Vol.2017-OS-141 No.15
2017/7/27

All-Flash Hyper-Converged Infrastructureに向けたアーキテクチャーの提案

加藤 純[1]　鈴木 康介[1]　岩田 聡[1]　大辻 弘貴[1]　佐藤 充[1]　吉田 英司[1]

概要：従来の仮想化基盤にストレージ機能を統合したHyper-Converged Infrastructure (HCI) では，ストレージデバイスをすべてSSDとするAll-Flashが注目されている．本稿では，HCIのアーキテクチャーをストレージの構成から2種類に分類して分析を行い，All-Flash化に伴うデバイス性能の向上によってI/O仮想化がオーバーヘッドとなることを明らかにした．このオーバーヘッドを解決するため，従来の仮想マシンとI/O仮想化のないコンテナをハイブリッドしたアーキテクチャーを提案する．仮想マシンとコンテナでI/O性能が異なるため，このようなアーキテクチャーを活用するためにはI/O性能差を前提としたI/Oスケジューリングが課題であり，今後はこの課題に取り組む予定である．

キーワード：All-Flash，Hyper-Converged Infrastructure，仮想マシン，コンテナ，I/Oスケジューリング

1. はじめに

Hyper-Converged Infrastructure (HCI) [1], [2], [3] は従来の垂直統合型基盤であるConverged Infrastructureとは異なり，1つのコモディティーサーバー上にコンピューティング機能とストレージ機能を統合した仮想化基盤である．HCIは外付けの共有ストレージ装置を持たないShared Nothing型のアーキテクチャー [4] であり，サーバー内蔵のストレージデバイスをソフトウェアにより仮想的なサーバー間共有ストレージとして利用する．このように，従来の仮想化基盤で前提とされていた共有の外付けストレージ装置とそれに付随する専用のストレージネットワークが不要となるため，省電力・省スペースや運用保守の効率化による総所有コストの削減が期待されている．

HCIではこれまでSSD (Solid State Drive) をキャッシュとしたHDD (Hard Disk Drive) ベースの設計が主流であったが，SSDの価格下落に伴い容量単価や性能単価の観点でSSDがHDDを逆転した [5] ことから，SSDのみでストレージを構成するAll-FlashのHCIも台頭し始めている．SSDはページ単位の読み書き，ブロック単位の消去とHDDの磁気ヘッドの動きによる読み書きとは原理的に動作が異なり，また性能に関してもSSDが1桁から2桁以上高速なため，All-Flashの利点を最大限享受するためには単純にHDDをSSDに換装するのではなく，All-Flash向けにアーキテクチャーを刷新することが重要である [6], [7], [8], [9]．実際に，外付けストレージではAll-Flash向けにアーキテクチャーを刷新した製品 [5], [10] が登場しており，HCIでも同様なアーキテクチャーレベルの再検討が必要である．

本稿では，All-Flash HCIの設計に向けて，ストレージ機能を提供するコントローラーの構成からアーキテクチャーを仮想マシン型とホスト型の2種類に分類，性能分析を行った．その結果，仮想マシン型はiSCSIによるVM間通信のオーバーヘッドが大きいためホスト型と比較して3分の1以下の性能であり，コントローラーにスケーラビリティー上のボトルネックがあることも判明した．ホスト型は仮想マシン型と比較するとコントローラーに対して直接I/Oを発行できるため低レイテンシーであり，VM数に比例してIOPSもスケールするが，VMあたりのレイテンシーとIOPSはストレージデバイスであるSSDの半分以下となることがあり，All-Flash化によりデバイス性能が向上したことでI/O仮想化のオーバーヘッドが顕在化した．

この性能分析の結果を踏まえて，All-Flash HCIに向けて仮想マシンとコンテナのハイブリッド型アーキテクチャーを提案する．従来ではI/O性能不足によりHCI基盤に集約できなかったI/O-intensiveなアプリケーションもI/O仮想化のオーバーヘッドがないためにSSDの性能を引き出すことができるコンテナを用いることでHCI基盤への集約が可能となる．このようなI/O性能が異なる仮想化方式のハイブリッドアーキテクチャーを活用するためには，仮想化方式によるI/O性能差を考慮したI/Oスケジューリングが課題であり，今後はこの課題に取り組む予定である．

[1] 株式会社 富士通研究所

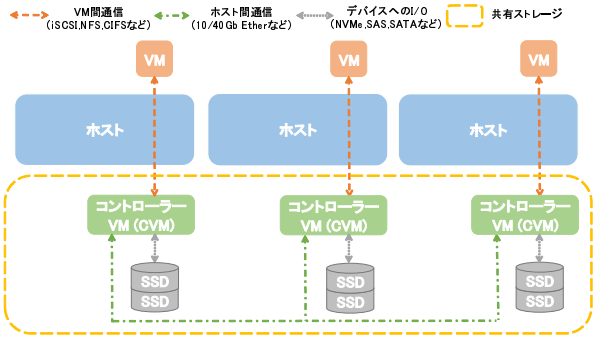


図 1　仮想マシン型の HCI アーキテクチャー

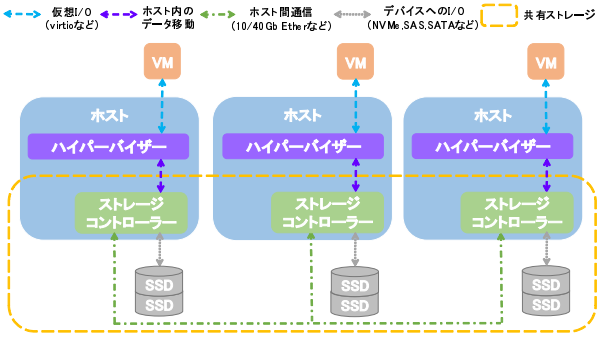


図 2　ホスト型の HCI アーキテクチャー

2. HCI の分析

2.1 アーキテクチャーの分類

HCI では，クラスター内の複数のサーバーにまたがって，サーバー内蔵のストレージデバイスをバックエンドデバイスとして仮想的な共有ストレージを提供するコントローラーがあり，このコントローラーが仮想マシンで実現されているか，ホスト内のコンポーネントとして実現されているかによりアーキテクチャーが異なる．本稿では，前者を仮想マシン型，後者をホスト型と定義することにより，HCI のアーキテクチャーを 2 つに分類する．

本稿で定義する仮想マシン型のアーキテクチャーは図 1 である．このアーキテクチャーでは，クラスターに参加するホスト 1 つにつきコントローラー VM (CVM) と呼ばれるシステム用の仮想マシンが 1 つあり，クラスター内の複数のコントローラー VM がホスト間ネットワークを通して協調することで，仮想的な共有ストレージを提供する．ユーザーの VM はこの共有ストレージに対して iSCSI，NFS，CIFS などのストレージプロトコルを通じてアクセスできる．これらのプロトコルを通じて共有ストレージに対して I/O が発行されると，まずユーザーの VM と同一のホスト上にある CVM に対して VM 間通信で I/O が送られる．I/O を受け取った CVM はクラスター内でのデータレイアウトを考慮してこの I/O 処理を担当する CVM を決定して，必要であればホスト間通信で I/O を転送する．CVM にはストレージデバイスである SSD がパススルーで接続されており，I/O 処理を担当する CVM はこの SSD に対して I/O を行う．

ホスト型のアーキテクチャーは図 2 が示すように仮想マシン型のような CVM はなく，ホスト内のストレージコントローラーと呼ばれるコンポーネントによって共有ストレージを実現する．共有ストレージはハイパーバイザーによって仮想化され，ユーザーの VM からは仮想デバイスとしてアクセスできる．この仮想デバイスに対して I/O が発行されると，ハイパーバイザーが I/O をハンドリングしてストレージコントローラーに対して I/O を通知する．ストレージコントローラーでの処理は仮想マシン型の CVM と同様に，データのレイアウトからこの I/O 処理を担当するストレージコントローラーを決定して，必要ならホスト間通信で I/O を転送した後に，I/O 処理を担当するストレージコントローラーで SSD に対して I/O を行う．

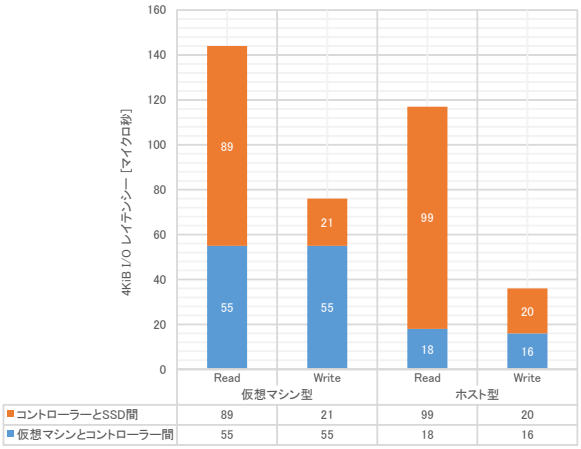
表 1　評価環境

物理マシン	
型版	Supermicro SYS-2028U-TN24R4T+
CPU	Intel Xeon E5-2697 v4 2.30GHz 36 コア (Hyper-Threading 有効), 2 ソケット
メモリー	PC4-17000 2133MHz DDR4 64GiB, 8 枚
NIC	Intel 10G Ethernet X550T, 4 枚
SSD	Intel NVMe SSD DC P3700 800GiB，24 本
仮想マシン	
vCPU	4
メモリー	8GiB
ソフトウェア	
OS	CentOS 7.3 (3.10.0-514.16.1.el7)
ハイパーバイザー	KVM (1.5.3-126.el7_3.6)
I/O 負荷	fio (2.2.8-2.el7)

仮想マシン型とホスト型の違いは，共有ストレージを扱うコントローラーとハイパーバイザーの結合性である．仮想マシン型のようにコントローラーを VM として独立させてハイパーバイザーとの結合を疎にすることで，ハイパーバイザーを問わない可搬性の高い HCI が実現できる．これに対して，ホスト型のように密結合にすると可搬性が犠牲になる代わりに VM 間通信がなくなるため高い I/O 性能を発揮できる．このように仮想マシン型とホスト型では可搬性と I/O 性能がトレードオフの関係にあり，All-Flash に向けた HCI のアーキテクチャーを検討する上では定性的な可搬性比較に加えて定量的な I/O 性能比較が必要である．そのため，以下では仮想マシン型とホスト型で I/O 性能比較を行い，定量的な面からも検討を行う．

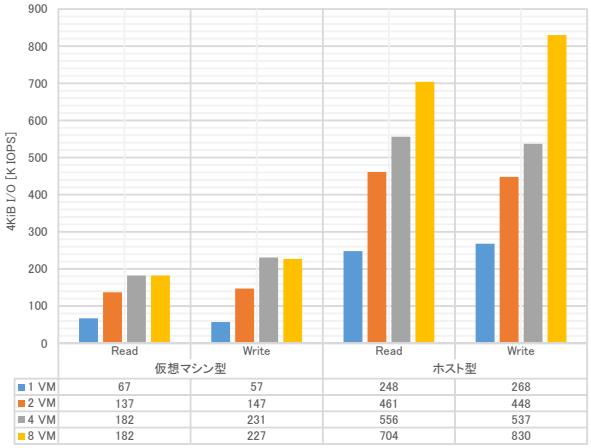
2.2 アーキテクチャーの性能比較

評価は表 1 の環境で行い，ハイパーバイザーとして KVM[11] を用いた．仮想マシン型では，プロトコルの点から高速な iSCSI を用いてユーザーの VM1 つにつき CVM



	仮想マシン型		ホスト型	
	Read	Write	Read	Write
コントローラーとSSD間	89	21	99	20
仮想マシンとコントローラー間	55	55	18	16

図 3 単発 I/O レイテンシーの内訳



	仮想マシン型		ホスト型	
	Read	Write	Read	Write
1 VM	67	57	248	268
2 VM	137	147	461	448
4 VM	182	231	556	537
8 VM	182	227	704	830

図 4 VM 数を増やしたときの合計 IOPS

上で iSCSI ターゲットを 1 つ作成して接続した．ホスト型では，ユーザーの VM からのアクセスのために virtio フレームワーク [12] を用いた準仮想化ブロックデバイスを作成した．この準仮想化ブロックデバイスではデバイスエミュレーションを行う QEMU[13] のジャイアントロックに伴う性能低下を回避するため data-plane[14], [15] の設定を行った．実際に，SSD の代わりに I/O を何も行わない Null ブロックデバイス [16] を用いて data-plane による準仮想化ブロックデバイスの性能比較を行ったところ，4KiB Read/Write のレイテンシーが 48/59 マイクロ秒から 18/16 マイクロ秒へと 3 分の 1 以下，IOPS が 172/180K IOPS から 395/406K IOPS と 2 倍以上に改善されており，この結果については 2.3 節で再び議論する．

図 3 はユーザーの VM から 4 KiB の I/O を 1 多重で発行した際のレイテンシーとその内訳である．内訳は，仮想マシン型の場合は iSCSI 経由で CVM，ホスト型の場合は仮想 I/O 経由でストレージコントローラーに I/O が行われるまでと，それらコントローラーと SSD 間の 2 つである．後者に関しては SSD のレイテンシーがそのまま反映されて大きな差は見られないが，前者に関しては仮想マシン型とホスト型で 3 倍近い差となっており，この差がそのままアーキテクチャーごとの性能差となって表れている．この原因は，仮想 I/O のように仮想マシンとホストで直接データの受け渡しを行わず TCP/IP を含む iSCSI のプロトコルオーバーヘッドもあるが，仮想マシン型の VM 間通信処理では送信側と受信側の両方でホストと仮想マシン間のコンテキストスイッチ [17], [18], [19] が必要であり，このコンテキストスイッチの回数がホスト型では 1 回であったのに対して仮想マシン型では 2 回に増えたことが性能差の主要因と考えられる．I/O の受け渡しを行うだけのホスト型でも 16 から 18 マイクロ秒かかっており，ストレージ処理とネットワーク処理の違いはあるがコンテキストスイッチの回数から少なくともこの 2 倍近いレイテンシーが想定されるからである．

図 4 は 128 多重で 4KiB の I/O を発行するユーザーの VM 数を同一ホスト上で最大 8VM まで増やした場合の全 VM の合計 IOPS である．1VM 時の結果から IOPS もレイテンシーと同様にホスト型は仮想マシン型と比べて 3 倍以上の IOPS であり，ホスト型は VM 数を増やすことで合計 IOPS がスケールするが，仮想マシン型は 4VM までしかスケールしないため性能差は開く一方である．この原因としては，上記のコンテキストスイッチ回数の増加による性能低下に加えて，CVM がスケーラビリティー上のボトルネックとなっているためである．仮想マシン型では同一ホスト上のすべての VM が 1 つの CVM に対して VM 間通信で I/O を行うため CVM の処理で律速されやすい．実際に，仮想マシン型は 4VM までしかスケールしなかったが，これは表 1 が示すように CVM を含む全仮想マシンの vCPU 数を 4 に設定したためで，CVM での処理が vCPU 数に律速されたからである．

2.3 All-Flash HCI に向けた考察

仮想マシン型はハイパーバイザーを問わない点で可搬性に優れるが，2.2 節の性能比較からレイテンシーと IOPS のどちらにおいてもホスト型の 3 分の 1 以下の性能であり，CVM によって IOPS が律速されることが分かった．CVM はユーザー VM が動作するホスト上での動作を前提とするため，CVM に CPU や Memory などのリソースを大量に割くことはユーザー VM に提供するリソースの削減につながるため期待できない．今後のハードウェアの進歩に伴う CPU コア数やメモリー容量の増加によりホスト当たりの VM 数が増加したとしても，同様に CVM にリソースを大量に割くことができず，逆に VM 数が増えることで CVM がボトルネックになりやすく，性能面で将来性が乏しい．複数の異なるハイパーバイザーを前提とする HCI であれば，仮想マシン型の可搬性を活かして 1 つだけの実装でハイパーバイザーに依存せず HCI を構築できる点を評価しつつも，本稿では，All-Flash HCI としては性能面の懸念

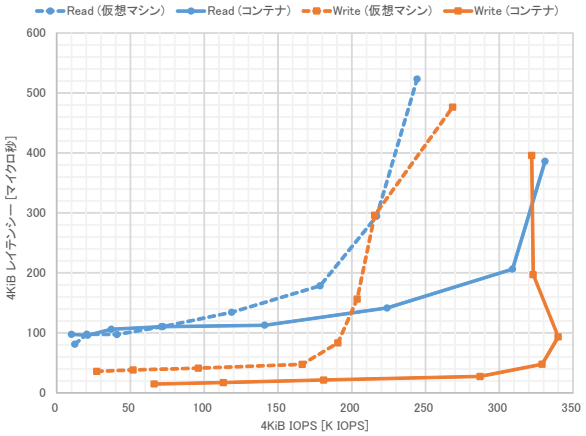


図 5　仮想マシンとコンテナの 4KiB I/O 性能

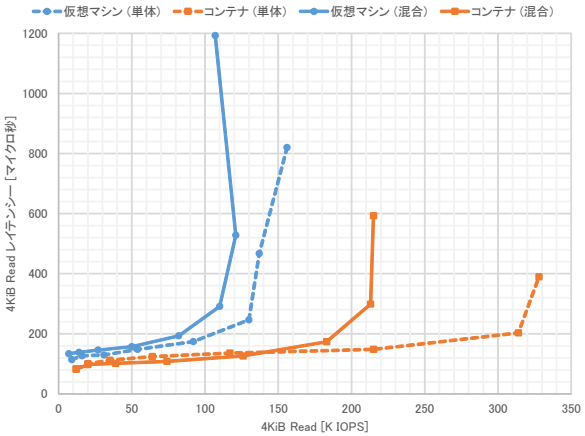


図 6　仮想マシンとコンテナの I/O 競合時の性能

が大きいためホスト型を採用する．

ホスト型は仮想マシン型と比較すると 3 倍近い性能でありスケーラビリティー上のボトルネックもないが，ストレージデバイスである SSD と比較すると仮想マシンからストレージコントローラーまでの処理に当たる I/O 仮想化のオーバーヘッドが大きい．図 3 のレイテンシーの内訳に注目すると，Read は SSD のレイテンシーが 99 マイクロ秒と I/O 仮想化の 18 マイクロ秒に比べて十分大きいが，Write は SSD のレイテンシーが 20 マイクロ秒と I/O 仮想化の 16 マイクロ秒とほぼ同程度であり，I/O 仮想化に伴いレイテンシーが 2 倍近くに伸びていることがわかる．IOPS に関しても，合計 IOPS は VM 数に比例してスケールするが，1 つの VM あたりの平均 IOPS に着目すると，VM 間で I/O が競合しない 1VM 時が最もよく，このときの Read の IOPS は図 4 によると 248K IOPS であるが，本評価で使用した SSD の諸元は 4KiB Read が 460K IOPS[20] であるため，VM あたりでは SSD1 本の半分程度の IOPS が限度である．2.2 節で述べた準仮想化ブロックデバイスだけでも 4KiB Read で 395K IOPS と SSD1 本分の性能にも達しておらず，I/O 仮想化がストレージデバイスの性能を引き出す上でのボトルネックとなっている．

このようにホスト型でも I/O 仮想化がボトルネックとして顕在化するため，HCI 基盤に I/O-intensive なアプリケーションを集約することは困難である．これを解決するために，4 節で述べる仮想 I/O 処理の高速化 [14], [15], [21], [22] がこれまで提案されてきたが，2 倍に改善してようやく SSD1 本の性能に到達する現状を踏まえると，この改善だけでは複数の SSD を束ねた場合に匹敵する性能を引き出すことは難しく，高速化と直交して I/O 仮想化自体を課題と捉えてアーキテクチャーレベルで改善に取り組むのが望ましい．本稿では，I/O 仮想化のないコンテナ [23] に着目して，従来のホスト型にコンテナを組み合わせたハイブリッドアーキテクチャーを提案する．

3. 仮想マシンとコンテナのハイブリッド

コンテナは仮想マシンのような OS レベルの仮想化ではなく，ホスト OS からプロセス ID やファイルシステムのマウントポイントなどの名前空間を分離 [24] させ，cgroups[25] などで CPU やメモリーなどのリソースを制約することで実現するプロセスレベルの仮想化である．ゲストがホスト OS に密結合してしまうため OS を選べないというデメリットがあるものの，I/O 仮想化のような処理がないため軽量 [26], [27] である．実際に，表 1 の環境において仮想マシンと同等の諸元のコンテナを Docker[28] で配備を行い，1 から 2 冪で 128 多重まで 4KiB の Read/Write を行った結果が図 5 である．I/O 仮想化がないコンテナでは 2.3 節で述べたような低多重度における Write のレイテンシーの増加はなく，仮想マシンと比較して半分近くに短縮とストレージデバイスである SSD と同等の値となった．IOPS に関しても同様の理由によって改善しており，コンテナを活用することで I/O-intensive なアプリケーションの性能要件を満たしやすくなり，これまで以上の HCI 基盤への集約が期待できる．

コンテナを扱うハイブリッド型における課題は，仮想マシンとの I/O 競合時の性能低下である．図 6 は同一ホスト上の仮想マシンとコンテナから同一の SSD に対して，同時に 1 から 2 冪で 128 多重で Read 負荷をかけて I/O を競合させた場合の結果である (なお，この評価では data-plane の設定により挙動が不安定になったため，標準で data-plane のコードが含まれる QEMU 2.9.0 を用いた)．仮想マシンとコンテナのどちらも 32 多重以上の場合に I/O の競合による単体時との性能差が読み取れるが，このような高多重時の性能低下の傾向は仮想マシンとコンテナで異なっており，仮想マシンではレイテンシーが，コンテナでは IOPS が大幅に低下していることがわかる．コンテナ側の IOPS の低下幅は VM 側の IOPS 値に等しいことから，コンテナ

のIOPSが低下するのはVMのI/Oを処理する分だけコンテナのI/Oが処理できないためであり，逆に仮想マシン側はI/O仮想化のオーバーヘッドがないコンテナからの高速なI/Oが高多重度で混じったことによりI/Oの処理待ちが増えたため，レイテンシーが増加したと考えられる．本稿の提案するハイブリッド型の狙いからコンテナには高I/O性能が期待されるが，このように競合する仮想マシンからのI/Oによる影響を受けやすく，実際に期待されるI/O性能を提供できるかどうかはワークロード依存である．

そのような課題を解決するために，本稿ではハイブリッド型向けのI/Oスケジューラーを提案する．ハイブリッド型では図5が示すようにI/Oの発行元が仮想マシンかコンテナかによって大きくI/O性能が異なるため，従来のI/Oスケジューラーが目指した公平の方向ではなく，アーキテクチャーとしてI/O性能差があることを前提としてシステムの全体最適化を目指す．図6のI/O競合の結果から，仮想マシンからのI/Oはコンテナからの高多重なI/Oがあると I/O待ちになりやすい傾向があり，スロットリングを行ったとしてもそれによる性能低下は小さい．このことを利用して，コンテナからの高多重なI/O発行により高いI/O性能が求められる場合には仮想マシンからのI/Oに対してはスロットリングを行い，代わりにコンテナからのI/Oを優先して処理することでコンテナ側のI/O性能を向上させることができる．また，仮想マシンからのI/O負荷に応じてスロットリングを行うことでワークロードによる依存度合いを減らし，仮想マシン側がどのようなワークロードであったとしてもコンテナ側は安定した高いI/O性能が期待できる．

仮想マシンからのI/Oのスロットリングは I/Oのスケジューリングを行うホストではなく仮想マシン内で行うことにより，ホストと仮想マシン間のコンテキストスイッチの回数が減少して CPU 使用率を削減できる．スロットリングは I/O 待ちによるレイテンシーの増加という欠点があるものの，I/O 待ち中に I/O をまとめることができるという利点もある．実際に，複数の I/O をまとめて 1 つのシーケンシャルな I/O にすることでデバイスの性能を引き出す最適化 [29], [30] もあるが，このような最適化と直交して仮想マシン内の準仮想化ブロックデバイスでスロットリングを行い，ホストへの I/O をまとめることでホストへのコンテキストスイッチを引き起こす I/O 発行の回数を削減できる．

4. 関連研究

本稿の 2.1 節で定義した仮想マシン型は HCI に限らずハイパーバイザーとしてみれば Xen[31] のアーキテクチャーと類似している．Xen はハイパーバイザーのため VM 間通信は準仮想化による高速化が可能であったが，HCI の観点からは準仮想化のようなハイパーバイザーに依存する高速化はアーキテクチャーが仮想マシン型からホスト型に変わってしまうため，仮想マシン型の利点であった可搬性を保ちつつ高速化することができない．また，たとえ準仮想化により高速化できたとしてもホスト型と同程度までであり，2.3 節で述べた I/O 仮想化のボトルネックは依然として残ったままである．

仮想 I/O の高速化の 1 つとしては Intel VT-d[32] や AMD IOMMU[33]，SR-IOV[34] などのハードウェア支援により仮想マシンに対して直接デバイスを割り当てることが挙げられる．ELI[35] により割り当てたデバイスからの割り込み時にホストにコンテキストスイッチする回数を減らすことで，物理デバイスに近い性能を実現できることが報告されている．また，これらハードウェア支援機構によらず，準仮想化を使いソフトウェアだけで直接デバイスにアクセスする手法 [36] も提案されている．このような高速化は仮想マシンとサーバーに内蔵されたデバイスが紐づけされてしまうため，VM マイグレーション [37] ができず，HCI の共有ストレージを活かすことができない．

SSD の特性に合わせた仮想 I/O の高速化も提案されている．SSD の標準インターフェースになりつつある PCIe ベースの NVMe(Non-Volatile Memory Express) は SSD 内部の並列度を活かすために規格上 64K 個のコマンドを受け付ける I/O キューを最大 64K 個持つことができる [38]．Linux のブロックレイヤーはこの複数の高深度な I/O キューを活用して SSD の性能を引き出すために新しく MQ(Multi-Queue) フレームワーク [16] を導入したが，同様に 2.2 節で述べた KVM の data-plane[14], [15] に vCPU ごとに専用の I/O キューとスレッドを用意することで並列度を高める手法 [21] がある．2.3 節で述べたように，本稿の提案はこのような I/O 仮想化の高速化と直交しているため，その成果をそのまま享受することができる．

コンテナは仮想マシンとの性能比較 [26], [27] が示すようにベアメタルに近い性能を示すことができる仮想化方式であるが，これまでは主に開発や運用の効率化に重点が置かれ，コンテナの配備を自動化する Docker[28] やコンテナのクラスタ管理を行う Kubernetes[39] などと組み合わせてマイクロサービスを実現する手法としてとらえられている．これまで，本稿のような仮想マシンとコンテナを扱うハイブリッドなアーキテクチャーが提案 [40] されてきたが，これらは仮想マシン内にコンテナを作成するため I/O 仮想化のオーバーヘッドを受けてしまう．本提案はホスト内にコンテナを作成することでそのようなオーバーヘッドを避けることができる．

5. おわりに

本稿では，現状の Hyper-Converged Infrastructure をストレージコントローラーの構成から仮想マシン型とホスト型の 2 つにアーキテクチャーを分類・分析し，高速なホス

ト型でも All-Flash 化に伴い仮想マシンの I/O 仮想化がボトルネックとなることを明らかにして，コンテナとのハイブリッド型アーキテクチャーの提案を行った．仮想マシン型はハイパーバイザーに依らない可搬性を可能とする代わりに VM 間通信によりホスト型と比較して 3 分の 1 以下の性能であり，高速なホスト型でも I/O 仮想化により SSD1 本の性能の半分程度とストレージデバイスの性能を引き出すことができない．提案するハイブリッド型では I/O 仮想化がないコンテナをホスト内で実行することでストレージデバイスの性能を引き出せるようになり，I/O-intensive なアプリケーションの HCI 基盤への集約が可能となる．このようなハイブリッド型を活用するための課題が I/O 競合による性能低下であるが，仮想マシンからとコンテナからの I/O の性能差を考慮した I/O スケジューリングによりこの性能低下を緩和することができる．今後の課題としては，この I/O スケジューリングの実装・評価を行い，提案するハイブリッド型アーキテクチャーの有効性を示すことである．

参考文献

[1] Nutanix: Nutanix, http://www.nutanix.com/.
[2] VMWare: VMWare vSAN, http://www.vmware.com/products/virtual-san.html.
[3] HPE: SimpliVity, https://www.hpe.com/jp/ja/integrated-systems/simplivity.html.
[4] Stonebraker, M.: The Case for Shared Nothing, *IEEE Database Eng. Bull.*, Vol. 9, No. 1, pp. 4–9 (1986).
[5] Fujitsu: ETERNUS AF series, http://www.fujitsu.com/jp/products/computing/storage/all-flash-arrays/af/.
[6] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S. and Panigrahy, R.: Design Tradeoffs for SSD Performance., *USENIX Annual Technical Conference*, pp. 57–70 (2008).
[7] Deng, Y. and Zhou, J.: Architectures and Optimization Methods of Flash Memory Based Storage Systems, *Journal of Systems Architecture*, Vol. 57, No. 2, pp. 214–227 (2011).
[8] Shin, W., Chen, Q., Oh, M., Eom, H. and Yeom, H. Y.: OS I/O Path Optimizations for Flash Solid-state Drives., *USENIX Annual Technical Conference*, pp. 483–488 (2014).
[9] Lee, S., Kim, J. and Mithal, A.: Refactored Design of I/O Architecture for Flash Storage, *IEEE Computer Architecture Letters*, Vol. 14, No. 1, pp. 70–74 (2015).
[10] Dell EMC: VMAX All Flash Storage, https://www.emc.com/ja-jp/storage/vmax-all-flash.htm.
[11] 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).
[12] Russell, R.: virtio: Towards a De-Facto Standard for Virtual I/O Devices, *ACM SIGOPS Operating Systems Review*, Vol. 42, No. 5, pp. 95–103 (2008).
[13] Bellard, F.: QEMU, a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46 (2005).
[14] Huynh, K.: Exploiting The Latest KVM Features For Optimized Virtualized Enterprise Storage Performance, *CloudOpen, North America* (2013).
[15] He, A.: Virtio-blk Performance Improvement, *KVM Forum* (2012).
[16] Bjørling, M., Axboe, J., Nellans, D. and Bonnet, P.: Linux block IO: Introducing Multi-queue SSD Access on Multi-core Systems, *Proceedings of the 6th International Systems and Storage Conference*, ACM, p. 22 (2013).
[17] Gordon, A., Har'El, N., Landau, A., Ben-Yehuda, M. and Traeger, A.: Towards Exitless and Efficient Paravirtual I/O, *Proceedings of the 5th Annual International Systems and Storage Conference*, ACM, p. 10 (2012).
[18] Landau, A., Ben-Yehuda, M. and Gordon, A.: SplitX: Split Guest/Hypervisor Execution on Multi-Core., *WIOV* (2011).
[19] Adams, K. and Agesen, O.: A Comparison of Software and Hardware Techniques for x86 Virtualization, *ACM SIGOPS Operating Systems Review*, Vol. 40, No. 5, pp. 2–13 (2006).
[20] Intel: SSD DC P3700 Series Specifications, https://www.intel.co.jp/content/www/jp/ja/solid-state-drives/ssd-dc-p3700-spec.html.
[21] Kim, T. Y., Kang, D. H., Lee, D. and Eom, Y. I.: Improving Performance by Bridging the Semantic Gap Between Multi-queue SSD and I/O Virtualization Framework, *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, IEEE, pp. 1–11 (2015).
[22] Kim, J., Ahn, S., La, K. and Chang, W.: Improving I/O Performance of NVMe SSD on Virtual Machines, *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, New York, NY, USA, ACM, pp. 1852–1857 (2016).
[23] Soltesz, S., Pötzl, H., Fiuczynski, M. E., Bavier, A. and Peterson, L.: Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors, *ACM SIGOPS Operating Systems Review*, Vol. 41, No. 3, ACM, pp. 275–287 (2007).
[24] Biederman, E. W.: Multiple Instances of the Global Linux Namespaces, *Proceedings of the Linux Symposium*, Vol. 1, Citeseer, pp. 101–112 (2006).
[25] Rosen, R.: Resource management: Linux kernel namespaces and cgroups, *Haifux, May*, Vol. 186 (2013).
[26] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J.: An Updated Performance Comparison of Virtual Machines and Linux Containers, *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, IEEE, pp. 171–172 (2015).
[27] Morabito, R., Kjällman, J. and Komu, M.: Hypervisors vs. Lightweight Virtualization: a Performance Comparison, *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, IEEE, pp. 386–393 (2015).
[28] Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment, *Linux Journal*, Vol. 2014, No. 239, p. 2 (2014).
[29] Li, C., Shilane, P., Douglis, F., Sawyer, D. and Shim, H.: Assert (! Defined (Sequential I/O))., *HotStorage* (2014).
[30] Axboe, J.: Linux Block IO - Present and Future, *Ottawa Linux Symposium*, pp. 51–61 (2004).
[31] 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, *ACM SIGOPS operating systems review*, Vol. 37, No. 5, ACM, pp. 164–177 (2003).
[32] Abramson, D., Jackson, J., Muthrasanallur, S., Neiger, G., Regnier, G., Sankaran, R., Schoinas, I., Uhlig, R., Vembu, B. and Wiegert, J.: Intel Virtualization Technol-

ogy for Directed I/O., *Intel technology journal*, Vol. 10, No. 3 (2006).
[33] AMD: Technology (IOMMU) Specification (2007).
[34] PCI SIG: PCI-SIG Single Root I/O Virtualization, http://pcisig.com/specifications/iov/single_root/.
[35] Gordon, A., Amit, N., Har'El, N., Ben-Yehuda, M., Landau, A., Schuster, A. and Tsafrir, D.: ELI: Bare-Metal Performance for I/O Virtualization, *ACM SIGPLAN Notices*, Vol. 47, No. 4, pp. 411–422 (2012).
[36] Liu, J., Huang, W., Abali, B. and Panda, D. K.: High Performance VMM-Bypass I/O in Virtual Machines., *USENIX Annual Technical Conference, General Track*, pp. 29–42 (2006).
[37] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I. and Warfield, A.: Live Migration of Virtual Machines, *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, USENIX Association, pp. 273–286 (2005).
[38] Cobb, D. and Huffman, A.: Nvm Express and the PCI eEpress SSD Revolution, *Intel Developer Forum, San Francisco, CA, USA* (2012).
[39] Bernstein, D.: Containers and Cloud: From LXC to Docker to Kubernetes, *IEEE Cloud Computing*, Vol. 1, No. 3, pp. 81–84 (2014).
[40] VMWare: vSphere Integrated Containers, https://www.vmware.com/products/vsphere/integrated-containers.html.