Vol.2022-OS-156 No.12
2022/7/28

ハードウェアオフロード技術の仮想マシンゲストからの制御に関する研究

鷲津 優維[1,a)]　牧田 俊明[1,b)]

概要：今後，5G/6G の普及により，大容量・低遅延を活かしたサービスが MEC 上において展開されることが予測される．また，コンテナ技術が広まることで，この MEC においてもコンテナの利用が予測されるが，Kubernetes などのコンテナ基盤では，ネットワークの様々な機能をソフトウェア実装するため，性能面のオーバヘッドが大きくネットワーク性能向上においてボトルネックとなりうる．このオーバーヘッドの抜本的な解決のために，物理マシン上においてコンテナ基盤を展開する場合，ハードウェアオフロードという手法が存在する．一方で，柔軟性向上等の目的で VM 上においてコンテナ基盤を展開する場合，ゲスト内で動作するコンテナ基盤の管理ソフトウェアから，直接ホストのハードウェアオフロードを制御できないため，この手法を適用できない．そこで，本研究では，VM 上において動作するコンテナ基盤において，ゲストから制御可能なハードウェアオフロードを用いて高速化するために，SR-IOV を用いたホストの仮想ファンクション (VF) をゲストに物理ファンクション (PF) として認識させる手法を検討した．本稿では，SR-IOV の L2 スイッチング機能に対応した，VM 上の仮想ネットワークのハードウェアオフロードを実装し，ネットワーク性能の向上を，スループットとレイテンシの 2 点で確認した．また，SR-IOV の L2 以外のパケットヘッダマッチ機能にも対応させ，コンテナネットワーク機能をオフロードするための実装方針の検討と，それにあたっての課題を述べる．

キーワード：仮想ネットワーク，コンテナ，MEC，エッジコンピューティング，オフロード，SR-IOV，vDPA

Research on controlling hardware offload technology from virtual machine guests

Abstract: With the spread of 5G and 6G, it is expected that services that take advantage of high capacity and low latency will be deployed to MEC platforms. In addition, with the spread of container technology, it is expected that containers will also be used in MEC. However, container infrastructures such as Kubernetes require software implementation of various network functions, which imposes a large performance overhead. To solve this overhead drastically, in the case of deploying container infrastructure to physical machines, a technology called hardware offloading is employed. On the other hand, when deploying container infrastructure to VMs for improving flexibility or other reasons, this technique cannot be applied because the management software of the container infrastructure running in the guest cannot directly control the hardware offloading of the host. Therefore, in this study, in order to speed up the container infrastructure running on VMs using hardware offloading controllable from guests, we investigated a method to make guests recognize a virtual function (VF) of a SR-IOV device on hosts as a physical function (PF) in guests. In this paper, we implemented hardware offloading of virtual networks on VMs to support SR-IOV's L2 switching functionality, and confirmed the improvement in network performance in two aspects: throughput and latency. In addition, we also show the outline and challenges of the implementation for offloading container network functions other than L2 switching.

Keywords: Virtual network, Container, MEC, Edge Computing, Offloading，SR-IOV，vDPA

1. はじめに

1.1 背景

5G などによるアクセス回線の大容量・低遅延化が進んでおり，それらの特徴を活かしたサービスの普及が予測される．例えば，イベント中継などの 4K ライブストリーミングや，ゲームサーバからのストリーミングによるクラウドゲーミング，多くのセンサをリアルタイムで監視することが必要なコネクテッドカーが挙げられる．このような，大容量・低遅延を活かしたサービスの実現に必要な技術として，Multi-access Edge Computing[1] (以下，「MEC」という) 技術がある．MEC とは，端末に近い位置で IT サービス環境やクラウドコンピューティング機能を提供するシステムである．MEC 上でシステムを構築することで，端末から地理的に近い MEC のサーバ上でデータの処理・分析が可能になるため，リアルタイム性が高く，また，負荷が分散されることで通信の大容量・低遅延化が可能になる．

また，近年 IT システム環境において，サービス開発のアジリティ向上や運用管理の単純化などを目的としてコンテナ技術の採用が進んでいる [2]．クラウドプロバイダ各社がマネージド Kubernetes のようにクラウド上でコンテナを利用するサービスを提供する [3] [4] など，クラウド上でのコンテナ利用も進んでおり，MEC においても今後コンテナが普及していくことが考えられる．

クラウド上でコンテナを利用する場合，仮想マシン (Virtual Machine，以下「VM」という) 上でコンテナを利用することが多い．例えばマネージド Kubernetes は，Kubernetes の管理を行う Master ノードの機能をクラウドベンダが提供し，コンテナを動作させる Worker ノードには VM を利用する構成が一般的である．マネージドサービスを利用しない場合も，運用の柔軟性を考慮し VM 上にコンテナをデプロイすることが考えられる．

1.2 本研究の想定環境と対象領域

1.1 で述べた背景より，本研究では MEC において VM によるコンピューティングリソースの提供を行うサービスで，VM 上でコンテナを動作させる環境を想定する．

端末から MEC の入口までのアクセスはさまざまな経路が想定されているが，例えば無線の場合は 6G により将来的に 5G の 10 倍以上の大幅な性能向上が期待されている．一方，端末から MEC 上で動作するコンテナまでのエンドツーエンドの性能を高めるためには，MEC における VM の提供やコンテナの動作に仮想ネットワークが必要となるため，これらの仮想ネットワークについてもアクセス経路と同等の性能が求められる．

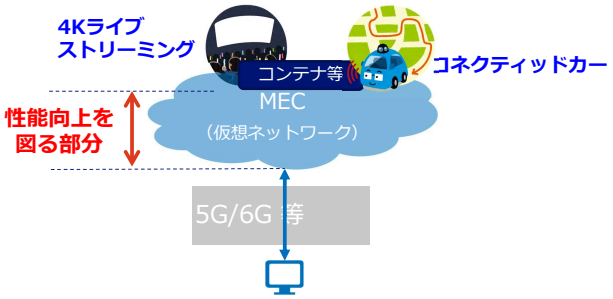

図 1 本研究で性能向上を図る区間
Fig. 1 Target area of our performance improvement

そのため，本研究では，MEC の VM においてコンテナを動作させる場合の，MEC 内の仮想ネットワークの性能向上を検討する．本研究で性能向上を図る対象区間について，図 1 に示す．

2. 技術課題

本研究では，仮想ネットワークの大幅な性能向上実現のため，ハードウェアオフロードを活用する．まずハードウェアオフロードについて 2.1 で説明し，これを 1.2 で説明した環境に適用するための課題について 2.2 で述べる．

2.1 ハードウェアオフロード

1.2 で述べたように，アクセス経路の性能は今後従来の 10 倍以上に飛躍的に向上していくことが期待される．それに伴い，仮想ネットワークの性能も抜本的に向上させる必要がある．そのため，本研究ではハードウェアオフロードを利用する．

一般的に VM やコンテナの仮想ネットワークは，デフォルト設定ではネットワーク上の様々な機能がソフトウェアを通した実装となる．VM やコンテナ基盤を構成するソフトウェア (OpenStack や Kubernetes) により，仮想ネットワーク上でルーティングやスイッチング，暗号化，ファイアウォールなどさまざまなネットワーク機能が提供されるが，それらが全てソフトウェアによる実装となる．このソフトウェア実装により，エミュレーションレイヤのオーバヘッドが発生し，ボトルネックとなりうる [5] [6]．例えば，ハッシュテーブルを利用したネットワーク機能のソフトウェア実装に起因する CPU キャッシュミスヒットや，ユーザ空間において機能が実装される場合のシステムコールなどのオーバヘッドが挙げられる．

これらのオーバーヘッドを解消する技術として，ハードウェアオフロードが挙げられる．ハードウェアオフロードは，ネットワーク機能等をハードウェアで動作させ，ソフトウェアによるオーバーヘッドを抜本的に低減させる技

[1] NTT Open Source Software Center
Shinagawa Season Terrace 29F, 1-2-70, Konan, Minato-ku, Tokyo, 108-0075, Japan
a) yui.washidu.bz@hco.ntt.co.jp
b) toshiaki.makita.vh@hco.ntt.co.jp

術である．例えば，VM を利用したクラウドの基盤ソフトウェアである OpenStack や，コンテナ基盤ソフトウェアの Kubernetes では，ハードウェアオフロードを利用した仮想ネットワークの構築が可能である．

これらの仮想ネットワークでは，Single Root I/O Virtualization [7] (以下，「SR-IOV」という) を利用したハードウェアオフロードを行う．SR-IOV は，通常の PCI ファンクション (物理ファンクション，以下「PF」という) を通じて軽量なファンクション (仮想ファンクション，以下「VF」という) を同一 PCI デバイス上に作成し，VM やコンテナに占有的に VF を割り当てられるようにする仕組みである．VF と PF 間あるいは外部ネットワークとの通信は当該 PCI デバイス，すなわち NIC によりスイッチングされる．また，NIC の機能によるが，ファイアウォール等の各種ネットワーク機能を NIC において提供することも可能である．VM やコンテナの基盤ソフトウェアは，これらスイッチングやネットワーク機能の設定を行うことで，ソフトウェアによるネットワーク機能実装を利用することなく，オーバーヘッドを低減できるため，大幅な性能向上が見込める．

SR-IOV を利用するには，各基盤ソフトウェアがその設定 (VF の割り当てや，NIC 内のネットワーク機能設定) を行う必要がある．一例として，コンテナ基盤 Kubernetes においては，ネットワーク設定は CNI[8] 仕様に準拠したプラグインにより実現する．SR-IOV を利用可能なプラグインとしては，SR-IOV CNI プラグインや，OVN CNI プラグイン，Antrea CNI プラグインなどが挙げられる．

2.2 本研究で取り扱う技術課題

次に，1.2 で述べた環境に，ハードウェアオフロードを適用する際の技術課題を述べる．

2.1 で例として挙げた Kubernetes における SR-IOV を活用可能なプラグインは，現時点では，VM 上の仮想ネットワークのオフロードまで考慮しておらず，図 2 のような物理サーバ上で動作させることを前提としており，PF にアクセスすることで，新たな VF の作成や各種ネットワーク機能の設定を行っている．しかし，本研究で対象としている VM 上で展開されたコンテナ環境では，一般的に VM 上で PF にアクセスすることができないため，コンテナ仮想ネットワークのハードウェアオフロードができない．

以下，PF にアクセスできない理由について説明する．ここで，VM の仮想ネットワークは，ソフトウェアによるオーバーヘッドを回避するため，PCI デバイス割り当て等によりハードウェアオフロードを行っているものとする．PCI デバイス割り当てとは，PF，VF を問わず，VM に直接 PCI デバイスを使用させることを可能とする機能である．しかし，PF を VM に直接利用させると，VM から物理デバイスを直接操作でき，VM からホストへのアタックベクタが増加するため，セキュリティ上の懸念がある．また，特定の NIC を 1 つの VM が占有するため，スケールしない．そのため，ホストでは SR-IOV を用いて VF を作成し，図 3 のように VF を PCI デバイス割り当て等により VM に使用させる．

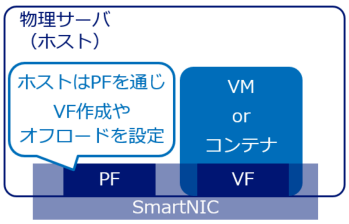

図 2 物理サーバにコンテナをデプロイ
Fig. 2 Deploy containers to a physical machine

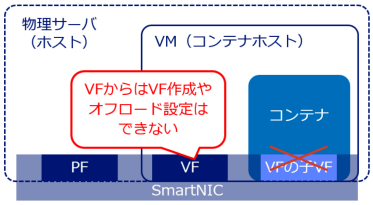

図 3 仮想マシンにコンテナをデプロイ
Fig. 3 Deploy containers to VM

このように，VM が PF にアクセスできず，直接ホストのハードウェアオフロードを制御できない環境で，ハードウェアオフロードを実現することが技術課題となる．

3. 提案方式

2.2 で述べた技術課題を解決するために，VM に割り当てられた VF をゲストに対して PF として認識させる方式を提案する．この方式の概要について 3.1 で，実装については 3.2 で述べる．

3.1 提案方式概要

本研究では，VM が PF にアクセスできない環境において，VM からハードウェアの制御を行える仕組みとして，ホストの VF をゲストに対して PF として認識させ (以下「仮想 PF」という)，ゲストからのハードウェア制御要求を仮想 PF を通じてホストでハンドリングしてハードウェア設定を行うことで，新たな VF (以下「仮想 VF」という) の作成等を可能にし，実質的にゲスト内で SR-IOV が制御できるように見せる方式を提案する．

しかし，VM のホストにおいて VF を PCI デバイス割り当てにより VM に使用させると，VF を PCI として VM に見せることができず，本研究で提案する方式を実現できない．これは，PCI デバイス割り当てという技術が，エミュレーションを行わずにホストの PCI デバイスを直接

ゲストに使用させることにより性能向上を実現しているためである．

ここで，2020 年に Linux 導入された vDPA 技術 [9] を応用することで，この方式を実現する．vDPA は，データプレーンが virtio 仕様，コントロールプレーンがベンダ仕様であるデバイスを用いることで，データプレーンのみを直接ゲストにアクセスさせ，コントロールプレーンをエミュレーションするという方式で，ライブマイグレーションなどに使用される技術である．このような，本来 SR-IOV をエミュレーションするためのものではない技術を応用することで，ホストの VF をゲストに対して PF と認識させ，ゲストにとっての PF から新たな VF を作成し，VM 内のコンテナへ割り当てる方式である．この方式は，ゲスト内からの SR-IOV の利用方法が，物理環境における SR-IOV の利用方法と同等になるため，2.1 で述べた，Kubernetes の各種 CNI プラグインなど，既存の SR-IOV を利用する基盤ソフトウェアのプログラムの変更を最小限に抑えることが可能な利点がある．

他にも，解決手法として，以下の 3 つの方式を検討した．

【方式 1】SR-IOV と SF 併用方式

【方式 2】エージェント方式

【方式 3】VM 内スイッチング方式

方式 1 はゲストに割り当てられた VF から新しい仮想ファンクションを作成する方式であり，Scalable IOV や Sub Function(SF) の技術を応用することで，この方式の実現を検討した．また，方式 2 は VM のエージェントがホストに VF の作成・割り当てを依頼することで，VM からは VM 内のコンテナへの VF 追加を可能にする方式である．最後に，方式 3 は コンテナに VF を割り当てるのではなく，コンテナ基盤の NW 機能をオフロードする方式であり，VM 内でスイッチングのみを行う機能をソフトウェア実装し，スイッチング以外の機能を NIC にオフロードすることで，仮想ネットワークの負荷軽減を図る方式である．ただし，方式 3 はパケット転送にオーバーヘッドが残る点，方式 2 は CNI プラグインなど既存のプログラムの大幅な変更が必要な点が欠点となり，方式 1 については，SF 等の技術のベンダ依存度が高いこと，及び現時点では VF から SF 等を作成できる機能をもつ NIC がないことから，本研究では SR-IOV のホスト VF をゲストに対して PF と認識させる方式を採用した．ただし，方式 1 については，将来的にそのような機能を持つ NIC が利用可能になれば，再検討の余地があると考える．

3.2 実装

提案方式実現のための実装として，Linux 上の qemu における PF・VF・外部ネットワーク間のスイッチング機能の実装を行った．図 4 にコントロールプレーン，図 5 にデータプレーンの実装を示す．

3.1 で述べたような，「ホストの VF をゲストに対して PF として認識させる」部分については，Linux 及び qemu の実装では，vDPA を利用したゲストに割り当てた VF が，ゲストからは virtio-net-pci の PF デバイスとして見えるため，vDPA を適用するのみで実現可能となっている．しかし，3.1 の方式を実現するためには，これに加えて，vDPA 技術によって割り当てられたデバイスに対して以下の 2 点を満たす必要がある．

【要件 1】ゲストから SR-IOV が可能な仮想 PF として認識される

【要件 2】ホストはゲストの SR-IOV 設定に応じて，適切にハードウェアでデータプレーンのスイッチングが可能なように仮想 VF を構成する

以降，要件 1,2 の実現のために必要となる実装を述べる．

3.2.1 要件 1

要件 1 の実現に必要な実装を以下に示す．

(1) PCI Configuration Space に SR-IOV Capability Structure を追加する

(2) SR-IOV Capability Structure へのゲストの書き込みをホストがハンドリングしてエミュレートした仮想 VF を作成する

図 4 に示すように，(1) は，仮想 PF が SR-IOV 機能を持つデバイスとしてゲストに認識されるために必要であり，qemu の virtio-net-pci デバイスエミュレーションコードにおいて，PCI Configuration Space の実装を書き換えて，Capability Structure に SR-IOV が含まれるように修正を加えることで実現した．

また，同様に図 4 に示すように，(2) も，qemu の virtio-net-pci デバイスのエミュレーションコードにおいて，SR-IOV Capability Structure の書き込みをハンドリングし，ゲストからの VF 作成要求に対して仮想 VF を作成する処理を追加した．ここでは，仮想 VF は virtio-net-pci デバイスとして認識されるように実装することで実現した．

3.2.2 要件 2

要件 2 の実現に必要な実装を以下に示す．

- 仮想 VF がハードウェアを通じて他の仮想 VF，仮想 PF，ホストの PF 及び外部ネットワークと通信できるように仮想ネットワークを構成する

これを実現するための実装について述べる前に，qemu のネットワークについて説明する．qemu のネットワークは，ゲストに提供される仮想ネットワークデバイスと，エミュレートされた NIC の通信を担うバックエンドの 2 つで構成される．vDPA 技術を使用する場合，仮想ネットワークデバイスは virtio-net-pci，バックエンドは vhost-vdpa を利用することとなる．提案方式の仮想 PF も通常の vDPA と同様，仮想ネットワークデバイスに virtio-net-pci，バックエンドに vhost-vdpa を使用する．このとき，仮想 PF は vDPA の機能により，データプレーンはエミュレーショ

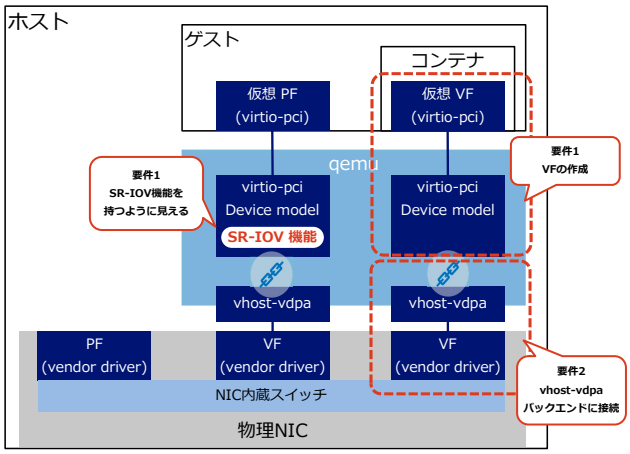

図 4 提案方式のコントロールプレーン
Fig. 4 Control Plane of the proposed method

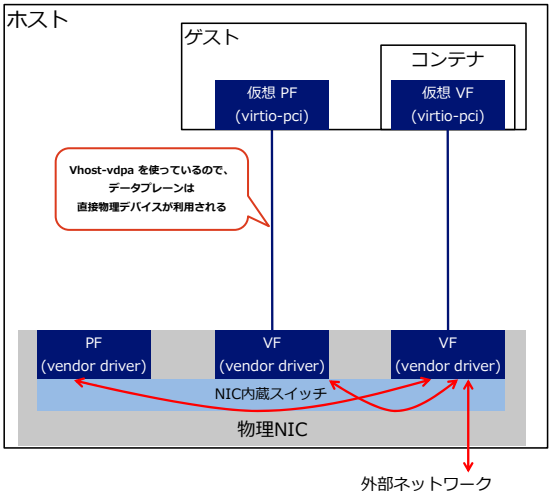

図 5 提案方式のデータプレーン
Fig. 5 Data Plane of the proposed method

ンを行わず，直接物理 NIC が利用される．通常 SR-IOV 機能を持つ物理 NIC には，PF，VF，及び外部ネットワークの通信をスイッチングする内臓のスイッチ機能が具備されている．そのため，図 5 に示すように，仮想 VF についても，vDPA を通じてデータプレーンで物理 NIC の VF に直接アクセスすれば，仮想 VF は他の仮想 VF，仮想 PF，ホストの PF，外部ネットワークとハードウェアを通じたスイッチングが可能となる．

よって，図 4 に示すように，仮想 VF 作成時に作成される virtio-net-pci デバイスのバックエンドは自動的に vhost-vdpa となるように実装した．ただし，バックエンドに vhost-vdpa を利用するには，対応するホスト物理 NIC の VF が必要となる．今回の実装では，あらかじめ仮想 VF 用にホストで VF を作成し，また，qemu の vhost-vdpa で利用可能なように設定を事前に行ったうえで，それらの情報を qemu 起動時にオプションで渡すことで，仮想 VF 用の vhost-vdpa バックエンドをプールするようにしている．仮想 VF 作成時にはプールから vhost-vdpa バックエンドが選択され，作成された仮想 VF のバックエンドとして使われる．

これによって，仮想 VF のデータプレーンは，ハードウェアにより直接通信が可能になる．

4. 性能検証

本章では，3 で行った実装による性能向上を確かめるための検証について述べる．

本研究では，VM 上の仮想ネットワークで発生するオーバーヘッドを対象として，それを低減させるために，ハードウェアオフロードを利用している．仮想ネットワークを使用しない環境 (仮想化もコンテナも使用せず物理マシン上のプロセスで直接通信を行う環境，以後「ベアメタル環境」という) ではこのオーバーヘッドが存在しないので，ベアメタル環境での性能以上の性能向上は，本方式では実現できない．そのため，まず理想値として，ベアメタル環境での性能検証を行った．次に，代表的なコンテナ環境の例として Kubernetes を取り上げ，これに対して，提案方式を適用し，ゲストにおいて SR-IOV CNI によるハードウェアオフロードを使用した環境で性能検証を行った．そして，性能向上を確かめるための比較対象として，ゲスト上でハードウェアオフロードを使用しない Kubernetes の環境 (以後，方式適用前環境と呼ぶ) でも性能検証を行った．

まず，4.1 で性能検証環境を説明し，次に 4.2 でスループットの検証方法と検証結果，4.3 でレイテンシの検証方法と検証結果について述べる．

また，ここでの性能検証の指標は，スループットとレイテンシの 2 点とし，スループットは測定区間内のロスなしの伝送速度，レイテンシは 60 秒間に測定区間でパケットを送って返ってくるまでの往復の時間の最大値を指す．

4.1 検証環境

ベアメタル環境、方式適用前，適用後の性能測定にあたって，それぞれどのような環境で測定を行ったかを述べる．

4.1.1 ベアメタル環境

まず，ベアメタルの検証環境について述べる．片方の物理マシン上で netperf のサーバプロセスを起動させ，もう一方の物理マシンから通信することで，ベアメタル環境での性能検証を行った．ベアメタル環境で使用したサーバ機種, CPU, OS, NIC 等は表 1 に示す．

4.1.2 方式適用前検証環境

次に，方式適用前の検証環境について述べる．PCI デバイス割り当てを用いて，仮想マシンまでのネットワークの高速化を行っている環境で測定を行った．PCI デバイ

表 1　環境構築に使用したハードウェアおよびソフトウェア
Table 1　Hardware and software used to build the environment

	ベアメタル環境	方式適用前検証環境	方式適用後検証環境
サーバ機種	HPE ProLiant DL360 Gen9		
CPU	Intel Xeon CPU E5-2600 @2.3GHz (CPU 数 2, 1 CPU あたりのコア数 10)		
NIC	Mellanox Technologies MT27710 family ConnectX-6 Dx		
ホスト OS	Red Hat Enterprise Linux 8.4		
ゲスト OS	-	CentOS 7.9	
ホスト kernel	5.15		
ゲスト kernel	-	3.10.0-1160.45.1.el7 (CentOS 7.9 カーネル)	5.18
qemu version	-	5.2.93	
Kubernetes version	-	1.21	
CRI プラグイン	-	containerd 1.4.4	
CNI プラグイン	-	Calico 3.18.1	Calico 3.18.1, SR-IOV CNI 2.6.2, Multus CNI 3.6
SR-IOV Device Plugin	-	-	3.5.0

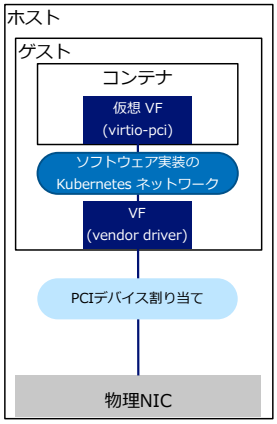

図 6　方式適用前環境
Fig. 6　Environment before application of the method

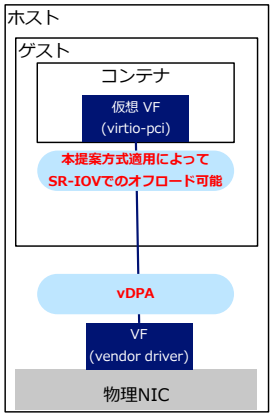

図 7　方式適用後環境
Fig. 7　Environment after application of the method

ス割り当てを使用した仮想マシンを 2 台起動，それぞれ Kubernetes の Master/Worker ノードとし，Worker ノードの中で，netperf のサーバプロセス (netserver) が動作する Pod を起動した．その上で，クラスタ外部の物理マシンから通信し，性能検証を行った．Kubernetes の仮想ネットワークには，プロキシ (kube-proxy の iptables 実装)，ルーティング (CNI プラグイン Calico)，ファイアウォール (CNI プラグイン Calico の NetworkPolicy 機能) が含まれ，これらはすべてソフトウェア実装となる．環境構築に使用したハードウェアおよびソフトウェアは表 1 に示す．

4.1.3　方式適用後検証環境

最後に，方式適用後の検証環境について述べる．vDPA を使用して，仮想マシンを 2 台起動，それぞれ Kubernetes の Master/Worker ノードとし，Worker ノードの中で，netperf のサーバプロセス (netserver) が動作する Pod を起動した．仮想マシンの起動に使用した qemu は，提案方式を実装した qemu を用いており，ホストの VF をゲストに PF として認識させることができる．また，CNI には，SR-IOV CNI を使用した．SR-IOV CNI は，Kubernetes で VF を割り当てる際に，SR-IOV Device Plugin と連携して動作し，Multus などのメタプラグインと併用することによって，割り当てられた VF の PCI アドレスを取得して，PCI アドレスで，SR-IOV CNI を呼び出すことができる．今回はメタプラグインとして，Multus を使用したが，Multus を使用するにあたって，事前に CNI プラグインがインストールされている必要があるため，ここでは，Calico を Multus 使用のために，利用している．

ただし，virtio-net-pci デバイスは他のネットワークデバイスと異なり PCI デバイスそのものがネットワークデバイスではなく，PCI デバイス配下の virtio bus にネットワークデバイスが接続される構成となっている．このハードウェア構成の見え方の差異のため Linux カーネルの sysfs ディレクトリ構成が他のネットワークデバイスと異なっており，SR-IOV CNI プラグインがこれに対応していなかったため，本環境で使用した SR-IOV CNI はこの対応のための修正を加えている．また，ゲストの Linux カーネルにおいて virtio-net-pci デバイスの PF から VF の情報を Netlink で取得する機能や，VF の vlan を設定する機能等が実装されておらず，SR-IOV CNI プラグインが動作しなかったため，この機能を実装するようカーネルに変更を加えている．

その上で，クラスタ外部の物理マシンから通信し，性能検証を行った．環境構築に使用したハードウェアおよびソフトウェアは表 1 に示す．

4.2　スループット

性能向上を確認するために，理想値であるベアメタル環境と方式適用前後，3 つの環境でのシングルコアでの性能

比較，性能値がスケールするか確認するために，方式適用後のシングルコア，マルチコア (2 コア，4 コア) の場合の性能比較を行った．ここでは，シングルコアでの性能検証方法，マルチコアでの性能検証方法，検証結果について述べる．

4.2.1 シングルコアでの性能検証方法

ベアメタル環境，方式適用前後それぞれの環境において，外部マシンと Kubernetes クラスタ間での 60 秒間 UDP のバルク転送を行うことで，性能検証を行った．測定は 5 回行い，その平均値を送受信それぞれのスループットの検証結果とした．ベアメタル環境では，物理マシン上で netserver を動作させ，送信性能は，他方のマシンへ UDP バルク転送を，受信性能は，他方のマシンから netserver が動作する物理マシンへ UDP バルク転送を行うことで検証した．また，方式適用前後の環境については，Kubernetes に netserver が動作する Pod を 1 つデプロイし，送信性能は，外部マシンへ Pod から UDP バルク転送を，受信性能は，外部マシンから Pod 宛に UDP バルク転送を行うことで検証した．送信性能については，Linux カーネルが適切に NIC からソケットへのバックプレッシャをかけるため，パケットロスが起きないことを確認の上，帯域制御を行わずに計測した．一方で，受信性能については，パケットロスが起きない上限性能を測るため，送信側で帯域制御を行い，帯域上限を徐々に上げて受信ソケット等でパケットロスが出ない最大のスループットを計測している．検証前に，割り込み発生のコアが検証中に変更されないように，ホストの irqbalance の動作を止め，ドライバからの割り込みを各 CPU コアを指定して発生するように設定している．

4.2.2 マルチコアでの性能検証方法

netperf の動作する Pod を 2 つデプロイした場合，4 つデプロイした場合それぞれの環境において，4.2.1 と同様に，外部マシンと Kubernetes クラスタ間で 60 秒間 UDP のバルク転送を行った．送信性能は，外部マシンへそれぞれの Pod から UDP バルク転送プロセスを 1 つずつ実行，受信性能は，外部マシンからそれぞれの Pod 宛に UDP バルク転送プロセスを実行，すなわち，外部マシンから同時にコア数分のプロセスを実行して検証した．測定は 5 回行い，プロセスごとに得られた測定結果の平均値を送受信それぞれのスループット検証結果とした．送信，受信性能測定時のパケットロスについては 4.2.1 と同様に考慮した．検証前に，割り込み発生のコアが検証中に変更されないように，ホストの irqbalance の動作を止め，ドライバからの割り込みを各 CPU コアを指定して発生するように設定している．また，受信測定では，割り込み発生時点で各 netperf プロセスのフローの使用コアが 2 コアないし 4 コアに分散されるように，Worker ノードで UDP フローごとに ethtool -U を使用してルールを設定している．

表 2 シングルコアでの方式適用前後の性能比較
Table 2 Performance comparison before and after application of the method on a single core

	送信性能 [Mbps]	受信性能 [Mbps]
ベアメタル環境 (理想値)	4029.0	6079.2
提案方式適用前	957.9	940.0
提案方式適用後	2493.8	2408.7

表 3 方式適用後のマルチコアとシングルコアの性能比較
Table 3 Comparison of multi-core and single-core performance after application of the method

	送信性能 [Mbps]	受信性能 [Mbps]
シングルコア	2493.8	2408.7
マルチコア (2 コア)	2461.9	2414.5
マルチコア (4 コア)	2395.7	2348.0

4.2.3 性能検証結果

シングルコアでの方式適用前後の性能比較を表 2 に，方式適用後のシングルコア，マルチコアでの性能比較を表 3 に示す．

送信性能，受信性能共に，適用前に対して適用後は 2.5 倍以上の性能となり，大きな性能向上を確認することができた．また，マルチコアとシングルコアで比較しても，今回検証した 4 コアまでの範囲ではスケールしていることを確認できた．ただし，理想値のベアメタル環境に対しては，提案方式適用後は送信性能は約 2/3, 受信性能は約 1/3 の性能となった．

4.3 レイテンシ

レイテンシについても，スループットと同様に，性能向上を確認するために，理想値とするベアメタル環境と方式適用前後のシングルコアでの性能比較，性能値がスケールするか確認するために，方式適用後のシングルコア，マルチコア (2 コア，4 コア) の場合の性能比較を行った．ここでは，シングルコアでの性能検証方法，マルチコアでの性能検証方法，検証結果について述べる．

4.3.1 シングルコアでの性能検証方法

方式適用前後それぞれの環境において，外部マシンと Kubernetes クラスタ間で，60 秒間 UDP において複数回 Request/Response 通信を行うことで測定した．測定は 5 回行い，得られた測定結果のうちの最大値をレイテンシの検証結果とした．そのため，物理マシン間でも，同様に 60 秒間 UDP における複数回の Request/Response 通信を 5 回行い，4.2.1 と同様に，ベアメタル環境では物理マシンで netserver プロセスを起動，方式適用前後環境では netserver プロセスが起動している Pod を 1 つデプロイして，UDP における複数回の Request/Response 通信を行うことで検証した．検証前に，割り込み発生のコアが検証中に変更されないように，ホストの irqbalance の動作を止

表 4　シングルコアでの方式適用前後の性能比較
Table 4　Performance comparison before and after application of the method on a single core

	レイテンシ [μsec]
ベアメタル環境 (理想値)	607
提案方式適用前	15400
提案方式適用後	4676

表 5　方式適用後のマルチコアとシングルコアの性能比較
Table 5　Comparison of multi-core and single-core performance after application of the method

	レイテンシ [μsec]
シングルコア	4676
マルチコア (2 コア)	4694
マルチコア (4 コア)	4764

め，ドライバからの割り込みを各 CPU コアを指定して発生するように設定している．

4.3.2　マルチコアでの性能検証方法

netserver の動作する Pod を 2 つデプロイした場合，4 つデプロイした場合それぞれの環境において，4.3.1 と同様に，外部マシンと Kubernetes クラスタ間で，60 秒間 UDP において複数回 Request/Response 通信を行った．検証前に，割り込み発生のコアが検証中に変更されないように，ホストの irqbalance の動作を止め，ドライバからの割り込みを各 CPU コアを指定して発生するように設定している．また，受信測定では，割り込み発生時点で各 netperf プロセスのフローの使用コアが 2 コアないし 4 コアに分散されるように，Worker ノードで UDP フローごとに ethtool -U を使用してルールを設定している．

4.3.3　性能検証結果

シングルコアでの方式適用前後の性能比較を表 4 に，方式適用後のシングルコア，マルチコアでの性能比較を表 5 に示す．

適用前に対して適用後のレイテンシは約 1/3 以下に小さくなり，大きな性能向上を確認することができた．また，マルチコアとシングルコアで比較しても，今回検証した 4 コアまでの範囲ではスケールしていることを確認できた．ただし，理想値であるベアメタル環境に対しては，提案方式適用後は約 8 倍という性能になった．

5.　今後の課題

性能検証結果を踏まえ，今後の課題を述べる．まず，性能向上については，方式適用前と比較すると，性能向上が確認できたが，理想値とするベアメタル環境に対しては，スループットの送信性能は約 2/3, 受信性能は約 1/3 の性能となり，レイテンシは約 8 倍と大きかった．これについては，今後要因を分析し，改善を検討する必要があると考えている．

次に，実装内容について，本稿では，L2 スイッチのみに対応した Linux 上の qemu の実装を行った．しかし，この実装のみでは，例えば Kubernetes の場合は，ファイアウォール等の Kubernetes が要求するネットワーク機能を実現できていないため，今回の検証環境で使用した SR-IOV CNI プラグインのように，他のプラグイン (Calico など) を併用する必要があった．この環境では，Calico で使用したネットワーク機能までオフロードすることができておらず，L2 スイッチ機能のみのオフロードとなり，一般的なコンテナ環境の仮想ネットワークがオフロードできているとまでは言えない．ファイアウォールやプロキシなどの一般的なネットワーク機能のオフロードには，転送以外に破棄などのパケット処理機能や，L2 以外のパケットヘッダフィールドによるパケット処理方法判定機能が必要になる．Linux では SR-IOV に switchdev モード [10] という機能があり，物理環境では，この機能に対応した NIC を用いることでこのような高度なパケット処理が可能になる．また，switchdev モードによるハードウェアオフロードを利用するためのソフトウェアとして Open vSwitch [11] があり，例えば Kubernetes の CNI プラグインでは OVN や Antrea などを使用することで Open vSwitch を通じた switchdev モードによって，高度なハードウェアオフロードが可能である．従って，仮想 PF に switchdev モードを実装することで，ファイアウォール等のネットワーク機能のハードウェアオフロードを実現する．

switchdev モードは，Open vSwtich のようなフローベースのスイッチ機能が組み込まれた NIC を利用する機能であるため，そのような NIC のエミュレーションとしては，Open vSwitch ないし類似の仮想スイッチをホストで動作させる手法が比較的簡単な方法だと考えられる．また，ソフトウェア実装でのエミュレーションではオーバーヘッドの低減にはならないため，この方法では，ホストで動作させる仮想スイッチの機能をホストの NIC へオフロードする必要がある．例えば，フローベースのルールのオフロードが可能な Open vSwitch の機能を利用して，エミュレートした NIC のオフロード機能を実現する方法が考えられる．よって，ホストの Open vSwitch のオフロード機能を通じて，SR-IOV の L2 以外のパケットヘッダマッチ機能にも対応させた，コンテナネットワーク機能をオフロードが可能な switchdev モードの実装を考えている．

6.　まとめ

本研究では，MEC の VM においてコンテナを動作させる場合の，MEC 内の仮想ネットワークの大幅な性能向上実現のため，ハードウェアオフロードの活用に焦点を当てた．例えば，コンテナ基盤ソフトウェアの Kubernetes では，ハードウェアオフロードを利用した仮想ネットワークの構築が可能であるが，これは，物理サーバでコンテナ基

盤を展開することを前提とした技術である．このように，現時点では，仮想ネットワークを構築する各種ソフトウェアは，一般的に VM 上の仮想ネットワークのオフロードまで考慮していない．

そのため，VM 上の仮想ネットワークをオフロード可能にするためには，仮想マシンゲスト内で動作する管理ソフトウェアから，PF にアクセスできず，直接ホストのハードウェアオフロードを制御できない環境において，ハードウェアオフロードを実現するという技術課題の解決が必要になる．技術課題解決のため，SR-IOV のホスト VF をゲストに対して PF と認識させる方式を提案した．この方式は，qemu で物理環境と仮想環境の差異を埋めることによって，仮想ネットワーク各種の管理ソフトウェアが VM 環境でのハードウェアオフロードを想定していない現状でオフロードを可能にする方式である．

そして，この方式の実現のための実装を行い，実装による性能向上を確かめるために，性能検証を行った．性能検証の指標は，スループットとレイテンシの 2 点とし，スループットは送受信性能ともに適用後は適用前の 2.5 倍以上の性能向上，レイテンシについても，適用後は適用前の 1/3 以下に小さくなり，性能向上を確認できた．ただし，理想値とするベアメタル環境での性能に対して，スループットの送信性能は約 2/3, 受信性能は約 1/3 の性能となり，レイテンシは約 8 倍と大きかったため，改善の余地があると考えている．

このように，本稿で提案した方式は，ベアメタル環境の性能には及ばないものの，現時点で VM 上で展開されたコンテナ基盤に対しては，適用することで大幅な性能向上を見込める方式であることが確認できた．今後は，性能面の課題の解決，L2 以外のパケットヘッダによるパケット処理判定のオフロード対応などの機能組み込みを行い，より実用的な方式にしていく予定である．

参考文献

[1] ETSI: ETSI - Multi-access Edge Computing - Standards for MEC, https://www.etsi.org/technologies/multi-access-edge-computing.
[2] IDCJapan 株式会社：2021 年 国内コンテナ／Kubernetes に関するユーザー導入調査結果を発表, https://www.idc.com/getdoc.jsp?containerId=prJPJ47597721 (2021).
[3] Google: Google Kubernetes Engine, https://cloud.google.com/kubernetes-engine.
[4] Amazon: Amazon Elastic Kubernetes Service, https://aws.amazon.com/jp/eks/.
[5] Istio: Performance and Scalability, https://istio.io/latest/docs/ops/deployment/performance-and-scalability/.
[6] Kapočius, N.: Performance Studies of Kubernetes Network Solutions, *2020 IEEE Open Conference of Electrical, Electronic and Information Sciences (eStream)*, pp. 1–6 (online), DOI: 10.1109/eStream50540.2020.9108894 (2020).
[7] Yu Zhao, D. D.: The Linux Kernel documentation PCI Express I/O Virtualization Howto, https://www.kernel.org/doc/html/v5.15/PCI/pci-iov-howto.html.
[8] The CNI Authors and The Linux Foundation: CNI, https://www.cni.dev/.
[9] Wang, J.: [PATCH V9 0/9] vDPA support, https://lkml.org/lkml/2020/3/26/481.
[10] Or Gerlitz, Hadar Hen-Zion, Amir Vadai and Rony Efraim: Introduction to switchdev SR-IOV offloads, https://legacy.netdevconf.info/1.2/session.html?or-gerlitz.
[11] : Open vSwitch, https://www.openvswitch.org/.