Vol.2020-OS-148 No.3
2020/2/27

ハイパーバイザーとNVMを用いたメモリトレイサーの検討

市川 遼[1]　坂本 龍一[2]　中村 宏[2]　並木 美太郎[1]

概要：メモリアクセスのパターンを解析することは，アプリケーションのパフォーマンス改善において非常に重要である．しかしメモリアクセスは頻繁に発行され，そのログが膨大になってしまうため，実行時間に影響を与えずにトレースを得ることは困難である．そこで本発表では，ハイパーバイザと Intel Optane DC Persistent Memory を用いたメモリトレースシステムを提案する．BitVisor を拡張した LVisor によって，Optane DCPM の持つ高速な大容量ストレージを解析用途に利用する方法を検討した．

1. 背景

近年，不揮発メモリ (NVM:Non-Volatile Memory) が普及し，高速大容量なストレージとしての利用が期待されている．さらに，NVM を主記憶として利用する NVMM(Non-Volatile Main Memory) の登場によりバイトアドレッサブルな主記憶の大容量化・不揮発化が実現されている．中でも Intel Optane DC Persistent Memory[1] は DRAM と同様に DIMM スロットにさすことができる不揮発メモリであり，DIMM1 枚当たり，128GB から 512GB の容量を有する．DRAM と読み書き速度を比較すると 5 倍程度の性能差 [2] があるものの，大容量・不揮発の恩恵は大きい．

また，近年アプリの多様化に伴い，セキュリティを担保することが重要であり，様々なアプリの解析や保護のために VMM (Virtual Machine Monitor) を用いてゲスト OS を保護する技術が注目されている．従来の VMM は複数の VM を管理する機能を備えており，VMM を動作させるためのカーネルまたは OS が必須であることが多く，デバイスの仮想化など機能を複雑にする要因によって VMM そのものを巨大にし，潜在的なバグや脆弱性などを排除しきれない状態であった．そこで，近年は Thin-Hypervisor[4] と呼ばれる，単一の VM のみを管理し，ほぼ全てのデバイスをパススルーするという準パススルー型アーキテクチャによって，最小限の機能をもった VMM でゲスト OS を保護する技術が登場した．Thin-Hypervisor を利用することで，VMM の保護を受けながら物理マシンとほぼ同一の環境でデバイスを扱うことが可能になり，Thin-Hypervisor の参考実装である BitVisor をベースに，ユーザを保護するための研究が数多く行われてきた．[6][7]

Thin-Hypervisor は VMM が最小限であるために利用できるメモリ量が限られており，解析用途に取得したデータを活用するには VMM の内部に組み込んだ解析器で処理を行うか，外部のリソースで処理するために逐次転送する必要があった．取得したデータを転送する手段はネットワークが主であったが，デバイスドライバを経由すること，ネットワークそのものの帯域幅の制約を受けることから，トレースログなど，解析データを大量に生成する場合は実行速度への影響が課題として残っていた．例えば SimuBoost[3] ではゲスト OS のメモリトレースを行うことが可能であったが，トレースデータをネットワーク経由で保存するため，解析時のアプリの実行速度に課題があった．

そこで，本研究では NVM を Hypervisor から利用することによる新しい応用を検討する．解析データの保存に NVMM を用いることで，データの保存にかかる時間を大幅に削減し，解析時のアプリ実行の高速化が見込める．また，NVM に大容量のトレースデータが保存可能になるため，長期の実行時間におけるトレースを可能にする．

NVM と Hypervisor の協調を想定し，いくつかの応用例の検討を行う．さらに，Hypervisor に必要な新たな機能について検討する．本研究の貢献は，

Hypervisor からの NVMM の応用方法について検討
Hypervisor から NVM を利用することで可能となりえる応用について検討する．これらの検討を基に，Hypervisor に必要となる機能要件をまとめる．

NVM を利用するための API を設計
機能要件をもとに既存の Hypervisor を拡張する形で，新たに必要となる API を検討する．

基礎評価
Hypervisor とゲスト OS 間で NVM を用いて通信を行

[1] 東京農工大学
Tokyo University of Agriculture and Technology
[2] 東京大学
The University of Tokyo

う評価を行い，基礎的な評価を示す．
である．

2 章では関連研究や NVM などの関連技術について示し，3 章で NVM と Hypervisor の協調例の検討を示す．4 章に協調のための要件を示す．5 章では設計と実装を示し，5 章で評価を示す．7 章でまとめを示す．

2. 関連研究と関連技術

2.1 Intel Optane

Intel Optane (以下 Optane) は Intel および Micron によって開発された 3 次元積層不揮発性メモリ技術 3D XPoint を用いたストレージデバイスである．SSD などの一般的な NVM ストレージに比べてアクセス速度に優れており，特にランダムアクセスの性能が非常に良い．これらの特徴を活かして 2 次記憶のキャッシュとして用いられることが多く，2 次記憶が低速なストレージでも PC の動作を高速にすることができる．また，不揮発性を活かして OS の起動高速化や，アプリケーションの高速化などへの利用が期待されている．

2.2 Intel Optane DC Persistent Memory

Intel Optane DC Persistent Memory (以下 Optane DCPM という) は Intel によって開発された 3 次元積層メモリ技術をベースとした Optane メモリのうち，データセンタなどでの利用を想定されたモデルである．Optane に比べてより高速かつ大容量 [*1] であり，大規模なマシンでの利用を想定されている．Optane は PCI デバイスだったのに対して，Optane DC は DIMM の規格で設計されており，メモリモジュールとして扱うことができる (**図 1**) ため，CPU から PCI を介さずに DMA (Direct Memory Access) によって高帯域アクセスを可能にしている．またドライバを必要とせず，直接 CPU 上からメモリアクセスとして扱うことが可能であるため，カーネルが存在しないベアメタル環境からでも利用可能である．

2.3 Intel VT

Intel VT (Intel Virtualization Technology) とは，Intel 製の CPU に搭載された仮想化支援機構である．CPU の仮想化を支援するための VMX root mode/VMX non-root mode の追加に加え，VM のページテーブルの管理を支援するための EPT (Extended Page Table) などが存在する．本研究では，メモリトレーサに必要と思われる機能を検討していく．

2.3.1 EPT

EPT (Extende Page Table) とは，Nehalem 以降の CPU に追加された，Intel VT におけるゲスト VM のページテーブルを管理するための機構である．EPT が存在しない従来の仮想化においては，ページテーブルエントリの内容を書き換えるたびに VMexit が発生するためオーバーヘッドとなっていたが，EPT によりハンドリングの処理が省かれ高速化につながる．EPT はページテーブルのキャッシュのような役割を果たし，EPT に存在しないページエントリに対するアクセスまたは設定した属性に違反するアクセスが発生した場合に VMexit を発生させ，VMM が適切にマッピングして再度 VMEnter を行う．

[*1] 一般向けの Optane が一モジュールあたり 32GB までなのに対し，Optane DCPM は 512GB まで存在する．

図 1 DIMM (写真上) と Optane DCPM (写真下) の比較

2.3.2 EPT page fault

EPT はページテーブルを管理するための機能であり，必要な構造体などは VMM が用意する必要がある．EPT に登録してあるエントリに違反すると VMexit が発生することを利用し，page fault を発生させて VMM 側でハンドルし，アドレスに対するアクセスをフックするのに用いる手法がある．しかし EPT page fault が頻繁に発生すると VMX root と VMX non-root 間のコンテキストスイッチが発生するため，VMM のパフォーマンスが低下するという問題が発生する．

2.3.3 PML

PML (Page-Modification Logging) とは EPT に付随する機能の一つであり，ゲスト VM 内におけるメモリ書き込みをトレースするための機能である．EPT で書き込み処理が発生した場合に，書き込みが発生したページのアドレスを特定の領域に書き込む．(**図 2** 参照)

PML はゲスト VM 内のメモリ書き込みをを page fault 無しに実現可能なため，低コストでメモリ書き込みのトレースを行うことができる．PML は 1 エントリにつき 8 バイト分の領域を 512 個分確保する必要があり，書き込みアクセスは最大で 512 つのアドレス分だけ保存される．エントリを全て使い果たしたときに VMexit が発生し，VMM がデータの保存などを行った後に再度 VMEnter する必要がある．

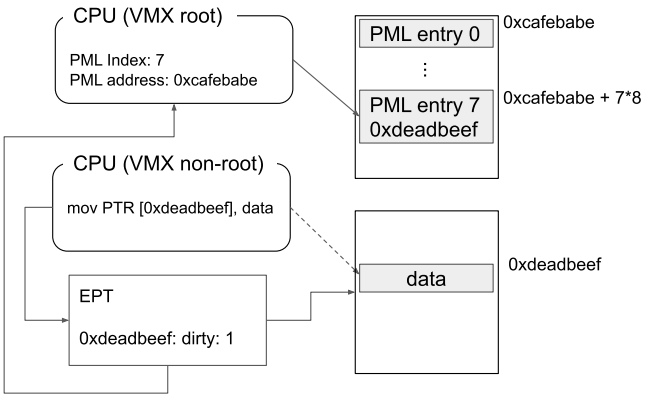

図 2　PML (Page-Modification Logging) の概略図

3. ハイパーバイザーによる NVM の有効利用

本章では，2 章で示した NVM の特性を生かす Hypervisor によって可能となる応用について検討を示す．

3.1 NVM の大容量を生かした大容量トレース

2020 年 2 月現在，第 2 世代 Xeon スケーラブル・プロセッサで利用できる代 1 世代の Optane DCPM は 128GB, 256GB，512GB の容量を有する．最大で 12 枚の Optane DCPM を 1 台の計算機に搭載できるため，512GB の Optane DCPM を 12 枚搭載した場合，6TB ものメモリを搭載することができる．一般にトレースデータは大規模な容量となるが，Optane DCPM を用いることで，従来よりも長期間のトレースが可能になると考えられる．

3.2 DRAM 性能に近い読み書き性能を生かした高速トレース

Optane DCPM は DRAM と比較すると 5 倍程度レイテンシが大きいが，SSD などの一般的な 2 次記憶と比較すると高速に読み書きが可能である．そのため，トレースデータを Optane DCPM に保存することによって，トレースデータ保存を高速化できる．

3.3 不揮発性を生かしたトレースデータのセロコピー

一般にトレースデータの解析にはトレースを行うターゲットの物理マシンと解析用のマシンの 2 台を用いる．そのため，トレースデータをターゲットの物理マシンから解析用マシンにコピーする必要がある．[3] では Xen でトレースしたデータをイーサネットを用いることで解析用マシンにトレースデータの転送を行っている．しかしながら，トレースデータのサイズは大きいためコピーに多くの時間が必要になる．トレースデータをターゲットマシン上に保存し，トーレース終了後に解析を行うことで，トレースデータのコピーをなくすことができるが，トレース中にターゲットがクラッシュしたり不安定になるため，現実的ではない．トレース後に再起動を行うことも考えられるが，トレースデータを DRAM に保存した場合，データが失われる可能性がある．

一方で，不揮発性を有する NVM にトレースデータを保存することで，トレースデータを永続化することが可能となる．トレース後に再起動を行い，解析を行う OS を起動することによってトレースデータの移動を行うことなく，解析が可能となる．これにより，解析に要するデータ転送時間を短縮できる．

しかしながら，トレースを行う際はアプリケーションの条件を変えつつ何度もトレースを行うことが考えられる．単純に Optane DCPM の先頭のアドレスからトレースデータを保存することも可能であるが，トレースを何度も繰り返す場合，何らかの形でトレースデータに名前をつける仕組みが必要であると考えられる．

3.4 メタデータ利用による詳細な分析

1 つのアプリケーションバイナリを対象として解析を行う場合，ターゲットアプリのシンボル情報などのメタデータが必要となる．従来は，ゲスト OS 上のアプリのシンボル情報を実行時に Hypervisor から取得することで，実行されている関数名を取得していた．

一方で近年のクラウド環境を代表するマイクロサービスでは，1 つの物理ノード上で多数のアプリが動作している．また，このような多数のアプリが同時に実行されうる環境での性能解析も重要である．このように，多数アプリを同時に解析したい場合，上記のシンボル情報などのメタ情報取得が頻繁に行われることになり，解析時の性能低下が懸念される．さらにマイクロサービス等のアプリは常に通信を行っており，トレース時の性能低下によってタイムアウトが生じてしまい，アプリの実行ができなくなる恐れがある．そこで，これらのメタ情報をターゲットアプリの実行前に静的に解析しておき，メタデータを不揮発領域に保存することで，解析時のオーバヘッド削減を期待できる．

しかしながら，アプリごとにメタデータ情報を持つ必要があり，メタデータが複数必要となる．単に不揮発領域にメタデータを保存することも考えられるが，多数のアプリを柔軟に解析するためには，複数のメタデータを効率的に管理するための仕組みが必要となる．

3.5 チェックポインティング

Page へのすべての書き込みをトレースできれば，ゲスト OS の内容をコピーすることができる．また，時系列毎にトレースできれば，ゲスト OS を任意の時刻の状態に復元することも可能である．意図的に VMexit を起こさせゲスト OS の書込みをハンドリングすることも可能であるが，オーバヘッドが大きい．そこで，Intel PML(Page-Modification Logging) を用いることで，ページに対する書き込みを CPU の機能として効率的にハンドリングすることが可能となる．

一方で，すべての書込みを保存するには大容量のメモリが必要である．そこで，大容量の Optane DC PM を用いることで，チェックポイントデータの長期間が可能となる．また，保存を高速化することによる，アプリ実行の高速化や，チェックポイントデータの永続化による，信頼性向上に期待できる．

4. 要件

3 章の応用を実現するために必要な Hypervisor の要件について示す．

4.1 プログラマブル VMM

既存の VMM は，VM の作成・設定・管理，外部ハードウェアの仮想化，VM 間仮想ネットワークの構築は独立環境の作成という目的に直接寄与するものである．一方で，提案する応用を想定する場合，VM を用いて作成したゲスト OS について，VM の外側または内側でなんらかの処理を加えることが必要となる．すなわち，VMexit の条件をプログラマブルに設定でき，且つ，その後のハンドリング処理をプログラマブルにできる必要がある．LibVMI[5] は VMM 側から制御を行うためのインタフェースを提供するが，制御を行う部分は C のコードとしてプログラミングする必要があるため，より柔軟な制御が必要である．

4.2 Hypervisor への汎用的なアプリケーション移植方法の検討および確立

NVM 上に複数のトレースデータやメタ情報を効率的に保持するためにはファイルシステムのようにデータの仮想化が行えることが重要である．また，SQLite のような DB を VMM 内で利用できるようにすることで，効率的な解析が可能となる．

一方で，Thin-Hypervisor の多くはベアメタルで状で動作する VMM のため，Linux で動作することを前提としている外部ライブラリはそのままでは動作させることが不可能であり，移植に際して解決すべき課題がある．

そこで，これらの課題を整理し汎用的に移植可能な方法を確立することにより，LibVMI やインタプリタ処理系のみならず様々なアプリケーションの移植を容易にする．

4.3 NVM アクセスインタフェース

詳細は実装に示すが Optane DCPM は App Direct Mode を利用した場合，CPU のアドレススペースの一部にマップされる．この領域に対してポインタなどを用いて直接読み書きすることで NVM への読み書きが可能となる．一方で Optane DCPM は非常に高性能であり，インターリーブ構成による高性能化，Namespace によるパーティショニング等の機能を提供しており，Optane や CPU の構成を同一にしても，CPU にマップされる NVM のアドレスは異なる可能性がある．そのため，VMM は NVM の構成情報を読み解き，NVM がマップされたアドレスを起動時に取得する必要がある．

4.4 NVM を介したゲスト OS との通信

アプリのメタデータであるシンボル情報はアプリが動作するゲスト OS 上にあり，シンボル情報を VMM に引き渡すことでアプリの解析が可能となる．そのため，ゲスト OS が VMM にシンボル情報を渡す必要がある．また，この際 NVM 領域にシンボル情報を書き出すようにすることで，物理マシンの再起動後も本領域を再利用することができる．

4.5 PML(Page-Modification Logging) ハンドリング

PML を用いることで，主記憶のチェックポイントを実装することができる．PML は CPU レベルでページに対する書込みをトレースし専用に確保した主記憶領域にトレース情報を書き出す．バッファがあふれた際には VMexit が発生するため VMM は PML による VMexit であることを検知し，ハンドラにてトレース情報を NVM に書き出すことで，NVM を用いてチェックポイントを実現することができる．

5. 設計と実装

本章では，4 章の要件を基に，設計と実装を示す．

5.1 プログラマブル VMM の実装

プログラマブルな VMM を実現するために本研究では，ベアメタルで動作する Thin-Hypervisor にスクリプト言語処理系機能を組み込むことによって，VMM 上で任意の処理を動的に実行可能にする．具体的には Thin-Hypervisor には BitVisor を用い，Lua 言語処理系を組み込む．

5.2 NVM 先頭アドレスの取得方法

NVM に対する読み書きを行う場合，CPU のアドレス空間にマップされた NVM のアドレスを取得する必要がある．本節では利用する Optane DCPM のモードについて説明し，実際にアドレスを取得するための方法を示す．

Memory Mode と App Direct Mode

Optane DCPM はメインメモリとして扱うモード (Memory Mode) と，ストレージデバイスとして扱うモード (App Direct Mode) が存在する．Memory Mode と App Direct Mode を混在させて使うことも可能であり，128GB の Optane DCPM について，50%をメインメモリとして，残りの 50%をストレージとして扱う，という例ではメインメモリ領域が 64GB 割り当てられ，OS から利用可能なストレージが 64GB となる．Optane DCPM の設定は物理レベルで行われており，メインメモリ領域を増やした場合は BIOS

コード 1　SRAT テーブル

```
1  ...
2  [EAAh 3754    4]              Proximity Domain : 00000001
3  [EAEh 3758    2]                    Reserved1 : 0000
4  [EB0h 3760    8]                 Base Address : 0000000C80000000
5  [EB8h 3768    8]               Address Length : 0000001F80000000
6  [EC0h 3776    4]                    Reserved2 : 00000000
7  [EC4h 3780    4]       Flags (decoded below) : 00000005
8                                       Enabled : 1
9                                  Hot Pluggable : 0
10                                  Non-Volatile : 1
11 [EC8h 3784    8]                    Reserved3 : 0000000000000000
```

から見える情報が書き換わり，CPU がブートローダを起動する段階で既に利用可能になっている．

Optane DCPM へのアクセス方法

Optane DCPM へアクセスするためには，物理アドレス空間上のどの位置にマッピングされているかどうかを知る必要がある．NVDIMM に関する情報は，Memory Mode の場合は ACPI の NFIT テーブルに，App Direct Mode の場合は SRAT テーブルに書き込まれ，App Direct Mode を扱うときは OS が ACPI から得た情報を頼りに初期化などの処理を行う．ゲスト OS と VMM の共有ストレージとして用いる場合は，OS に Optane DCPM をストレージとして認識させる必要があるため，以降では App Direct Mode での利用を前提とする．

VMM で NVDIMM のアドレスを取得するためには，ACPI テーブルのパースをする処理を書く必要があるため，今回はゲスト OS を Linux であると仮定して，sysfs の ACPI テーブルから物理アドレスを取得，VMM に通知する．Linux で Optane DCPM を App Direct Mode で使うためには，`ndctl` コマンドを用いて NVDIMM の namespace を作成する必要がある．

NVDIMM の namespace 作成コマンド

```
ndctl create-namespace
```

namespace を作成することによって，NVDIMM を App Direct Mode で利用するための初期化が行われ，ブロックデバイスとしてアクセス可能になる．

次に Linux のツールである `iasl` コマンドを用いて，SRAT テーブルをパースするとコード 1 の出力が得られる．

`Non-Volatile : 1` となっている部分が，当該エントリが NVDIMM であることを示している．`Base Address` は NVDIMM がマッピングされている物理アドレスを指しており，`Base Address` から `nd_pfn_sb` 構造体[*2]の位置を探し，`Base Address` に dataoff を加算することによってブロックデバイスのマッピング先アドレスが得られる．

表 1　評価環境

	構成
CPU	Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz
NVM	Intel Optane DC Persistent Memory 128GB x 2
DRAM	DDR4 64GB
VMM	BitVisor (tags:tips)
OS	Ubuntu 18.04

6. 性能評価

評価では Bitvisor を改良し，VMM から NVM を利用した際の簡単な性能評価を示す．

6.1 評価環境

評価には表 1 の環境を用いた．

6.2 Read/Write 性能

VMM から NVM の領域に対するデータの読み書きを行ったときのベンチマークを測定する．BitVisor ではゲスト VM の物理アドレスにアクセスする関数が用意されており，**表 2** の通りである．

BitVisor では 1, 2, 4, 8byte 単位でのみしかアクセスする手段がないが，NVM の性能を引き出すためには更に大きい単位でデータをコピーする必要がある．また，表 2 に挙げた関数はいずれもゲスト VM へのページマッピングを経由するため，オーバーヘッドが発生する．しかし NVM は物理アドレス上に直接マッピングされるため，VMM とゲスト VM において同じアドレスでアクセスすることが可能である．

そ こ で 任 意 の バ イ ト 数 を コ ピ ー す る 関 数

[*2] https://github.com/torvalds/linux/blob/master/drivers/nvdimm/pfn.hL16

関数	機能
`read_gphys_b(u64 phys, void* data, u32 attr)`	phys が指すアドレスから data が指す領域に 1byte 書き込む
`read_gphys_w(u64 phys, void* data, u32 attr)`	phys が指すアドレスから data が指す領域に 2byte 書き込む
`read_gphys_l(u64 phys, void* data, u32 attr)`	phys が指すアドレスから data が指す領域に 4byte 書き込む
`read_gphys_q(u64 phys, void* data, u32 attr)`	phys が指すアドレスから data が指す領域に 8byte 書き込む
`write_gphys_b(u64 phys, u32 data, u32 attr)`	phys が指すアドレスへ data の下位 1byte を書き込む
`write_gphys_w(u64 phys, u32 data, u32 attr)`	phys が指すアドレスへ data の下位 2byte を書き込む
`write_gphys_l(u64 phys, u32 data, u32 attr)`	phys が指すアドレスへ data を書き込む
`write_gphys_q(u64 phys, u64 data, u32 attr)`	phys が指すアドレスへ data を書き込む

表 2　BitVisor で提供されているメモリアクセス用関数

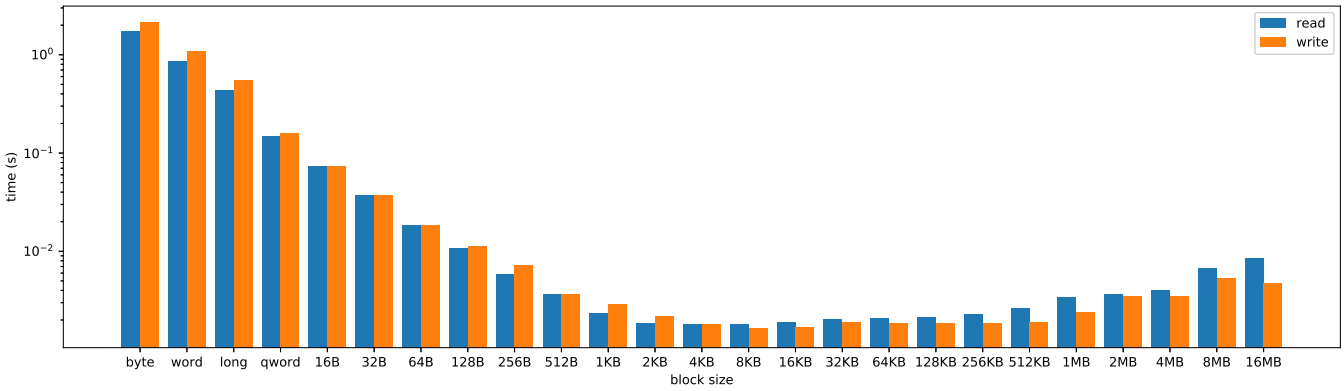

図 3　VMM から NVM へのアクセス (16MB)

`read_hphys_nbytes` および `write_hphys_nbytes` を実装した．これらの関数は `read_gphys_b` のようにページのマッピングを行わずにアドレスを直接解決するため，NVM の利点である DMA を活かし，より高速な読み書きを可能にする．

16MB のデータを VMM から読み書きするのにかかった時間を図 **3** に示す．横軸はブロックサイズを示しており，byte, word, long, qword はそれぞれ BitVisor 標準の関数を用いた結果である．

結果より，ブロックサイズが大きくなっていくと指数関数的に時間が短縮されていくが，4KB を過ぎると徐々に増加していることがわかる．4KB は実験した VMM におけるページサイズであり，マッピングしたページを全て転送に使い切ることから効率が上がったと考えられる．また，8KB や 16KB など数ページ分に相当するブロックサイズではパフォーマンスがほぼ変わらないものの，確保するページ数が増えるにつれて VMM のメモリ管理機構のオーバーヘッドが加わってきたと考えられる．今回実験した VMM である BitVisor のメモリ管理機構は一度に多くのページ数を確保することを前提としないため，性能低下が見られた．

6.3　NVM を介したゲスト OS と NVM 間のデータ転送

ゲスト OS から VMM にデータを転送するときに要する時間を，以下の手順を用いて計測する．

(1) ゲスト OS であるサイズのランダムデータを生成する
(2) 生成したデータのサイズを NVM に書き込む

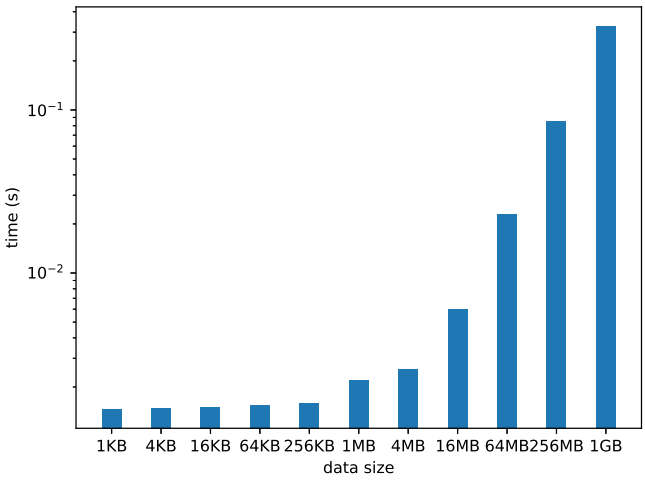

図 4　ゲスト OS と VMM 間でのデータ転送時間

(3) ゲスト OS で時間計測開始
(4) ゲスト OS から VMMCall を用いて VMM の関数を呼び出す
(5) VMM は書き込まれたサイズ情報を取得し，その分だけランダムデータを読み込む
(6) VMM がランダムデータを読み終わった段階で VMMCall の制御がゲストに戻る
(7) ゲスト OS で計測終了

6.2 の結果より，ブロックサイズには 4KB (0x1000) が最適であるとし，全体のサイズを変えながらデータの転送にかかる時間を計測する．計測結果を図 **4** に示す．

ゲスト OS と VMM 間でのデータ転送にかかる時間は

1GB の場合でも 0.327 秒と，極めて高速であった．通常のストレージに比べて遥かに高速にデータを転送可能なことが確認できる．

7. まとめ

本研究では NVM の特性を生かす Hypervisor の設計についての検討を示した．Intel Optane DCPM の利用を想定し，Bitvisor を拡張する方法について示した．また，NVM を介したゲスト OS と VMM の通信について初期評価を示した．

謝辞 本研究の一部は基盤研究 (B)17H01708 と JST さきがけ JPMJPR19M3 の助成を受けたものである．

参考文献

[1] Intel® Optane™ DC Persistent Memory Product Brief. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[2] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, Vol. abs/1903.05714, , 2019.

[3] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, Houston, Texas, March 16 2013.

[4] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, et al. Bitvisor: a thin hypervisor for enforcing i/o device security. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 121–130, 2009.

[5] Haiquan Xiong, Zhiyong Liu, Weizhi Xu, and Shuai Jiao. Libvmi: A library for bridging the semantic gap between guest os and vmm. In *CIT*, pp. 549–556. IEEE Computer Society, 2012.

[6] 小川夏樹, 大山恵弘ほか. Advisor: ゲスト os の操作に連動した広告を表示するハイパバイザ. 研究報告 システムソフトウェアと オペレーティング・システム (OS), Vol. 2011, No. 5, pp. 1–7, 2011.

[7] 大月勇人, 瀧本栄二, 齋藤彰一, 毛利公一ほか. マルウェア観測のための仮想計算機モニタを用いたシステムコールトレース手法. 情報処理学会論文誌, Vol. 55, No. 9, pp. 2034–2046, 2014.