軽量 OS を用いた並列処理のための仮想マシン間通信機構

2014SE013 古川善土　2014SE035 伊藤拓也　2014SE084 酒井優弥
指導教員：宮澤元

1　はじめに

近年，ネットワーク経由でサービスを提供するクラウドコンピューティング (クラウド) が普及している．クラウドではデータセンタに置かれた複数の計算機を多数のユーザが利用する．そのため，ユーザの一人ひとりに CPU やメモリ，ストレージなどの物理資源を仮想マシン (VM) の形で提供し効率的に運用している．クラウドでは，特に必要な資源を必要なときに必要なだけ利用することができる特徴から科学技術計算などのハイパフォーマンスコンピューティング (HPC) 用途にクラウドを用いる HPC クラウドが注目されている．

しかし，仮想マシンは物理マシンに比べパフォーマンスが劣ることが一般的である．例えば，ハイパーバイザが VM をスケジューリングしさらに VM が内部で動いているプロセスをスケジューリングした場合，スケジューリング処理の多重化が起こりオーバーヘッドが発生する．あるいは，ディスクやネットワークにアクセスするときに発生する I/O エミュレーションでもオーバーヘッドは発生する．これらのオーバーヘッドが VM のパフォーマンスに悪影響を及ぼす．

これらの問題を解決するアプローチとして，ライブラリ OS の技術を用いた Unikernels[1] が提案されている．クラウドサービスに特化したアプリケーションを動作させることを前提に，この Unikernels ではアプリケーションをライブラリ OS とリンクする形で原則としてシングルスレッドで動作させることによってハイパーバイザ型仮想化のオーバーヘッドを低減する．そのため，Unikernels を用いて複数スレッドを持つようなアプリケーションを動作させるには，複数の VM のそれぞれをスレッドとし相互に通信させる形で実現する必要がある．

先行研究 [2] では並列処理に用いる Unikernels をグループにまとめ，グループを考慮してスケジューリングを行うことで実行効率を改善している．しかし，グループ内でのデータの共有は VM 間のネットワーク通信で行われているので VM 間のデータの共有にオーバーヘッドが存在する．

本研究の目的は，Unikernels のようなアプローチにおいて VM 同士の高速な通信を可能とすることでマルチスレッドアプリケーションの効率的な実行を実現することである．ハイパーバイザが VM 間で利用できる共有メモリを提供することにより，同一物理マシン上の VM 同士が高速に通信することができる．

2　研究の背景

クラウドにおけるサーバーの運用では，様々な仮想化技術が提案され利用されている．本節では，軽量 OS を用いた並列処理が必要である背景として，ハイパーバイザ型の仮想化，コンテナ型の仮想化，軽量 OS について述べる．

2.1　ハイパーバイザ型の仮想化

ハイパーバイザを用いて 1 台の物理マシンに複数の VM を実行する方式をハイパーバイザ型の仮想化という．VM はプロセッサやメモリと言ったコンピュータリソースを論理的に割り当てられ，ハードウェア上で直接動作するハイパーバイザによって管理される．代表的なハイパーバイザに，Microsoft の Hyper-V[3] や VMware の ESXi[4]，Xen Project の Xen[5] などがある．VM は独立した 1 つのマシンとして動作するため，VM 毎に異なる OS を実行することができるなど自由度が高い．一方，1 台の物理マシン上で動作する複数の VM 上で，物理マシン用の OS がそのまま動作するので，スケジューリングやメモリ管理の多重化，I/O エミュレーションなどによるオーバーヘッドが発生し性能低下を引き起こすなど，コンピュータリソースへの負荷も大きい．

図 1 にハイパーバイザ型仮想化の構造を示す．ハードウェア上で動作するハイパーバイザによって複数の VM を動作させることができる．各 VM にはそれぞれ個別に OS がインストールされ，その上でアプリケーションが動作する．

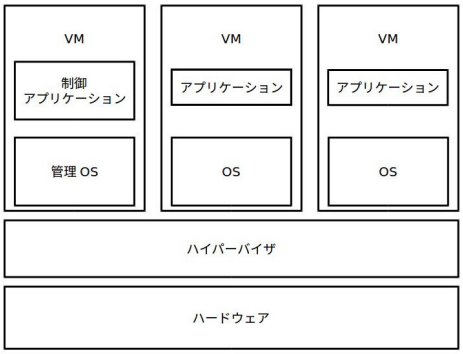

図 1　ハイパーバイザ型仮想化の構造

2.2　コンテナ型の仮想化

複数の VM を実行する際のオーバーヘッドを改善する手法の 1 つに，Docker[6] などのコンテナ型仮想化技術を利用したものがある．コンテナ型仮想化では 1 つの OS の上にコンテナと呼ばれる分離された空間を複数作成することができる．ハイパーバイザがハードウェアをエミュレーションして VM を実行させるのに対して，コンテナは 1 つの OS 上で通常のプロセスと同様に実行されるので，コンテナの起動や終了は高速で，ハイパーバイザに見られるよ

うなオーバーヘッドは比較的小さい．一方で，1 つのコンテナを複数の物理マシンに跨るように配置して並列に処理を行うことが難しいという問題がある．

コンテナ型仮想化の構造を図 2 に示す．1 つの OS 上に複数の互いに隔離された空間 (コンテナ) を作成し，アプリケーションはコンテナ内で動作する．

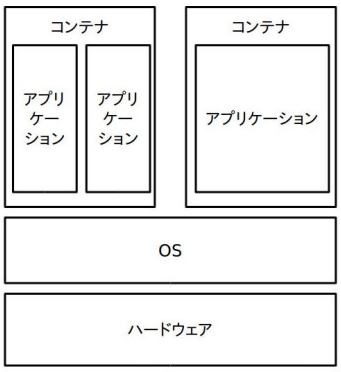

図 2 コンテナ型仮想化の構造

2.3 Unikernels

ハイパーバイザ型仮想化のオーバヘッドを低減する手法に Unikernels がある．Unikernels はライブラリ OS を用いてアプリケーションを軽量 OS として実現し，VM 毎に単一のサービスを提供する．アプリケーションの動作に必要な OS の機能をライブラリとして実装し，アプリケーションとともにコンパイルすることで，OS の軽量化やメモリフットプリントの削減が実現できる．また単一のサービスのみを提供するので，VM として実行された際のタスクスケジューリングやメモリ管理を簡素化し，仮想化によるオーバーヘッドを改善できる．

Unikernels の 1 つに MirageOS[7] がある．MirageOS では Mirage コンパイラを用いて，ソフトウェアスタックや言語ランタイム，システムライブラリをコンパイルし最適化することで軽量化を実現している．また，複数の VM を用いてアプリケーションを構成し，VM 間通信を行うことでマルチスレッドのような並列処理を実現することができる．しかし，VM 間通信はネットワークを介したメッセージ通信として実現されるので，オーバーヘッドが大きいという問題がある．

2.4 複数の VM による並列処理の問題

複数の VM をスレッドに見立てて並列処理を行う場合にまず問題になるのは VM 間の通信である．同一プロセス内の各スレッドはメモリ空間が共有されている．しかし，VM のそれぞれが持つメモリはハイパーバイザによって擬似物理アドレスが割り当てられ VM ごとに隔離されているので，一方の VM が持つデータを他方の VM が認識することはできない．そこで VM をスレッドに見立て並列処理を行う場合には，各 VM が他の VM にデータを送る必要がある．このとき，図 3 のようにハイパーバイザは物理メモリ上のデータのコピーを行う．並列処理の途中にメモリ上でデータのコピーが発生することは効率的ではない．

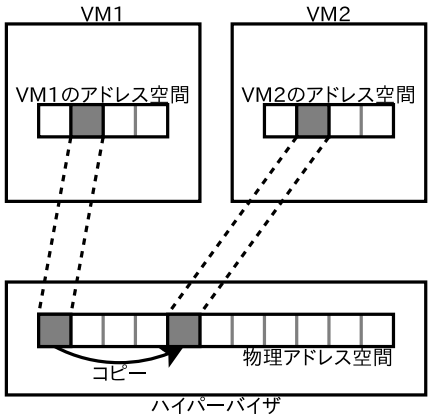

図 3 VM 間メッセージ通信時のデータコピー

3 アプローチ

本研究では Unikernels 型 VM で効率的にマルチスレッドアプリケーションを動作させるために，共有メモリを用いた VM 間通信機構を提案する．通信データを共有メモリを介してやり取りし，VM 間で高速な通信を実現できる．共有メモリはハイパーバイザがそれぞれの VM の持つメモリ空間にマップすることで構成される．

3.1 共有メモリによる VM 間通信のインターフェース

本 VM 間通信機構では，各 VM をまとめて管理しているハイパーバイザが共有メモリを提供する．ハイパーバイザは並列処理の実行に際して，新たに共有メモリのためのメモリ領域を確保する．ハイパーバイザは物理メモリ上で各 VM 用に確保したメモリ領域を仮想マシンのアドレス空間にマップしている．共有メモリを使う場合は物理メモリの同じ領域を共有するすべての VM の仮想マシンアドレスにマップする．VM 間の共有メモリを実現した場合，共有されたメモリ領域を複数の実行単位が同時にアクセスするので共有メモリにアクセスする VM 間の同期に注意が必要となる．各 VM で並列処理を行うアプリケーションはお互いのデータを破壊しないようにするため，これまでのマルチスレッドアプリケーションのようにセマフォやロックを用い競合状態を避けるように実装されなければならない．通常の OS 上のマルチスレッドアプリケーションでは各スレッドが一つのメモリ空間上で動いている．従来の Unikernels 型の VM でこれを模して実行する場合にはメッセージ通信で各 VM のデータを共有する必要がある．共有メモリを使うことでマルチスレッドアプリケーションをより自然に Unikernels 型の VM で実現することができる．

3.2 共有メモリによる VM 間通信の実現

図 4 はハイパーバイザが複数の VM に共有メモリを提供している様子を表している．ハイパーバイザは各 VM それぞれに隔離されたメモリ領域を用意し，VM の持つ仮想マシンアドレスにマップしている．通常，ハイパーバイザは同じ物理メモリの領域を同時に複数の仮想マシンアドレスにマップしない．VM が共有メモリを必要とした場合，ハイパーバイザの持つ物理メモリの一部の領域が共有メモリとして確保される．このとき共有メモリだけは特例として複数の仮想マシンアドレスにマップされることが許可される．共有メモリは各 VM から同時にアクセスされるので VM 間のデータの共有を高速に行うことができる．また，仮想マシンアドレスが同じになるようにマップした場合，データをコピーすることなく各 VM がポインタ変換などを行わずにデータ構造を共有することもできる．

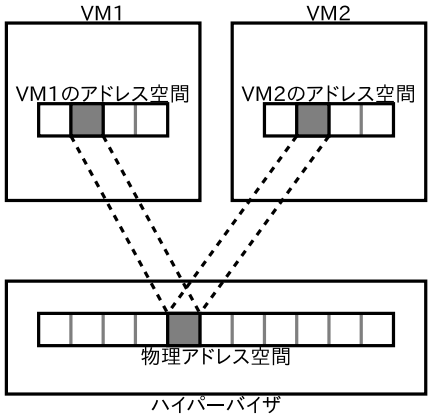

図 4 各 VM で同一に割り当てられたアドレス空間

3.3 共有メモリの利用

実際に VM が共有メモリを利用する手順を説明する．並列処理を行う VM が起動してから共有メモリを用意するため，ハイパーコールを使いハイパーバイザを呼び出す．ハイパーバイザから共有メモリが提供され，必要な VM が共有メモリを確認できたところで並列処理が開始される．実行中は同期機構を用いて排他制御を行う．

4 実装

複数の VM でメモリアドレス空間を共有する機能の実装について述べる．ハイパーバイザは Xen 4.6.5 を用いた．また，共有メモリにアクセスする Linux カーネルモジュールを作成した．

4.1 Xen を用いた VM 間の共有メモリ

VM 間の共有メモリを実現するために，Xen ハイパーバイザに以下の実装を行った．

4.1.1 共有メモリの確保

Xen が起動したときに共有メモリ用にマシンアドレスを確保する．Xen が管理するメモリ領域はマシンアドレスと呼ばれる．共有メモリのサイズは，第 5 章の実験を行うために充分なサイズとして，1025 ページとする．

4.1.2 仮想アドレスへの共有メモリのマッピング

共有メモリを VM が指定する仮想マシンアドレスにマップする関数 do_mapping_share_memory を追加した．現在は共有メモリに対するアクセス制御を行なっておらず，どの VM も共有メモリをマップできる．

4.1.3 複数の VM からのマッピングを許可

マッピングが行われるとき，共有メモリは例外としてエラーを出さないようにソースコードを変更した．VM 間の共有メモリを実現するためには，複数の VM からマップする必要がある．通常 Xen では物理メモリを複数の VM にマップすることはできないが，共有メモリは例外として複数 VM へのマップを許可することとした．

4.2 インタフェース

VM が共有メモリをマップするためのインタフェースとして mapping_share_memory ハイパーコールを作成した (図 5)．引数 va に共有メモリをマップする VM の仮想マシンアドレスを指定する．このハイパーコールによってハイパーバイザが呼び出されると，4.1.2 節の関数 do_mapping_share_memory が実行され，指定した仮想アドレスから始まる 1025 ページへ共有メモリをマップする．

```
int mapping_share_memory(unsigned long va)
```

図 5 共有メモリをマップするインタフェース

なお，このインタフェースはハイパーコール 39 番に割り当てている．ハイパーコール 39 番は XenClient のために予約された番号で，Xen では使用されていないので，今回の実装で使用した．

4.3 Linux カーネルモジュールの作成

VM から共有メモリにアクセスするために Linux カーネルモジュールを作成した．本来は Unikernels を使う想定ではあるが実験を容易に行えるように Linux を用いた．Linux の仮想アドレスにはユーザプロセス空間とカーネル空間があるが，カーネル空間を利用するためにカーネルモジュールとして実装した．このカーネルモジュールがロードされたとき，以下の手順により共有メモリにアクセスする．

1. 仮想アドレスを確保する．
2. mapping_share_memory ハイパーコールを呼び出し，Xen に対して共有メモリへのマッピングを要求する．引数として，確保した仮想アドレス空間の先頭アドレスを指定する．
3. 仮想アドレスに対してアクセスを行う．

5 実験

第 4 章で実装した共有メモリによる VM 間通信機構を用いると，仮想ネットワークを用いたメッセージ通信より高速な VM 間通信が可能となることを示す実験を行った．共有メモリを使った場合とメッセージ通信を使った場合のそれぞれについてデータを VM 間通信で往復させるのにかかった時間を計測し，比較する．実験では，VM 間での共有メモリを使用して 2 つの VM 間で 8192 個の int 型の数値を往復させた．メッセージ通信を使用した場合も同様に数値を往復させた．

5.1 実験環境

実験で使用した PC の仕様を表 1 に示す．この PC 上で本システムを実装した Xen を実行した．実験では，表 2 に示す VM を 2 つ作成した．

表 1 PC の仕様

CPU	Intel(R) Core(TM) i7-3770
クロック周波数	3.40GHz
コア数	4 コア 8 スレッド
メモリ	8GB
HDD 容量	1TB
OS	Ubuntu 16.04

表 2 VM の仕様

CPU コア数	1
メモリ	128MB
HDD 容量	4GB
OS	Ubuntu 16.04

5.2 実験結果

共有メモリを使った場合とメッセージ通信を使った場合の往復の通信時間を計測した．それぞれ 10 回ずつ計測を行い往復の通信時間の平均値を求めた．実験結果を表 3 に示す．

表 3 データ転送の時間

	時間 (μ s)
共有メモリ	116
メッセージ通信	14978

メッセージ通信を用いた場合，通信しているのはユーザプロセス同士であり，ユーザプロセスと VM のカーネルとのコンテキスト切替が行われるので表 3 の通信時間を直接比較することはできない．参考までに，gettimeofday システムコールにかかる時間を計測して，ユーザプロセスと VM のカーネルとのコンテキスト切替にかかる時間を見積もって見たところ，10326 マイクロ秒であった．表 3 のメッセージ通信の結果からこの数字を差し引くと，メッセージ通信を用いた場合，VM のカーネル同士はおよそ 4000 から 5000 マイクロ秒で通信を行うことができると考えられる．

この実験結果から，第 4 章で実装した VM 間の共有メモリを用いるとメッセージ通信を用いる場合より高速にデータを転送できると考えられる．

6 おわりに

我々は Unikernels のようなアプローチを用いたマルチスレッドアプリケーションを効率的に実現するための手法として，VM 間の共有メモリを用いた VM 間通信機能を提案した．VM 間の通信に共有メモリを用いることで，VM 間通信時に発生するオーバーヘッドを削減し並列処理を効率化できる．また，提案した機能を Xen ハイパーバイザに実装し共有メモリとメッセージ通信の比較実験を行った．実験結果より本論文で想定するような環境において共有メモリの効果を確認した．

本研究では提案手法に対して Linux を用い実装や実験を行ったが，今後は Unikernels を用いて提案手法の効果を確認する必要がある．

参考文献

[1] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand and Jon Crowcroft: “Unikernels: Library Operating Systems for the Cloud”, http://unikernel.org/files/2013-asplos-mirage.pdf (2013).

[2] 田尻 翔太，遠山 恭平：“Unikernel クラウド向けの VM グループ管理”，南山大学情報理工学部卒業論文 (2014).

[3] Microsoft: “Hyper-V の概要”, https://technet.microsoft.com/ja-jp/library/hh831531(v=ws.11).aspx (2018).

[4] VMware: “vSphere ESXi ベアメタル ハイパーバイザー”, https://www.vmware.com/jp/products/esxi-and-esx.html (2018).

[5] Xen Project, A Linux Foundation Collaborative Project: “The Xen Project, the powerful open source industry standard for virtualization.”, https://www.xenproject.org/ (2017).

[6] Docker Inc: “Docker - Build, Ship, and Run Any App, Anywhere”, https://www.docker.com/ (2017).

[7] “MirageOS”, https://mirage.io/ (2017).