情報処理学会論文誌 Vol.54 No.3 1153–1164 (Mar. 2013)

Non-Volatileメインメモリとファイルシステムの融合

追川 修一[1,a)]

受付日 2012年7月9日, 採録日 2012年12月7日

概要：近年，不揮発性の non-volatile（NV）メモリの性能向上が著しく，高速化，大容量化，低価格化が進んでいることから，それらをメインメモリとして用いる研究，またストレージデバイスとして用いる研究が，それぞれ別個に行われてきた．しかしながら，メインメモリおよびストレージの両方としても用いることのできる NV メモリは，その両方を融合できることを意味する．融合により，メインメモリとして使用できるメモリ領域が増加し，これまでメインメモリ容量を超えてメモリ割当て要求があった場合に発生していたページスワップが不要になることで，システムの処理性能が向上する．本論文は Linux を対象とし，NV メモリから構成されるメインメモリとファイルシステムの具体的な融合方法を提案する．そして，Linux をエミュレータ上で実行する評価実験を行い，融合が可能であること，また性能面でも有効であることを示す．

キーワード：オペレーティングシステム，メモリ管理，ファイルシステム，不揮発性メモリ

Unification of Non-Volatile Main Memory and a File System

Shuichi Oikawa[1,a)]

Received: July 9, 2012, Accepted: December 7, 2012

Abstract: Recent advances of non-volatile (NV) memory technologies make significant improvements on its performance including faster access speed, larger capacity, and cheaper costs. While the active researches on its use for main memory or storage devices have been stimulated by such improvements, they were conducted independently. The fact that NV memory can be used for both main memory and storage devices means that they can be unified. The unification of main memory and a file system based on NV memory enables the improvement of system performance because paging becomes unnecessary. This paper proposes a method of such unification and its implementation for the Linux kernel. The evaluation results performed by executing Linux on a system emulator shows the feasibility of the proposed unification method.

Keywords: operating systems, memory management, file systems, non-volatile memory

1. はじめに

近年，不揮発性の non-volatile（NV）メモリの性能向上が著しく，高速化，大容量化，低価格化が進んでいる．その中でも，相変化メモリ（PCM: Phase Change Memory）は，プロセッサがバイト単位で直接アクセス可能であり，DRAM に近いアクセス性能を持つことから，その大容量，省電力を活かし，メインメモリとして用いるための技術が研究されてきた．PCM が特性として持つ，読み込みと比較して長い書き込みの遅延，および書き込み回数の制限が，これまでメインメモリとして用いられてきた DRAM を置き換えるには不利な点である．これらの短所を克服するための技術が，計算機アーキテクチャの面から研究されてきた [1], [2], [3], [4], [5]．そこでは，オペレーティングシステム（OS）は，PCM への書き込みの検出や移動するページのリマップといった，補助的な役割を果たすのみであった [6], [7]．ほかに，メインメモリになりうる NV メモリとして，書き換え回数やアクセス速度の点で，PCM よりも優れている MRAM [8] や ReRAM などがある．これらのメモリは，容量や実用化の課題が解決されれば，PCM を置き換える存在になる可能性がある．

[1] 筑波大学システム情報系情報工学域
Faculty of Engineering, Information and Systems, University of Tsukuba, Tsukuba, Ibaraki 305–8573, Japan
[a)] shui@cs.tsukuba.ac.jp

一方，NV メモリの中で低速かつアクセス方法に制限のあるフラッシュメモリであっても，HDD と比較した場合，優位な性能を持つ．そのため，NV メモリ上にファイルシステムを構築する研究は以前から行われてきており [9]，組込みシステムでは実用的に用いられている [10]．近年では，高速アクセス可能な PCM 上にファイルシステムを構築する研究 [11], [12] や，PCM を従来のストレージの補助的手段として用いる研究 [13] も行われている．これらは，PCM をストレージと見なしており，メインメモリ同様にアクセスできることを利用してはいるものの，OS やアプリケーションの実行には DRAM メインメモリを用いる前提となっている．

以上のように，NV メモリをメインメモリとして用いる研究，およびストレージデバイスとして用いる研究は，それぞれ独立に行われてきた．メインメモリおよびストレージの両方としても用いることのできる NV メモリを，領域分割し，それぞれをメインメモリ，ストレージと領域を分けて使用することも可能である．しかしながら，メインメモリおよびストレージの両方として用いることのできる NV メモリは，その両方を 1 つに融合できることを意味する．融合することの利点は，メインメモリ，ストレージと領域を分けて使用する必要がなくなり，単一システムで直接使えるメモリ領域が増加する点である．領域を分けて使用する場合，メモリが不足してくると，メインメモリとスワップ領域間のページスワップ[*1]により，メインメモリに空き領域を確保する必要がある．融合することで，ページスワップのオーバヘッドを減少させることができ，性能的にも有効である．メインメモリとストレージの融合は，NV メモリをメインメモリとして用いた場合の可能性の 1 つとして文献 [14], [15] で議論されているが，具体的な実現方法には触れられていない．

本論文は，Linux カーネルを対象とし，NV メモリから構成されるメインメモリとファイルシステムの具体的な融合方法を提案する．そして，Linux をエミュレータ上で実行する評価実験を行い，融合が可能であること，また性能面でも有効であることを示す．本論文は，書き込み回数の制限，アクセス遅延の差は考慮しない．これは，PCM をメインメモリとして用いる研究 [1], [2], [3], [4], [5], [6], [7] により，書き込みバッファを設けることにより，実用的に用いることができるレベルまで，その制限や特性を隠蔽ができることが分かっているからである．また，NV メモリの 1 つである MRAM [8] は DRAM と速度差がないとしていることも，その理由となっている．以下，本論文では，特に指定しない場合，NV メモリはメインメモリとして用いることのできる PCM や MRAM，ReRAM などを指すものとする．

本論文の構成は次のとおりである．2 章では，NV メインメモリとファイルシステムの融合方法を提案し，3 章で Linux における具体的な実現方法について述べる．4 章では実験結果から融合が可能であることを示し，5 章では考察を行う．6 章で関連研究について述べ，7 章でまとめと今後の課題について述べる．

2. NV メインメモリとファイルシステムの融合方法

本章では，NV メインメモリとファイルシステムの融合方法について述べる．まず最初に，対象とするシステム構成を明確にする．次に融合するにあたり必要な要件を定義した後に，提案手法を述べる．本論文では，プロセッサがバイト単位で直接アクセス可能な 1 次記憶装置をメインメモリ，直接アクセスできない 2 次記憶装置をストレージと呼ぶものとする．

2.1 対象とするシステム構成

バイト単位でアクセス可能な NV メモリがシステムに導入された場合，記憶装置は DRAM，NV メモリ，HDD や SSD などのブロックデバイスの組合せから構成されることになる．これらのうち，DRAM はメインメモリ，ブロックデバイスはストレージとしてのみ使用可能であり，NV メモリはメインメモリ，ストレージの両方として使用可能である．大容量のストレージが必要なサーバでは，HDD や SSD などのブロックデバイスが必要となる場合が考えられるが，PC やタブレット，スマートフォンなどのクライアントでは，そのような大容量のストレージは不要である．すなわち，クライアントデバイスでは DRAM および NV メモリからなる構成となると考えられる．NV メモリのみからなる構成も考えられるが，今後の研究において，一時的にしか使用されないデータ，頻繁に読み書きされることが分かっているデータ領域（スタックなど）の処理に用いることも考慮し，本論文では DRAM と NV メモリの両方から構成されるシステムを対象とする．しかしながら，DRAM はメインメモリとしては補助的な位置づけとし，容量的には NV メモリがメインメモリの大部分を占め，DRAM は一部を占めるにとどまるものとする．また，実際のシステムを構築するためには，現状の Linux を含む OS カーネルはある程度の大きさの連続領域をブート時から占有できることを前提としており，またその時点ではファイルシステムからの領域確保はまだできないため，その用途にも DRAM を使用する．

図 1 に，本論文が対象とするシステムの物理アドレス空間を図示する．NV メモリは，DRAM と同様に，物理アドレス空間に配置される．DRAM はメインメモリとしての

[*1] 本論文では，メインメモリ上に空き領域を確保するため，ページサイズ単位で，メインメモリ上のデータをストレージのスワップ領域へ退避させる操作，逆に退避したデータが必要になりスワップ領域からメインメモリ上へデータを復帰させる操作をページスワップと呼ぶ．

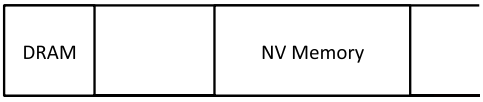


図 1 DRAM と NV メモリから構成される物理アドレス空間
Fig. 1 Physical address space of the target system.

み使用されるが，NV メモリは分割されることなく，メインメモリとファイルシステムの両方に使用される．また NV メモリは，DRAM と同じまたは別のメモリバスを通して，CPU と接続され，CPU から見たアクセスは DRAM と何ら変わるところはない．そのため，NV メモリをストレージとして用いる場合，DRAM 上のラムディスクと同様，CPU から直接 NV メモリにアクセスされる．そのため，HDD や SDD などのブロックデバイスとは異なり，DMA は不要である．

2.2 要件

以下に，NV メモリが分割されることなくメインメモリとファイルシステムの両方に使用できるようにするために，実現すべき要件をまとめる．

(1) NV メモリは一定サイズの領域に分割され管理されるが，メインメモリとしてはページフレーム，ファイルシステムとしてはブロックのどちらにも使用される．
(2) NV メモリの領域はあらかじめ用途が決められることなく管理され，必要に応じてメインメモリやファイルの保存領域として使用される．
(3) ファイルに格納されたプログラムテキストおよびデータは，可能な限りコピーせずに使用される．
(4) NV メモリの領域がメインメモリの一部として使用される場合，その用途にはできる限り制限を加えない．
(5) NV メモリ上に構築されるファイルシステムは，正常な終了時，再起動時に，ファイルシステムとしての一貫性を維持する．

(1)，(2) は，NV メモリがメインメモリとファイルシステムの両方に，あらかじめ用途が区別されることなく使用されることを意味する．(3) は，ファイルの内容がメインメモリとして直接参照されることを意味する．これは XIP（eXecution In Place）と呼ばれ，もともとは ROM 上のプログラムを直接実行するために開発された技術である．Linux では，Ext2 や PRAMFS [16] がバイト単位でアクセス可能なメモリ上に構成されている場合，実現されている．(4) は，NV メモリの領域がメインメモリとして使用できるのは，たとえば XIP によるプログラムテキストおよびデータの直接参照のみに限られることがないようにするための要件である．本論文では，容量的にメインメモリの大部分は NV メモリが占め，DRAM は補助的に一部を占めるだけとしているため，ユーザプロセス中ではヒープや BSS といった，ファイルの一部にはならない領域に対しても，NV メモリを割り当てられる必要がある．(5) は，NV メモリ上に構築されたファイルシステムの一部が，メインメモリとして破壊的に使用されることがないようにするための要件である．OS の異常終了時にも，ファイルシステムとしての一貫性を維持するべきではあるが，これは本論文では対象としない．

2.3 提案手法

本論文は，ファイルシステムを NV メモリの領域管理の基盤として使用する手法を提案する．以下に，2.2 節で述べた用件を実現するための提案手法をまとめる．

- NV メモリ上には，メインメモリとファイルシステムの両方として使用する全領域を管理するため，メインメモリのページサイズと同一にしたブロックサイズを持ち，また XIP の機能を提供するファイルシステムを構築する．
- NV メモリの空き領域は DRAM とは別管理とし，NV メモリからの領域確保要求はファイルシステムに対して行い，また NV メモリから確保され解放可能となった領域は，基本的にはファイルシステムに空き領域として登録する．
- システムの正常な終了時，再起動時には，NV メモリから確保された領域がすべてファイルシステムに空き領域として登録されるよう，解放処理を行う．

本手法は，ファイルシステムのデータ領域を，メインメモリとしても使用する．そのために，ブロックサイズはメインメモリのページサイズと同一に設定する．サイズが異なっていても対応は不可能ではないが，処理が煩雑化する．メモリ割当て・解放処理は効率的である必要があるため，煩雑な処理は避けるため，メインメモリのページサイズと同一のブロックサイズを用いる．

NV メモリの空き領域を DRAM とは別管理とするのは，確保・解放をファイルシステムに対し効率的に行うためである．現在の汎用 OS カーネルは，メインメモリ上にできる限りストレージのデータをキャッシュし，データ転送を効率化するために，複雑な処理を行っている．単一のメモリ領域がメインメモリとストレージの両方の役割を担う場合，そのような複雑なキャッシュ機構は不要になる．そのため，NV メモリの空き領域は別に管理する．

システムの正常な終了時，再起動時に，NV メモリから確保された領域がすべてファイルシステムに空き領域として登録されていなければ，次の起動時のファイルシステムにはファイルからの参照がない領域が使用されたままの状態で残ってしまうことになり，一貫性のない状態となってしまう．そのようなことのないよう，解放処理を行う．

次に，対象とするシステム構成に関連した割当て方針について述べる．容量的にメインメモリの大部分は NV メモリが占め，DRAM は補助的に一部を占めるシステムを対象

としている．そのため，NV メモリから優先的に割り当てる．NV メモリから優先的に割り当てることで，DRAM に空き領域を残しておくことができる．したがって，NV メモリ上のファイルシステムがファイルでフルの状態になったとしても，まだ DRAM からメモリを割り当てられる可能性が残される．DRAM から優先的に割り当ててしまうと，DRAM に空きがなくなった状態で，NV メモリから割当てが始まることになり，NV メモリがファイルでフルの状態になってしまうと，どこからもメモリを割り当てることができなくなってしまう．したがって，NV メモリから優先的に割り当てることにメリットがあるといえる．

3. 実装

2.3 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，エミュレータを用いたターゲット環境の構築について述べ，次に提案手法の実装について述べる．

3.1 ターゲット環境の構築

NV メモリをメインメモリとして持つシステムは，一般的に入手可能な状態ではない．そこで，NV メモリをメインメモリとして持つシステムをエミュレーションし，ターゲット環境とする．以下に，NV メインメモリをエミュレーションする環境と，その上での Linux のブートについて述べる．

3.1.1 NV メインメモリのエミュレーション

QEMU システムエミュレータを変更することで，NV メインメモリをエミュレーションする環境を作成し，ターゲット環境として用いる．NV メモリをメインメモリとして用いる場合，広い物理アドレス空間を持つ 64 bit 環境である方が望ましいため，x86_64 プロセッサをエミュレーションする QEMU を用いる．NV メモリをエミュレーションするために，NV メモリの内容を保存するファイルを用いる．そのファイルの内容を，QEMU がエミュレートするプロセッサ環境の物理メモリにマップするための変更を加えた．たとえば，以下のようにオプションを指定すると，128 MB の DRAM の領域が確保され，またファイル nvmemory.img のすべてが NV メモリとして物理アドレス 0x100000000 からの領域にマップされた状態で，QEMU が起動される．

```
% qemu -m 128 -nvmemory
  file=nvmemory.img,physaddr=0x100000000
```

DRAM 領域の大きさは BIOS に渡され，Linux カーネルはブート時にそのメモリマップ情報を受け取る．一方，NV メモリの情報は BIOS には渡されず，Linux カーネルもその情報を BIOS からは受け取らない．これは，NV メモリは DRAM とは別管理とするためである．

3.1.2 ターゲット環境における Linux のブート

上記のターゲット環境上での Linux のブートについて述べる．NV メモリをメインメモリとして用いることで，Linux のブートも，ブロックデバイスのストレージ上に置かれた Linux カーネルを DRAM 上にコピー，展開してブートする従来の方式とは異なったブート方式も考えられる．しかしながら，現状の Linux カーネルは，メインメモリ上にある程度の大きさの連続領域を，ブート時から占有できることを前提としている．カーネルの初期化プロセスでは，その連続領域上に，カーネルの実行に必要な，すなわちメインメモリを含む資源管理のためのデータ構造体を置いていく．そのため，従来の方式とは異なったブート方式を導入するには，カーネルの初期化プロセスを大きく変更しなければならない可能性がある．

本論文は Linux カーネルを対象とし，2.3 節で述べた提案手法を実装，評価することを目的としているため，ブート方式としては従来方式を用いる．すなわち，Linux カーネルを DRAM 上にコピー，展開してブートする．ターゲット環境として用いる QEMU は，ブートローダとしての機能も持っているため，これを利用する．`-kernel` オプションでブートする Linux カーネルの bzImage を指定することで，このファイルの内容を QEMU がエミュレーションする DRAM 上にコピー，展開し，そのカーネルをブートすることができる．

実機ではブートローダが必要となるが，現状のブロックデバイスのストレージの先頭にブートローダが置かれている場合と同様に考えることで，ブート可能である．すなわち，BIOS などのファームウェアから，NV メモリの先頭に置かれているブートローダを起動する．そのブートローダが NV メモリ上のファイルシステムを読み，ファイルシステム上のカーネルを DRAM 上にコピー，展開し，実行を開始すればよい．

3.2 提案手法の実装

次に，提案手法の Linux カーネルにおける実装について述べる．実装は，NV メモリ上に構築するファイルシステムの選択，NV メモリの空き領域管理と領域確保・解放処理，Linux カーネルにおける領域確保・解放処理への変更，そしてシステム正常終了時の解放処理からなる．**図 2** に，仮想記憶システム，メモリアロケータ，ファイルシステムの関係を図示する．実線は実装済みの関係であり，点線が提案手法の実装に必要な部分である．仮想記憶システムは，メモリアロケータを通して，DRAM のページを利用する．また，XIP を用いる場合，仮想記憶システムは，ファイルシステムから直接 NV メモリのページを利用する．しかし，メモリアロケータが NV メモリのページを使用するためのパスは用意されておらず，提案手法を実現するためには，この部分を実装する必要がある．

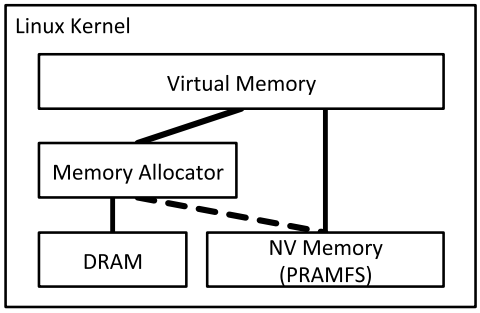


図 2 仮想記憶システム，メモリアロケータ，ファイルシステムの関係

Fig. 2 Relationship between a memory allocator and a file system.

3.2.1 NV メモリ上に構築するファイルシステム

提案手法を実装するには，基盤となるファイルシステムが必要となる．メインメモリとファイルシステムの融合に適したファイルシステムを，別途開発することも考えられるが，それは本論文の主旨とは外れる．そのため，既存のファイルシステムを利用し，必要に応じて変更することとした．

既存のファイルシステムとしては，XIP をサポートし，読み書き可能なファイルシステムとして Ext2，PRAMFS [16] が候補となる．ほかに候補として検討したファイルシステムはいくつかあるが，以下の理由で候補とならなかった．XIP をサポートするファイルシステムとして AXFS，CRAMFS があるが，書き込みをサポートしない．メモリ上に構築されるファイルシステムとして ramfs があるが，その管理構造体は現在実行中のカーネルの関数ポインタを含むため，異なるカーネルを用いて再起動することができない．SCMFS [12] は NV メモリを対象として開発されたが，オープンソースではないため使用できない．

Ext2 は，ブロックデバイスのストレージ上に構築されることを前提としており，その特性を考慮することで高い性能を達成するための工夫がしてある．そのため，構造，機構ともに複雑になっている．また，NV メモリ上に構築するためには，NV メモリのラムディスクドライバが必要となる．一方，PRAMFS（Persistent and protected RAM File System）は，その名のとおり，NV メモリ上に構築するために開発されたファイルシステムであり，ブロックデバイスは考慮されておらず，単純な構造，機構となっている．

提案手法を実装し，十分な性能を達成するには，より単純である方が有利であると考えられるため，PRAMFS を基盤として開発することにした．提案手法を実装するために，メインメモリのページサイズ 4 KB と同一のブロックサイズでファイルシステムを構築し，XIP を用いるように設定を行う．

3.2.2 空き領域管理と領域確保・解放

ファイルシステムを NV メモリの領域管理の基盤として使用するため，ファイルシステムが空き領域を管理し，領域の確保および解放はファイルシステムを通して行う．PRAMFS では，1 ブロックの確保および解放を行う関数として `pram_new_block()`，`pram_free_block()` が提供されている．1 ブロック単位で確保・解放処理を行う場合は，これらの関数を呼び出せばよい．

当初の実装でこれらの関数を用いて領域の確保および解放を行い，簡単な性能評価実験を行ったところ，単純なブロック単位でのファイルシステムから領域の割当てでは，性能への影響が大きいことが分かった．そこで，ファイルシステムからは一定数の連続したブロックを一括確保する．そして，要求されている数のブロックを要求元に返し，残りのブロックはファイルシステムからは確保済みの空きブロックを管理するリスト（フリーリスト）に追加し，以降の要求のためにキャッシュする方式とした．この方式では，領域確保と解放は次のように行う．領域確保要求に対し，フリーリスト中に空き領域があればそれを返す．なければ，一括領域確保を行った後に，空き領域を返す．一括領域確保には，PRAMFS では，空き領域はスーパーブロックのビットマップで管理されていることろ利用する．これを `unsigned long` の単位（64 bit）で検索し，空き領域が見つかったら，64 ブロックをフリーリストに加えることにより，一括確保する．解放時には，一定数（現在の実装では 128 ブロック）以上のブロックがフリーリストにある場合は，ファイルシステムに解放し，そうでなければフリーリストに追加する．一括領域確保とフリーリストに空き領域をキャッシュする効果は，4.2，4.3 節の実験結果で示す．

領域確保要求では，連続する複数ページが要求される場合がある．連続ページは，開始アドレスが要求サイズにアラインされている必要がある．たとえば，拡張ページサイズ[*2]は，x86_64 アーキテクチャでは 2 MB のページサイズを利用可能にするが，この場合 2 MB のアドレスにアラインされた連続領域が必要になる．このような連続ページの領域確保要求に対して，1 ページ確保の場合と同様に，一括確保とキャッシュを行うと，ファイルシステムから確保済みとなる領域が多くなりすぎる可能性がある．そのため，連続ページ要求時にはファイルシステムの空き領域を検索し，解放時に少数の領域のみをキャッシュすることとした．

3.2.3 Linux カーネルにおける領域確保・解放処理への変更

Linux カーネルのメモリアロケータが，NV メモリの領域を確保・解放できるようにするための変更点について述べる．

[*2] Linux カーネルのコンパイル時に CONFIG_TRANSPARENT_HUGEPAGE を設定することで，利用可能になる．

メモリアロケータは，__alloc_pages_nodemask()，free_hot_cold_page() においてページの確保，解放を行う．2.3 節で述べたとおり，提案手法では NV メモリを優先的に割り当てる方針を採用する．そのため，__alloc_pages_nodemask() では，DRAM の前に，NV メモリからの確保を試みる．NV メモリ領域を確保できた場合は，DRAM 領域の確保は行わない．解放時には，NV メモリは DRAM とは別に解放処理を行う必要がある．そのため，NV メモリ領域の確保時に，そのページの属性として NVmemory を追加する．そして，解放するページにこの属性が付いている場合は，NV メモリの領域解放を行う関数を呼び出す．

空きページを確保するために __alloc_pages_nodemask() が呼び出されるとき，確保するページが満たす必要のある属性が引数 gfp_mask に指定される．カーネル内の割当てに使用されるページは GFP_KERNEL となり，この場合も NV メモリからの割当ては可能である．しかし，次に述べるシステム正常終了時の解放処理が煩雑になるため，現状では，NV メモリはユーザプロセスへ割当てを行うようにしている．そこで，gfp_mask が GFP_HIGHUSER_MOVABLE の場合，NV メモリからの割当てを行っている．この場合も，XIP でマップされる以外の，ヒープや BSS 領域の割当てに NV メモリが使用される．

連続する複数ページの確保・解放も，上述の 2 つの関数を通して行われる．したがって，上述の変更により，NV メモリに対して確保・解放要求が行われ，3.2.2 項で述べたように処理される．

3.2.4 システム正常終了時の解放処理

システム正常終了時に，ファイルシステムの一貫性を保つため，NV メモリから確保された領域が，使用されたままの状態で残らないようにするための処理について述べる．解放する必要があるのは，次のページである．

- NV メモリから割り当てられ使用中のページ
- 割当てのためにキャッシュされているページ

システムを正常終了するためには，一般的に，すべてのプロセスにシグナルを送り終了処理を行い，ファイルシステムをアンマウントまたはリードオンリーに再マウントした後に，カーネルの終了処理に入る．リードオンリーに再マウントするときに，NV メモリからの領域確保を停止することで，NV メモリから割り当てられ使用中のページはなくなることになる．リードオンリーに再マウント後は，NV メモリからの領域確保は行われないため，このときに，割当てのためにキャッシュされているページも解放する．

4. 評価

本章では，まず NV メモリが分割されることなく，メインメモリとファイルシステムの両方に使用できることを示す．次に，提案手法の実装の性能面への影響について調査した実験結果を示す．実験環境には，3.1 節で述べた，QEMU によるエミュレーションを行うターゲット環境を用いる．DRAM として 128 MB，NV メモリは 4 GB を割り当てる．実験に用いた QEMU のバージョンは 1.0.1，Linux カーネルは 3.4 である．

エミュレータでは，タイマデバイスによる時間の計測は十分に信用できない．QEMU は，-icount 0 オプションを付けて実行することで，1 命令実行ごとに TSC（Time Stamp Counter）が 1 ずつ増えるため，実行命令数を計測することができる．そこで，性能測定では，実行時間の代わりに，実行命令数を性能測定の指標として用いる．TSC の値は RDTSC 命令により取得することができる．

4.1 NV メモリからの割当て・解放

NV メモリが，分割されることなくメインメモリとファイルシステムの両方に使用できることを示すため，メインメモリとファイルへの領域割当ておよび解放の実験を行った．提案手法ではファイルシステムが空き領域を管理しているため，プロセスがメモリ領域を確保することで，ファイルシステムの空きスペースが減少し，また解放することで，ファイルシステムの空きスペースが増加することになる．同様に，ファイルを作成，消去時にも，ファイルシステムの空きスペースが減少，増加する．df コマンドを用いると，領域確保および解放によるファイルシステムの使用スペースの増加，減少を調べることができる．malloc() によるメインメモリへの割当て，dd コマンドによるファイルの作成を行うプログラムを作成した．**図 3** に，その実行結果をスクリーンダンプにより示す．

スクリーンダンプの最初の行は，PRAMFS をルートファイルシステムとしてマウントし，Linux がブートしていることを示している．ルートファイルシステム上には，ベンチマークプログラムのほか，シェルなどのコマンドとして busybox がインストールされている．そのため，ブートすると，まず busybox のシェルが起動する．そこで，メインメモリとファイルへの領域割当ておよび解放を行うプログラムを実行している．プログラムの引数に 16 を指定することで，1 MB から倍に増やしながら，16 MB までの領域割当てを行う．メインメモリへの割当て，ファイルの作成を行うたびに，ファイルシステムの使用スペースが，その分だけ増加していくことが分かる．最後に，割り当てた領域の開放，ファイルの消去を行うことで，ファイルシステムの使用スペースは減少している．メインメモリからの解放時には一定量をキャッシュとして残すため，完全に最初の値には戻らない．

2.3 節で述べたとおり，NV メモリの空き領域は DRAM とは別に，ファイルシステムにより管理される．そのため，使用および空きスペースの取得も，NV メモリについ

```
[    1.092768] VFS: Mounted root (pramfs filesystem) on device 0:10.
[    1.093687] devtmpfs: mounted
[    1.100968] Freeing unused kernel memory: 288k freed
[    1.112993] Write protecting the kernel read-only data: 4096k
[    1.117175] Freeing unused kernel memory: 80k freed
[    1.171474] Freeing unused kernel memory: 1472k freed
Mounting filesystems.

Please press Enter to activate this console. [    1.832023] Refined TSC
ation: 2526.978 MHz.
[    1.832511] Switching to clocksource tsc

~ # /bench/mem_alloc+dd+df-root 16
max alloc_size: 16 MB, use malloc
No allocated memory.
Filesystem                Size      Used Available Use% Mounted on
mtd                       3.8G     14.0M      3.8G   0% /
1 MB allocated and touched.
mtd                       3.8G     15.3M      3.8G   0% /
1 MB file created.
mtd                       3.8G     16.4M      3.8G   0% /
2 MB allocated and touched.
mtd                       3.8G     16.2M      3.8G   0% /
2 MB file created.
mtd                       3.8G     18.4M      3.8G   0% /
4 MB allocated and touched.
mtd                       3.8G     18.2M      3.8G   0% /
4 MB file created.
mtd                       3.8G     22.4M      3.8G   1% /
8 MB allocated and touched.
mtd                       3.8G     22.2M      3.8G   1% /
8 MB file created.
mtd                       3.8G     30.4M      3.8G   1% /
16 MB allocated and touched.
mtd                       3.8G     30.2M      3.8G   1% /
16 MB file created.
mtd                       3.8G     46.5M      3.8G   1% /
No allocated memory.
mtd                       3.8G     14.4M      3.8G   0% /
~ #
```

図 3 NV メモリ割当て後のディスク使用量
Fig. 3 Disk usage after NV memory allocation.

```
~ # /bench/mem_alloc+df+free-root 16
max alloc_size: 16 MB, use malloc
No allocated memory.
Filesystem                Size      Used Available Use% Mounted on
mtd                       3.8G     14.9M      3.8G   0% /
             total         used         free       shared      buffers
Mem:        124644        68532        56112            0            0
-/+ buffers:              68532        56112
Swap:            0            0            0
16 MB allocated and touched.
mtd                       3.8G     31.1M      3.8G   1% /
             total         used         free       shared      buffers
Mem:        124644        68540        56104            0            0
-/+ buffers:              68540        56104
Swap:            0            0            0
~ #
```

図 4 NV メモリ割当て後の DRAM 使用量
Fig. 4 DRAM usage after NV memory allocation.

てはファイルシステムより行うことになる．`malloc()` を用いて領域割当てを行い，領域割当て前後に，`df` コマンドを用いて NV メモリの使用スペース，`free` コマンドを用いて DRAM の使用スペースを表示するプログラムを作成した．**図 4** に，その実行結果をスクリーンダンプにより示す．ユーザプロセスへのメモリ割当ては NV メモリから行われるため，メモリ割当て後，`df` コマンドが表示するファイルシステムの使用スペースは増加している．一方，DRAM の使用スペースは，領域割当て前後でほとんど変化していない．これは，DRAM はカーネル内への割当てにのみ使用されるからである．

4.2 OS 実行性能への影響

2 つめの実験では，提案手法の OS 実行性能への影響を調べる．そのために，LMbench3 ベンチマークスイートに含まれる fork+exit，fork+execve，fork+sh を実行した結果を，**表 1** に示す．表中の値は，RDTSC により取得した実行命令数を 1000 で割った値である．表中の DRAM は，NV メモリを用いない場合の結果である．また，NV メモリ（キャッシュ）は，3.2.2 項で述べたファイルシステムか

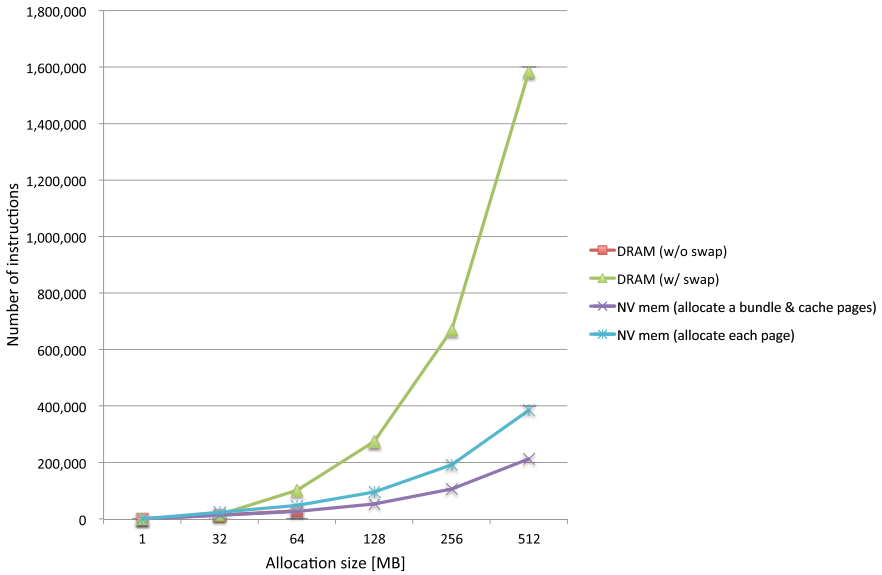


図 5 ページスワップ機構を用いた場合との性能比較
Fig. 5 Performance comparison with the case that uses the paging mechanism.

表 1 マイクロベンチマークの結果
Table 1 Micro benchmark results.

ベンチマーク	fork+exit	fork+execve	fork+sh
DRAM	72	162	405
NV メモリ（キャッシュ）	72	162	398
NV メモリ（ページごと）	91	200	499

ら一括領域確保を行い，フリーリストにキャッシュする方式を用いた場合の結果，NV メモリ（ページごと）は，この方式を用いずに，ファイルシステムからブロック単位での領域確保を行った場合の結果である．fork+exit は，fork システムコールによりプロセスを作成し終了，fork+execve は，さらに execve システムコールにより別プログラムを起動，fork+sh は，シェルから別プログラムを起動するようにしたものである．fork システムコールによるプロセス作成，execve システムコールによるプログラムの起動は，カーネル内でのメモリ割当てをともなうため，これらのベンチマークを選択した．

実験結果から，メインメモリに割り当てるための領域をフリーリストにキャッシュすることで，23〜26%の性能向上が実現でき，DRAM の場合と同等の性能を実現することができていることが分かる．逆に，ファイルシステムをそのまま空き領域の管理機構として用いるだけでは効率が悪く，OS 実行性能への影響があることが分かる．少数の領域をキャッシュすることで，領域割当て時のコストを削減し，ファイルシステムを空き領域の管理機構とするオーバヘッドを隠蔽することができる．

4.3 ページスワップ機構を用いた場合との性能比較

3 つめの実験では，ページスワップ機構との性能比較を行う．NV メモリをファイルシステムとしてのみ使用する場合には，ファイルシステム上にスワップファイルを置き，メインメモリとスワップファイル間でページスワップを行うことにより，仮想的にメインメモリの容量を増やすことができる．一方，NV メモリがメインメモリとファイルシステムの両方に使用できる場合，メモリ消費量の多いプログラムの実行時にも，容量の大きい NV メモリから直接メモリを割り当てて実行できる．そこで，ページスワップを行う場合と，NV メモリから直接メモリを割り当てることのできる場合で，大量のメモリアクセスする場合に，性能差が生じるかどうかを計測する実験を行った．

実験には，引数で指定された容量のメモリ領域を確保し，その領域の各ページの先頭部分に対し書き込みを行うプログラムを作成し，実行した．測定結果には，メモリ割当てコストは含まれるが，解放処理のコストは含まれない．実行したのは，DRAM のみ，DRAM とスワップファイルの組合せ，NV メモリ（キャッシュ），NV メモリ（ページごと）の場合である．その結果を図 5 に示す．横軸は割当てメモリサイズ，縦軸は RDTSC により取得した実行命令数を 1000 で割った値である．

DRAM のみと NV メモリ（キャッシュ）は，ほぼ同等の性能であるが，DRAM のみの場合はメモリが足りないため 128 MB 以上の割当てができない．DRAM とスワップファイルの組合せは 512 MB までの割当てができているが，NV メモリ（キャッシュ）よりも約 7.4 倍のコストがかかっている．また，NV メモリ（ページごと）は NV メモリ（キャッシュ）よりも約 1.8 倍のコストがかかっている．したがって，NV メモリをメインメモリとファイルシステムの両方に使用することで，ページスワップが不要になり，DRAM のみの場合と同等の性能を維持することができることが分かった．

4.4 ファイルの読み出し・書き込みコスト

最後の実験では，NV メモリに構築された PRAMFS 上のファイルの読み出し・書き込み性能の比較を行う．Linux では，ファイルのデータにアクセスするために，2 種類の方法が提供されている．1 つは，通常の `read`，`write` システムコールを用いる方法である．もう 1 つは，`mmap` システムコールを用いてユーザプロセスの仮想アドレス空間にマップする方法である．PRAMFS は，NV メモリ上に構築するために開発されたファイルシステムであり，ファイルのデータにカーネルが直接メモリアクセスできることを前提としている．そのため，`read`，`write` システムコールを用いる場合でも，バッファリングを行うページキャッシュは用いられず，ファイルシステムへ直接読み書きが行われる．また，`mmap` システムコールを用いる場合，ユーザプロセスはファイルのデータに直接メモリアクセス可能である．実験には，それぞれの方法で，指定されたサイズのファイルに対し，読み出し・書き込みを行うプログラムを作成，実行した．読み出しの計測には，ファイルを作成し書き込みを行った後に，そのファイルを読み出す場合（hot）と，ページキャッシュのフラッシュ[*3]またはページのアンマップ操作後に読み出す場合（cold）を計測した．書き込みの計測には，ファイルを作成し指定サイズの書き込みを行った後に，もう 1 度書き込みを行う場合（hot）と，ファイル作成後に初めて書き込みを行う場合（cold）を計測した．その結果を**図 6**，**図 7** に示す．横軸は割当てファイルサイズ，縦軸は RDTSC により取得した実行命令数を 1000 で割った値である．

読み出しの場合，最も高速なのが `mmap` により仮想アドレス空間にマップされ，すでにページテーブルが構築されている場合（hot），最も低速なのがページテーブルが構築されていない場合（cold）となった．`read` システムコールによるアクセスは，その中間となり，ページキャッシュは用いられないため，hot と cold の場合に差はなかった．`read` システムコールは，カーネルで処理されるため，カーネルへ実行を移すコストがかかり，それが差となっている．ページフォルトも同様にカーネルで処理されるが，その処理コストは，システムコールよりも大きいことが分かる．

書き込みの場合，hot と cold の場合で，大きく差が出る結果となった．どちらの場合も，`mmap` システムコールでマップしての書き込みの方が高速であった．`write` システムコールによる書き込みにおける hot と cold の差は，ブロック割当てのコストであると考えられる．`mmap` システムコールでマップしての書き込みにおける hot と cold の差は，ブロック割当てのコストに，ページテーブルが構築されていないことに起因するページフォルトのコストが加わる．

[*3] PRAMFS はページキャッシュを用いないが，念のためフラッシュの操作を行った．

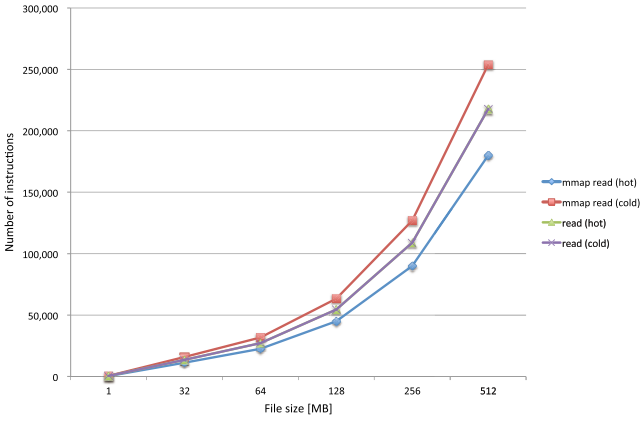


図 6 ファイル読み出し性能の比較
Fig. 6 Performance comparison of file reading.

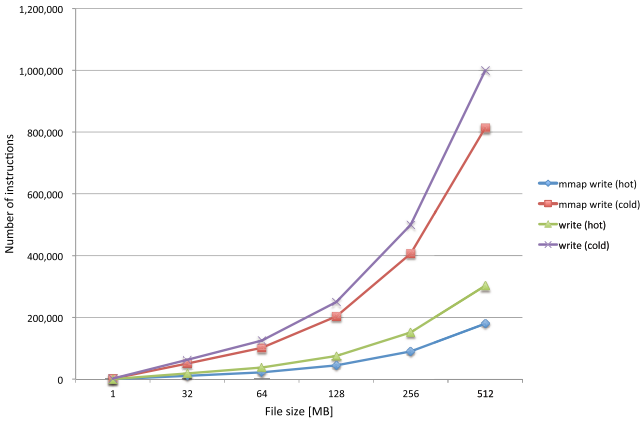


図 7 ファイル書き込み性能の比較
Fig. 7 Performance comparison of file writing.

実験より，PRAMFS の領域割当てのコストが大きいことが分かった．3.2.2 項では，メインメモリへのページ割当てコストが大きいことから，1 ページの割当てについては一括確保とキャッシュの仕組みを導入した．メインメモリへの連続ページ確保の場合も含め，効率の良い領域割当て手法の導入が必要である．

5. 考察

提案手法における，設計・利用方針，異常終了時の対処方法，適用領域と使用可能メモリ容量の表示について考察を行う．

5.1 NV メモリ領域管理の設計方針

本論文で述べた提案手法は，ファイルシステムを NV メモリの領域管理の基盤として使用する．この設計方針に至った理由として，ファイルシステムは，非揮発性のストレージを前提として，長期にわたり安定したデータの保存ができるように設計されていることがあげられる．そのため，ストレージの領域管理を行うにあたり，OS 終了時から次の起動にデータが引き継がれる，領域管理構造体がカーネルに直接依存しない，OS 異常終了後の起動時に行う一

貫性の検査・修復手法が確立されている，といった利点がある．これらは，実装に用いたPRAMFSとは独立した，ファイルシステムに一般的な特徴である．一方，メインメモリ管理は，揮発性メモリが前提となっており，その管理はOSの起動から終了までの間に限定される．そのため，ファイルシステムとは逆に，実行効率を高めるため管理構造体に関数ポインタを含めるなど，実行中のカーネルに依存した構造となっており，またOSの終了・再起動間でのデータの引き継ぎは考慮されない．これらの理由から，ファイルシステムをNVメモリの領域管理の基盤として使用する方が，現状では有利であると考え，その設計方針を採用するに至った．

本論文の提案手法では採用しなかったが，メインメモリ管理をNVメモリの領域管理の基盤として使用することも可能である．たとえば，ファイルシステムに最初は必要最小限のNVメモリ領域を割り当て，必要に応じてリサイズする方法が考えられる．BTRFSやZFSなど近年開発されたファイルシステムでは，オンラインリサイズが可能である．しかしながら，バイト単位でアクセス可能であることを生かすことのできるXIPはサポートしない．Ext2はXIPをサポートするが，オンラインではファイルシステムの伸張のみ可能である．理論的には，縮小のサポートも不可能ではないが，XIPでマップされたメモリの移動が必要な場合もあり，そのサポートは用意ではないと考えられる．また，ファイルシステムに割り当てられた領域を，OSの終了・再起動間で引き継ぐための仕組みも必要となる．

5.2 NVメモリ領域の利用方針

メインメモリやファイルシステム上で，領域確保と解放操作を繰り返すと，断片化を引き起こす．提案手法を導入した場合，NVメモリ上のファイルシステムに断片化が発生する．発生すると，断片化により連続領域が不足することによる問題が生じる．断片化のNVメモリへの影響はHDDとは異なり，アドレスが分散したブロックへのアクセス時間が長くなることはないため，ファイルを構成するブロックが分散することによる問題は生じない．また，メインメモリとして用いる場合も，物理アドレスが分散したブロックは，連続した仮想アドレスにマップして使用できるため，分散することによる問題は生じない．しかしながら，いくつかの点で性能面への影響が生じる．まず，空きブロックの探索時間が増加し，領域割当てに遅延が生じる．領域割当ての遅延は，4.2節，4.4節に示したように，性能に影響する．提案手法では，連続した64ブロックを一括確保する．断片化により，このような連続した空きブロックが減少すると，多くのブロックについて状態を確認する必要が生じ，探索時間が増加してしまう．最悪，連続した空きブロックを確保することができず，1ブロックずつの確保が必要になる状態になることも考えられる．ファイルの書き込み時にブロックを確保する際にも，探索時間が増加してしまう．次に，拡張ページサイズの確保失敗が起こりうる．Linuxは2MBの拡張ページサイズをサポートする．拡張ページサイズを用いることで，大容量メモリを必要とする際には，TLBへの負荷が減少し，性能向上を見込むことができる．断片化により，2MBの連続ブロックが確保できなくなると，拡張ページサイズも使用できなくなり，性能が低下する可能性がある．したがって，断片化状態は放置すべきではない．

断片化は，原因となっているブロックを移動することで，解消することができる．HDDでは，連続してアクセスする可能性が高いブロックが位置的に分散するとアクセス時間が長くなるため，ファイルを構成するブロックを連続ブロックに構成し直す操作を行う．NVメモリでは，しかしながら，分散したブロックへのアクセス時間が長くなることはないため，そのような複雑なファイルの再構成は必要としない．したがって，断片化の原因となっているブロックを，使用中でないものについて，単純にNVメモリの先頭に近い部分に詰めていくような操作を，システムがアイドル時に行えばよい．それでも断片化が解消されないような状態の場合，OSの再起動が可能であれば，NVメモリ上のファイルシステムをマウントする前では使用中のブロックはないため，すべてのブロックを移動し，断片化を解消できる．

5.3 OS異常終了時の対処方法

OS異常終了時には，NVメモリ上のファイルシステムも，一貫性が失われた状態になる可能性がある．この点に関しては，通常のファイルシステムと何ら変わるところはない．さらに，メインメモリとしての領域割当てが行われたブロックに関しては，異常終了時には解放されず，その後のOS起動時にはメモリリーク同様の状態として残ってしまう．メインメモリとして確保された領域は，正常終了時にはすべて解放されることが前提となっている．そのため，これらの領域の解放が必要となる．

このような，ファイルから参照されないが割り当てた状態になっている領域の解放処理は，ファイルシステムの修復処理の1つに含まれる．したがって，OS起動時にNVメモリ上のファイルシステムをマウントするとき，ファイルシステムのスーパブロックのフラグから正常終了しなかったことが分かった場合，ファイルシステムの検査・修復を行うコマンド（fsck）を起動すればよい．本論文で述べた実装では，NVメモリ上のファイルシステムをマウントした時点から，NVメモリからメインメモリとしての領域割当てが始まるため，マウント前にfsckを起動する必要がある．Linuxでは初期ラムディスクinitrdを用いて起動した場合，initrdが最初にマウントするファイルシステムとなる．initrd上にfsckコマンドを配置し，必要に応じて

NV メモリ上のファイルシステムの一貫性を回復するための修復処理を行ってからマウントすることで，OS 起動時に一貫性を保つことが可能となる．

5.4 提案手法の適用領域・使用可能メモリ容量の表示

提案手法は，バイト単位アクセス可能な NV メモリを前提とし，NV メモリをメインメモリおよびストレージの両方に使用可能にすることで，メインメモリの大容量化をもたらす．一方で NV メモリは，従来の一般的なストレージデバイスである HDD と比較して高価であるため，ストレージの大容量化には適さない．そのため，NV メモリをストレージとして使用することが機能的に求められる，ノート PC やタブレット，スマートフォンなどのモバイル系のクライアントデバイスが，提案手法の適用領域として第 1 に考えられる．これらのデバイスでは，近年のクラウドコンピューティング環境の広まりもあり，手元に大容量ストレージを持つ必要性が薄れてきている．一方，アプリケーションの高度化により，メインメモリは大容量化が必要とされてきている．そのため，提案手法により利便性を上げることができる．また，あらかじめ決まったプログラムを搭載・動作させる組込みシステムでも，メインメモリとストレージの一元化により，管理面での利便性が上がると考えられる．

これら主な適用領域として考えられるクライアントデバイスでは，使用可能メモリ容量を表示するためのアプリケーションが提供されるのが一般的であり，そのアプリケーションが NV メモリと DRAM の両方の空き容量を統合して表示すれば，メモリ領域が分離されていたとしても問題はない．また，組込みシステムでは一般ユーザが使用可能メモリ容量を気にする必要はなく，その表示手段が提供されることも考えにくい．表示する必要がある場合も，クライアントデバイスと同様，アプリケーション側で対応可能である．

6. 関連研究

メインメモリとストレージの融合について触れている論文として，文献 [14], [15] がある．文献 [14] は，NV メモリをメインメモリとして用いた場合の様々な可能性について議論した論文であり，メインメモリとストレージの融合も可能性の 1 つとして取りあげられている．文献 [15] は，メインメモリとストレージを融合するうえでの方針と有効性について議論している．どちらも，メインメモリとストレージを融合する具体的な手法については触れられておらず，実際に Linux で実現しその有効性を評価した本論文とは異なっている．

その他の研究では，NV メモリをメインメモリとして用いる場合，ファイルシステムとして用いる場合，仮想記憶システムのページスワップデバイスとして用いる場合とで，別個に研究が行われてきた．NV メモリをメインメモリとして計算機アーキテクチャに組み込むための研究の多くでは，基本的に OS は介在しない [1], [2], [3], [4], [5], [17]．文献 [6], [7] は，同様に計算機アーキテクチャの面からの研究であるが，耐久性向上のために，ページ単位でデータを DRAM と NV メモリ間で移動させるために OS を用いている．しかしながら，NV メモリはメインメモリとしての利用のみ考慮されている．NV メモリ上にファイルシステムを構築する研究は古くからあり [9]，組込みシステムではすでに実用的に用いられている [10]．近年では，バイト単位で直接アクセス可能であることを利用して，高速なファイルシステムを構築する研究 [11], [12] や，NV メモリを従来のストレージの補助的手段として用いる研究 [13] が行われているが，これらにおいては NV メモリはファイルシステムとしての利用にとどまっている．また，NV メモリをページスワップデバイスとして活用することで，ページスワップを高速化する研究も行われている [18], [19], [20]．これらの研究では，NV メモリはブロックデバイスとして利用しており，メインメモリの一部として用いることはできない．

Multics [21] は論理的な単一レベルストレージを提供したシステムであるが，実装レベルではメインメモリとストレージは分離されており，本論文で実現したメインメモリとストレージの融合とは異なっている．

7. まとめ

近年，不揮発性の non-volatile（NV）メモリの性能向上が著しく，高速化，大容量化，低価格化が進んでいることから，それらをメインメモリとして用いる研究，またストレージデバイスとして用いる研究が，それぞれ別個に行われてきた．しかしながら，メインメモリおよびストレージの両方としても用いることのできる NV メモリは，その両方を融合できることを意味する．本論文は Linux カーネルを対象とし，NV メモリから構成されるメインメモリとファイルシステムの具体的な融合方法を提案した．そして，Linux をエミュレータ上で実行する評価実験を行い，融合が可能であること，また性能面でも，ページスワップが不要になり，DRAM のみの場合と同等の性能を維持することができることが分かった．

メインメモリおよびストレージの融合は，様々な可能性をもたらすと考えられる．また，DRAM と NV メモリのアクセス遅延の差や書き込み回数の制限を考慮したシステムを，実際に OS が動作するシステムに組み込むことも必要である．これらは今後の課題である．

謝辞 本研究は，（株）フィックスターズ三木聡氏との議論から着想を得た．ここに感謝の意を表します．

参考文献

[1] Lee, B.C., Ipek, E., Mutlu, O. and Burger, D.: Architecting phase change memory as a scalable dram alternative, *Proc. 36th Annual International Symposium on Computer Architecture* (*ISCA '09*), pp.2–13 (2009).

[2] Qureshi, M.K., Srinivasan, V. and Rivers, J.A.: Scalable high performance main memory system using phase-change memory technology, *Proc. 36th Annual International Symposium on Computer Architecture* (*ISCA '09*), pp.24–33 (2009).

[3] Zhou, P., Zhao, B., Yang, J. and Zhang, Y.: A durable and energy efficient main memory using phase change memory technology, *Proc. 36th Annual International Symposium on Computer Architecture* (*ISCA '09*), pp.14–23 (2009).

[4] Qureshi, M.K., Franceschini, M.M. and Lastras-Montano, L.A.: Improving read performance of Phase Change Memories via Write Cancellation and Write Pausing, *Proc. 2010 IEEE 16th International Symposium on High Performance Computer Architecture* (*HPCA*), pp.1–11 (2010).

[5] Ramos, L.E., Gorbatov, E. and Bianchini, R.: Page placement in hybrid memory systems, *Proc. International Conference on Supercomputing* (*ICS '11*), pp.85–95 (2011).

[6] Zhang, W. and Li, T.: Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures, *Proc. 18th International Conference on Parallel Architectures and Compilation Techniques*, pp.101–112 (2009).

[7] Mogul, J.C., Argollo, E., Shah, M. and Faraboschi, P.: Operating system support for NVM+DRAM hybrid main memory, *Proc. 12th Conference on Hot topics in Operating Systems* (*HotOS '09*) (2009).

[8] Tehrani, S.: Status and Prospect for MRAM Technology, *Proc. 22nd Symposium on High Performance Chips* (*Hot Chips*) (2010).

[9] Baker, M., Asami, S., Deprit, E., Ouseterhout, J. and Seltzer, M.: Non-volatile Memory for fast, reliable file systems, *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-V*), pp.10–22 (1992).

[10] Woodhouse, D.: JFFS: The journaling flash file system, *Ottawa Linux Symposium* (July 2001), available from ⟨http://sources.redhat.com/jffs2/jffs2.pdf⟩.

[11] Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D. and Coetzee, D.: Better I/O through byte-addressable, persistent memory, *Proc. ACM SIGOPS 22nd Symposium on Operating Systems Principles* (*SOSP '09*), pp.133–146 (2009).

[12] Wu, X. and Reddy, A.L.N.: SCMFS: A file system for storage class memory, *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (*SC '11*), pp.1–11 (2011).

[13] Park, Y. and Park, K.H.: High-Performance Scalable Flash File System Using Virtual Metadata Storage with Phase-Change RAM, *IEEE Trans. Comput.*, Vol.60, No.3, pp.321–334 (2011).

[14] Bailey, K., Ceze, L., Gribble, S.D. and Levy, H.M.: Operating system implications of fast, cheap, non-volatile memory, *Proc. 13th USENIX Conference on Hot topics in operating systems* (*HotOS 13*) (2011).

[15] Jung, J-Y. and Cho, S.: Dynamic co-management of persistent RAM main memory and storage resources, *Proc. 8th ACM International Conference on Computing Frontiers* (*CF '11*) (2011).

[16] Protected and Persistent RAM Filesystem (2012), available from ⟨http://pramfs.sourceforge.net/⟩.

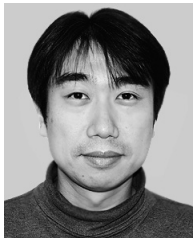
[17] Wu, M. and Zwaenepoel, W.: eNVy: A non-volatile, main memory storage system, *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS-VI*), pp.86–97 (1994).

[18] Saxena, M. and Swift, M.M.: FlashVM: virtual memory management on flash, *Proc. 2010 USENIX Conference on Annual Technical Conference* (*USENIX ATC '10*) (2010).

[19] Badam, A. and Pai, V.S.: SSDAlloc: hybrid SSD/RAM memory management made easy, *Proc. 8th USENIX Conference on Networked Systems Design and Implementation* (*NSDI '11*) (2011).

[20] Guerra, J., Mármol, L., Campello, D., Crespo, C., Rangaswami, R. and Wei, J.: Software Persistent Memory, *Proc. 2012 USENIX Conference on Annual Technical Conference* (*USENIX ATC '12*) (2012).

[21] Bensoussan, A., Clingen, C.T. and Daley, R.C.: The Multics virtual memory: Concepts and design, *Comm. ACM*, Vol.15, No.5, pp.308–318 (1972).



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，電子情報通信学会各会員．