Vol.2015-ARC-215 No.11
Vol.2015-OS-133 No.11
2015/5/27

メモリストレージ容量拡張手法の Linux 4.0における実現について

追川 修一[1]

概要：近年，メモリインタフェースを提供するストレージとして注目されている次世代不揮発性メモリやNV-DIMM 等のメモリストレージは，高性能ではあるが，容量に制限があり，主ストレージとして使用することは困難である．従って，従来のブロックストレージと組み合わせ，透過的に十分な容量を提供することが重要になる．本論文では，メモリストレージとブロックストレージとの組み合わせを，デバイスドライバレベルで行うのではなく，ファイルシステムのレイヤで行う手法を提案する．そして，提案手法の Linux 4.0 カーネルにおける実現について述べる．

1. はじめに

メモリインタフェースを提供するストレージとして，現在，次世代不揮発性メモリ（NV メモリ）や NV-DIMM 等の開発が進んでおり，これらメモリストレージの実用化が近いと考えられる．NV メモリとしては，MRAM, PCM (Phase Change Memory), ReRAM 等が開発されている．PCM, ReRAM は書き込み時の性能や耐久性に制限があるが，MRAM にはそのような制限がない．また，MRAM は DRAM に相当する性能を持つ点で優れている．NV-DIMM には，DRAM の DIMM にキャパシタ等を搭載することで，不揮発性を提供しているものがある．

MRAM や NV-DIMM といったメモリストレージは，DRAM に相当する性能を提供するが，容量に制限がある．従って，これらのメモリストレージのみが構成するストレージは，現在広く用いられているブロックストレージよりも遙かに高性能となるものの，非常に高価となり，その用途は限られたものになると考えられる．しかしながら，メモリストレージは，そのメモリインタフェースによる直接アクセスが可能であり，またその性能から同期アクセスが可能であることから，処理コストの大きいブロックデバイスドライバが不要または大幅に簡素化でき，アクセスコストを低減できるという長所がある．

メモリストレージ容量拡張手法 VEMS (Virtually Extended Memory Storage) [1] は，メモリストレージとブロックストレージを組み合わせ，メモリストレージの容量を仮想的に拡張し，ブロックストレージの容量を提供する手法である．VEMS は，メモリストレージとしてのアクセスを提供する．そのため，メモリストレージの特徴である，直接同期アクセスによるアクセスコストの低減が可能である．一方，ブロックストレージと組み合わせることで，ブロックストレージと同じ容量を提供する．透過的に大容量を提供可能にすることで，メモリストレージをストレージの主体デバイスの一部として利用可能にする．VEMS は Linux カーネルのデバイスドライバとして実装した．ブロックデバイスドライバとして実現したことで，ファイルシステムとは分離されており，そのための制約を受けることとなった．その制約を回避するため，VEMS は，XIP (eXecution-In-Place) インタフェースの拡張を行ったが，そのインタフェースは煩雑なものとなった．

そこで本論文は，メモリストレージ容量拡張を，ファイルシステムレイヤで実現する方法を提案する．そもそもメモリストレージをアクセスするために、デバイスドライバは必要ではない。従って、PRAMFS [2], PMFS [3] 等、ファイルシステムレイヤでメモリストレージにアクセスするファイルシステムも実現可能である。ファイルシステムレイヤで、個別のファイルシステムとは独立に、メモリストレージにアクセスし、メモリストレージ容量拡張を実現する。これにより，ファイルシステムとブロックデバイスドライバの分離に起因する問題点を解決し、またメモリストレージを有効かつ柔軟に管理することを可能にする．

以下，2 章で，背景となる技術として VEMS について述べる．3 章はファイルシステムレイヤにおけるメモリストレージ容量拡張手法について述べ，4 章は Linux 4.0 における実装について述べる．5 章で関連研究について述べ，6 章で本論文をまとめる．

[1] 筑波大学　システム情報系情報工学域
University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan

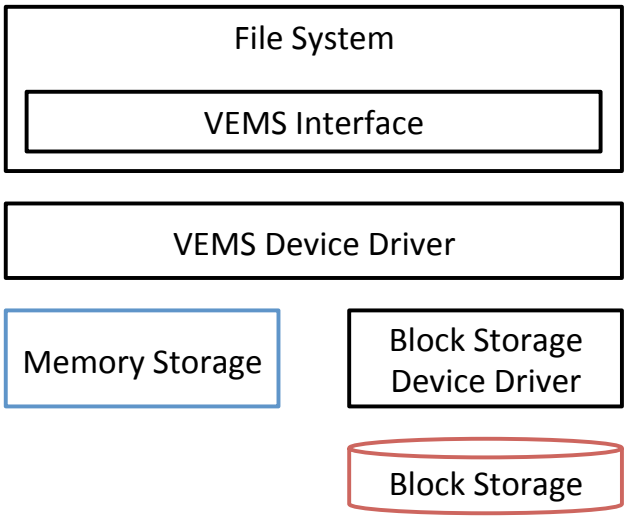

図 1 VEMS の実装形態

2. 背景

背景となる技術として，メモリストレージ容量拡張手法 VEMS (Virtually Extended Memory Storage) [1] について述べる．VEMS は，メモリストレージとブロックストレージを組み合わせ，メモリストレージの容量を仮想的に拡張する手法である．VEMS は，メモリストレージとしてのアクセスを提供する一方，ブロックストレージと組み合わせることで，ブロックストレージと同じ容量を提供する．そのため，メモリストレージの特徴である直接同期アクセスによりアクセスコストを低減可能であり，また透過的にブロックストレージの大容量が利用可能である．

VEMS は Linux カーネルのデバイスドライバとして実装されている．ファイルシステムからのアクセスを効率化するため，VEMS は，XIP インタフェースを拡張したファイルシステムインタフェースを必要とする．現状では，VEMS ファイルシステムインタフェースは，Ext2 に実装されている．ブロックストレージへのアクセスには，そのためのブロックデバイスドライバが必要になる．ブロックデバイスドライバは，VEMS デバイスドライバが必要に応じて呼び出す．従って，アクセス要求の処理がメモリストレージへのアクセスで済む場合は，ブロックデバイスドライバは呼び出されない．図 1 に，VEMS の実装形態を示す．

VEMS の特徴を以下にまとめる．

- 同期・非同期アクセスの適応的切替
- ページキャッシュが不要
- メモリストレージへの書き込みによるデータ永続化

以下，それぞれについて述べる．

同期・非同期アクセスの適応的切替は，メモリストレージは同期アクセス，ブロックストレージは非同期アクセスであることから，アクセス先に応じて，同期・非同期を切り替えることで実現する．読み出しの場合，データがメモリストレージ上にあれば，同期アクセスとなる．書き込みの場合，メモリストレージ上に書き込み可能領域があれば，同期アクセスとなる．それ以外の場合は，ブロックストレージへのアクセスが必要となるため，非同期アクセスとなる．このように必要に応じて同期・非同期を切り替えることで，高性能を実現する．

VEMS は XIP インタフェースを拡張したファイルシステムインタフェースを提供し，メモリストレージへの直接アクセスを行っている．直接アクセスできないブロックストレージは，ページキャッシュを経由することでアクセス処理を効率化するが，直接アクセス可能なメモリストレージはページキャッシュの機能をはたすため，メモリストレージとは別にページキャッシュを用いる必要はない．そのため，VEMS へのアクセス時には，ページキャッシュは用いられない．従って，メモリストレージから読み出す場合は，ページキャッシュを経ずに，データはユーザプロセスへ直接コピーされる．また，書き込みの場合は，ユーザプロセスからメモリストレージへ，直接コピーされる．

メモリストレージは，メモリアクセスインタフェースを提供するが，そのメモリは不揮発性である．従って，メモリストレージへ書き込んだデータは，その時点で永続化が保証される．ページキャッシュは，揮発性のメインメモリから割り当てられるため，ページキャッシュへ書き込んだ時点では，データは永続化されていない．データの永続化は，ページキャッシュからブロックストレージへの書き込み終了を待つ必要がある．VEMS は，メモリストレージへ書き込み時点で，データの永続化を保証する．さらに，ページキャッシュを用いないため，アプリケーションが書き込みを終了した時点で，データが永続化されることになる．

3. ファイルシステムレイヤにおけるメモリストレージ容量拡張手法

本章では，まず VEMS の問題点を述べ，ファイルシステムレイヤにおけるメモリストレージ容量拡張手法を提案する．

3.1 VEMS の問題点

VEMS は，メモリストレージとブロックストレージの組み合わせを，ブロックデバイスドライバとして実現した．それは，メモリストレージの管理をデバイスドライバレベルに隠蔽し，メモリストレージへの直接アクセス機能を提供する XIP の枠組みを拡張するかたちで，VEMS を実現することが，最も単純な実現方法であると考えたからである．XIP インタフェースは，特定ファイルシステムから，アクセス対象のデータの場所に対応するブロックデバイスのブロック番号を取得，そのブロック番号をデバイスドライバへ渡し，そのブロック番号のデータがあるメモリアド

レスを取得する．

VEMS は，メモリストレージの効率的な使用を実現するため，XIP インタフェースを拡張，引数を増やし，ファイルシステムレイヤから VEMS デバイスドライバへ，より多くの情報を渡せるようにする必要があった．VEMS が追加で必要とする情報は，汎用ファイルシステムレイヤにあり，アクセスが読み書きのいずれかであるか，ページフレームサイズにアラインしたアクセスであるか，メモリストレージに領域を確保する必要があるか，である．XIP は，汎用ファイルシステムレイヤが特定ファイルシステムを呼び出し，特定ファイルシステムがブロックデバイスドライバを呼び出すパスとなっている．VEMS もそのパスを踏襲しているため，VEMS が追加で必要とする情報を，汎用ファイルシステムレイヤから，特定ファイルシステムを経由し，VEMS デバイスドライバへ渡している．そのような引数の受け渡しは，ファイルシステムとブロックデバイスドライバの分離に起因していると考えられる．

ファイルシステムとブロックデバイスドライバの分離という観点では，ブロックデバイスドライバが取得する情報は，基本的にはブロックに関連する情報に限られるという問題がある．いかなる情報も引数として渡すことは可能ではあるが，例えば，ファイルシステムレイヤが扱うファイル情報をブロックデバイスドライバに渡すことは，ブロックデバイスドライバとしてのモジュール化に反するものとなる．また，ブロックデバイスドライバがファイル情報を取得しても，ブロックデバイスドライバがファイルシステムレイヤをアップコールすることは，上に示した階層化された呼び出しパスをさらに煩雑なものとしてしまう．

3.2 提案手法

前節で述べた，ファイルシステムとブロックデバイスドライバの分離に起因する問題点を解決する 1 つの方法として，ファイルシステムレイヤにおいてメモリストレージを管理し，その容量を拡張する手法を提案する．

提案手法を，図 2 に示す．提案手法は，図 1 では VEMS デバイスドライバの管理下にあったメモリストレージを，ファイルシステムの直接管理下に置く．そして，汎用ファイルシステムレイヤに，メモリストレージを直接管理するためのインタフェースを導入する．これにより，汎用ファイルシステムレイヤは，メモリストレージをアクセスするにあたり，ブロックデバイスドライバを経由する必要がなくなり，ファイル情報をメモリストレージ管理に用いることが可能になる．

メモリストレージはメモリアクセスインタフェースを提供するため，そのアクセスにデバイスドライバを必要としない．そのため，ファイルシステムレイヤがメモリストレージに直接アクセスできる．従って，ファイルシステムレイヤにおいてメモリストレージを管理する提案手法が実現可能となる．

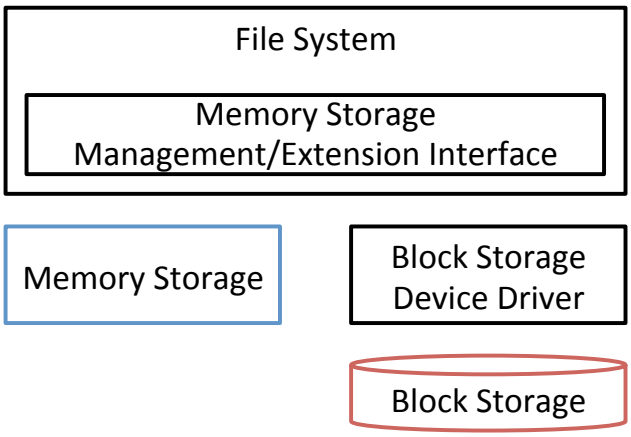

図 2 ファイルシステムレイヤにおけるメモリストレージ容量拡張手法の概観

提案手法のもっとも単純な実現形態は，ファイルシステムのメモリストレージ管理機構に，VEMS のメモリストレージおよびブロックストレージ管理機構を導入する方法である．この方法では，基本的にはブロック情報を用い，メモリストレージ上のデータを管理することになる．即ち，まず特定ファイルシステムを呼び出し，ファイル情報をブロック情報に変換する．そのブロック情報を用い，ストレージ上のデータにアクセスする．この形態では，アクセス対象のファイルの位置を，対応するブロック番号に変換するために，特定ファイルシステムを呼び出すことになる．そして，メモリストレージまたはブロックストレージ上のデータへのアクセスは，汎用ファイルシステムレイヤに導入された，それぞれにアクセスする機能によって実現される．この方法は，ストレージへのアクセスに際し，基本的にはブロック情報を用いるという点では，VEMS と大きな違いはない．しかしながら，汎用ファイルシステムレイヤがメモリストレージを直接管理することで，ファイル情報を管理に用いることが可能になる点が異なる．

汎用ファイルシステムレイヤがメモリストレージを直接管理する提案手法は，より柔軟なメモリストレージ管理も可能である．例えば，メモリストレージ上のファイルデータへのアクセスを実現するために，必ずしも，そのファイルのブロック情報は必須ではない．ファイルの識別子とアクセスする位置のデータのみに基づき，ファイルデータにアクセスすることも可能である．これは，オブジェクトストレージデバイス (OSD) の概念を，メモリストレージの管理に適用したものと言える．この場合，メモリストレージ上のデータをブロックストレージに書き戻す際には，そのデータのファイル情報からブロック情報へ変換する必要がある．しかしながら，メモリストレージ上のデータにアクセスする際には，ブロック情報は必要とされない．このような管理形態の実現は，今後の課題である．

4. Linux 4.0 における実装

メモリストレージ容量拡張機能の Linux 4.0 における実装は，メモリストレージへのアクセス機能を提供する DAX をベースとする．そのため，本章では，まず DAX について述べた後に，Linux 4.0 におけるメモリストレージ容量拡張機能の実装について述べる．

4.1 DAX

本節は，DAX について述べる．DAX は，XIP を置き換えるかたちで Linux 4.0 に導入された，メモリストレージに対応するための機能である．XIP と DAX の実装の両方について，Ext2 ファイルシステムの場合における，ファイル読み出し時の関数呼び出しパス，ページフォルト時の関数呼び出しパスの概要を，図 3, 4 に示す．それぞれ，例えば vfs_read に対し__vfs_read 等の同一関数名の先頭に__を付け見通しを良くしているような場合など，細かい点は省略している．以下の説明についても，それは同様である．

XIP のファイル読み出し処理では，vfs_read は，ファイルへの操作を抽象化し，ファイルが属するファイルシステムを隠蔽する file_operations 構造体の read メンバを呼び出す．実際には，その値に代入されている xip_file_read を呼び出される．xip_file_read は do_xip_mapping_read を呼び出し，これが XIP の読み出し処理の本体となっている．do_xip_mapping_read は，ファイルと仮想記憶管理のインタフェースを定義する address_space_operations 構造体の get_xip_mem メンバを呼び出し，ファイルとオフセットから対応するメモリストレージのアドレスを取得する．get_xip_mem はファイルシステムごとに定義され，Ext2 ファイルシステムでは，ext2_get_xip_mem が呼び出される．ext2_get_xip_mem は，ext2_get_block を呼び出し，ファイル情報をブロック情報に変換した後に，ブロックデバイスの direct_access を呼び出し，ブロック情報をメモリストレージのアドレスに変換する．do_xip_mapping_read は，__copy_to_user を用い，メモリストレージのアドレスからユーザプロセスのバッファにデータをコピーし，データの読み出しを完了する．

DAX のファイル読み出し処理では，vfs_read は file_operations 構造体の read メンバを呼び出すが，その値に代入されているのは new_sync_read である．new_sync_read は，file_operations 構造体の read_iter メンバを呼び出し，それは address_space_operations 構造体の direct_IO メンバを呼び出す．direct_IO には，Ext2 ファイルシステムでは，ext2_direct_IO が代入されている．DAX が有効なファイルに対し，ext2_direct_IO が行う処理は，ext2_get_block を引数として dax_do_io を，呼び出すのみである．dax_do_io が，DAX における読み書き処理の本体である．ここで，引数として受け取った ext2_get_block を呼び出し，ファイル情報をブロック情報に変換した後に，ブロックデバイスの direct_access を呼び出し，ブロック情報をメモリストレージのアドレスに変換，最後に__copy_to_user を用い，メモリストレージのアドレスからユーザプロセスのバッファにデータをコピーし，データの読み出しを完了する．

XIP および DAX における書き込み処理の呼び出しパスは，read が write になる等の関数名の違いを除けば，読み出し処理とほぼ同様である．DAX の場合，書き込みの場合も，dax_do_io が処理の本体となる．

ページフォルト処理では，handle_pte_fault が，仮想アドレス空間への操作インタフェースを定義する vm_operations_struct 構造体の fault メンバを呼び出すまでは，一般のページフォルト処理と変わらない．XIP の場合，fault メンバには xip_file_fault が代入されている．xip_file_fault は，get_xip_mem メンバを呼び出し，ファイルとオフセットから対応するメモリストレージのアドレスを取得した後に，vm_insert_mixed を呼び出し，そのアドレスのページフレームを仮想アドレス空間にマップする．DAX の場合，dax_fault が xip_file_fault に対応する処理を行う．Ext2 ファイルシステムを呼び出し，ファイル情報をブロック情報に変換するために，ext2_get_block を呼び出すが，これは vm_operations_struct 構造体の fault メンバに代入されている ext2_dax_fault から引数として渡される．

XIP と DAX で行っている処理は，基本的に同一である．読み出し時には，まず Ext2 ファイルシステムを呼び出し，ファイルとオフセットのファイル情報をブロック情報に変換する．次に，ブロックデバイスドライバを呼び出し，ブロック情報からメモリストレージのアドレスに変換する．最後に，そのアドレスから，データをユーザプロセスのバッファにコピーする．ページフォルト処理時も，同様に，まずファイル情報をメモリストレージのアドレスに変換した後に，そのアドレスのページフレームを仮想アドレス空間にマップする．

XIP と DAX のもっとも大きな違いは，XIP では，file_operations 構造体，address_space_operations 構造体のメンバに，XIP のための処理を行う専用の関数が代入されているのに対し，DAX では，汎用の関数となっている点である．そのため XIP では，XIP の実装にほぼ閉じており，ページキャッシュを用いる従来型の実装とは，呼び出しパスが大きく異なっている．また，呼び出しの階層も，DAX よりも低く済んでいる．これは，特に読み出し処理で顕著である．一方 DAX では，読み出し処理では dax_do_io が呼ばれるまでは，基本的に汎用の関数を経由する．また，DAX では，address_space_operations 構造体に get_xip_mem メンバがなくなり，DAX の処理を行うためだけに用いられるメンバは含まれない．get_xip_mem は，特定ファイルシステムを呼び出すために用いられていたた

Call hierarchy of XIP

```
vfs_read
|
+- f_op->read = xip_file_read
  |
  +- do_xip_mapping_read
    |
    +- a_ops->get_xip_mem = ext2_get_xip_mem
    | |
    | +- ext2_get_block
    | |
    | +- inode->i_sb->s_bdev->bd_disk->fops->direct_access
    |
    +- __copy_to_user
```

Call hierarchy of DAX

```
vfs_read
|
+- f_op->read = new_sync_read
  |
  +- filp->f_op->read_iter = generic_file_read_iter
    |
    +- mapping->a_ops->direct_IO = ext2_direct_IO
      |
      +- dax_do_io
        |
        +- ext2_get_block
        |
        +- bdev_direct_access
        | |
        | +- bdev->bd_disk->fops->direct_access
        |
        +- copy_to_iter
          |
          +- __copy_to_user
```

図 3 ファイル読み出し時の関数呼び出しパスの比較

Call hierarchy of XIP

```
handle_pte_fault
|
+- vm_ops->fault = xip_file_fault
  |
  +- xip_file_fault
    |
    +- a_ops->get_xip_mem = ext2_get_xip_mem
    | |
    | +- ext2_get_block
    | |
    | +- inode->i_sb->s_bdev->bd_disk->fops->direct_access
    |
    +- vm_insert_mixed
```

Call hierarchy of DAX

```
handle_pte_fault
|
+- vm_ops->fault = ext2_dax_fault
  |
  +- dax_fault
    |
    +- ext2_get_block
    |
    +- dax_insert_mapping
      |
      +- bdev_direct_access
      | |
      | +- bdev->bd_disk->fops->direct_access
      |
      +- vm_insert_mixed
```

図 4 ページフォルト時の関数呼び出しパスの比較

め，代わりに関数ポインタを引数として渡す方法が取られている．そのため，dax_do_io, dax_fault の呼び出しもとなる関数は，特定ファイルシステムごとの実装となっている．

DAX は，汎用の関数で DAX 固有の処理を行う必要がある．そのため，IS_DAX(inode) マクロにより，DAX の対象ファイルかどうかを調べ，対象ファイルならば DAX 固有の処理を行うようになっている．DAX が有効なファイルシステム上のファイルを open した場合，そのファイルの inode 構造体の i_flags メンバに S_DAX フラグをセットする．このマクロは，このフラグがセットされているかどうかを調べる．

4.2 メモリストレージ容量拡張機能の実装

メモリストレージ容量拡張機能は，拡張された容量のメモリストレージのインタフェースを提供することを目的としているため，その Linux 4.0 における実装は，メモリストレージへのアクセス機能を提供する DAX をベースとし，容量を拡張する機能を追加したものとなる．また，本論文の提案手法は，ファイルシステムレイヤがメモリストレージを管理する．即ち，メモリストレージを管理するブロックデバイスドライバを用いず，ファイルシステムレイヤがメモリストレージに直接アクセスする．

ファイルシステムレイヤに，メモリストレージ容量拡張機能のためのメモリストレージを管理する機構を導入するには，大きく分けて以下の 2 つの方法があると考えられる．

- DAX 相当の機構を別途追加
- DAX を必要に応じて拡張

以下，それぞれについて述べる．

DAX 相当の機構を別途追加する方法は，提案手法を使用するためには，DAX とは相互排他的なマウントオプションを指定する．dax_do_io, dax_fault に相当する関数を別途実装し，メモリストレージを管理し，必要に応じてブロックストレージとデータのやり取りを行うことで，メモリストレージの容量拡張を実現する．また，DAX と同様に，汎用の関数で提案手法固有の処理を行う必要があるため，IS_DAX(inode) に相当するマクロを定義し，必要となる処理を別途記述可能にする．この方法では，DAX とは別の実装となるため，実装の自由度は高まる．一方で，DAX と提案手法でほぼ同様の処理をする際に，処理自体は同じであっても，設定するフラグ等の違いのみで，DAX と提案手法で別途のコードが並ぶこととなり，冗長な実装となる可能性がある．

DAX を必要に応じて拡張する方法は，提案手法を使用するためには，DAX を有効にするマウントオプションに加えて，提案手法を有効にするマウントオプションを指定する．従って，基本的には DAX が定義する関数を用い，必要な箇所で提案手法固有の処理を行うことで，メモリストレージの容量拡張を実現する．この方法でも提案手法固有の処理を行う必要があるため，IS_DAX(inode) に相当するマクロを定義するが，DAX の処理を行う中で，必要な場合に提案手法固有の処理を行うことになる．この方法の長所は，DAX と提案手法で同様の処理をする際に，追加の必要がなくなり，実装の冗長性がなくなる点である．一方，DAX の実装の制約を受ける可能性がある．しかしながら，DAX は基本的にはファイルシステムレベルでの実装となっており，dax_do_io, dax_fault の引数にはファイル情報が含まれているため，DAX を拡張し提案手法を実装することは十分に可能であると考えられる．従って，提案手法の実装には，DAX を必要に応じて拡張する方法をとることとし，現在実装を行っている．

5. 関連研究

VEMS [1] は，ブロックデバイスレベルでメモリストレージとブロックストレージを組み合わせ，メモリストレージ容量を拡張する手法である．本論文は，VEMS をもとに，ファイルシステムレベルでメモリストレージ容量を拡張する手法を提案した．

PRAMFS [2], PMFS [3] は，デバイスドライバを使用せず，ファイルシステムレイヤでメモリストレージにアクセスするファイルシステムである．メモリストレージのみを管理し，ブロックストレージとの組み合わせはサポートしない点で異なっている．

複数のブロックストレージの組み合わせにより，アクセス性能を向上させる初期の試みとしては，DCD [4] がある．DCD は，SSD 出現以前に，ブロックストレージはシーケンシャルアクセスの方が高速であることに着目し，キャッシュとするブロックストレージに，シーケンシャルアクセスを行うようにすることで，高速化を実現した．その後，高速なブロックストレージとして SSD が出現したことにより，同様な手法の研究開発が行われた [5], [6], [7], [8]．これらはいずれも，複数のブロックストレージを組み合わせ，ブロックストレージのインタフェースをブロックデバイスレベルで提供する点で異なっている．

6. まとめ

近年，メモリインタフェースを提供するストレージとして注目されている次世代不揮発性メモリや NV-DIMM 等のメモリストレージは，高性能ではあるが，容量に制限があり，主ストレージとして使用することは困難である．従って，従来のブロックストレージと組み合わせ，透過的に十分な容量を提供することが重要になる．本論文では，メモリストレージとブロックストレージとの組み合わせを，デバイスドライバレベルで行うのではなく，ファイルシステムのレイヤで行う手法を提案した．そして，提案手法の Linux 4.0 カーネルにおける実現について議論した．

参考文献

[1] 追川修一：ブロックストレージとの組み合わ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， Vol. 8, No. 2 (ACS 50) (2015).
[2] PRAMFS: Protected and Persistent RAM Filesystem, http://pramfs.sourceforge.net/ (2013).
[3] Persistent Memory File System, https://github.com/linux-pmfs/pmfs/ (2013).
[4] Hu, Y. and Yang, Q.: DCD – Disk Caching Disk: A New Approach for Boosting I/O Performance, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp. 169–178 (online), DOI: 10.1109/ISCA.1996.10021 (1996).
[5] Kgil, T. and Mudge, T.: FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers, *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, New York, NY, USA, ACM, pp. 103–112 (online), DOI: 10.1145/1176760.1176774 (2006).
[6] FlashCache: https://github.com/facebook/flashcache (2014).
[7] Saxena, M., Swift, M. M. and Zhang, Y.: FlashTier: A Lightweight, Consistent and Durable Storage Cache, *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, New York, NY, USA, ACM, pp. 267–280 (online), DOI: 10.1145/2168836.2168863 (2012).
[8] Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N. and Zhao, M.: Write Policies for Host-side Flash Caches, *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, Berkeley, CA, USA, USENIX Association, pp. 45–58 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2591272.2591278⟩ (2013).