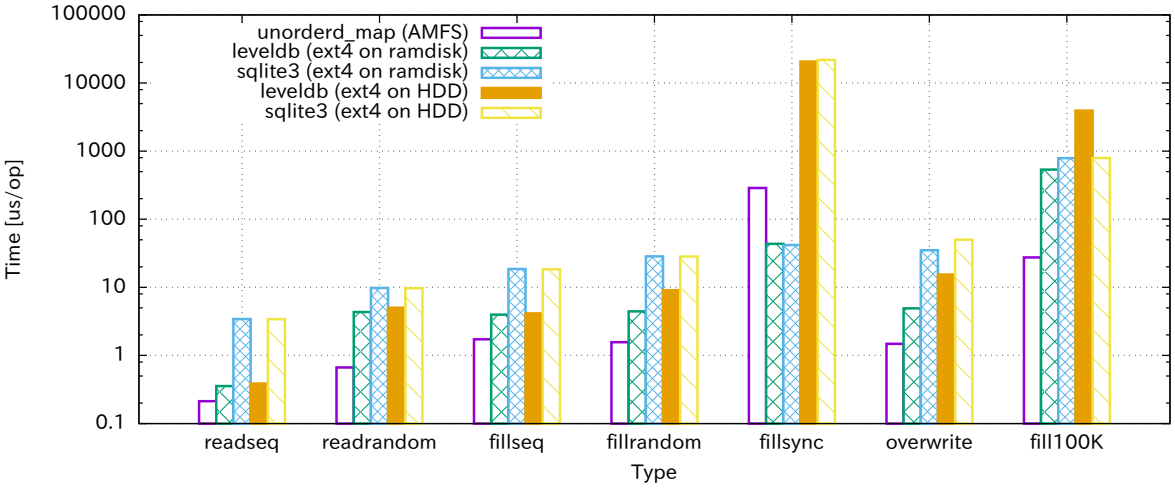

図 5.5. Comparison between pm::unordered_map on AMFS, leveldb 1.18 and sqlite3.8.2

5.5.5　まとめ

この評価により, 直接 STL を永続化して利用できることによるパフォーマンス上のメリットがあることが確認できた. 複数のプロセスからの isolation が必要ないのであれば, fbegin-fsync のセマンティクスが十分な役割を果たすと言える.

第6章　結論

6.1　まとめ

本論文では, 格納してあるデータに直接アクセスができ, かつアトミックなデータ永続化 API をそなえた PM 向けのファイルシステム, AMFS を提案し, 実装及び評価を行った. AMFS では OS レベルのページキャッシュをうまく活用することにより実装されており, その結果, AMFS 上のヒープファイルに STL のオブジェクトで KVS を実現したものが軽量な KVS のライブラリである leveldb より 1.6 倍高速に動作することが確認できた. また, AMFS において実装された fbegin という API を利用した場合, ユーザープログラムが簡潔にアトミックなデータ永続化を利用することができることも示すことができた. アトミックなデータ永続化が不要であるシチュエーションというのは考えにくいため, ここで提案したファイルシステムのセマンティクスは汎用性が高いものであると考えられる.

6.2　今後の展望

AMFS の実装, 評価するにあたって必要になったり, できると今後の PM 向けデータ構造の開発がより容易になりそうな分野について簡単に触れておく.

6.2.1　named malloc の導入と anonymous malloc を含めた利用

List 6.1. Example of named malloc

```
1  struct tree_node {
2    tree_node *left, *right;
3  };
4
5  void named_malloc_sample() {
6    tree_node *root = pm::find_object("sampleapp/test_root");
7    if (!root)
8      root = pm::named_malloc("sampleapp/test_root", 0644, sizeof(tree_node));
9    if (!root->left)
10     root->left = pm::malloc(sizeof(tree_node));
11   if (!root->right)
12     root->right = pm::malloc(sizeof(tree_node));
13 }
```

PM では, KVS のワークロードでも行ったように, メモリのアロケーションによって永続化領域を確保することが可能であると非常に便利である. このようなオブジェクトストアが実現された場合, 現在の malloc の API ではすでに永続化されたオブジェクトを取ってくることができない.

これはなぜかというと, 名前がついていないため, どのオブジェクトを指定しているのかわからないからである. そこで, 名前を指定して領域を確保するような API を `named_malloc` と名付け, サンプルプログラムとしたものが List 6.1 である.

この例では, まずはじめに 2 分木の根を `find_object` という API を用いて探す. これによって, 既に以前永続化された”sampleapp/test_root” オブジェクトがあればその領域を取得してくる. もし見つからなければ `named_malloc` を呼び出し, オブジェクトを生成する. ここで重要なのは, パーミッションもオブジェクトに設定しなければならない点である. これに関しては次に説明する.

また, `root->left`, `root->right` に関しては `named_malloc` ではなく通常の `malloc` を利用しているのも特徴である. すべてのオブジェクトが名前を持つ必要はなく, 名前が必要なのはコンテナだけであることがこのような例からわかる. 今回の場合はファイルではなくオブジェクト単位なので, データの管理が非常に細粒度になる. そのため, 現在のファイルシステムでも行っている名前の管理やパーミッションの管理に関しても, 数が多くなることによってコストがかさむと考えられる. このように `named_malloc` と通常の malloc に相当する `anonymous_malloc` を分離することで, この問題が緩和されると考えられる.

6.2.2　パーミッションの管理

先述した `named_malloc` に関連し, パーミッションも重要となる. 細粒度にデータが分割されるため, 現在の Linux のようなユーザー, グループ, その他というたった 3 段階のパーミッション分けでは不十分になる可能性がある. 少なくとも, ユーザー単位で指定したり, グループを簡単に作成したり, 自動でプロセスの実行バイナリやユーザー, 環境などと対応づけるなどしてプライベート空間とそれ以外の共有されるかもしれない領域を分けるなどの工夫が必要となる可能性がある. また, 自由度を高めるだけではなく, 煩わしさにも配慮が必要になる. 複雑なプロトコルは好まれないだろう. このようなパーミッションに関する議論は実際のアプリケーションを書いてその有効範囲を検証するのがよいと考えられるので, `named_malloc` の話と一緒に議論すると良いだろうと考えている. また, このパーミッションの話は, single address space の業界でも同じ議論がなされているようである. これについては次で説明する.

6.2.3　複数プロセスによるポインタの管理

PM 領域の仮想アドレスを動的に割り振ってしまうと, 永続化したオブジェクト内で保持されるポインタが無効になってしまう. この問題を回避するためにすべてのポインタをオフセットに変更するのは, オフセットの計算のコストやすでに書かれたライブラリの再利用性といった点で不利である. そこで, 現在の Linux とは異なる, Opal[2] のような Single address space operating system が参考になりそうである. Linux のカーネル空間のように, PM の領域は仮想アドレスを固定とし, single address space OS の実装の際に用いられているアドレス空間の確保と共有の技術を利用することで PM 内のポインタは常に有効にすることができる.

プロセス間でのアドレス空間の共有の情報をどのように管理し, 永続化するかという点を考えることで, 現在のユーザー, グループ, その他というパーミッションより柔軟なパーミッションの与え方も考えられるのではないだろうか.

6.2.4　アトミック操作のファイルシステムによるサポートが及ぼす影響範囲の調査

ファイルシステムがアトミックなデータ永続化をサポートすることによって, 複雑なシステムソフトウェアからはじまり簡単なトイアプリまで影響が及ぼされると考えられる. そこで, このデータ永続化のセマンティクスの仕様によって, どの程度のアプリケーションが高信頼に記述可能かという定量的な評価が行えると好ましい.

R. Verma らは, ALICE と BOB というツールを作成し, 各ファイルシステム上でアプリケーションが正しく動作するかどうかを自動で判定できるようにした [33]. このツールを改変し, fbegin に対応させるということを当研究室の伊藤が現在行っている [1].

各種アプリケーションを fbegin に対応させて, この fbegin への対応済みの ALICE を用いてチェックすることで, 安全にプログラムを記述することができているかどうか判断することができるようになる. そこで, 逆にファイルシステムの仕様を変数とし, 複数ファイルのアトミックな永続化が必要かどうかや, fsync がサポートすべきアトミックな永続化の最大サイズ, 複数スレッドやプロセスへの対応の必要性といった点を評価することで, より現実のアプリケーションに即したファイルシステムの仕様を策定することができると考えられる.

6.2.5　PM におけるフラグメンテーションの回避の重要性

PM ではデータが永続化されてしまう. そのため, malloc のようなアロケータを作成した場合, フリーリストがリセットされることなく常にフラグメンテーションが進行するになる. そこで, PM 向けのメモリアロケータには GC のコンパクションのような機能, ファイルシステムでいうところのデフラグが必要となると考えられる. 特に, 永続化したい領域のサイズは数バイトのオブジェクト 1 つという単位から 1GB を超えるような巨大なサイズまでありえる. それだけに, オブジェクト単位での永続化が可能になった場合にはアロケータの実装が重要である.

そこで, まず PM 向けにアロケータを実装し, その上でどのような実アプリケーションではどのようなアロケータが必要になるかを評価していく必要があると考える. このとき, データサイズや個数の特性が変わると予想されるので, 設計したいのは永続化したいデータのアロケータであって一時領域のものではないことに注意する必要があると思われる.

6.2.6　デバイスレベルでのアトミック操作のサポート

AMFS では, ハードウェアによって 8 バイトのアトミックな更新を保障するだけで任意のサイズのデータ永続化をアトミックにすることが可能になる. 一方で, NVMe 1.2 のようにハードウェアがある一定サイズ以下の書き込みはアトミックにすることができる [10] ものもあるため, このような API を用いればいくつかのツリーの更新を CoW を用いずにアトミックに書き出すように処理を変更することも可能であると考えられる. たとえば 64KB をまとめて書き出すことができるのであれば 8 ページまで同時に書き出すことができるので, 64KB までのユーザーデータの更新であればメタツリーを更新せず, 直接データを更新してしまうことが考えられる. このように, ある程度のサイズまでのアトミック性をハードウェアによって保障することで高速化の余地がある. しかし, このようなハードウェアにおいても任意のサイズのアトミック性が担保されることは考

[1] 今年度の卒業論文となる予定である.

えにくく, CoW を用いてディスクの許す限り任意のサイズのアトミックな更新が可能にする必要はあると言える.

6.2.7　AMFS の Linux4.2 以降への移植と実装の拡充

AMFS は現在 Linux 3.11 上に実装されている. これは PMFS がもともと Linux 3.11 上に実装されていたことによる. しかし, Linux 4.2 ではカーネルに PMEM という PM を抽象化したブロックデバイスのインターフェースが定義されるなど, mainline にすでに一部 PMFS の機能が実装され始めている. そこで, AMFS もその開発に追いつき, このインターフェース上に再実装される必要がある.

実際の実装としてはおそらく PMEM を利用することになると予想されるが, 問題となるのはページキャッシュの扱いである. NVDIMM に関する最新の動向 [27] を見ると, ページキャッシュが一切排されているように見える. 実際の実装を見たわけではないが, このあたりが実装の面倒な点ではあると思われる.

また, huge page に対応することも必要だと考えられる. 現在は mmap した領域はすべてページサイズを 4K としているが, TLB ミスを考えるとこれは得策ではない. PMFS では 2M や 1G を境界として動的に huge page を選択するようになっていたので, AMFS においても同様の実装を行うのが都合がよいと考えられる.

謝辞

まず, 田浦先生には 3 年間, 学部の実験や OS の授業から合わせたら 4 年間, 大変お世話になりました. なかなかテーマが決まらず悩んでいた M1 の冬に, 先生と相談してテーマがきちんと決まったときの盛り上がりは忘れられないです. また, 修士の生活で PILE やらインターンやら自由に生活させていただいたのも感謝しています. ご迷惑をお掛けしていたかもしれませんが, 僕としては, いろいろな経験をすることができたと感じております. 最後までたくさんのご指導をいただき, 大変感謝しております. ありがとうございます.

また, 研究室の M2 の同期は, 研究室生活を一緒にとても楽しく過ごさせてもらいました. なごさんは隣の席から人知れずお菓子を餌付けしてくれ, チャーリーくんはイカの面白さを共有してもらいました.

M1 の後輩には, 技術的や研究的な面でとてもお世話になりました. めりんぎさん, 研究室のサーバー・ネットワーク管理をサポートして頂いてたすかりました. 今後も頑張って下さい. 岩崎くんはガンガン進捗を生む姿勢にとても感動していました. 見習って頑張りたいです.

他にもたくさんの人にお世話になりました. この場を借りてお礼申し上げます. ありがとうございました。

発表文献

国内発表（査読なし）

[1] Makoto Shimazu，Kenjiro Taura．ASDB: Direct Search Database System on Any Data Source. *SWoPP2014*, Niigata, 2014/8.

[2] Makoto Shimazu，Kenjiro Taura．AMFS: A File System for Emerging Persistent Memory Supporting Atomic Data Durability. *SWoPP2015*, Oita, 2015/8.

参考文献

[1] Daniel P. Bovet and Marco Cesati. 詳解 Linux カーネル 第 3 版. O'REILLY, 2007.

[2] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, Vol. 12, No. 4, pp. 271–307, November 1994.

[3] Leon O Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 1971.

[4] J Coburn and et al. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ASPLOS '11*, 2011.

[5] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Doug Burger, Benjamin Lee, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Symposium on Operating Systems Principles (SOSP '09)*, 2009.

[6] Berkley DB. http://www.oracle.com/jp/products/database/berk eley-db/overview/index.html.

[7] Dormando. memcached, 2015.

[8] Doug Lea. A memory allocator, 1996.

[9] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys 14)*, 2014.

[10] NVM Express. *NVM Express 1.2a Specification*. NVM Express, 2015.

[11] R Fackenthal and et al. 19.7 A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. *ISSCC'14*, 2014.

[12] Ilya Grigorik. Sstable and log structured storage: Leveldb, 2012.

[13] Condit J. and et al. Better I/O through byte-addressable, persistent memory. *SOSP '09*, p. 133, 2009.

[14] Cooke Jim. The inconvenient truths about nand flash memory. Micron MEMCON ' 07 presentation, 2007.

[15] D-H. Kang, J.-H. Lee, J.H. Kong, D. Ha, J. Yu, C.Y. Um, J.H. Park, F. Yeung, J.H. Kim, W.I. Park, Y.J. Jeon, M.K. Lee, J.H. Park, Y.J. Song, J.H. Oh, H.S. Jeong, and H.S. Jeong. Two-bit cell operation in diode-switch phase change memory cells with 90nm technology. In *VLSI Technology, 2008 Symposium on*, 2008.

[16] Michael Kerrisk. Linux プログラミングインタフェース. O'REILLY, 2012.

[17] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. *SIGARCH '09*, 2009.

[18] Leveldb Core Team. Leveldb, 2016.

[19] Memcached. http://memcached.org.

[20] C. Mohan. Repeating history beyond aries. VLDB '99, 1999.

[21] OpenNVM. http://opennvm.github.io/.

[22] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[23] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 14–23, New York, NY, USA, 2009. ACM.

[24] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. Osborne/McGraw-Hill, 2000.

[25] redislabs. Redis, 2016.

[26] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, Vol. 9, No. 3, p. 9, 2013.

[27] Ross Zwisler. Persistent memory, 2015.

[28] Samsung. High-performance, low-latency and cost-optimized data center ssd, 2015.

[29] Clinton W Smullen, Vidyabhushan Mohan, Anurag Nigam, Sudhanva Gurumurthi, and Mircea R Stan. Relaxing non-volatility for fast and energy-efficient stt-ram caches. In *HPCA '11*, 2011.

[30] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*, 2009.

[31] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *nature*, 2008.

[32] R Takemura and et al. 2mb spram design: Bi-directional current write and parallelizing-direction current read schemes based on spin-transfer torque switching. *ICICDT'07*, 2007.

[33] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-atomic updates of application data in a linux file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[34] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 14:1–14:14, New York, NY, USA, 2014. ACM.

[35] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, 2011.

[36] WesternDigital. Wd xe datacenter hard drives, 2015.

[37] Wolfram Gloger. Wolfram gloger's malloc homepage, 2006.

[38] Yole Développement. Emerging non volatile memory (nvm) technological choices are about to be made ... sttmram or mram?, 2015.

[39] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, 2009.