Vol.2014-ARC-210 No.16
Vol.2014-OS-129 No.16
2014/5/15

OS 軽量化のためのストレージ仮想化手法とその応用

追川 修一[1]

概要：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，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，資源共有の効率化を目的とし，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とその応用について述べる．

1. はじめに

プロセッサの高性能化，マルチコア化がすすみ，計算機単体が処理能力が大きく向上したことで，PC ベースのコモディティシステムにおいても，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]．近年では，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て，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] の提供等にとどまり，カーネルの構造，および VM が提供するカーネルへのインタフェースは，実機が実行するカーネル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，Intel VT-x 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．

VMM としては，KVM [6] のような，VMM として OS（ホスト OS）を用いるタイプ 2 VMM も，ホスト OS の提供する機能やデバイスドライバを再利用でき，開発コストを低減できることから，普及してきた．このような，ホスト OS とゲスト OS の組み合わせでは，ホスト OS とゲスト OS は，ほぼ同一の機能を持つことになる．

本論文は，ゲスト OS の軽量化，資源共有の効率化を目的とし，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とその応用の可能性について述べる．本ストレージ仮想化手法は，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 [7]．本手法は，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．さらに，ストレージアクセスを効率化するために用いられるページキャッシュを，ホスト OS に集約することができ，メモリ資源の共有を効率化することができる．

仮想ストレージをメモリとして仮想化する手法は，さまざまな応用を持つ．まず，仮想ストレージと VM に割り当てられるメモリの区別がなくなり，不揮発性メモリとして扱うことが出来る．これにより，ストレージとメモリ管理の融合が可能になる [8]．ストレージとメモリ管理の融合により，割り当てられたメモリをそのままファイルの一部とすることができるため，チェックポイント・リスタート

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

© 2014 Information Processing Society of Japan

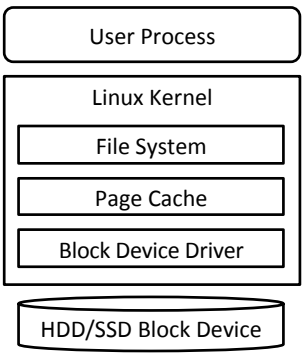

図 1 ストレージアクセスのためのカーネル構造

の高速化が可能になる [9]．高速化されたユーザプロセスのチェックポイント・リスタートにより，カーネルのソフトウェア若化が可能になる [10]．

仮想ストレージをメモリとして仮想化する手法は，Linux をホスト OS として用いる KVM に実装されている．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．

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 章は応用について述べる．5 章は関連研究を述べ，6 章で本論文をまとめる．

2. 背景

本章では，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る．

2.1 仮想化環境におけるストレージアクセス

図 1 に，一般的な OS がストレージアクセスのために構成するカーネルの構造を示す．ユーザプロセスは，カーネルを介して，HDD や SSD など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 などのストレージは，ブロックデバイスであるため，DRAM など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．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，ページキャッシュは，ブロック単位でしかアクセスできないストレージへの，プロセッサによるバイト単位でのアクセスを仲介し，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構造を導入し，

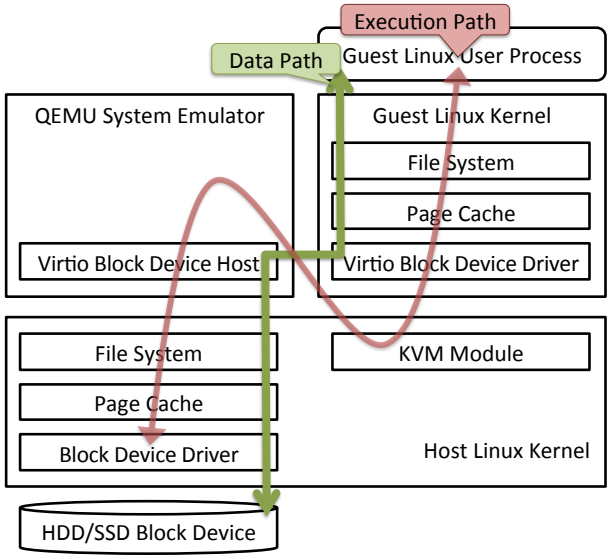

図 2 KVM 仮想化環境におけるゲスト OS からのストレージアクセスの実行パスとデータパス

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 2 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を用いて，VM を構成，実現する．

図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] を用いている．virtio は，仮想デバイスを，デバイスドライバとデバイスホストの組み合わせから構成する．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，デバイスホストはアクセス要求を処理する．デバイスホストは，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，デバイスホストは，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ホストは，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．デバイスホスト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．

ホストおよびゲスト OS のカーネル構造は，図 1 に示した構造と同一である．従って，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

© 2014 Information Processing Society of Japan

クデバイスのデバイスホスト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ホスト間に作られる 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ホスト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 [11]．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遙かに高い，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 [12], [13]．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 [14] 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．割り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他，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び替え，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以上の効果が得られた．

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数十倍，HDD よりも高速である．次世代技術により，さらに一桁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，従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わかってきた [15], [16]．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させる．

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 への切り替え，仮想ブロックデバイスのデバイスホスト処理を行うユーザプロセスの起動を伴う．即ち，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り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故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伴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伴う場合は，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得る．

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．

© 2014 Information Processing Society of Japan

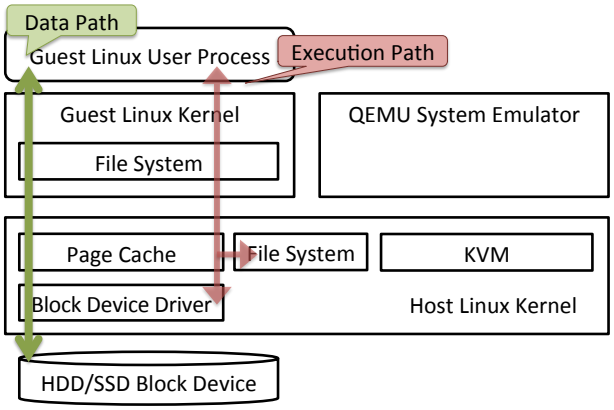

図 3 仮想ストレージをメモリとして仮想化した場合のストレージアクセスの実行パスとデータパス

3. 仮想ストレージのメモリとしての仮想化

2.3 節で述べた問題点を解決するためには，短い実行パスおよびデータパス，ゲスト OS からホスト OS への切替数の削減，同期アクセス処理との親和性が必要である．以下，これらの問題を解決するための，仮想ストレージをメモリとして仮想化する手法について述べる．

仮想ストレージをメモリとして仮想化する手法は，仮想ストレージを提供するファイルを，ゲスト OS を実行する 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 [7]．ゲスト物理アドレス空間にマップされるのは，ホスト OS のページキャッシュである．即ち，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．

図 3 に，仮想ストレージをメモリとして仮想化した場合のホスト 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．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，デバイスホストは存在しないため，仮想ストレージファイルへのアクセスはホスト OS カーネルで完結する．

本手法は，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言うことができる．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．本手法は，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 (EPT: Extended Page Table)[*1] を介して，VM のゲスト物理アドレス空間にマップする．

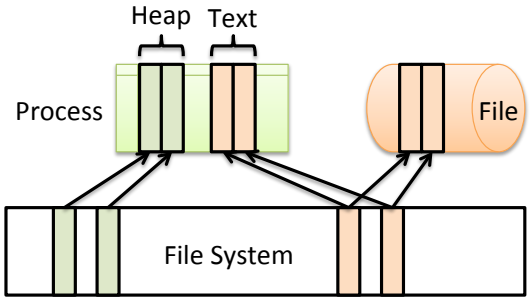

図 4 メインメモリとファイルシステムが融合されたシステム

本手法は，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ため，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．さらに，ストレージアクセスを効率化するために用いられるページキャッシュを，ホスト OS に集約することができ，メモリ資源の共有を効率化することができる．

4. 応用

既存のシステムは，揮発性のメモリと不揮発性のストレージを前提として構成されてきた．仮想ストレージをメモリとして仮想化することで，ゲスト OS はメモリを不揮発性として扱うことが可能になる．このことから，これまで不揮発性メモリを対象として行われてきた研究を，仮想ストレージをメモリとして仮想化する手法の応用として用いることができるようになる．本章では，本論文の著者が不揮発性メモリを対象として行ってきた研究を，応用例として述べる．

仮想ストレージをメモリとして仮想化しマップしたメモリ領域は，ストレージとして使用可能なだけでなく，メインメモリとしても使用可能になるため，ストレージとメインメモリ管理の融合が可能になる [8]．この管理の融合にあたり，不揮発性のメモリ領域をファイルシステムが管理するものとし，その一部を一時的にメインメモリとしての用途に割り当てる．それにより，図 4 に示すように，ファイルシステムが管理する領域を分割することなく，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なる．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，ファイルシステムの機能として提供される，XIP (eXecution In Place) を用いることにより，ページキャッシュを経由することなく，メインメモリとして直接参照す

[*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ページテーブルを，Intel は EPT と呼び，AMD は NPT (Nested Page Table) と呼ぶ．

© 2014 Information Processing Society of Japan

ることができる．

ストレージとメインメモリ管理の融合により，プロセスに割り当てられたメモリをそのままファイルの一部とすることができるようになる．これを利用することで，チェックポイント・リスタートを高速化が可能になる [9]．即ち，1) チェックポイントに必要な実行状態の大部分を占めるメモリデータを，そのままチェックポイントファイルの一部とすることによる，チェックポイントの高速化，2) リスタート時にも，チェックポイントファイルのデータを，そのまま実行状態のメモリの一部とすることによる，リスタートの高速化により，チェックポイント・リスタートの両方の高速化が実現できる．高速化されたユーザプロセスのチェックポイント・リスタートを利用することで，ユーザプロセスの状態は保ったまま，カーネルだけ実行イメージをを入れ替えることで，カーネルのソフトウェア若化も可能になる [10]．

5. 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，仮想化に伴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伴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，それをデバイスホストで処理する点で，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．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 層の呼び出しの度にホスト 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 [19], Unikernels [20], ClickOS [21] 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他，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．

ハードウェアの機能としてデバイスを仮想化するものに，SR-IOV [5], Moneta-Direct [22], [23] がある．SR-IOV は，単一のネットワークデバイスを，ハードウェアにより複数デバイスに見せる機能を提供することで，ゲスト OS カーネルからパススルーによる直接アクセスを可能にする．Moneta-Direct は，ハードウェアの機能としてストレージを仮想化し，ユーザプロセスからの直接アクセスを可能にする．どちらもハードウェアの機能により，単一デバイスを複数化するものであり，ソフトウェアで実現するストレージをメモリとして仮想化する手法とは異なっている．

SSD の高性能化にともなう，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ついて述べた研究に，[15], [16] がある．高性能 SSD を前提とすると，非同期アクセス処理のオーバヘッドが大きいため，同期アクセス処理を行った方が全体のオーバヘッドが小さいことを示している．詳細は，2.2 節で述べたとおりである．これらの研究は，仮想化環境におけるストレージアクセス手法については言及していない．

ストレージをメモリとして仮想化する手法は，単一レベルストアの基盤として使用可能である．EROS [24] のような既存の単一レベルストアシステムは，アクセスするオブジェクトをメモリ上にキャッシュする．ストレージをメモリとして仮想化する手法を用いることで，OS 側でそのようなオブジェクトのキャッシュ管理を行う必要がなくなり，仮想化環境に委任することができる．

6. 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，資源共有の効率化を目的とし，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について述べた．

© 2014 Information Processing Society of Japan

SSD の高性能化にともなう，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うけ，ストレージをメモリとして仮想化する手法は，ストレージの高速化を活かすことができると考えられる．さらに，仮想ストレージをメモリとして仮想化することで，仮想的に不揮発性メモリを構成することができ，ストレージとメインメモリ管理の融合等，さまざまな応用を持つ．

今後の課題としては，ストレージをメモリとして仮想化する手法の性能評価，および手法の応用があげられる．

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 (online), DOI: 10.1109/MC.2005.176 (2005).

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, ACM, pp. 164–177 (online), DOI: 10.1145/945445.945462 (2003).

[3] Russel, R.: lguest: Implementing the little Linux hypervisor, *Proceedings of the Linux Symposium*, 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 (online), DOI: 10.1145/1400097.1400108 (2008).

[5] PCI-SIG: Single Root I/O Virtualization, http://www.pcisig.com/specifications/iov/single_root/ (2007).

[6] Kivity, A., Kamay, Y., Laor, D., Lublin, U. and Liguori, A.: kvm: the Linux virtual machine monitor, *Proceedings of the Linux Symposium*, Vol. 1, pp. 225–230 (2007).

[7] Oikawa, S.: Virtualizing Storage as Memory for High Performance Storage Access (2014), Manuscript submitted for publication.

[8] 追川修一：Non-Volatile メインメモリとファイルシステムの融合，情報処理学会論文誌， Vol. 54, No. 3, pp. 1153–1164 (2013).

[9] 追川修一， 三木聡：Non-Volatile メインメモリを用いたチェックポイント・リスタートシステム，情報処理学会論文誌：コンピューティングシステム， Vol. 6, No. SIG 4(ACS 44), pp. 49–57 (2013).

[10] Oikawa, S.: Independent Kernel/Process Checkpointing on Non-Volatile Main Memory for Quick Kernel Rejuvenation, *Proceedings of International Conference on Architecture of Computing Systems*, ARCS '14, Springer, pp. 234–245 (2014).

[11] Josephson, W. K., Bongo, L. A., Li, K. and Flynn, D.: DFS: A file system for virtualized flash storage, *Trans. Storage*, Vol. 6, No. 3, pp. 14:1–14:25 (online), DOI: 10.1145/1837915.1837922 (2010).

[12] Akel, A., Caulfield, A. M., Mollov, T. I., Gupta, R. K. and Swanson, S.: Onyx: a protoype phase change memory storage array, *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, Berkeley, CA, USA, USENIX Association, pp. 2–2 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2002218.2002220⟩ (2011).

[13] 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, *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 226–227 (online), DOI: 10.1109/ISSCC.2013.6487711 (2013).

[14] Huffman, A. and Juenemann, D.: The Nonvolatile Memory Transformation of Client Storage, *Computer*, Vol. 46, No. 8, pp. 38–44 (online), DOI: http://doi.ieeecomputersociety.org/10.1109/MC.2013.223 (2013).

[15] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K. and Swanson, S.: Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories, *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, Washington, DC, USA, IEEE Computer Society, pp. 385–395 (online), DOI: 10.1109/MICRO.2010.33 (2010).

[16] Yang, J., Minturn, D. B. and Hady, F.: When poll is better than interrupt, *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, Berkeley, CA, USA, USENIX Association, pp. 1–7 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2208461.2208464⟩ (2012).

[17] Eiraku, H., Shinjo, Y., Pu, C., Koh, Y. and Kato, K.: Fast Networking with Socket-outsourcing in Hosted Virtual Machine Environments, *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, New York, NY, USA, ACM, pp. 310–317 (online), DOI: 10.1145/1529282.1529350 (2009).

[18] Jujjuri, V., Van Hensbergen, E., Liguori, A. and Pulavarty, B.: VirtFS - A virtualization aware File System passthrough, *Proceedings of the Ottawa Linux Symposium*, pp. 109–120 (2010).

[19] Cloudius-Systems: OSv: the operating system designed for the cloud, http://osv.io (2014).

[20] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S. and Crowcroft, J.: Unikernels: Library Operating Systems for the Cloud, *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, New York, NY, USA, ACM, pp. 461–472 (online), DOI: 10.1145/2451116.2451167 (2013).

[21] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. and Huici, F.: ClickOS and the Art of Network Function Virtualization, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, USENIX, pp. 459–473.

[22] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J. and Swanson, S.: Providing Safe, User Space Access to Fast, Solid State Disks, *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, New York, NY, USA, ACM, pp. 387–400 (online), DOI: 10.1145/2150976.2151017 (2012).

[23] Swanson, S. and Caulfield, A. M.: Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage, *Computer*, Vol. 46, No. 8, pp. 52–59 (online), DOI:

© 2014 Information Processing Society of Japan

http://doi.ieeecomputersociety.org/10.1109/MC.2013.222 (2013).

[24] Shapiro, J. S. and Adams, J.: Design Evolution of the EROS Single-Level Store, *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, Berkeley, CA, USA, USENIX Association, pp. 59–72 (online), available from ⟨http://dl.acm.org/citation.cfm?id=647057.713855⟩ (2002).

© 2014 Information Processing Society of Japan