Vol.2017-OS-141 No.9
2017/7/26

仮想化環境に特化した単一アドレス空間OSにおける疑似マルチプロセス実行

縣 直道[1]　大須賀 敦俊[1]　窪田 貴文[1]　河野 健二[1]

概要：仮想化を利用したクラウド環境では，ひとつの仮想マシン上でひとつのアプリケーションのみを動作させる構成が一般的になっている．このようなクラウド環境の構成に特化した OS が提案・開発されている．これらの OS では，アプリケーション間の保護をハイパーバイザに任せることで機能を削減し，軽量化・省メモリ化を実現している．さらに，ベアメタルハイパーバイザ上で動作させることで，OS をアプリケーションに特化させることも可能である．一方で，プロセスの機能を提供しないため，マルチプロセスで動作するアプリケーションに対応することができない．また，ホスト OS の機能を利用することでマルチプロセスに対応した仮想化環境特化 OS の場合，ホスト OS が提供していない機能を実装することができない．本論文では，単一アドレス空間内で複数のプロセスを擬似的に実行させる手法を提案する．これによって，ホスト OS の存在を前提にせずマルチプロセスに対応することができ，既存のクラウド環境特化 OS と同様の軽量化を可能にする．

キーワード：クラウド・コンピューティング，仮想化，ライブラリ OS

1. Introduction

利用者に仮想マシン (VM) を提供する形態のクラウド環境が普及してきている．ある VM 内での不正な動作が他の VM に影響を及ぼさないよう，それぞれの VM 間は，ハイパーバイザによって隔離されている．このようなクラウド環境では，ひとつの VM 内で単一のアプリケーションのみを実行する構成が一般的であり，ゲスト OS は従来の汎用 OS が持つプロセス間の保護機能を必要としない．さらに，ゲスト OS 内では単一のアプリケーションのみが動作するため，カーネルとアプリケーションを隔離する必要もない．

これらの特徴に注目し，クラウド環境に特化した OS が開発されている [1], [2]．これらは，OS の機能を Library OS として実装し，アプリケーションとリンクする設計になっている．アプリケーション間の保護をハイパーバイザに任せることで，プロセス間の保護機能とカーネルの保護機能を削除し，軽量化・省メモリ化を実現している．ベアメタルハイパーバイザ上で動作するクラウド環境特化 OS の場合，アプリケーションに特化した機能を OS が提供することも可能である．一方で，これらのクラウド環境特化 OS は，プロセスの機能を提供していないため，マルチプロセスで構成されるアプリケーションを実行することができない．

また，ホスト OS の機能を利用することでマルチプロセスに対応する Library OS も開発されている [3]．ホスト OS の存在を前提にするため，ホスト OS が提供する機能に制限される．そのため，アプリケーションに特化した機能を OS が提供することができない．

既存のクラウド環境特化 OS では，1) アプリケーションに特化した機能を提供すること，2) 既存のアプリケーションを改変することなく動作させること，3) マルチプロセスアプリケーションに対応する，の 3 点を同時に満たすことができない．

本論文では，ホスト OS を前提としないクラウド環境特化 OS 上で，単一のアドレス空間における疑似マルチプロセス実行のプロセスモデルを提案する．単一アドレス空間上を複数の領域に分割し，プロセスを割り当てることで，単一のアドレス空間上での擬似的なマルチプロセス実行に対応する．

提案手法は，ホスト OS を前提としないため，アプリケーションに特化した機能を提供することが可能である．また，マルチプロセス実行に対応することで，既存のアプリケーションを再開発することなく動作させることができる．

本論文の構成は次のとおりである．2 章では，研究背景と関連研究について述べる．3 章では，提案システムにお

[1] 慶應義塾大学
Keio University

けるプロセスモデルとシステムの概要を述べる．4 章では，提案システムの設計と実装について述べる．5 章では，提案システムにおけるパフォーマンス評価について述べる．6 章では，本論文のまとめを述べる．

2. Background and Design Goals

2.1 Background

仮想マシンを用いたクラウド環境では，ひとつの VM 内で単一のアプリケーションのみを実行する構成が一般的になっている．このような環境では，ハイパーバイザが提供する VM 間隔離のための機能とゲスト OS が提供するプロセス間隔離のための機能が重複しており，パフォーマンス・メモリ使用量のオーバヘッドとなっている．ゲスト OS 上で単一のアプリケーションのみを実行する場合，ゲスト OS 内のプロセス間保護の機能やカーネル保護の機能は不要である．この重複を取り除くために，仮想化環境でゲスト OS として動作することを前提とした特化型 OS が設計・開発されるようになっている．

仮想化環境特化 OS の利点は次の 2 点である．1) VM 内でのプロセス間保護・カーネル保護を必要としないため，ゲスト OS を軽量にできる．2) VM 内で動作するアプリケーションに特化した OS が利用できる．

ハイパーバイザが VM 間の隔離を行うため，ある VM 内で発生した不正なアドレスへのアクセスなどの動作が他の VM に影響を及ぼすことがない．また，クラウド環境ではひとつの VM 内で単一のアプリケーションのみを実行する構成が一般的であるため，VM 内での保護機能が不要である．そこで，プロセス間保護やカーネルの保護を提供しないことで，ゲスト OS の軽量化が実現できる．

ベアメタルハイパーバイザを用いる場合には，アプリケーションに特化した OS を開発することも可能となる．通常の汎用 OS の場合，カーネルを介してしか使用することのできない CPU の特権命令などを，アプリケーションに特化した機能として提供することができる．この場合も，VM 間はハイパーバイザによって適切に隔離されているため，他の VM に影響を与えることはない．

一方，仮想化環境特化 OS の問題点として，1) カーネルを開発する必要がある 2) アプリケーションを特化 OS に合わせて設計・開発し直す必要がある 3) 実行できるアプリケーションに制限がある という 3 点が挙げられる．仮想化環境特化 OS は，新しい設計であり，既存の汎用 OS を流用することができない．したがって，新しい設計にもとづいてカーネルを開発する必要がある．カーネルの開発は難しく，コストが高い．また，特化 OS は既存の汎用 OS と同様の機能をすべて提供できるとは限らない．そのため，既存のアプリケーションを実行するためには，アプリケーションを特化 OS に合わせるよう変更しなければならない．さらに，プロセスの機能を提供しない特化 OS の場合，マルチプロセスから構成されるアプリケーションを実行することができない．

以下で，既存の仮想化環境特化 OS について詳しく述べる

2.2 Single-Process Library Operating System

isolation 機能の重複を解消するために，ゲスト OS 内では単一プロセスのみを動作させる OS として，Unikernels[1] や OSv[2] がある．これらはプロセス間保護の機能を提供しないことで，アプリケーションのオーバヘッドを削減し，省メモリ化を実現している．

OSv は，Linux API の多くを提供しており，既存のアプリケーションをほぼ変更することなく実行することができる．また，独自にゼロコピーのネットワーク API を提供しており，アプリケーションをこれらを利用するよう変更することで，更にパフォーマンスを向上できる．

Unikernels は，OS の機能を Library OS として実装し，アプリケーションにリンクすることで，オーバヘッドを削減している．しかし，OS が独自の API のみを提供するため，アプリケーションを新たに開発しなおす必要がある．

Unikernels と OSv に共通する問題として，ネットワークサーバやシェルスクリプトなどのマルチプロセスで動作するアプリケーションをそのまま実行することができないという点がある．これらの OS は単一プロセスを実行することに特化しており，プロセスの機能を提供していないためである．マルチプロセスアプリケーションへの対応としては，VM が軽量であることを利用し，プロセスの代替として複数の VM を起動し通信させる方法が提案されているが，VM 間通信はプロセス間通信と比較してオーバヘッドが大きい．また，VM 間でファイルやネットワークの状態などを共有するのは難しく，対応は容易でない．

2.3 Multi-Process Library Operating System

仮想化環境に特化した軽量 OS 内で複数のプロセスを実行する手法として picoprocess を利用する Graphene[3] がある．picoprocess はホスト OS 上のプロセスとして実行され，内部で LibraryOS として実装された軽量ゲスト OS とアプリケーションが動作する．picoprocess におけるゲスト OS は，ホスト OS の一部機能を利用することで軽量化を実現している．そのため，通常の汎用 OS をゲスト OS として使用する場合と比較して，オーバヘッドが小さい．また，Graphene は pipe に似たバイトストリームを用いて，効率的な picoprocess 間の通信・情報共有を可能にしている．これは VM 間通信よりオーバヘッドが小さいため，picoprocess を複数起動することで，マルチプロセスアプリケーションを効率的に実現している．

一方，picoprocess による特化 OS は，ホスト OS の機能を利用することで実現されているため，ベアメタルハイ

表 1　仮想化環境特化 OS の特徴

	アプリケーションに特化した機能の提供	既存アプリケーションの実行	マルチプロセス対応
OSv	○	△	×
Unikernels	○	×	×
Graphene	×	○	○
Dune	○	×	○

パーバイザ上で実行できない．ゲスト OS の実装がホスト OS の提供する機能に制限を受けるため，OSv や Unikernels と異なり，アプリケーションに特化した機能を提供することができない．ゲスト OS は，ホスト OS が提供している方法のみを使用できるため，ホスト OS が許可していない特権命令をアプリケーション内で使用させるといったことは不可能である．

他にも，仮想化技術を応用して，アプリケーションが安全にハードウェアの機能の使用できるようにするシステムとして Dune [4] がある．Dune は，ハードウェアの仮想化を利用することで，プロセス間を適切に隔離したうえで，安全にハードウェアに直接アクセスできるユーザプロセスとして Dune Process を提供する．Dune Process は，ホスト OS の制限を受けずに, アプリケーションの特性に応じたハードウェアの使用が可能であり，アプリケーションに特化した OS を開発するのと同等のパフォーマンスを実現することができる．また，Dune が提供するのは仮想マシンではなくプロセスであるため，マルチプロセスアプリケーションについても制限なくこれらの利点が適用できる．

しかし，Dune Process 内からの通常のシステムコール呼び出しは，ハイパーバイザを介した通所のハイパーコールになる．したがって，アプリケーションを Dune Process に合わせて最適化しなければ，パフォーマンス向上が期待できない．

2.4 Design Goals

提案手法の目的は以下の 3 点である．

(1) アプリケーションに特化した機能の提供を可能にする
(2) 既存のアプリケーションを改変することなく実行する
(3) マルチプロセスアプリケーションに対応する

2.2，2.3 で述べたように，既存の仮想化環境特化 OS はこれらを同時に満たすことができない (表 1)．本論文では，これらの 3 点を同時に満たす OS の設計のためのプロセスモデルを提案する．

3. Process Model and System Overview

3.1 Process Model

2.1 で述べたように，単一のアプリケーションのみを実行するゲスト OS 内では，ユーザプロセス間の isolation が不要であり，カーネルをユーザプロセスから保護する必要がない．そこで，本論文の提案システムでは，カーネル内で複数のユーザプロセスを実行させる．複数のプロセスを可能な限り単一のアドレス空間内で実行させ，ユーザプロセスをカーネル空間内で実行させる．

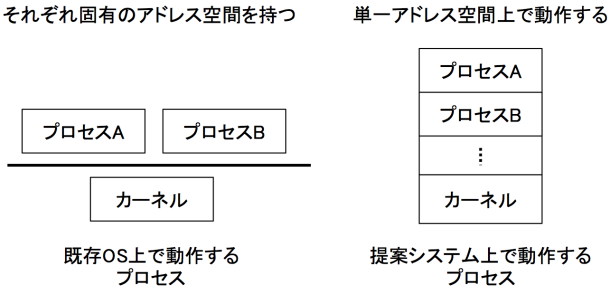

図 1　既存 OS におけるプロセスと提案システムにおけるプロセス

単一のアドレス空間を複数のユーザプロセスで共有することによって，コンテキストスイッチ時に Translation Lookaside Buffer (TLB) キャッシュをフラッシュする必要がなくなる．これによって，TLB キャッシュヒット率が向上しアドレス変換のコストを削減することができる．また，通常のプロセスの場合，カーネルスペースとユーザスペースの間でのメモリコピーが発生する．カーネルとユーザプロセスを同じアドレス空間内で実行させることで，メモリコピーを排除することができる．

通常のプロセスがカーネルの機能を利用するためにはシステムコールを使用する．システムコールは関数呼び出しと比較してオーバヘッドが大きい．ユーザプロセスをカーネル空間内で実行する場合，カーネルの機能をアプリケーションが直接関数呼び出しとして使用することが可能になるため，システムコールのオーバヘッドを回避することができる．

3.2 fork and exec

複数のユーザプロセスを単一のアドレス空間内で実行させるため，本論文ではアドレス空間を一定幅の領域に区切り，各プロセスは 1 領域内のアドレスのみを使用するようにする．

ただし，ユーザプロセスが fork する際は，親プロセスと子プロセスは同じ仮想アドレスを使用しなければならない．親プロセスのアドレス空間の内容を異なる番地にコピーしてしまうと，ポインタを使用したデータの整合性が保てず，正常に動作させられない．したがって fork 時に

子プロセスを同一アドレス空間の別領域に割り当てることはできない．

そこで，fork を実行した時点では，親プロセスのアドレス空間と同じ内容をもつ新しい仮想アドレス空間を作成する．このアドレス空間は通常の Linux のプロセス同様，Copy on Write で形成される．

fork によって作られた子プロセスが exec 系システムコールによってプログラムを実行する際に，複数のプロセスで共有して使用するアドレス空間内でプログラムを実行するよう統合する．exec 系システムコールを発行した場合は，それまでのメモリの状態はすべて破棄され，仮想アドレス空間はプログラムを実行するために初期化される．そのため，ポインタを使用したデータの整合性の問題が生じない．そこで，exec 系システムコールが発行されたタイミングで，それまで子プロセスが使用していた仮想アドレス空間を破棄し，共有アドレス空間内の未使用領域にプログラムを展開する．

また，共有アドレス空間を使用しないプロセスのページテーブルは Tagged TLB を用いて管理する．Tagged TLB を用いることで，コンテキストスイッチ時の TLB フラッシュを回避することができる．通常，Tagged TLB で使用できるタグの数には限りがあるため，仮想アドレス空間ごとにタグを割り当てると，タグの枯渇が問題となる．Unix 系のプログラムでは，fork をした直後に exec を用いて別プログラムを起動するというのが fork の典型的な利用例であるため，本システム上で共有アドレス空間を使用せず固有のアドレス空間をもつプロセスの数は少ない．そのため，各アドレス空間ごとにタグを割り当てた場合でも，タグの枯渇が問題になりにくい．

4. Design and Implementation

本論文では提案するプロセスモデルを Linux 4.4.10 を対象に実装した．Position Independent Executable (PIE) として出力された ELF バイナリを対象に，仮想アドレス空間の併合を行う．プロセス間で仮想アドレス空間を共有する場合，通常のバイナリが配置されるアドレス範囲と異なるアドレスに配置される．PIE 形式でないバイナリは配置を変更することができないため，exec 時に共有アドレス空間を使用することができない．本論文では，PIE 形式のバイナリのみを共有アドレス空間内で実行する．

仮想アドレス空間を複数に分割するために，Linux のメモリ管理に領域を表す構造体を追加する．ページテーブルごとに，各領域の使用状況と参照カウントをもたせる．共有アドレス空間への exec を行う際には，空いている領域を探索する．

また，スケジューラを改変し，異なるプロセスへのコンテキストスイッチ時であっても，参照するページテーブルが同一であれば，ページテーブルの切り替えと TLB キャッシュのフラッシュを行わないようにする．

4.1 fork

exec 時に親プロセスのアドレス空間に併合できるようにするために，各プロセスがもつ mm は，親プロセスのページテーブルや使用状況をもつ必要がある．そこで fork を改変し，仮想アドレス空間を複製する際に，複製元となった仮想アドレス空間をもたせる．仮想アドレス空間の複製自体は Linux の既存実装をそのまま利用し，Copy on Write で複製する．ここで，複製先の仮想アドレス空間の領域使用状況を初期化し，親プロセスが使用していた領域のみを使用中とマークし，参照カウントを 1 に初期化する．

4.2 exec

通常の Linux の exec は以下のように実行される．

(1) 新しい空の仮想アドレス空間を作成する
(2) 環境変数 / 実行時引数をもとのアドレス空間からコピーする
(3) 実行するファイルの種類を識別する
(4) mmap やファイル種類の応じたローダを用いて実行ファイルをメモリ上に展開する
(5) もとのアドレス空間を破棄し，新しいアドレス空間を使用するよう切り替える

この手順のうち，手順 4 で実行ファイルの展開先を共有アドレス空間の空き領域に切り替えることで，アドレス空間の併合を行う．また，この際にもとのアドレス空間を破棄し，ページテーブルを共有アドレス空間に切り替える．

Linux における PIE 形式の ELF バイナリのロードは，ld-linux.so を通じて行われる．ld-linux.so は，配置するアドレスとバイナリファイルを受け取り，メモリ上に展開する．そこで，ld-linux.so に指定するアドレスを共有アドレス空間の空き領域に変更する．

また，環境変数と実行時引数は exec 前のプロセス固有のアドレス空間上にある．そのため，ページテーブルを共有アドレス空間に切り替えてもとのアドレス空間を破棄する前に，これらを共有アドレス空間の特定領域にコピーする．

5. Evaluation

提案システムによってプロセス間のコンテキストスイッチのパフォーマンスが向上することを確認するため，qemu-kvm [5], [6] 上でコンテキストスイッチを頻繁に発生させるプログラムを実行し，性能を評価した．評価環境を表 2 にまとめる．

実験では，コンテキストスイッチを意図的に誘発するために，sched_yield を親プロセス・子プロセスの両方で繰り返し実行するプログラムを実行した．実験結果を表 3 に示す．

表 2　評価環境

項目	詳細
Linux Kernel	4.4.10
CPU	Intel Core i7-3820QM
RAM	16 GB

表 3　sched_yield の呼び出し回数と実行時間

実行回数	提案システム (usec)	Linux(usec)
100,000	158294.4	224670.5
1,000,000	1264861.5	2295291.8

通常の Linux に対し，提案システムでは最大で 45%パフォーマンスが向上している．提案システムでは，同一アドレス空間内でのコンテキストスイッチではページテーブルを変更せず TLB キャッシュのフラッシュも行わない．実験では 2 つのプロセスを単一アドレス空間内で実行しているため，コンテキストスイッチが軽量になることでパフォーマンスが向上した．

6. Related Work

既存のクラウド環境特化 OS には，Unikernels [1]，OSv [2]，EbbRT [7] などがある．これらはクラウド環境の性質に特化することでパフォーマンスの向上を実現している．

Unikernels，EbbRT は，Library OS のデザインになっており，アプリケーションとライブラリ OS をリンクする．アプリケーションと OS が同じ空間で動作するため，プロセスのオーバヘッドを削減できるが，プロセスの機能を提供しない．

OSv では，プロセスによるオーバヘッドを回避しつつ，Linux の提供する API の多くをサポートしている．これによって既存のアプリケーションの多くを，改変することなく実行することが可能になっている．

これらの OS は，ベアメタルハイパーバイザ上で動作し，アプリケーションに特化した機能を提供することが可能である．一方，プロセスを提供しないため，マルチプロセスで構成されるアプリケーションを実行できない．VM の起動やオーバヘッドが十分に軽量であることを活かし，プロセスの代わりに VM を起動し VM 間通信をすることで，擬似的にマルチプロセスに対応することも可能だが，VM 間通信のオーバヘッドは大きく共有情報も持ちにくいため難しい．

また，Library OS と仮想化技術を応用し，マルチプロセスに対応する OS のデザインとして，Drawbridge [8] や Graphene [3] がある．これらは picoprocess という形式でゲスト OS を実行する．picoprocess はホスト OS のプロセスとして実行され，picoprocess 内で Library OS とアプリケーションが動作する．picoprocess 内の Library OS は，アプリケーションを実行するのに必要な最小限の機能のみをもち，ホスト OS の機能を活用する．picoprocess はホスト OS から見た場合，プロセスと同様の isolation であるため，VM 間通信より軽量な通信が可能であり，picoprocess 間の共有情報をもつことも容易である．しかし，ホスト OS の機能を利用するため，ホスト OS が提供していないハードウェアの機能などをアプリケーションに提供することはできず，アプリケーションに特化した機能の提供が不可能である．

Arrakis [9]，IX [10] は，コントロールプレーンとデータプレーンを分離し，アプリケーションからデバイスの機能を使用可能にする OS の設計を提案している．通常アプリケーションはカーネルを経由してデバイスを操作し IO 処理を行うが，カーネルを経由することによるオーバヘッドが大きい．そこで，Arrakis，IX はアプリケーションから少ない OS の関与でデバイスを操作することを可能にし，オーバヘッドを回避している．Arrakis，IX はプロセス間保護の機能を維持しながら，デバイスを操作するオーバヘッドを削減することを実現しているが，本論文では，プロセス間保護の機能を削減することでパフォーマンス向上・省メモリ化を実現する．

7. Conclusion

クラウド環境の特徴に着目しアプリケーションのパフォーマンス向上・省メモリ化を実現するクラウド環境特化 OS 上で，マルチプロセスで構成されるアプリケーションに対応する方法として，アドレス空間を複数の区間に分割しプロセスごとに割り当てるプロセスモデルを提案した．これによって，ベアメタルハイパーバイザ上で動作しアプリケーションに特化した機能を提供できるクラウド環境特化 OS 上で，マルチプロセスアプリケーションを改変することなく動作させることが可能となる．

提案したプロセスモデルでは，複数のプロセスを可能な限り単一のアドレス空間内で実行する．exec 時にプログラムを共有アドレス空間上で実行させ，同一アドレス空間内のプロセス間コンテキストスイッチ時にはページテーブルを書き換えないようにすることで，コンテキストスイッチのコストを削減する．

提案システムを Linux に対して実装し，コンテキストスイッチを頻繁に繰り返すプログラムを実行させたところ，最大で 45% のパフォーマンス向上を達成した．

今後の課題として，共有メモリやパイプといったプロセス間通信機能の最適化や，カーネルとユーザプロセスの isolation の削除がある．本論文ではプロセスモデルのみを実装したが，このモデル上でカーネルの保護を取り除き，システムコールのオーバヘッド削減やゼロコピー API の提供などを行うことで，パフォーマンスの向上を期待できる．

参考文献

[1] 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), ACM, pp. 461–472 (2013).

[2] Kivity, A., Costa, D. L. G. and Enberg, P.: OS v— Optimizing the Operating System for Virtual Machines, Proceedings of USENIX ATC' 14: 2014 USENIX Annual Technical Conference, p. 61 (2014).

[3] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D. and Porter, D. E.: Cooperation and security isolation of library OSes for multi-process applications, *Proceedings of the Ninth European Conference on Computer Systems*, ACM, p. 9 (2014).

[4] Belay, A., Bittau, A., Mashtizadeh, A. J., Terei, D., Mazières, D. and Kozyrakis, C.: Dune: Safe User-level Access to Privileged CPU Features., *Osdi*, Vol. 12, pp. 335–348 (2012).

[5] Bellard, F.: QEMU , a Fast and Portable Dynamic Translator, *USENIX Annual Technical Conference. Proceedings of the 2005 Conference on*, pp. 41–46 (2005).

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

[7] Schatzberg, D., Cadden, J., Dong, H., Krieger, O. and Appavoo, J.: EbbRT: A Framework for Building Per-application Library Operating Systems, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, (OSDI'16), Berkeley, CA, USA, USENIX Association, pp. 671–688 (2016).

[8] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R. and Hunt, G. C.: Rethinking the library OS from the top down, *ACM SIGPLAN Notices*, Vol. 46, No. 3, ACM, pp. 291–304 (2011).

[9] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T.: Arrakis: The Operating System is the Control Plane, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, (OSDI'14), Berkeley, CA, USA, USENIX Association, pp. 1–16 (2014).

[10] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C. and Bugnion, E.: IX: A Protected Dataplane Operating System for High Throughput and Low Latency, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, (OSDI'14), Berkeley, CA, USA, USENIX Association, pp. 49–65 (2014).