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.1　1–11 (Mar. 2015)

ゲストOS軽量化のためのストレージ仮想化手法

追川 修一[1,a)]

受付日 2014年7月15日, 採録日 2014年10月28日

概要：計算機が高性能化し，またクラウドコンピューティングが普及するにともない，オペレーティングシステム（OS）が仮想化環境で使われることが多くなっている．仮想化環境では，仮想マシン（VM）がOSを実行する．VMは仮想化環境が定義するものであるが，実機上で動作するOSをそのまま実行できる実機に相当するVM，そしてOSとVMが連携することで処理を軽量化するVMが，これまで提供されてきた．しかしながら，OSの構造，およびVMがOSに提供するインタフェースは，実機上で動作するOSのものから大きく変更されることはなかった．本論文では，VMが実行するOSの軽量化を目的とし，VMがOSに提供するインタフェースを変更するかたちでのストレージ仮想化手法について述べる．提案手法を，Linuxをホスト OSとして用いるKVMに実装した．実験結果から，従来手法と比較して，提案手法はアクセスを高速化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，仮想化，ストレージ

Storage Virtualization Method for a Lightweigt Guest OS

Shuichi Oikawa[1,a)]

Received: July 15, 2014, Accepted: October 28, 2014

Abstract: As the performance of computing platforms becomes higher and cloud computing becomes popular, it is common to execute operating systems (OSes) on virtualized environments. Such virtualized environments employ a virtual machine (VM) to execute an OS. While VMs can be defined by virtualized environments, they are defined to be the same as or similar to real hardware; thus, their interface to OSes also remain mostly unchanged. This paper describes a storage virtualization method that changes the VM's storage interface in order to make guest OSes lightweight. We implemented the proposed method in KVM, which utilizes Linux as its host OS. The evaluation results show that the method can improve the data access by comparing with the existing method.

Keywords: operating systems, virtualization, storage

1. はじめに

近年，プロセッサの高性能化，マルチコア化がすすみ，計算機単体の処理能力が大きく向上したことで，オペレーティングシステム（OS）が仮想化環境で使われるようになって久しい [1]．さらに，クラウドコンピューティングの普及により，仮想化環境の重要性はさらに増している．仮想化環境では，仮想マシンモニタ（VMM: Virtual Machine Monitor）が仮想マシン（VM: Virtual Machine）を構成し，VMがOS（ゲストOS）を実行する．実機が実行できるOSカーネルと同一のカーネルを，VMが実行可能な場合，仮想化環境は完全仮想化されているという．一方，VMがゲストOSカーネルを実行することを前提とし，VMおよびゲストOSカーネルを特化することで，仮想化環境におけるOSの実行を効率化することができる．このような仮想化環境は，準仮想化されているという [2]．現在のカーネルは，ブート時に実行環境を識別し，動的に準仮想化環境に適用するようにカーネル自体を変更するものもある [3]．

そのような準仮想化環境におけるVMであっても，実機からの変更点は，プロセッサの特権命令の置き換えや，軽量なソフトウェア処理が可能な仮想デバイス [4] の提供等

[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

にとどまり，VM が提供するカーネルへのインタフェースが，実機から大きく変更されることはなかった．その理由としては，完全仮想化環境の効率化を目的とし，VM を実機にできるだけ近づけるべく，プロセッサ仮想化支援機能やネットワーク仮想化機能 [5] のような，ハードウェアによる仮想化のサポートの方向へ，開発が進んでいたことが考えられる．

本論文は，ゲスト OS の軽量化を目的とし，VM が提供する OS へのインタフェースを変更するかたちでのストレージ仮想化手法 *VMMS*（*Virtual Main Memory Storage*）について述べる．VMMS はストレージをメモリとして仮想化する [6]．そのため，仮想ストレージへのインタフェースは，メモリアクセスインタフェースとなる．これにより，ゲスト OS は複雑なブロックデバイスドライバを必要としなくなり，ゲスト OS を軽量化することができる．メモリとしての仮想化は，仮想ストレージを提供するファイルを VM のゲスト物理アドレス空間にマップすることで実現する．

VMMS は，Linux をホスト OS として用いる KVM [7] に実装した．KVM を制御するために用いられる，QEMU システムエミュレータを変更することで，仮想ストレージファイルを VM のゲスト物理アドレス空間にマップする．実験結果から，VMMS は，ホスト OS のページキャッシュ有効時に，読み出しで最大 9.0 倍，書き込みで最大 10.6 倍，従来手法である virtio よりもアクセスを高速化することができることが分かった．また，VMMS は，ホスト OS のページキャッシュに依存する構造となっているが，攪乱用 VM を実行する実験から，cgroups によるメモリ資源管理がページキャッシュ干渉抑制に一定の効果があり，干渉による性能低下を抑えることができることが分かった．

以下，2 章で背景を述べる．3 章は仮想ストレージをメモリとして仮想化する手法について述べ，4 章はその実装について述べる．5 章は提案手法を評価し，6 章で考察を行う．7 章は関連研究を述べ，8 章で本論文をまとめる．

2. 背景

本章では，本論文の背景として，仮想化環境におけるストレージアクセス，および SSD 高性能化にともなうストレージアクセス手法の変化について述べ，既存のストレージ仮想化手法の問題点についてまとめる．

2.1 仮想化環境におけるストレージアクセス

一般に，ユーザプロセスは，カーネルを介して，HDD や SSD 等のストレージにアクセスする．ストレージアクセスのために，カーネルはファイルシステム，ページキャッシュ，ブロックデバイスドライバといった機能を提供する．HDD や SSD 等のストレージは，DRAM 等のバイト単位でのアクセスが可能なメモリとは異なり，ある一定サイズのブロック単位でアクセスする必要があるブロックデバイス

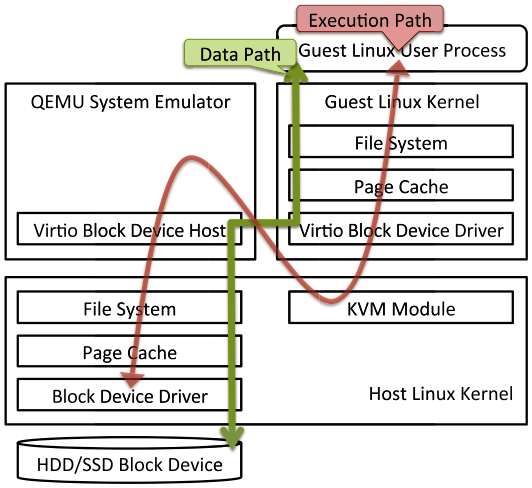


図 1　KVM 仮想化環境におけるゲスト OS からのストレージアクセスの実行パスとデータパス

Fig. 1　The execution and data paths of storage access on KVM virtualization software.

である．ストレージアクセスは，プロセッサの速度と比較すると非常に低速である．そのため，ストレージアクセスを制御するブロックデバイスドライバは，アクセスを効率化するための機構を提供する．また，ブロック単位でしかアクセスできないストレージに対して，ページキャッシュはプロセッサによるバイト単位でのアクセスを仲介し，また低速なアクセス時間を緩和する機構を提供する．ファイルシステムは，単純な一次元構造しか持たないストレージに論理的な階層構造を導入し，ファイルやディレクトリからなる木構造を構成する機構を提供する．

図 1 に，KVM 仮想化環境がストレージアクセスのために構成するホスト OS とゲスト OS の構造，およびゲスト OS からストレージアクセスするための実行パスとデータパスを示す．KVM 仮想化環境は，ハードウェア仮想化機能を制御する KVM カーネルモジュール，KVM カーネルモジュールの制御およびデバイスをエミュレートする QEMU システムエミュレータを用いて，VM を構成，実現する．

図 1 に示した仮想化環境は，仮想ブロックデバイスとして virtio [4] を用いており，デバイスエミュレータが，仮想デバイスを実現する．すなわち，デバイスドライバは仮想デバイスにアクセス要求を出し，デバイスエミュレータはアクセス要求を処理する．デバイスエミュレータは，VM の外部，ホスト OS 上で実行されるプログラムに実装される．図中では，デバイスエミュレータは，QEMU システムエミュレータに含まれ，ユーザプロセスとして実行される．virtio 仮想ブロックデバイスのデバイスエミュレータは，ストレージへのアクセス要求を処理するため，システムコールを発行し，仮想ストレージファイルにアクセスする．デバイスエミュレータによる仮想ストレージファイルへのアクセスは，通常のユーザプロセスによるファイルへ

のアクセスと何ら変わらない．

図 1 が示しているように，ホストおよびゲスト OS のカーネル構造は，基本的に同じである．したがって，ゲスト OS のユーザプロセスからのストレージアクセス要求の実行パスは，1) ゲスト OS カーネルにおけるユーザプロセスからの，仮想ブロックデバイスのドライバの呼び出し，2) ゲスト OS を実行する VM からホスト OS 側への実行の移行，および仮想ブロックデバイスのエミュレーション処理を行うユーザプロセスの起動，3) ホスト OS カーネルにおけるユーザプロセスからの，実ブロックデバイスのドライバの呼び出し，からなる．一方，データパスは，1), 3) については実行パスと基本的に同一であるが，2) については，仮想デバイスのデバイスドライバとデバイスエミュレータ間に作られる共有メモリ（virtio queue）を経由するため，ホスト OS カーネルを経由せずに，直接デバイスドライバとデバイスエミュレータ間で通信が行われる．

2.2　SSD の高性能化

近年，フラッシュメモリを記憶デバイスとする SSD が普及し，複数デバイスへの並列アクセスや，フラッシュメモリの特徴を活かすアクセス処理の工夫により，SSD の高性能化がすすんでいる [8]．そして，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，PCM（phase change memory）や MRAM，ReRAM といった次世代不揮発性メモリを用いた SSD の研究開発も行われている [9], [10]．さらに，SSD の高性能を活かすための基盤として，次世代の I/O バスやコントローラ仕様である PCI-Express Gen 3 や NVM-Express [11] がある．

このように SSD の高性能化がすすむにつれ，もともと低速な HDD へのアクセスを前提に開発された，ブロックデバイスドライバの処理コストが大きいことが顕在化してきた．HDD の場合，アクセス要求を出してから，その要求処理が終了するまでの遅延が大きい．そのため，処理が終了するまでの待ち時間に，別プロセスを起動・実行することで，CPU 時間を無駄にしないようにしてきた．割込みによる処理終了の通知を受けて，要求を出したプロセスは実行を再開する．ブロックデバイスドライバは，このような非同期アクセス処理を行う機構のほか，連続するアクセス要求のとりまとめ，磁気ヘッドの動きを最小化するためのアクセス要求の並べ替え，プロセス間のアクセス要求の調停といった機能を提供するフレームワークを持つ．このような様々な機能の処理コストは大きいが，HDD が低速であるため，処理コストが顕在化することはなく，むしろ処理コストをかけただけの効果が得られた．

しかしながら，SSD は，現状でも連続アクセスで数倍，ランダムアクセスの場合は数十倍，HDD よりも高速である．次世代技術により，さらに 1 桁以上の高性能化が期待されている．このような高性能 SSD では，これまでのブロックデバイスドライバの処理は性能向上に寄与せず，したがって処理コストは単なるオーバヘッドとなる．そして，SSD が十分に高速である場合，これまでの非同期アクセス処理ではなく，アクセス要求を同期的に処理する方が処理コストが小さくてすみ，待ち時間を考慮してもアクセス全体のオーバヘッドが小さいことが分かってきた [12], [13]．アクセス要求の同期処理は，SSD に要求を出し，その処理終了を検知するためにポーリングする．ポーリング中の CPU 時間は無駄になるが，同期的に処理することにより処理コストが減少するため，非同期アクセス処理の場合の使用可能 CPU 時間を，同期アクセス処理の場合の使用可能 CPU 時間が上回る結果となる．

以上のように，SSD の高性能化は，ストレージアクセス手法の非同期アクセス処理から同期アクセス処理への変化をもたらす．

2.3　問題点

2.1，2.2 節で述べた背景から，SSD の高性能化にともなうストレージアクセス手法の変化を考慮した場合の，既存の仮想化環境におけるストレージアクセスの問題点についてまとめる．

2.1 節で述べたように，ホスト OS とゲスト OS は，ストレージアクセスに関して基本的に同じカーネル構造を持つ．そして，ストレージアクセスを処理するにあたり，ゲスト OS からホスト OS への切替え，仮想ブロックデバイスのエミュレーション処理を行うユーザプロセスの起動をともなう．すなわち，仮想化環境における 1 回のストレージアクセスは，2 つのユーザプロセスからのストレージアクセスおよびプロセスの切替えに相当するコストを要することになる．そして，ホスト OS とゲスト OS のそれぞれがブロックデバイスドライバを持つため，その両方が非同期処理を行う．この非常に長い実行パスとデータパスは，HDD が低速であり，非同期処理が有効に働くがゆえに問題にならなかった．

既存の仮想化環境におけるストレージアクセス方式は，しかしながら，2.2 節で述べた，SSD の高性能化によるストレージアクセス手法の非同期アクセス処理から同期アクセス処理への変化に対応できない．まず，単純にホスト OS とゲスト OS のそれぞれのブロックデバイスドライバで同期アクセス処理を行うこととした場合，非常に長い実行パスにともなう大きな遅延が問題になる．ストレージアクセスに大きな遅延がともなう場合は，既存のブロックデバイスドライバが行っているように，ゲスト OS では，連続するアクセス要求をできるだけとりまとめ，非同期アクセス処理を行う方が有利であることになってしまう．そこで，ホスト OS では同期アクセス処理，ゲスト OS では非同期アクセス処理を行うこととしても，ホスト OS 側では SSD の性能を活かした同期アクセス処理が行えるが，ゲ

スト OS 側のブロックデバイスドライバに起因するオーバヘッドはそのままである．また，データパスが長いままであることも，SSD の性能を活かせない原因となりうる．

以上のように，既存の仮想化環境におけるストレージアクセス方式は，ホスト OS とゲスト OS がストレージアクセスに関して基本的に同じカーネル構造を持つことに起因し，SSD の高性能化による同期アクセス処理への変化に対応できない．

3. VMMS：仮想ストレージのメモリとしての仮想化

本章は，2.3 節で述べた問題点を解決するための，仮想ストレージをメモリとして仮想化する手法 [6] について述べる．以下，メモリとして仮想化したストレージを *VMMS*（*Virtual Main Memory Storage*）と呼び，またそのストレージ仮想化手法をも表すものとする．VMMS は，短い実行およびデータパス，実行コンテキスト切替え数の削減，同期アクセス処理との親和性を実現し，既存手法の問題点を解決する．

3.1 動作原理

VMMS は，仮想ストレージを提供するファイルを，VM のゲスト物理アドレス空間にマップすることで，ストレージをメモリとして仮想化する．VM が実行するゲスト OS カーネルは，マップされたアドレスにメモリアクセスを行うことで，ストレージへのアクセスが可能になる．**図 2** に，VMMS を導入したホスト OS とゲスト OS の構造，およびゲスト OS からストレージアクセスするための実行パスとデータパスを示す．ストレージは，一般に直接メモリアクセスができないブロックデバイスであるため，そのデータを仲介するホスト OS のページキャッシュが，ゲスト物理アドレス空間にマップされる．すなわち，ゲスト OS カーネルのファイルシステムは，ホスト OS のページキャッシュに直接アクセスすることになる．

VMMS は，仮想ストレージのインタフェースをメモリアクセスインタフェースとする．そのためゲスト OS は，VMMS が提供する領域をアクセスするために，ページキャッシュおよび複雑なブロックデバイスドライバを必要としない．そして，アクセスするデータがホスト OS のページキャッシュにあり，すでに VM のゲスト物理アドレス空間にマップされていれば，VM 内で処理は完結する．アクセスするデータがホスト OS のページキャッシュになければ，ホスト OS カーネルに実行を切り替え，ホスト OS が仮想ストレージを提供するファイルにアクセスする．この場合，仮想ストレージファイルへのアクセスはホスト OS カーネルで完結し，デバイスエミュレータは介在しない．

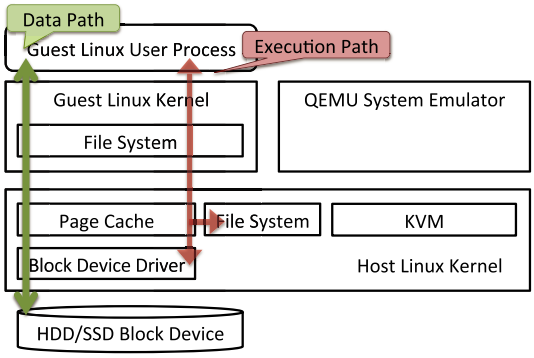


図 2　VMMS におけるストレージアクセスの実行パスとデータパス
Fig. 2　The execution and data paths of storage access enabled by VMMS.

3.2 特徴

VMMS では，ゲスト OS はページキャッシュおよび仮想ブロックデバイスドライバを必要としないこと，ホスト OS は仮想ブロックデバイスドライバに対応したデバイスエミュレータを必要としないことから，データアクセスに必要な実行パスは短縮される．データパスについても，同様の理由により，少なくとも，ゲスト OS におけるデバイスドライバとページキャッシュ間のデータコピーがなくなり，短縮される．VMMS はまた，仮想ストレージファイルへのアクセスにデバイスエミュレータを必要としないことから，ホスト OS からデバイスエミュレータへの切替え，それに続くファイルアクセスのためのホスト OS の呼び出しが不要になり，実行コンテキスト切替え数が削減される．VMMS は，ストレージへの同期的メモリアクセスインタフェースを提供するという点で，2.2 節で述べた SSD の高速化がもたらす同期アクセス処理への対応性が高い．

VMMS は，mmap システムコールによるファイルのマップを，仮想化環境に応用したものということができる．mmap システムコールは，ファイルをユーザプロセスの仮想アドレス空間にマップする．この場合，ページキャッシュに読み込まれたファイルのデータを持つページフレームを，ページテーブルを介して，仮想アドレス空間にマップする．VMMS は，仮想ストレージを提供するファイルを，VM のゲスト物理アドレス空間にマップする．この場合，ホスト OS のページキャッシュに読み込まれたファイルのデータを持つページフレームを，拡張ページテーブル[*1]を介して，VM のゲスト物理アドレス空間にマップする．

VMMS は，ホスト OS のページキャッシュを，VM のゲスト物理アドレス空間にマップする．マップされるページキャッシュは，仮想ストレージを提供するファイルのデータが読み込まれたページフレームに限られる．ゲスト物理アドレスとホスト OS のページキャッシュとの対応付けは，ホスト OS 側で制御され，ゲスト OS が関与すること

[*1] このようなゲスト物理アドレスからホスト物理アドレスへの変換を行う拡張ページテーブルを，Intel は EPT（Extended Page Table）と呼び，AMD は NPT（Nested Page Table）と呼ぶ．

はできない．これは，mmap における仮想アドレスとページキャッシュとの対応付けに，ユーザプロセスが関与することができないのと同様の理由による．そのため，VMMS により，ゲスト OS が任意のページキャッシュにアクセス可能になることはなく，VM 間の保護には影響を与えない．

VMMS は，仮想ストレージを提供するファイルを，VM のゲスト物理アドレス空間にマップするが，マップされた領域は，メインメモリとして使用される領域とは別の管理としている．そのため，ゲスト OS がメインメモリとして認識する物理メモリは，従来どおりである．

3.3　メモリストレージへの対応

VMMS は，VM にバイトアクセス可能なメモリストレージを提供する．ファイルシステムがメモリストレージへアクセスする方法は，主に 2 通りある．1 つは，PRAMFS [14] や PMFS [15] のような，メモリストレージ上の構築を前提に設計されたファイルシステムを用いる方法である．もう 1 つは，メモリストレージにアクセスするために，PRD [16] のような，指定された領域を RAM ディスクとするデバイスドライバを用い，その上に通常のファイルシステムを構築する方法である．どちらの方法でも，メモリストレージ領域の情報として，開始アドレスおよびサイズが必要となる．その情報取得については，4.3 節で述べる．RAM ディスクドライバを用いる場合，そのドライバが，ブロックデバイスインタフェースを通した，メモリストレージへのアクセスを提供する．そのため，ブロックデバイス上で動作する，任意のファイルシステムと組み合わせて用いることができ，それらのファイルシステムへの変更は不要である．

ファイルシステムまたは RAM ディスクドライバのどちらの方法でも，メモリストレージがバイトアクセス可能であることを活用するためには，XIP（eXecution-In-Place）への対応が重要となる．XIP は，メモリストレージ上のデータを，ページキャッシュへコピーすることなく参照，およびユーザアドレス空間へマップすることを可能にする．RAM ディスクドライバを用いる方法で XIP を用いるには，RAM ディスクドライバおよびファイルシステムの両方が XIP に対応している必要がある．現状のメインライン Linux カーネルでは，Ext2 ファイルシステムのみが XIP に対応している

また，SSD 高速化の延長線上には，次世代不揮発性メモリを用いたバイトアクセス可能なメモリストレージが考えられる．ホスト OS において，そのようなメモリストレージに仮想ストレージを提供するファイルが置かれる場合でも，従来手法は VM にブロックデバイスインタフェースを提供するため，メモリストレージを十分に活用できない．一方，VMMS は，メモリストレージを，ホスト側においても XIP を用い，VM のゲスト物理アドレス空間にマップすることで，VM から直接メモリストレージへのアクセスを可能にする．

3.4　汎用性

VMMS の汎用性として，KVM 以外の仮想化ソフトウェアでの，VMMS の実現可能性について考察する．仮想化ソフトウェアは，大きく分けて，実機上で動作する VMM 上にすべての VM が構成される Type-1 VMM，実機上で動作するホスト OS に VMM 機能が含まれる Type-2 VMM，に分類される [1]．KVM は Linux をホスト OS とする Type-2 VMM である．Type-2 VMM としては，他に VirtualBox が広く用いられている．Type-1 VMM として代表的なものには Xen [2] がある．

VMMS は，Type-2 VMM では容易に実現可能であると考えられる．Type-2 VMM では，メモリを含めた資源管理はホスト OS で行う．そのため，VM の物理メモリ領域を確保するためには，ホスト OS で割り当てたアノニマスメモリ領域を，VM のゲスト物理アドレス空間にマップする操作が必要となる．VMMS の実現には，アノニマスメモリ領域の代わりに，仮想ストレージファイルをマップしたメモリ領域を，VM のゲスト物理アドレス空間にマップすればよい．

メモリ資源を VMM が直接管理する Type-1 VMM での VMMS の実現は，Type-2 VMM ほど容易ではなく，また実現可能性は VMM の実装に大きく依存すると考えられる．VMM が，ホスト OS 相当の機能を持ち，メモリだけでなくデバイスの仮想化まで行うものであるとすると，VMMS が必要とするホスト側のページキャッシュが提供されない可能性があり，その場合，既存機能の改変だけでは実現は困難である．Type-1 VMM には，VMM は必要最低限の資源管理を行い，OS が動作する特権的な管理 VM で，デバイスの仮想化等その他の資源管理を行うものもある．たとえば，Xen はこの形態である．デバイスの仮想化には，メモリマップ IO（MMIO）領域を，VM のゲスト物理アドレス空間にマップする操作が必要となる．MMIO 領域は，デバイス仮想化を行う管理 VM の OS が割り当てる．そこで，MMIO 領域をマップする機能を用いて，VMMS のメモリ領域をゲスト物理アドレス空間にマップできる可能性がある．

4.　実装

VMMS は，拡張ページテーブルをサポートする x86_64 CPU をターゲットとし，Linux をホスト OS として用いる KVM 仮想化環境に実装した．本章では，VMMS の実装として，ホスト OS における仮想ストレージファイルのゲスト物理アドレス空間へのマップ方法，このマップ方法に起因する Linux 仮想メモリシステムとの整合性の問題，そしてゲスト OS への改変とメモリストレージ領域の検出について述べる．なお，VMMS はカーネル内の KVM モジュールへの変更は必要としない．

4.1　ファイルのゲスト物理アドレス空間へのマップ

KVM を制御する QEMU システムエミュレータを変更し，ホスト OS において，仮想ストレージファイルを VM のゲスト物理アドレス空間へマップ可能にする．そのために，VMMS として仮想化するファイル名を，QEMU 起動時の引数として指定する．ファイル名とあわせて，マップ先のゲスト物理アドレス空間のアドレスも指定する．

QEMU は，まず指定されたファイルを，mmap システムコールにより，自身のプロセス空間にマップする．そして，そのマップした領域をメモリ領域として登録する．メモリ領域としての登録は，QEMU 内で抽象化された操作となっている．KVM を制御するように起動された QEMU は，登録対象のメモリ領域を，カーネル内の KVM モジュールに通知する．KVM モジュールは，通知された領域を，VM のゲスト物理アドレス空間に設定することで，登録処理は終了する．

KVM モジュールは，登録された領域から，ゲスト物理アドレスに対応するホスト OS における仮想メモリ領域の情報，そしてその仮想メモリ領域に対応する物理メモリ領域の情報を取得することができる．この情報を用いて，KVM モージュルは，ゲスト物理アドレスとホスト物理アドレスを対応づける．ホスト物理アドレスの実体は，対応する仮想メモリ領域がホスト OS 側でどのように確保されたかに依存する．ホスト OS 側の仮想メモリ領域が `malloc()` されたヒープ領域であれば，アノニマスページが用いられる．また，VMMS のように仮想ストレージファイルがマップされた領域であれば，ファイルが読み込まれたページキャッシュが用いられることになる．したがって，ゲスト物理アドレスとホスト物理アドレスの対応づけに，ゲスト OS が関与する必要はなく，そのために，ゲスト OS がハイパーコール等によりホスト OS を呼び出すことはない．

4.2　Linux 仮想メモリシステムとの整合性

前節で述べた方法で，ホスト OS において，ファイルを VM のゲスト物理アドレス空間へマップしようとすると，Linux カーネルの仮想メモリシステムとの整合性がとれず，マップに失敗する場合が生じた．この問題は，ホスト側においても XIP を用い，メモリストレージとして用いていた RAM ディスク上のファイルを，VM のゲスト物理アドレス空間にマップした場合に発生した．

この問題は，VM のゲスト物理アドレス空間へマップできる物理ページフレームは，カーネルが管理するメインメモリのみに制限されていたことに起因していた．そこで，XIP によりマップされる物理ページフレームの属性を変更することで，整合性の問題を解決した．具体的には，`xip_file_mmap()` は，マップ先の仮想メモリ領域の属性を以下に設定していた．

```
vma->vm_flags |= VM_CAN_NONLINEAR | VM_MIXEDMAP;
```

`VM_MIXEDMAP` は，デバイスのメモリ領域をマップ可能にするための属性である．KVM モジュールは，この属性がついている領域のゲスト物理アドレス空間へのマップを禁止している．そこで，マップするファイルが，カーネルが管理可能なメインメモリ，すなわち `struct page` が付与されているメモリ領域にある場合，この属性を削除するようにした．そして，`xip_file_fault()` が，実際に物理ページフレームをページテーブルに設定する際に，`vma->vm_flags` に `VM_MIXEDMAP` が含まれていなければ，`vm_insert_page()` を呼ぶようにした．

この変更により，メモリストレージの提供するメモリ領域に `struct page` が付与されている場合は，ホスト OS の XIP により，その領域上のファイルをゲスト物理アドレス空間にマップし，使用可能となる．付与されていない場合は，ゲスト物理アドレス空間にマップした領域に，実行中アクセスが生じたときにエラーが発生するが，これはそもそも KVM モジュールの制約である．また，KVM を用いない場合は，`struct page` が付与されている場合とされていない場合で，適切な方法でページテーブルに設定するため，他の機能に影響を及ぼすことはない．

4.3　ゲスト OS への改変とメモリストレージ領域の検出

VMMS を用いるにあたり，ゲスト OS カーネルは，3.3 節で述べたメモリストレージへのアクセスが可能なファイルシステム，または RAM ディスクドライバを必要とする．どちらも，現時点でのメインラインの Linux カーネルには含まれていない．そのため，ファイルシステムとしては PRAMFS や PMFS，RAM ディスクドライバとしては PRD 等を導入する必要がある．ホスト側において XIP を用い，メモリストレージ上のファイルを，VM のゲスト物理アドレス空間にマップする場合は，前節で述べたとおり，メモリストレージの提供するメモリ領域に `struct page` を付与する必要がある．メモリストレージへのアクセスに対応したファイルシステムまたは RAM ディスクドライバは，メモリストレージ領域の開始アドレスとサイズを指定することで，使用することができる．メモリストレージ領域は，ホスト OS 側でゲスト物理アドレス空間へマップすることで作成されるため，その領域を使用するにあたり，ゲスト OS カーネルは，その他の変更は必要としない．

ゲスト OS カーネルは，メモリストレージ領域へアクセスするにあたり，その領域の開始アドレスとサイズを取得する必要がある．現状では，ホスト OS からゲスト OS への情報提供手段は実装しておらず，暫定的に，ゲスト OS 側は開始アドレス，サイズともに固定されているものとして扱っている．しかしながら，物理メモリマップ情報は，BIOS またはファームウェアから提供されるのが一般的であるため，メモリストレージ領域の情報も，同様に提供されるのが妥当である．そのためには，たとえば x86_64 シ

ステムにおいては，E820 または EFI のメモリタイプを拡張し，メモリストレージ領域の情報を取得するためのインタフェースを実装する必要がある．

5. 評価

VMMS を評価するため，読み書きコストを計測し，既存手法と比較する．また，ページキャッシュの干渉についても実験を行う．

5.1 実験環境

実験に用いた OS 環境は，ホスト，ゲストともに，Fedora 19，Linux カーネルは 3.4 [*2]である．KVM を制御する QEMU のバージョンは 1.6.1 である．VMMS およびホストシステムのメモリストレージ領域を管理する RAM ディスクドライバは，I/O リクエストスケジューリングの有無，XIP 機能を提供するものを実装し，用いた．実験は Intel Core i7-3770 3.4 GHz 上で行い，実行時間は，RDTSC 命令により計測した．

ホストシステムのメモリは，メインメモリに 4 GB，メモリストレージに 8 GB 割り当てた．メモリストレージ領域は，RAM ディスクドライバが管理し，その上に Ext2 ファイルシステムを構築した．Ext2 は，現状で，既存のブロックデバイス上に構築でき，かつ XIP をサポートする Linux メインラインカーネルに含まれる唯一のファイルシステムである．

ゲスト VM のメモリは，256 MB 割り当てた．VMMS 領域は，ホストシステム同様，RAM ディスクドライバが管理し，その上に Ext2 ファイルシステムを構築し，XIP を有効にした．従来方式を用いる場合，virtio ブロックデバイス上に Ext2 ファイルシステムを構築した．virtio は XIP をサポートしない．

5.2 読み書きコスト

VMMS および従来手法による 512 MB ファイルの読み書きコストを計測し，比較する．read，write システムコールを用いた場合と，mmap システムコールを用いた場合の両方で計測を行った．読み出しの計測には，ファイルを作成し，指定サイズの書き込みを行った後に，そのファイルを読み出す場合（warmup）と，ページキャッシュのフラッシュまたはページのアンマップ操作後に読み出す場合（no-warmup）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 度書き込む場合（warmup）と，ファイル作成後に初めて書き込む場合（no-warmup）を計測した．

計測対象は，VMMS および従来手法（virtio）である．ホスト OS における RAM ディスクを管理するブロックデバイスドライバには，I/O リクエストスケジューリングを用いる場合（queueing），用いずに直接アクセス方式を用いる場合（direct），XIP 機能を用いる場合（XIP）があり，それぞれの場合で計測を行った．queueing は非同期アクセス処理，direct，XIP は同期アクセス処理を行う．virtio を用いる場合，QEMU の設定により，仮想ストレージファイルに対してホスト OS のページキャッシュを有効にする場合（wb）としない場合（none）があり，それぞれの場合で計測を行った．なお，以下 none は表記から省略している．したがって，virtio/xip/wb は，アクセス手法として従来手法の virtio，RAM ディスクドライバは XIP 機能を用い，仮想ストレージファイルに対してホスト OS のページキャッシュを有効にした場合を表す．**表 1** に，計測対象となる手法・機能における選択肢をまとめる．

表 1　計測対象の各手法・機能における選択肢
Table 1　Selection options for each method/mechanism.

手法・機能	選択肢
アクセス手法	vmms，virtio
ブロックデバイスドライバ	queueing，direct，xip
QEMU キャッシュモード	wb，none

図 3，**図 4**，**図 5** に，ホスト OS におけるブロックデバイスドライバの設定で分類した計測結果を示す．(dev) は，仮想ストレージファイルとして，RAM ディスクドライバのデバイスファイルを指定し，ホスト OS のファイルシステムを経由しないようにした場合である．VMMS は，機構的にページキャッシュを無効にできないため，計測していない．

ホスト OS のページキャッシュ有効時の比較で，VMMS のコストは従来手法である virtio のコストを下回っていることが分かる．VMMS と virtio のコストの差は，読み出しで 2.8 倍から最大 9.0 倍，書き込みで 1.4 倍から最大 10.6 倍となった．特に，mmap した領域に対する読み出し（warmup），書き込み（warmup）の性能差が最も大きく，それぞれ 8.4 から 9.0 倍，9.8 から 10.6 倍の差となった．一方，その他の計測結果では，読み出しが 2.8 から 3.8 倍，書き込みが 1.4 から 3.4 倍 VMMS が高速との結果となった．

ホスト OS のページキャッシュを有効にした場合，RAM ディスクへのアクセスは初回アクセス時のみに限られるため，ブロックデバイスドライバの選択肢は，アクセスコストへの影響はわずかである．一方，ページキャッシュ無効時には，ブロックデバイスドライバの選択肢は，アクセスコストに大きく影響している．

5.3 ページキャッシュの干渉

VMMS は，ホスト OS のページキャッシュに依存する手法である．そのため，ホスト OS 上で複数 VM が動作する場合，一方の VM でのファイルアクセスによるページ

[*2] Fedora 19 付属のカーネルでは XIP が正常動作しなかった．そのため，XIP の使用実績のある 3.4 を用いた．

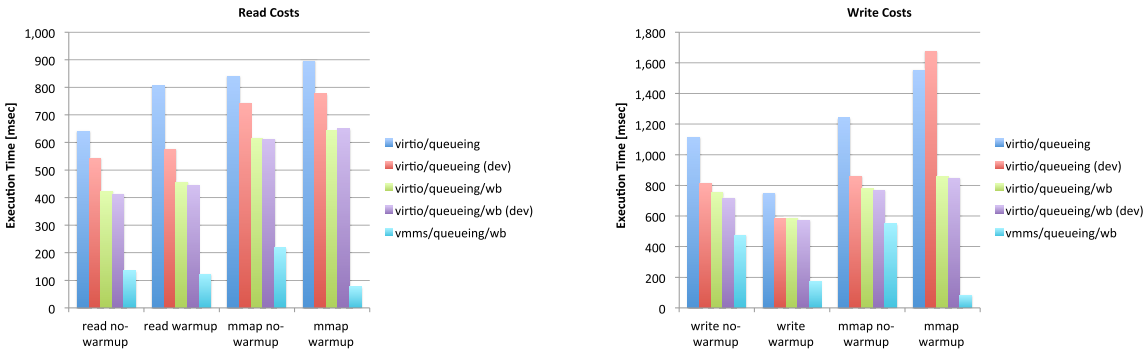


図 3　ホスト OS において I/O リクエストスケジューリングを用いた場合の 512 MB ファイルの読み書きコスト

Fig. 3　Costs to read/write 512 MB on ramdisk with the I/O request scheduling.

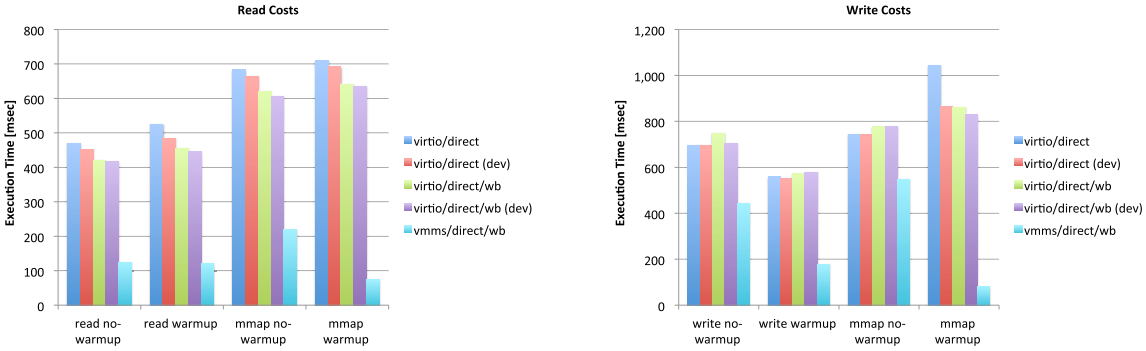


図 4　ホスト OS において直接アクセス方式を用いた場合の 512 MB ファイルの読み書きコスト

Fig. 4　Costs to read/write 512 MB on ramdisk with the direct access method.

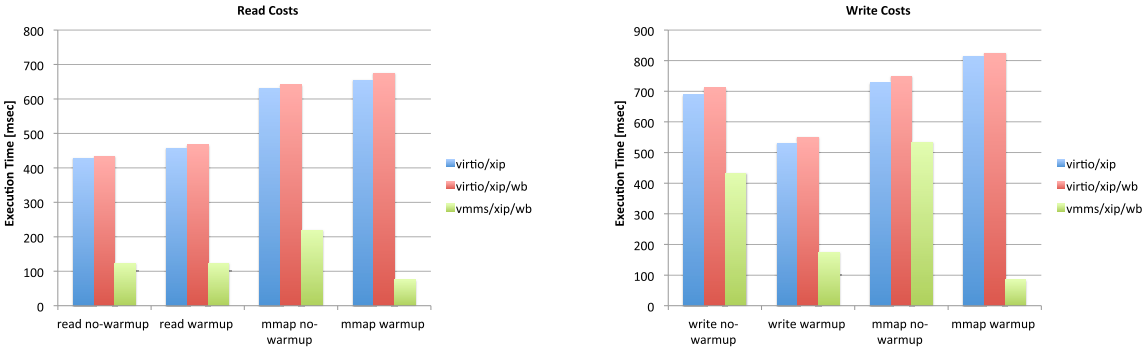


図 5　ホスト OS において XIP 機能を用いた場合の 512 MB ファイルの読み書きコスト

Fig. 5　Costs to read/write 512 MB on ramdisk with the XIP mechanism.

キャッシュの消費が，他方の VM でのファイルアクセスコストに影響を与えることが考えられる．そこで，複数 VM 動作の場合のページキャッシュの干渉について実験を行う．また，Linux カーネルが提供する資源管理機構 cgroups により，干渉を抑制できるかどうか，実験を行う．

計測対象となる VM では，前節同様，VMMS および従来手法による 512 MB ファイルの読み書きコストを計測する．計測対象 VM とは別に，攪乱要因となる VM を立ち上げ，4 GB ファイルの読み書きを実行した．ホスト OS における RAM ディスクのブロックデバイスドライバの設定は direct とした．また，仮想ストレージファイルに対してページキャッシュを有効に設定した．攪乱用 VM のアクセス手法は virtio に固定したため，その構成は，virtio/direct/wb となる．

cgroups を有効にする際には，攪乱用 VM のみを管理下に置いた．メモリの階層的管理を有効とし，1 GB の使用制限を設定した．メモリ資源管理はオーバヘッドがともなうことから，一般に有効な設定にはなっていない．このことも考慮し，cgroups を有効にした実験を行う場合のみ，メモリ資源管理が有効なカーネルを用いた．

図 6 に計測結果を示す．+disturb は攪乱用 VM を実行しながらの結果であること，+cgroups は cgroups を有効

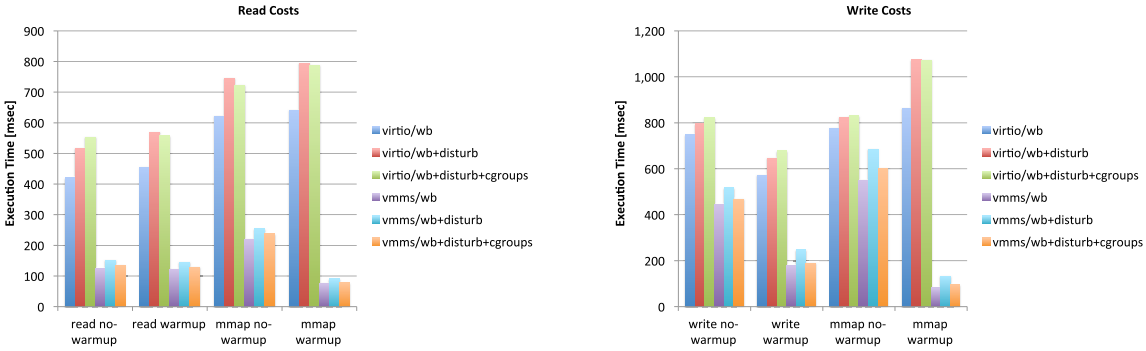


図 6　攪乱要因となる VM の実行の有無における 512 MB ファイルの読み書きコスト
Fig. 6　Costs to read/write 512 MB with/without a disturbing VM.

にした結果であることを示す．攪乱用 VM の実行により，VMMS では，読み出しで 16～22%，書き込みで 17～63%，コストが増加した．virtio では，読み出しで 20～25%，書き込みで 7～25%，コストが増加した．次に，cgroups を有効にすることで，攪乱用 VM を実行しても，VMMS では，読み出しで 2～9%，書き込みで 6～17%，のコスト増加に抑えられた．virtio では，読み出しで 16～31%，書き込みで 7～24%，コスト増加となった．

実験結果より，攪乱用 VM の実行によるページキャッシュの干渉は，VMMS の特に書き込みにより大きな影響があるが，virtio にも影響があることが分かる．cgroups によるメモリ資源管理によるページキャッシュ干渉の抑制は，VMMS には一定の効果がある．一方で，virtio には，VMMS ほどの効果が見られず，かえってコスト増となっているが，これは cgroups のオーバヘッドが原因である．cgroups を有効とし，メモリの使用制限を最大値とする，すなわち実質的には使用制限を設けない設定で実験を行った結果と比較したところ，ページキャッシュの干渉が抑制されている結果となった．

6. 考察

VMMS について，高速化の要因，およびページキャッシュ置換や先読みの影響について考察する．

6.1 VMMS 高速化の要因

VMMS が実現した高速化の主な要因について，5.2 節の結果から考察する．まず，読み出しについては，read システムコール（no-warmup）の結果を用いる．（no-warmup）の結果を用いるのは，書き込みをともなうページキャッシュ置換の影響が排除されており，より単純な場合となっているからである．VMMS を用いた読み出しコストは約 120 ミリ秒となっている．これは，その構造から，ファイルシステムを経由し，ページキャッシュからユーザプロセスへ読み出しを行うコストであると考えることができる．そして，virtio/queueing と virtio/xip の差は，約 210 ミリ秒である．これは，I/O リクエストスケジューリングを用いた場合の，ブロックデバイスドライバからの読み出しコストであると概算することができる．従来方式 virtio/queueing は，両者をゲスト OS とホスト OS で行う．そのため，約 660 ミリ秒のコストになると概算され，virtio/queueing の実際コストである 641 ミリ秒に近似した値となる．

同様に，書き込みについて，write システムコール（warmup）の結果を用いて概算を行う．書き込みについては（warmup）の結果を用いるのは，ファイルへのブロック割当てコストの影響を排除するためである．VMMS を用いた書き込みコストは約 170 ミリ秒であり，virtio/queueing と virtio/xip の差は約 220 ミリ秒である．そのため，これらの値を用いた従来方式 virtio/queueing のコストの概算は約 780 ミリ秒となり，実際コストである 749 ミリ秒に近似した値となる．

以上のストレージアクセスの構成要素の概算から，VMMS 高速化には，I/O リクエストスケジューリングを用いたブロックデバイスドライバの排除が最も大きく貢献しており，次にデバイスエミュレータによるデータアクセス仲介の排除が貢献していることが分かる．これらは，VMMS の短い実行およびデータパスが高速化を実現していることを示している．一方，実行コンテキスト切替え数の削減は，上記のコストの概算からは効果を確認することができなかった．

6.2 ページキャッシュ置換や先読みの影響

VMMS は，ゲスト OS にメモリストレージを提供するため，それを活用するため，ゲスト OS では，XIP を用い，ページキャッシュを経由しないアクセスを行う．ホスト OS で，メモリストレージ上の仮想ストレージファイルを XIP を用いてゲスト物理アドレス空間にマップしている場合，ホストおよびゲスト OS ともに，ページキャッシュを使用する必要がなくなる．ホスト OS で，通常のブロックストレージ上の仮想ストレージファイルをゲスト物理アドレス空間にマップしている場合，ゲスト OS はホスト OS のページキャッシュを使用する．したがって，ゲスト OS は，ホスト OS におけるページキャッシュ置換の影響を受け，またその置換にはゲスト OS における使用状況が十分

に反映されない可能性がある．

Linux カーネルは，どのプロセスも参照していないページキャッシュを，ユーザプロセスにマップされているページよりも，優先して回収するアルゴリズムを採用している．しかしながら，ホスト OS では，仮想ストレージファイルのページキャッシュが，ゲスト OS でどのように使用されているか把握することができないため，そのすべてのページキャッシュのアクセス頻度だけに依存して，回収するページを判断することになる．したがって，ホスト OS とゲスト OS においては，異なったページキャッシュ置換が行われることになる．しかしながら，この違いが及ぼす性能への影響を判断することは，非常に難しい．なぜならば，どのプロセスも参照していないページキャッシュを優先して回収する理由は，回収コストが低いからである．したがって，ゲスト OS でページキャッシュ置換を行うと，アクセス頻度がやや高いがマップされていないページキャッシュが回収され，その結果，VMMS を導入しホスト OS でページキャッシュ置換を行った場合よりも，性能が低下する可能性がある．一方で，VMMS のメモリストレージは，マップされたページを参照するため，回収コストはより高いが，ページスワッピングにともなう入出力コストは低い．また，回収を行う必要性は，ホスト OS でのメインメモリの逼迫度合いに依存する．ホスト OS におけるメインメモリに余裕がある場合，VMMS を導入した VM の構成を変更することなく，メインメモリの余裕を活用することができる．上記の考察から総合的に判断すると，VMMS を導入することのメリットは，ページキャッシュ置換の違いのデメリットを，やや上回ると考えられる．

次に，ディスクの先読みの影響について考察する．VMMS を導入した場合，ゲスト OS では，XIP を用い，ページキャッシュを経由しないアクセスを行うため，先読みは行われない．ホスト OS で，通常のブロックストレージ上の仮想ストレージファイルをゲスト物理アドレス空間にマップしている場合，ホスト OS のページキャッシュへの先読みが行われる．ゲスト OS で先読みを行う場合と異なるのは，ゲスト OS で先読みを行うファイルは読み込み中のファイルであるが，ホスト OS では仮想ストレージファイルである点である．したがって，ホスト OS での先読みが，ゲスト OS でアクセスしているファイルの先読みとなっているとは限らない．しかしながら，ファイルシステムは，できるだけ連続領域をファイルに割り当てようとするのが一般的である．そのため，ホスト OS での先読みによる影響はあるが，限定的であると考えられる．

7. 関連研究

仮想化環境に合わせた OS カーネルの構成手法としては，準仮想化（paravirtualization）[2] やアウトソーシング [17] がある．準仮想化は，実行環境に影響を及ぼすがその実行を検知できない命令，エミュレーションにコストがかかるデバイス等を，仮想化環境での処理に適した命令やデバイスモデルに変更することで，仮想化にともなうオーバヘッドを軽減する手法である．Linux における仮想デバイスフレームワークには，Virtio [4] がある．Virtio は，2.1 節に述べたとおり，デバイスのエミュレーションにともなうコストは軽減するが，ゲスト OS カーネルの構造を変更するものではない．

VirtFS [18] は，ゲスト OS カーネルのファイルシステム（VFS）機能が呼び出されると，それをデバイスエミュレータで処理する点で，ゲスト OS カーネルの構造を簡略化している．しかしながら，Virtio の仕組みを使用しているため，2.1 節で示した長い実行パスに変わりはない．

Vhost-blk [19] は，仮想ブロックデバイスのエミュレーションをホスト OS カーネル内で行う手法を提案している．従来方式では必要であった，デバイスエミュレータを実行するユーザプロセスを呼び出しを不要とし，ホスト OS カーネルがゲスト OS カーネルからのブロックデバイスアクセス要求を直接処理可能としている点では，VMMS と類似している．ゲスト OS で XIP 機能を使用せず，ゲスト OS のページキャッシュへのデータコピーが発生する場合，VMMS と Vhost-blk の差は，ゲストおよびホスト OS 間の通信経路だけの違いとなり，性能差はわずかであると考えられる．ホスト OS において次世代不揮発性メモリを用いたメモリストレージ上に仮想ストレージファイルが置かれる場合，VMMS ではホストおよびゲスト OS で XIP 機能を使用し，ゲスト OS はホストのメモリストレージを直接アクセスできる．一方，Vhost-blk では，ゲスト OS のページキャッシュへのデータコピーを避けることができない．この場合，VMMS はより高速にストレージアクセスを行うことができることになる．

アウトソーシングは，ゲスト OS カーネルの高水準モジュールからホスト OS カーネルを呼び出す手法である．ファイルアクセスのアウトソーシングは，VirtFS 同様，ゲスト OS カーネルの VFS 層の呼び出しを置き換えるが，アウトソーシングの場合，呼び出し先はホスト OS の VFS になる．ファイルシステム機能はホスト OS カーネルのものを使用し，また，ゲスト OS カーネルの VFS 層の呼び出しのたびにホスト OS が呼び出される．アウトソーシングもゲスト OS カーネルの構造を簡略化するが，本論文で述べたストレージをメモリとして仮想化する手法と異なり，ホスト OS への依存度がより高くなる．

仮想化環境で実行することを前提に開発されている OS として，OSv [20]，Unikernels [21]，ClickOS [22] 等がある．OSv は，Hadoop 等での利用を目的に，仮想化環境で Java VM（JVM）のみを実行可能にするために開発されている OS である．JVM が実行するアプリケーションプログラムのほか，任意のプログラムを実行することは想定していな

いため，JVM はカーネル空間で実行する．しかし，OSv カーネルの構造自体は，特に既存のカーネルと変わるところはない．Unikernels は，OSv と同様，アプリケーションプログラムは言語ランタイムでの実行を前提とし，言語ランタイムをカーネル空間で実行する．Unikernels は言語ランタイムとして，OCaml を採用している．Unikernels のストレージアクセスは，仮想化環境として使用する Xen が提供する I/O 機能を活かす形態となっているが，メモリとしての仮想化は行っていない．ClickOS は，高速なネットワーク処理を可能にするために仮想化環境を活かす形態としているが，ストレージアクセスの高速化は扱っていない．

8. まとめ

計算機の高性能化，クラウドコンピューティングの普及にともない，OS が仮想化環境で使われることが多くなっている．仮想化環境では OS は VM 上で実行されるが，OS の構造，および VM が提供する OS へのインタフェースは，これまで実機上で動作する OS から大きく変更されることはなかった．本論文は，ゲスト OS の軽量化を目的とし，ストレージの仮想化手法として，ストレージをメモリとして仮想化する手法 VMMS について述べた．VMMS は，ホスト OS のページキャッシュ有効時に，読み出しで最大 9.0 倍，書き込みで最大 10.6 倍，従来手法である virtio よりもアクセスを高速化することができる．VMMS は，ホスト OS のページキャッシュに依存する構造となっているが，攪乱用 VM を実行する実験から，cgroups によるメモリ資源管理がページキャッシュ干渉抑制に一定の効果があり，干渉による性能低下を抑えることができる．

参考文献

[1] Rosenblum, M. and Garfinkel, T.: Virtual machine monitors: Current technology and future trends, *Computer*, Vol.38, No.5, pp.39–47 (2005).

[2] Barham, P., Dragovic, B., Fraser, K., et al.: Xen and the Art of Virtualization, *Proc. SOSP '03*, pp.164–177, ACM (2003).

[3] Russel, R.: lguest: Implementing the little Linux hypervisor, *Proc. Linux Symp.*, Vol.2, pp.173–178 (2007).

[4] Russell, R.: Virtio: Towards a De-facto Standard for Virtual I/O Devices, *SIGOPS Oper. Syst. Rev.*, Vol.42, No.5, pp.95–103 (2008).

[5] PCI-SIG: Single root I/O virtualization, available from ⟨http://www.pcisig.com/specifications/iov/⟩ (2007).

[6] Oikawa, S.: Virtualizing Storage as Memory for High Performance Storage Access, *Proc. ISPA-14*, pp.18–25, IEEE (2014).

[7] Kivity, A., Kamay, Y., Laor, D., et al.: KVM: The Linux virtual machine monitor, *Proc. Linux Symp.*, Vol.1, pp.225–230 (2007).

[8] Josephson, W.K., Bongo, L.A., Li, K. and Flynn, D.: DFS: A file system for virtualized flash storage, *ACM Trans. Storage*, Vol.6, No.3, pp.14:1–14:25 (2010).

[9] 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).

[10] 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).

[11] Huffman, A. and Juenemann, D.: The Nonvolatile Memory Transformation of Client Storage, *Computer*, Vol.46, No.8, pp.38–44 (2013).

[12] Caulfield, A.M., De, A., Coburn, J., et al.: Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories, *Proc. MICRO '43*, pp.385–395, IEEE/ACM (2010).

[13] Yang, J., Minturn, D.B. and Hady, F.: When poll is better than interrupt, *Proc. FAST '12*, pp.1–7, USENIX (2012).

[14] PRAMFS: Protected and Persistent RAM Filesystem, available from ⟨http://pramfs.sourceforge.net/⟩ (2013).

[15] Persistent Memory File System, available from ⟨https://github.com/linux-pmfs/pmfs/⟩ (2013).

[16] Persistent RAM Driver, available from ⟨https://github.com/01org/prd/⟩ (2014).

[17] Eiraku, H., Shinjo, Y., Pu, C., et al.: Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments, *Proc. SAC '09*, pp.310–317, ACM (2009).

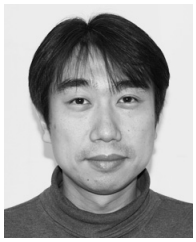
[18] Jujjuri, V., Van Hensbergen, E., Liguori, A., et al.: VirtFS — A virtualization aware File System passthrough, *Proc. Linux Symp.*, pp.109–120 (2010).

[19] He, A.: Virtio-blk Performance Improvement, KVM Forum (2012).

[20] Cloudius-Systems: OSv: The operating system designed for the cloud, available from ⟨http://osv.io⟩ (2014).

[21] Madhavapeddy, A., Mortier, R., Rotsos, C., et al.: Unikernels: Library Operating Systems for the Cloud, *Proc. ASPLOS '13*, pp.461–472, ACM (2013).

[22] Martins, J., Ahmed, M., Raiciu, C., et al.: ClickOS and the Art of Network Function Virtualization, *Proc. NSDI '14*, pp.459–473, USENIX (2014).



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．