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　15–24 (June 2015)

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

追川 修一[1,a)]

受付日 2014年10月18日, 採録日 2015年1月24日

概要：フラッシュメモリよりも記憶デバイスとしての性能がはるかに高い MRAM，PCM（phase change memory），ReRAM といった次世代不揮発性メモリ（NV メモリ）の実用化が進んでおり，これらのメモリを用いた SSD の研究開発も行われている．NV メモリのなかでも MRAM は，DRAM に相当する性能と耐久性を持つという点で優れているが，集積度では劣っているため，ストレージの主体デバイスとしての用途は，現状では考えにくい．そこで，大容量のブロックストレージと組み合わせ，MRAM の容量を仮想的に拡張して用いることで，その高速性と不揮発性を活かす手法を提案する．提案手法は，Linux カーネルに実装した．実験結果から，従来手法よりも低コストなアクセスを実現できることが分かった．

キーワード：オペレーティングシステム，不揮発性メモリ，ストレージ

Memory Storage Capacity Extension Combining with Block Storage

Shuichi Oikawa[1,a)]

Received: October 18, 2014, Accepted: January 24, 2015

Abstract: The next generation non-volatile (NV) memory technologies are emerging as their development is actively conducted to bring them into practice. There are also efforts to develop SSDs that employ these NV memory technologies. Among them, MRAM has superior characteristics to the others since its performance and endurance are comparable to DRAM while the difficulty to increase its density is its downside; thus, it is less likely for MRAM to be used as the main storage device. This paper presents a technique that virtually enhances the capacity of MRAM by combining it with larger capacity block storage. It enables the combined storage to provide the large capacity of block storage and also the performance comparable with MRAM. The proposed technique was implemented in the Linux kernel. The experiment results show that its performance is better than the existing methods.

Keywords: operating systems, non-volatile memory, storage

1. はじめに

フラッシュメモリよりも記憶デバイスとしての性能がはるかに高い MRAM，PCM（phase change memory），ReRAM といった次世代不揮発性メモリ（NV メモリ）の実用化が進んでいる．これらのメモリを用いた SSD の研究開発も行われており，たとえば，Moneta [1] は PCM をストレージデバイスとした高性能 SSD である．また，フラッシュメモリに ReRAM をキャッシュとして組み合わせることで高性能化を実現した例 [2] もある．本論文では，不揮発性のバイトアクセス可能なメモリをメモリストレージと呼び，SSD や HDD のようなブロック単位でアクセスする従来のストレージをブロックストレージと呼ぶ．

本論文では，Linux カーネルを対象とし，メモリストレージに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拡張する手法 VEMS（Virtually Extended Memory Storage）について述べる．VEMS は，バイトアクセス可能なメモリストレージへの直接アクセスを活用することで，ストレージへのアクセスコストの低減を可能に

[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
[a)] shui@cs.tsukuba.ac.jp

する．そのために，メモリストレージを隠蔽するのではなく，むしろ逆に露出させ，ブロックストレージと合わせたストレージ全体を，メモリストレージとしてアクセス可能にする．メモリストレージは，CPU が直接アクセスできるため，オペレーティングシステム（OS）カーネルがメインメモリ上に管理するページキャッシュを介さずに，アクセス可能である．全体をメモリストレージと見なすことで，いったんページキャッシュへデータをコピーする必要がなくなる．またこれにより，ページキャッシュとして用いられるメモリ容量を削減することができる．

VEMS に適したメモリストレージとして，MRAM を想定する．MRAM は，NV メモリの中では，DRAM に相当する性能と耐久性を持つという点で優れている．しかし，集積度では劣っているため，ストレージ全体を MRAM で構成することは，現状では考えにくい．そこで，大容量のブロックストレージと組み合わせ，MRAM の容量を仮想的に拡張して用いることで，その高速性と不揮発性を活かすことができる．

主ストレージに，より高速なストレージを組み合わせ，アクセスを高速化する手法は，これまで数多くの研究開発が行われてきた [3], [4], [5], [6]．これらはいずれも，低速大容量な HDD を主ストレージとし，高速な SSD を組み合わせるものである．どちらのストレージもブロックデバイスであり，OS カーネルはブロックストレージとしてアクセスする．一方，VEMS は，大容量の主ストレージと高速なストレージを組み合わせるという点は共通しているが，全体をメモリストレージとして提供するという点で，これらの既存手法とは異なっている．

VEMS は，Linux カーネルに実装した．VEMS は，メモリストレージのデバイスドライバと，ファイルシステムからメモリストレージへアクセスするためのレイヤから構成される．実験結果から，従来手法よりも低コストなアクセスを実現できていることが分かった．

本論文の構成は以下のとおりである．2 章で背景を述べる．3 章では VEMS の設計について，4 章では実装について述べる．5 章は実験結果を示す．6 章は VEMS についての考察を述べる．7 章は関連研究を述べ，8 章で本論文をまとめる．

2. 背景

本論文の背景として，NV メモリ，高速ストレージとの組合せによるアクセス高速化手法，NV メモリとの組合せにおける問題点について述べる．

2.1 NV メモリ

MRAM, PCM (phase change memory), ReRAM といった次世代不揮発性メモリ（NV メモリ）は，不揮発性であることからストレージデバイスとして用いることできる一方，バイトアクセス可能であるためメインメモリの一部として用いることができるという特徴を持つ．これらの NV メモリは，実現するための技術が異なるため，それぞれ異なった特徴を持つ [7]．そのため，用途も異なってくると考えられる．

PCM，ReRAM は，読み出しの遅延は比較的短いが，書き込みについて遅延および耐久性の問題がある．しかしながら，高い集積度を実現できるため，主にストレージでの利用が研究されている [1], [2]．

MRAM は，DRAM に相当する性能と耐久性を持つという点で優れている．MRAM の特徴として，PCM，ReRAM と異なり，書き込みについて遅延や耐久性といった問題を持たないため，メインメモリの一部，またはプロセッサのキャッシュとしての利用が研究されている [8]．集積度では劣っているため，ストレージの主体デバイスとしての用途は，現状では考えにくい．しかしながら，他の大容量ストレージと MRAM の組合せで構成されるストレージは，十分に考えられる．

2.2 NV メモリの OS サポート

NV メモリをメモリストレージとすると，その上にファイルシステムを構築して使用することになる．その場合，単にメモリストレージをブロックストレージと同様に見なし，メモリストレージのデバイスドライバは，ブロック単位のアクセス要求を処理するようにすることができる．しかしながら，この方法では，メモリストレージが直接バイトアクセス可能であるという特徴を活かしておらず，メモリストレージ上のデータはいったんページキャッシュに読み出され，必要に応じてまた書き戻されることになる．

そこで，メモリストレージへの直接アクセスを可能にするために導入されたのが，XIP（eXecution-In-Place）機能である．XIP を用いることで，ページキャッシュの介在はなくなる．read，write システムコールを用いたアクセスの場合，ユーザプロセスのバッファとメモリストレージ間で読み書きが行われるようになる．また，mmap システムコールがファイルがマップする場合，ユーザプロセスの仮想アドレス空間が，メモリストレージの領域を参照する．XIP を用いるためには，ファイルシステムとデバイスドライバ両方のサポートが必要となる．

NV メモリの管理については，上記のようにメインメモリとは別にメモリストレージとして管理する方法のほかに，不揮発性ではないメインメモリ上にも動的にメモリ割当てを行うことでファイルシステムも構築可能であるため，メインメモリと同様に管理可能であるとの主張もあり，両者の間で議論がある [9]．NV メモリをメインメモリと同様に管理可能であるとすれば，既存のページキャッシュ機構に組み入れることで，ページキャッシュとストレージの統合の可能性がある．しかしながら，ページキャッシュ機構を

含め，メインメモリの管理機構は，実行中のカーネルと密接に関係しており，カーネルの再起動やバージョン変更をまたいでデータを保持する前提とはなっていない．そのため，ページキャッシュとストレージの統合には，大きな変更が必要であり，これは将来的な課題である．

2.3　高速ストレージとの組合せによるアクセス高速化

主ストレージに，より高速なストレージを組み合わせ，アクセスを高速化する手法は，これまで数多くの研究開発が行われてきた [3], [4], [5], [6]．これらはいずれも，低速大容量な HDD を主ストレージとし，高速な SSD を組み合わせるものである．頻繁にアクセスされるデータを，SSD 上に置くことで，アクセスを高速化する．一方，アクセスされないデータは HDD 上に置き，全体としては大容量を提供する．

これらは，別個の HDD と SSD を，ソフトウェアまたはハードウェアにより組み合わせ，全体として単一のストレージとする．そのため，最新データの所在が分散する，組合せのための情報が失われると単一ストレージに戻すことができない，SSD の書き込み耐久性は限りがある，といった問題点があり，これらに対処するための研究が行われてきた．

2.4　NV メモリとの組合せにおける問題点

NV メモリを高速なメモリストレージとし，主ストレージとなるブロックストレージと組み合わせる場合，組み合わせたストレージのインタフェースを，メモリストレージとするか，ブロックストレージとするかが問題となる．HDD に SSD を組み合わせる場合，どちらのストレージもブロックデバイスであるため，OS カーネルは，組み合わせたストレージを，ブロックストレージとしてアクセスする．同様に，メモリストレージをブロックストレージとして扱い，全体としてもブロックストレージとしてアクセスすることができる．この場合，単に SSD の代替として，NV メモリを使用することになる．

メモリストレージは，1) CPU からの直接アクセスが可能であり，ページキャッシュを介する必要がないことから，アクセスコストが低減される，2) ページキャッシュのためのメモリ容量が不要になる，といった利点を持つ．しかしながら，NV メモリをブロックストレージとして扱う方法では，これらの長所を活かすことができないという問題点がある．

3.　設計

本章では，Linux カーネルを対象とし，メモリストレージに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拡張する手法 VEMS（Virtually Extended Memory Storage）について述べる．まず，対象とするシステム構成を明確にする．次に，目的と実現すべき要件を定義した後に，提案手法について述べる．

3.1　対象とするシステム構成

メモリストレージ，ブロックストレージの組合せ方には，コンピュータシステムへの接続方法およびストレージ間の制御方法について，それぞれ以下のいくつかの形態が考えられる．まず，コンピュータシステムへの接続方法としては，以下の 3 つの形態がありうる．

(1) メモリストレージ，ブロックストレージの両方が，I/O バスを通してコンピュータシステムに接続．
(2) メモリストレージはメモリバス，ブロックストレージは I/O バスを通し，それぞれ別個にコンピュータシステムに接続．
(3) メモリストレージ，ブロックストレージの両方が，メモリバスを通してコンピュータシステムに接続．

また，メモリストレージ，ブロックストレージ間の，データ転送を含む制御方法としては，以下の 2 つの形態がありうる．

(a) ストレージ間のデータ転送は，デバイス側で制御．
(b) ストレージ間のデータ転送は，コンピュータシステム側のソフトウェアで制御．

接続方法と制御方法の組合せのうち，現実的な形態としては，(1-a)，(2-b) が考えられる．(1-a) 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た，単一のストレージデバイスとしての提供となる．(2-b) は，メモリストレージとブロックストレージは，別個のデバイスとしての提供となる．(1-a)，(2-b) の形態を，**図 1**，**図 2** に示す．

本論文では，メモリストレージとして用いる NV メモリとして MRAM を想定し，(2-b) の組合せ形態をターゲットとする．その理由として，メインメモリとして適した特性を持つ MRAM はメモリバスに接続することでその性能を発揮することができ，また別途制御用のハードウェアを用意することなく，既存のブロックストレージと組み合わせ

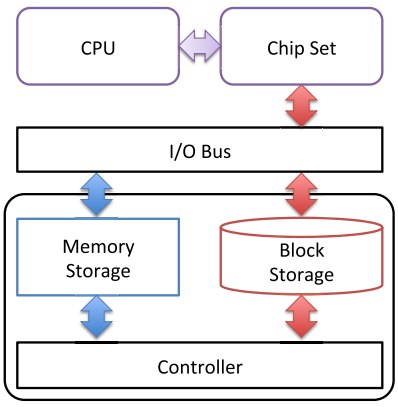


図 1　メモリストレージ，ブロックストレージの組合せ形態 (1-a)
Fig. 1　Combination form of memory and block storage (1-a).

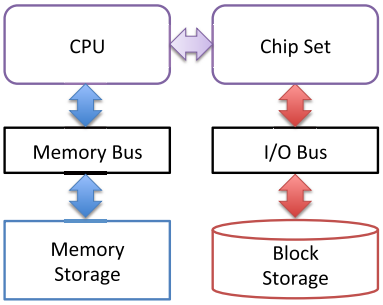


図 2　メモリストレージ，ブロックストレージの組合せ形態 (2-b)
Fig. 2　Combination form of memory and block storage (2-b).

て使用することができるからである．さらに，サーバシステムでは，その管理は一定のスキルが必要であり，デバイスの交換時に必要な手順をふむ必要があり，またタブレットやスマートフォン等のモバイルシステムでは，基本的にメモリやブロックストレージ等のデバイスは交換不可能である．その場合，ソフトウェアでストレージ間の制御を行ったとしても，単一のストレージデバイスと見なすことができると考えるからである．

3.2　目的と要件

本論文で提案する手法の目的は，メモリストレージとブロックストレージを組み合わせ，ブロックストレージの容量を持つメモリストレージを提供することである．メモリストレージを提供することは，すなわち，メモリストレージのインタフェースを提供することである．

この目的のために実現すべき要件を以下にまとめる．

(1) ブロックストレージの大きさのアドレス空間を提供する．
(2) 提供するアドレス空間の範囲内で指定されたブロックの最新データを提供する．
(3) アクセスに際しては，基本的には同期的なインタフェースを提供する．
(4) 仮想アドレス空間へマップ可能にする．そのために，少なくともページフレームのサイズの領域に分割し管理する．

(1)，(2) は，組合せにより提供されるストレージが，ブロックストレージと同等に機能することを意味する．(1) は，容量はブロックストレージと等しくなることを意味する．(2) は，最新データが，メモリストレージにあればメモリストレージから，ブロックストレージにあればブロックストレージから提供されることを意味し，古く無効になったデータが提供されることがないことを意味する．

(3)，(4) は，そのストレージが，基本的にはメモリストレージとしてアクセス可能になることを意味している．(3) は，メモリストレージへのアクセスは同期的であること，(4) は，メモリストレージは仮想アドレス空間へマップ可能であることを意味する．

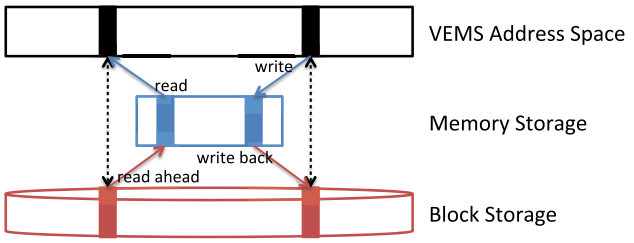


図 3　提案手法の概観
Fig. 3　Overview of the proposed technique.

3.3　VEMS: Virtually Extended Memory Storage

本論文は，メモリストレージの容量を拡張する手法 VEMS（Virtually Extended Memory Storage）を提案する．**図 3** に提案手法の概観を示す．VEMS はブロックストレージと同じ大きさのアドレス空間を提供し，VEMS とブロックストレージのアドレスは 1 対 1 の対応をとる（黒破線）．メモリストレージは，ブロックストレージよりも容量が小さいため，VEMS のアドレスと 1 対 1 に対応する特定の領域は存在せず，ある領域はその時々で異なる VEMS のアドレスのデータを格納することとなる（赤実線）．ある VEMS のアドレスに対応するデータをメモリストレージ上に格納することで，メモリストレージのインタフェースを提供可能にする（青実線）．

以下に，VEMS の実現手法をまとめる．

- メモリストレージは，プロセッサのページフレームサイズに分割し管理する．分割した各領域が，ブロックストレージのどのブロックに対応するか，また各領域の状態遷移の情報を，各領域の属性として別途管理する．
- メモリストレージに書き込まれたデータは，適宜，ブロックストレージへの書き込みを行うことで，メモリストレージ上の再利用可能な領域を確保する．
- メモリストレージには，メインメモリを仮想アドレス空間へのマップを管理するために用いられる，ページフレーム管理のためのデータ構造体 `struct page`[*1] を割り当てる．`struct page` を割り当てることで，あるページが仮想アドレス空間にマップされた状態にあるかどうか，またアンマップ後にそのページへの書き込みが起こったかどうかの情報を取り出すことが容易になる．そこで，ページへの書き込みが起こったかどうかの管理は，`struct page` に統合する．
- 必要に応じて，データアクセスに際し，メモリストレージへの先読みやメモリストレージをバイパスする処理を行う．読み出しに際しては，メモリストレージ上にデータがない場合，ブロックストレージからの読み出しには大きな遅延がともなうため，それを回避するための先読み処理を行う．また，アクセスにあたり，メ

[*1] `struct page` は，Linux カーネルにおけるページフレーム管理データ構造体の実装である．

モリストレージを活用できない場合，できる状態になるまで待つのではなく，メモリストレージをバイパスし，ブロックストレージへアクセスする．

メモリストレージは，仮想アドレス空間へマップ可能にするため，プロセッサのページフレームと同じサイズに分割し管理する．そのため，メモリストレージとブロックストレージの間では，ページフレームサイズでデータをやりとりする．しかしながら，一般に，ブロックストレージは512バイトのセクタに分割され，この単位でアドレスが割り振られている．ストレージは，最小アクセス単位のサイズをOSカーネルに伝えることで，基本的にはそのサイズでのみアクセスされるが，マウント時等で例外的にセクタ単位でのアクセスが要求される場合もある．セクタサイズはページフレームサイズよりも小さい場合が多いため，ページフレームサイズにアラインされていないアクセスにも，対応が必要である．

メモリストレージに書き込まれたデータは，適宜，ブロックストレージへの書き込みを行う．メモリストレージへの書き込が起こった領域は，最新データがメモリストレージ上にあるため，ブロックストレージへの書き込みが終了するまで，ブロックストレージの異なるブロックのデータを置くために再利用できない．メモリストレージ上の，ブロックストレージへの書き込みが終了した領域，またブロックストレージから読み出されただけで書き込まれていない領域は，再利用可能である．

ブロックストレージへの書き込みは，ブロックストレージ上で連続するブロックを検索し，できるだけ大きなサイズで書き込みを行う．SSDであっても，シーケンシャルアクセスの方が高速であるため，連続ブロックを構成することで，書き込みの効率を高めることができる．

メモリストレージのある領域に書き込みが起こったかどうかは，writeシステムコールを通して書き込む場合は，容易に把握することができる．しかしながら，XIPを通して，ユーザプロセスの仮想アドレス空間にマップされた場合，書き込みがカーネルの実行をともなうとは限らない．この場合，ページテーブルエントリに含まれるdirtyビットの情報を使用する必要がある．この情報は，アンマップ時に`struct page`に反映されることから，取り出すことができる．そのため，ページへの書き込みが起こったかどうかの管理は，`struct page`に統合する．

上述したように，VEMSは，メモリストレージに`struct page`を割り当て，`struct page`によって提供される機能を活用する．`struct page`の割当ては，メインメモリを消費する．そのため，メモリストレージがブロックストレージ相当の容量を提供する場合に，`struct page`を割り当てることには慎重な検討が必要である．しかし，VEMSが対象とするシステム構成では，メモリストレージとしてMRAMを想定しており，その容量としてはメインメモリ程度が考えられるため，`struct page`を割り当てることに問題はない．

遅延を削減するため，データアクセスに際し，必要に応じて，メモリストレージへの先読みやメモリストレージをバイパスする処理を行う．読み出しに際しては，メモリストレージ上にデータがない場合，ブロックストレージからの読み出しには大きな遅延がともなう．先読み処理を行い，あらかじめデータをメモリストレージ上に置くことで，遅延を回避することができる．また，書き込みに際して，メモリストレージ上に使用可能領域がない場合，使用可能な領域ができるまで待つと，大きな遅延が生じる．また，強制的にメモリストレージ上のデータのブロックストレージへの書き込みを起動し，使用可能領域を確保しようとしても，小さなサイズの書き込みを行うことになり，書き込みサイズに対する遅延は大きい．そこで，メモリストレージ上に使用可能領域がない場合，メモリストレージをバイパスし，ブロックストレージへのアクセスを行う．メモリストレージをバイパスする処理は，読み出し時にメモリストレージ上にデータがない場合にも行う．

4. 実装

3.3節で述べた提案手法の，Linuxにおける具体的な実現方法について述べる．まず，メモリストレージの管理について述べた後に，XIPをベースしたメモリストレージへのインタフェースを提供する実装について述べる．そして，従来のブロックデバイスインタフェースを提供する実装について述べる．

4.1 メモリストレージの管理

メモリストレージは，プロセッサのセット連想方式キャッシュと同様に管理する実装を行った．セット連想方式キャッシュとしたのは，メモリストレージがメインメモリ相当の容量の場合であっても，ブロックストレージのブロックアドレスに対応するメモリストレージ領域の検索を，単純なアルゴリズムにより一定コストで行えるようにするためである．メモリストレージを，プロセッサのページフレームサイズに分割したデータ領域が，キャッシュラインに相当することになる．セット数は，デバイスドライバのモジュールパラメータで指定可能である．デフォルトのセット数は16とした．

セット連想方式キャッシュでは，各キャッシュラインのデータのメインメモリでのアドレスや状態を，対応するタグに格納し，キャッシュラインを管理する．VEMSでのメモリストレージの管理でも，同様に，メモリストレージを分割した各データ領域にもタグを付与する．タグは，ブロックストレージのセクタアドレスを格納するために十分な大きさとし，さらに状態遷移情報を格納する．

メモリストレージのタグは，ページフレームサイズに分

割したデータ領域とは別に，まとめて管理する．データ領域とタグ領域を別にすることで，各データ領域の先頭アドレスは，無駄なくページフレームサイズにアラインすることができる．

タグには，VEMS での管理に必要となる状態遷移情報として，UPDATE，WRITEBACK を格納する．UPDATE は，対応するデータ領域が更新中であることを表す．WRITEBACK は，書き込みが起こった領域を対象にした，ブロックストレージへの書き出しの最中であることを表す．データ領域に書き込みが起こったかどうかの状態は，3.3 節で述べたとおり，タグではなく struct page で管理される．

VEMS のアドレスに対応するメモリストレージ領域があるかどうかの検索は，アクセスするアドレスからセットを計算し，該当するセットに含まれるタグに，そのアドレスを格納したタグがあるかどうかを検索する．検索は，セット数が少ないことから，単純に線形探索を用いている．アクセスするアドレスを格納したタグが見つかれば，対応するデータ領域とタグへのアドレスを返し，データ領域への読み書き，および必要に応じたタグの更新を行う．見つからなかった場合の処理は，アクセスが読み出しか書き込みかで異なる．読み出しの場合は，メモリストレージをバイパスするため，見つからなかったことを表す値を返す．書き込みの場合は，再利用可能な領域として，書き込まれておらず，UPDATE，WRITEBACK のどちらの状態でもない領域を検索し，見つかれば，そのデータ領域とタグへのアドレスを返す．使用可能な領域が見つからなかった場合は，そのことを表す値を返す．この場合，該当するセットのブロックストレージへの書き戻しを行った後に，再検索を行う．

書き込まれた領域は，再利用可能な領域にするため，ブロックストレージへ書き出す．書き出しは非同期的な処理として行われるため，UPDATE 状態の領域は，書き出しの対象から除外する．一方，WRITEBACK 状態の領域への，書き込みは許可している．これは，ブロックストレージへの書き出し開始時に，struct page で管理される書き込み状態を消去しておくことで，書き出し中に，領域への書き込みが起こったかどうか把握できるためである．

4.2　VEMS インタフェースの提供

VEMS が提供する，メモリストレージへのインタフェースの実装について述べる．メモリストレージへのインタフェースは，XIP のインタフェースをもとに，拡張したものとなっている．

まず，元となった XIP のインタフェースの概要について述べる．XIP を有効にしてマウントされたファイルシステムのファイルを，read，write システムコールによりアクセスする場合，xip_file_read()，xip_file_write() 関数が呼ばれる．どちらの関数も，ファイルシステムが提供する get_xip_mem() インタフェースを呼び出すことで，アクセス先のページフレームのアドレスを取得し，データの読み出し，または書き込みを行う．ファイルシステムが直接メモリストレージを管理しない場合，メモリストレージを管理するドライバが提供する direct_access() インタフェースを呼び出し，セクタアドレスに対応するメモリストレージのアドレスを取得する．

VEMS が，XIP のインタフェースをそのまま使用すると，以下の情報および機能不足により，処理を効率的に行うことができない．

(1) アクセスが読み出しなのか，書き込みなのかの情報が渡されない．そのため，読み書きのいかんにかかわらず，書き込みと見なす必要があり，無駄なブロックストレージへの書き戻しが必要になる．
(2) アクセスするデータサイズの情報が渡されない．そのため，書き込み時にアクセス先アドレスのデータがメモリストレージ上にない場合，すべて新しいデータで上書きされてしまう場合でも，無駄にブロックストレージからメモリストレージへの読み出しが必要になる．
(3) アクセス先アドレスのデータがメモリストレージ上にない，またはメモリストレージ上に再利用可能なデータ領域がないことを，xip_file_read()，xip_file_write() 関数のレベルで知ることができない．そのため，要求されたデータサイズに対応して，メモリストレージをバイパスする処理を行うことができない．
(4) XIP は，すべてのデータがメモリストレージ上に置かれることを前提としている．そのため，xip_file_read() 関数が，ブロックストレージからメモリストレージへの先読みを指示するためのインタフェースを提供していない．

上記の情報不足を解消するため，XIP のインタフェースの引数を拡張し，VEMS のためのインタフェースを定義した．拡張したインタフェースの実装では，上記の情報を用いた処理を行い，メモリストレージとブロックストレージ間の無駄なデータの移動を省き，処理を効率化する．既存インタフェースの引数の拡張では，メモリストレージをバイパスする処理を要求するためのインタフェース，先読みを指示するインタフェースが不足するため，それらを追加した．**図 4**，**図 5** に，VEMS インタフェース実装のプロトタイプ宣言と階層構造を示す．vems_file_read()，vems_file_write() は，XIP インタフェースとは名前が異なるのみであるため，図 4 からは省略した．

ext2_get_vems_mem()，vems_direct_access() における，rw，page_aligned，allocate が，拡張された引数である．rw は，上記 (1) に対応し，アクセスが読み書きのどちらなのかを示す．page_aligned は，上記 (2) に対

```
int ext2_get_vems_mem(struct address_space *mapping, pgoff_t pgoff, int create,
    int rw, int page_aligned, int allocate, void **kmem, unsigned long *pfn)
void ext2_read_vems_mem(struct address_space *mapping, pgoff_t pgoff, struct page *page)
void ext2_read_ahead_vems_mem(struct address_space *mapping, pgoff_t pgoff)

int vems_direct_access(struct block_device *bdev, sector_t ksector, void **kaddr, unsigned long *pfn,
    int rw, int page_aligned, int allocate)
void vems_read_page(struct block_device *bdev, sector_t ksector, struct page *page)
void vems_read_ahead(struct block_device *bdev, sector_t ksector)
```

図 4　VEMS インタフェースのプロトタイプ宣言
Fig. 4　Prototype declaration of VEMS interface.

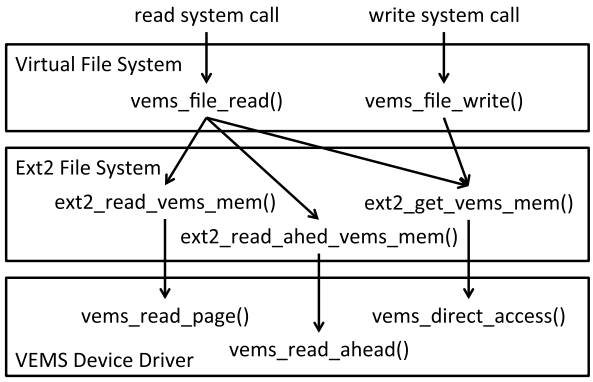


図 5　VEMS インタフェースの実装の階層構造
Fig. 5　Hierarchy of VEMS interface implementation.

応し，データサイズがページフレームサイズにアラインしているかを示す．アラインしている場合は，書き込み時にブロックストレージからの読み出しが不要になる．`allocate` は，メモリストレージに領域を確保する必要性を示し，mmap への対応で必要となる．上記 (3) に対応しメモリストレージをバイパスする処理を行うため，`vems_direct_access()` は該当する場合に，`ENOMEM` をエラーコードとして返す．そしてバイパスする読み出し処理のために，`ext2_read_vems_mem()`，`vems_read_page()` を追加した．上記 (4) に対応し先読みを行うため，`ext2_read_ahead_vems_mem()`，`vems_read_ahead()` を追加した．`vems_file_read()`，`vems_file_write()` は，これらの拡張に対応する処理を実装する．

メモリストレージをバイパスする処理は，現状では読み出しについてのみ行っている．バイパスする場合，`vems_file_read()` は，`ext2_read_vems_mem()` を呼び出す．そこでファイルのブロック情報を取得し，そこから `vems_read_page()` を呼び出し，そのブロックを読み出す．そのため，ある程度大きなサイズの連続したブロックに対して読み出しを行わないと，性能が低下する．そのために，指定された読み出しサイズだけ上記の処理を行った後に，一括して `vems_read_page()` からブロックストレージへ読み出し要求を送るようにしている．また，ページキャッシュは用いないため，読み出し先は，ユーザプロセスのバッファ領域となる．

VEMS は，理想的にはファイルシステムから中立に実装できることが理想である．しかしながら現状では，図 5 における，ファイルシステムに汎用的な処理をまとめた仮想ファイルシステム（VFS）層，個別のファイルシステム（Ext2）を実装する層にまたがった実装が必要となっている．それは，VFS 層が扱うファイルとその中のオフセットを，Ext2 層でブロックアドレスに変換する必要があるからである．実際，Ext2 層で VEMS に対応するために実装した 3 つの関数は，ブロックアドレスに変換した後に，対応する VEMS ドライバ層の関数を呼ぶだけとなっている．したがって，ファイルとその中のオフセットからブロックアドレスに変換するインタフェースを定義することにより，少なくとも個別のファイルシステム層から中立に実装することが可能になる．

4.3　VEMS-block の実装

メモリストレージのインタフェースとの性能比較のために，従来のブロックデバイスのインタフェースも実装した．ブロックデバイスのインタフェースを提供する実装を，VEMS-block と呼ぶ．

VEMS-block は，VEMS と同様に，メモリストレージとブロックストレージを組み合わせる．Linux は，複数のブロックストレージを組み合わせるためのフレームワークとして，device mapper を提供している．device mapper は，たとえば，複数のブロックストレージから 1 つの論理ボリュームの構成を可能にする LVM（Logical Volume Manger）の実現に用いられている．VEMS-block は，当初，メモリストレージを管理するラムディスクドライバとブロックストレージのドライバを，device mapper により組み合わせる実装を検討した．しかしながら，プロトタイプ実装の評価から，device mapper を経由することのオーバヘッドが若干あるため，device mapper は用いない実装を行った．

ブロックデバイスのインタフェースには，基本的には 2

種類，1) ブロックストレージへのアクセスをより効率良くするため，アクセス要求のスケジューリングを行うものと，2) 単純にアクセス要求をブロックストレージへ渡すだけのもの，がある．VEMS-block には，2) のインタフェースを用いた．2) の実装には，`blk_queue_make_request()` 関数を呼び出し，ブロック単位でのアクセス要求を受け取る関数を登録する．以下に，VEMS-block が登録する関数のプロトタイプ宣言を示す．

```
void vems_make_request(struct request_queue *q,
    struct bio *bio)
```

`vems_make_request()` 関数は以下の処理を行う．各アクセス要求は，第 2 引数 `bio` で渡される．読み出しの場合，要求されたデータがメモリストレージ上にあるか検索する．あれば，そのデータを `bio` が指定するバッファ領域にコピーする．なければ，ブロックストレージのドライバに `bio` を転送し，ブロックストレージから読み出しを行う．書き込みの場合，書き込み先のアドレスのデータがメモリストレージ上にあるか，もしなければ，メモリストレージ上に再利用可能なデータ領域があるか検索する．あれば，`bio` が指定するデータをメモリストレージへコピーする．なければ，ブロックストレージのドライバに `bio` を転送し，ブロックストレージへ書き込みを行う．メモリストレージへのアクセスで処理が完結した場合，`vems_make_request()` 関数での処理の最後に，`bio_endio()` 関数を呼ぶことで，アクセス要求の同期的な処理が可能になる．一方，ブロックストレージのドライバに転送した `bio` は，ブロックストレージのドライバが非同期的に処理する．

なお，`bio` は連続する領域へのアクセス要求を指定することができる．VEMS-block では，`bio` がページフレームサイズを超えたサイズの要求を含むと，処理が煩雑となるため，1 つの `bio` にページフレームサイズを超えたサイズの要求が入らないように設定している．

5. 実験結果

本章では，VEMS を用いて実験を行った結果を示す．まず，実験環境をまとめ，ファイルアクセス性能を示す．そして，読み出し性能について，先読みとバッファサイズの影響を示す．

5.1 実験環境

MRAM を装備したシステムは一般に入手することは困難であるため，実験には MRAM の代わりに通常の DRAM を用いた．実験には，Intel Core i7-3770 3.4 GHz を搭載する PC 互換機を用いた．メインメモリに 256 MB，メモリストレージに対応するメモリに 256 MB を割り当てた．ブロックストレージには，PCIe Gen2 x2 接続の Plextor M6e PCI Express SSD を用いた．

VEMS を実装した Linux カーネル 3.14.12 上で，ファイルシステムとして Ext2 を用い，ファイルの読み出し，書き込みを行うベンチマークプログラムを実行し，実行時間を計測した．実行時間の計測には，TSC（Time Stamp Counter）を用いた．

比較対象は，ラムディスク，Linux カーネルに含まれている device mapper を用いてラムディスクとブロックストレージの組合せを可能にする dm-cache，そして SSD である．ラムディスクは，XIP インタフェースと，単純にアクセス要求をブロックストレージへ渡すだけのブロックデバイスのインタフェースを提供する．ラムディスクのみの場合は XIP インタフェース，dm-cache と用いる場合はブロックデバイスのインタフェースを用いた．ラムディスクのみの性能計測時には，4 GB のメモリを割り当てた．

5.2 ファイルアクセス性能

ファイルを作成し書き込みを行い，次にそのファイルの読み出しにかかった実行時間を計測した結果を，**図 6**，**図 7** に示す．図では，実行時間を示す縦軸はログスケールとなっている．VEMS は 4.2 節で述べた VEMS インタフェースを使用した場合，VEMS-block は 4.3 節で述べた従来のブロックインタフェースを使用した場合を表す．

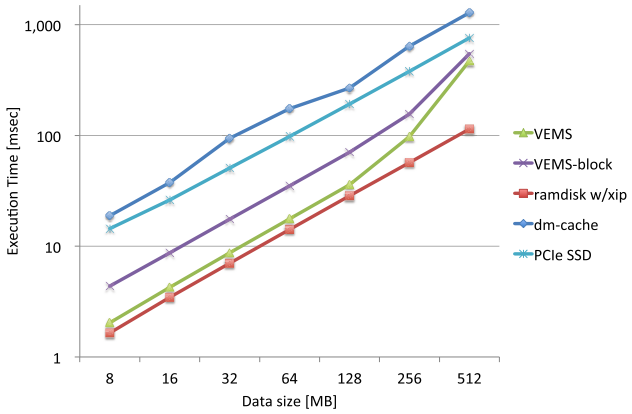


図 6　ファイル読み出し性能の比較
Fig. 6　File read performance comparison.

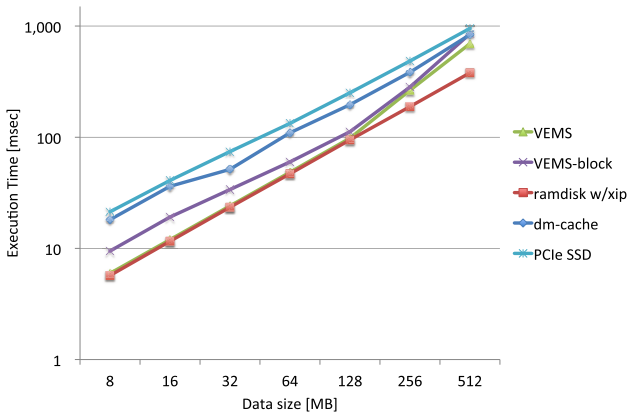


図 7　ファイル書き込み性能の比較
Fig. 7　File write performance comparison.

ファイルを新たに作成し，書き込みを行っているため，書き込みに関しては，ファイルへのブロック割当てのコストを含む．また，書き込みと読み出しの間に，ページキャッシュを無効化する操作を行っているため，読み出しに関しては，ストレージからの読み出しコストとなっている．

ラムディスクを除くと，ファイルサイズが 8 MB から 512 MB までの読み出し，書き込みの両方で，VEMS が最も高速，その次が VEMS-block という結果となった．VEMS は，8 MB から 128 MB までは，ラムディスクに近い性能となっており，読み出しで 23〜26%，書き込みでは 3〜5%の性能低下にとどまっている．VEMS-block と比較すると，読み出しで 96〜115%，書き込みでは 14〜59%高速である．メモリストレージのサイズと同じ 256 MB，またそれを超える 512 MB のファイルサイズとなると，読み出し，書き込みともに，VEMS-block の性能に近づいてはいるが，ファイルサイズ 512 MB では，読み出しで 16%，書き込みで 23%，VEMS の方が高速である．

5.3　先読みとバッファサイズの影響

VEMS によるファイルアクセス高速化の要因を解析するため，読み出しの場合について，先読みを行わない場合，先読みを行わずユーザプロセスにおけるバッファ領域を拡大した場合について計測を行った．ユーザプロセスにおけるバッファ領域は，デフォルトでは `BUFSIZ` マクロに定義された 8 KB を用いており，拡大した場合は 128 KB とした．また，VEMS はページキャッシュを介さずに読み出しを行うため，比較のため，SSD からダイレクト I/O（DIO）による読み出し性能を計測した．なお，DIO は先読みを行わない．計測した結果を**図 8** に示す．

結果からは，ファイルサイズ 512 MB では，デフォルトのバッファ領域サイズにおいて，VEMS の先読みを行う場合と行わない場合の性能差は 7.0 倍あり，先読みを行わない場合の性能劣化が著しいことが分かる．先読みを行わない VEMS は，その結果が DIO とほぼ同じになっていることから，バッファ領域のサイズで同期的に読み出した結果，性能劣化を引き起こしていると考えられる．バッファ領域サイズを 128 KB に拡大することで，先読みを行わない VEMS と DIO の性能は改善され，VEMS の先読みを行う場合と行わない場合の性能差は 31%まで縮まる．バッファ領域サイズの拡大は，一度のアクセス要求で SSD から読み出すデータ量を増加するため，全体としては読み出しの遅延が減少し，読み出し性能が向上する．それでも，先読みを行わない VEMS は VEMS-block 以下の性能にとどまっている．上記の結果から，読み出し性能の向上には，先読みの影響が大きいことが分かる．

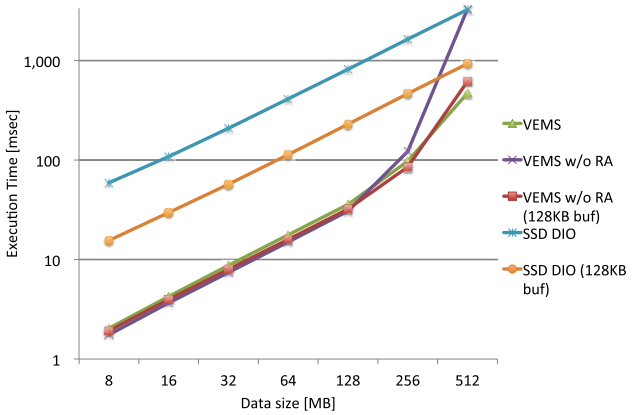


図 8　ファイル読み出し性能への先読みの影響

Fig. 8　File read performance comparison with and without read ahead.

6.　考察と今後の課題

実験結果から，データサイズがメモリストレージに収まる場合に，大きな性能向上が見込めることが分かった．このことから，VEMS が特に有効と考えられる対象について考察する．

まず候補となるのは，データベースである．データベースは，システムの障害に耐えつつ，データの一貫性を保つ必要がある．そのため，復旧可能なデータ更新処理を行うため，多くのデータベースシステムが先行書き出しログ（WAL）を用いているが，ストレージの不揮発性デバイス部への複数回の書き込みを行う必要があり，大きなオーバヘッドとなっている．DuraSSD [10] は，SSD が一般的に搭載する DRAM キャッシュをバッテリにより不揮発性メモリ化することにより，オーバヘッド削減に成功している．VEMS も，メモリストレージへの書き込み終了時点で，不揮発化を保証するため，同様のオーバヘッド削減が可能である．VEMS は，メモリストレージへの同期アクセスおよび直接書き込みによる，さらに低いアクセスコストを提供するため，オーバヘッドの削減度合いはより大きいと考えられる．

次に候補となるのは，Hadoop 環境の分散ファイルシステムを実現する HDFS [11] である．HDFS は，既存のファイルシステム上に構築された，ユーザレベルサービスである．データの保存には既存のローカルファイルシステムを使用するため，ローカルファイルシステムの複数のブロックファイルが，HDFS の 1 つのファイルを構成する．HDFS は，シーケンシャルの読み書きに最適化した設計となっており，ブロックファイルは 64 MB がデフォルトの最大サイズとなっている．MapReduce の map 処理は，ブロックファイル単位に分散処理を行う．すなわち，1 つのブロックファイルをシーケンシャルに読み出し，結果をまたシーケンシャルに書き込む．そのため，結果の書き込みと次の処理の読み出しがオーバラップするとしても，現実的なサイズのメモリストレージは，複数のブロックファイルを配置することができる．そのため，メモリストレージ

に対する低いアクセスコストを十分に活用でき，VEMS は性能向上に寄与すると考えられる．

これらの候補における VEMS の有効性の実証は，今後の課題である．また，VEMS は XIP 機能を拡張したインタフェースを提供するため，それを通して，メモリストレージをユーザプロセスの仮想アドレス空間にマップ可能である．そのための実装は行われているが，現状では性能は十分ではないため，その性能向上も今後の課題である．

7. 関連研究

複数のブロックストレージの組合せにより，アクセス性能を向上させる初期の試みとしては，DCD [12] 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 [3], [4], [5], [6]．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るものであり，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インタフェースを提供する VEMS とは異なる．

eNVy [13] は，バイトアクセス可能かつ読み出しは DRAM 相当の性能を持つ NOR 型フラッシュメモリに，SRAM を書き込みバッファとして組み合わせ，メインメモリの一部として使用可能にするシステムである．バイトアクセス可能なメモリを組み合わせている点が VEMS と類似しているが，ブロックストレージとの組合せでない点で異なっている．

8. まとめ

フラッシュメモリよりも記憶デバイスとしての性能がはるかに高い MRAM，PCM（phase change memory），ReRAM といった次世代不揮発性メモリ（NV メモリ）の実用化が進んでおり，これらのメモリを用いた SSD の研究開発も行われている．そのなかでも MRAM は，DRAM に相当する性能と耐久性を持つという点で優れているが，集積度では劣っているため，ストレージの主体デバイスとしての用途は，現状では考えにくい．そこで，大容量のブロックストレージと組み合わせ，MRAM の容量を仮想的に拡張して用いることで，その高速性と不揮発性を活かす手法として VEMS（Virtually Extended Memory Storage）を提案した．VEMS は，Linux カーネルに実装し，実験結果から，従来手法よりも低コストなアクセスを実現できていることが分かった．

参考文献

[1] Akel, A., Caulfield, A.M., Mollov, T.I., et al.: Onyx: a protoype phase change memory storage array, *Proc. HotStorage'11*, p.2, USENIX (2011).

[2] Tanakamaru, S., Doi, M. and Takeuchi, K.: Unified solid-state-storage architecture with NAND flash memory and ReRAM that tolerates 32x higher BER for big-data applications, *Conf. Digest ISSCC'13*, pp.226–227, IEEE (2013).

[3] Kgil, T. and Mudge, T.: FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers, *Proc. CASES '06*, pp.103–112, ACM (2006).

[4] Facebook: FlashCache, available from ⟨https://github.com/facebook/flashcache⟩ (2014).

[5] Saxena, M., Swift, M.M. and Zhang, Y.: FlashTier: A Lightweight, Consistent and Durable Storage Cache, *Proc. EuroSys '12*, pp.267–280, ACM (2012).

[6] Koller, R., Marmol, L., Rangaswami, R., et al.: Write Policies for Host-side Flash Caches, *Proc. FAST'13*, pp.45–58 USENIX (2013).

[7] Song, Y.-J., Jeong, G., Baek, I.-G. and Choi, J.: What Lies Ahead for Resistance-Based Memory Technologies?, *Computer*, Vol.46, No.8, pp.30–36 (2013).

[8] 藤田　忍，安部恵子，野村久美子，野口紘希：携帯情報端末におけるノーマリーオフコンピューティング—STT-MRAM で実現するノーマリーオフメモリ技術，情報処理，Vol.54, No.7, pp.668–676 (2013).

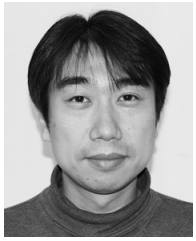
[9] Supporting filesystems in persistent memory, available from ⟨http://lwn.net/Articles/610174/⟩ (2014).

[10] Kang, W.-H., Lee, S.-W., Moon, B., et al.: Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases, *Proc. SIGMOD '14*, pp.529–540, ACM (2014).

[11] Shvachko, K., Kuang, H., Radia, S., et al.: The Hadoop Distributed File System, *Proc. MSST '10*, pp.1–10, IEEE (2010).

[12] Hu, Y. and Yang, Q.: DCD – Disk Caching Disk: A New Approach for Boosting I/O Performance, *Proc. ISCA '96*, pp.169–178, ACM (1996).

[13] Wu, M. and Zwaenepoel, W.: eNVy: a non-volatile, main memory storage system, *Proc. ASPLOS '94*, pp.86–97, ACM (1994).



追川 修一　（正会員）

平成 8 年慶應義塾大学より博士（工学）．平成 16 年筑波大学大学院システム情報工学研究科助教授に着任．現在，筑波大学システム情報系情報工学域准教授．オペレーティングシステムに関する研究に従事．IEEE 会員．