VM 内コンテナを用いたサービス単位のオートスケール機構

植木 康平　光来 健一

IaaS 型クラウドでは仮想マシン (VM) 内で動作するサービスの負荷に対応するために，オートスケールと呼ばれる機能が提供されている．オートスケールにより，VM の負荷が高まった時には自動的に VM の数を増やすスケールアウト処理が行われる．しかし，1 つの VM で複数のサービスを動作させている場合，その中の 1 つのサービスに負荷が集中したとしても VM 全体がスケールアウトされる．その結果，新しく起動される VM においてスケールアウトが不要なサービスも動作するため，必要以上に大きな VM を用意しなければならず，スケールアウトを完了させるのにも時間がかかる．本稿では，VM 内の各サービスをコンテナを用いて動作させることにより，サービス単位でのきめ細かいオートスケールを実現するシステム Ciel を提案する．Ciel では，VM 内コンテナに対して VM イントロスペクションと呼ばれる技術を用いることで，各サービスのリソース使用量を VM の外部から正確に監視する．負荷の高いコンテナが見つかれば，そのコンテナが動作するのに必要十分な大きさの VM を新たに作成し，その VM ではスケールアウトする必要のあるコンテナだけを起動する．これにより，VM のコストとスケールアウトにかかる時間を最小限に抑えることができる．我々は Ciel を Xen と Docker を用いて実装し，有用性を確認する実験を行った．

1　はじめに

近年，Amazon AWS や Microsoft Azure をはじめとするクラウドサービスの利用が広まってきている．その利用形態の一つである IaaS 型クラウドでは利用者に仮想マシン（VM）の提供を行う．VM には利用者の用途に応じて搭載する OS やアプリケーションなどを自由にインストールすることが可能である．VM のメモリ量や仮想 CPU 数，ディスク容量などはあらかじめ用意されたテンプレートの中から選択することが多いが，リソースごとにきめ細かく設定できるクラウドも存在する．利用料金は VM 単位で課金され，大きな VM ほど単位時間あたりのコストがかかる．一般的に，仮想 CPU の性能や個数，メモリ量，ディスク容量に比例した料金体系となっている．

クラウドでは VM 内で動作するサービスの負荷上昇に対応するために，スケールアウトと呼ばれる処理を行う．スケールアウトは VM の負荷が高まった際に，新たに VM を起動することで負荷を分散させて各 VM の負荷上昇を抑える手法である．多くのクラウドでは，VM の負荷に応じて自動的にスケールアウトを行う機能を提供しており，この機能はオートスケールと呼ばれる．オートスケール機能は VM の負荷を監視し，負荷の上昇を検知した際にスケールアウトを行う．

しかし，1 つの VM 内で複数のサービスが動作している場合，その中の 1 つのサービスの負荷が高まったとしても VM 全体がスケールアウトされる．その結果，必要以上に大きな VM を起動することになり，コストが上昇する．また，不要なサービスの起動にもリソースが消費されるため，スケールアウトの完了にも時間がかかる．本来，高負荷なサービスのみをスケールアウトするべきであるが，複雑に絡み合った個々のプロセスやファイルがどのサービスで使用されているかをクラウド側から正確に把握するのは容易ではない．そのため，VM 外部から各サービスのリソース使用量を監視するのも困難であり，VM 全体のリソース使用量やサービスへのリクエストを基に推

Kohei Ueki，Kenichi Kourai, 九州工業大学, Kyushu Institute of Technology.

測するしかない．

本稿では，VM 内の各サービスをコンテナを用いて実行することにより，サービス単位の柔軟なオートスケールを実現するシステム Ciel を提案する．コンテナの中でサービスを実行することにより，各サービスが使用しているリソースを容易に特定することができる．Ciel では VM イントロスペクション [6] を用いて VM の外から VM 内コンテナを監視することで，各サービスのリソースの使用量を監視する．負荷の高いコンテナがあれば，そのコンテナが動作するのに必要十分な VM を起動し，新たに起動した VM ではスケールアウトする必要のあるコンテナだけを起動する．

我々は Ciel を Xen 4.6.0 [3] と Docker 17.05.0-ce [14] に実装した．VM 外部から VM のメモリ上にある Linux の cgroup の階層構造を解析し，コンテナのリソース使用量を取得できるようにした．このリソース監視機能は OS の仮想アドレスを VM の物理メモリアドレスに変換する必要があるため，LLVM を用いてプログラム変換を行う LLView [25] を使用した．Ciel を用いてスケールアウトにかかる時間やスケールアウト時のリソース消費量を測定し，従来システムと比べて大幅に改善できることを確認した．

以下，2 章で従来のオートスケール手法とその問題点について述べる．3 章で VM 内コンテナを用いてオートスケールを行う Ciel を提案し，4 章でその実装について述べる．5 章で Ciel の有用性を調べた実験について述べる．6 章で関連研究に触れ，7 章でまとめと今後の課題について述べる．

2 従来のオートスケール

クラウドでは VM 内で動作するサービスの負荷に対して柔軟に対応するために，スケールアウトおよびスケールインと呼ばれる処理を行う．スケールアウトは VM の負荷が高まった際に，図 1 のように新たに VM を起動することで負荷を分散させて VM の負荷の上昇を抑える手法である．例えば，Web サーバが動作している VM の CPU 使用率が 80%を超えた時には，新しい VM を起動してリクエストを分散させることにより，それぞれの VM の CPU 使用率を

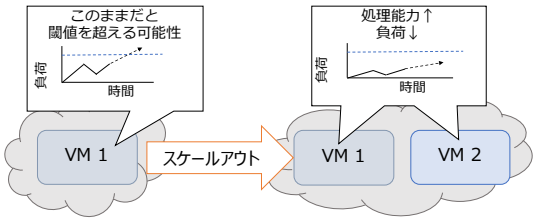

図 1　VM のオートスケール

40%程度に低下させることができる．一方，スケールインはスケールアウトによって増加した VM の負荷が低くなった際に，コストを削減するためにいくつかの VM を停止する手法である．例えば，2 つの VM の CPU 使用率がそれぞれ 20%であれば，一方の VM を停止させても VM の負荷は 40%程度に抑えられる．

多くのクラウドでは，VM の負荷に応じてスケールアウトとスケールインを自動的に行うことが多く，この機能はオートスケールと呼ばれる．オートスケール機能は VM の負荷を監視し，負荷の上昇を検知した際にはスケールアウトを行う．逆に，スケールアウトによって増やした VM の負荷が低いことを検知した際にはスケールインを行う．負荷上昇の検知は VM 全体のリソース使用量に基づいて行われ，VM の CPU 使用率やメモリアクセス量，ディスクアクセス量，接続クライアント数などの推移によるトレンド分析が主流となっている．

クラウドでは，1 つのサービスに対して 1 つの VM を割り当てることも多いが，複数のサービスを 1 つの VM に集約することでリソースをより有効に活用し，コストを削減することができる．しかし，VM 内の特定のサービスに負荷が集中することによって VM の負荷が高まった場合でも，VM 全体をスケールアウトすることになる．そのため，必要以上に大きな VM を新たに作成して起動しなければならず，その分だけ VM にかかるコストが上昇する．また，新しく起動した VM ではスケールアウトの必要がないサービスも起動されるため，スケールアウトの完了にも時間がかかる．これは VM 内の複数サービス間でリソース競合が引き起こされ，サービスの起動が遅くなるためである．

本来，VM の中で高負荷の原因となったサービスの

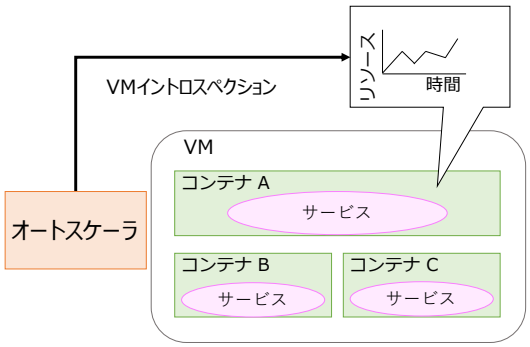

図 2　Ciel のシステム構成

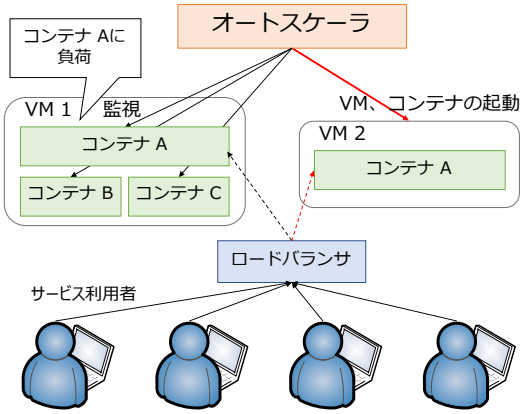

図 3　Ciel におけるオートスケール

みをスケールアウトすることが望ましい．しかし，複雑に絡み合ったプロセスやファイルなどがどのサービスによって使われているかを VM の外部から特定するのは困難である．その上，サービスごとのリソース使用量の監視についても，VM 外部のクラウド側からでは各サービスに送られたリクエストを基に推測するしかなく，正確な監視，計測は困難である．VM 内部で監視を行う手法も考えられるが，監視ソフトウェアをクラウドの利用者にインストールさせる負担が大きく，監視データを外部に送信する必要もある．

3　Ciel

本稿では，VM 内コンテナを用いて各サービスを実行することにより，サービス単位でのより柔軟なオートスケールを行うシステム Ciel を提案する．図 2 に Ciel のシステム構成を示す．コンテナは計算機全体を仮想化する VM とは異なり，OS が提供する仮想実行環境である．Ciel では VM 内のサービスごとにコンテナを起動し，コンテナ内で 1 つのサービスだけを動作させる．これにより各サービスが使用するプロセスやファイルなどが分離されるため，サービスごとのリソース使用量を正確に計測することが容易になる．

VM 内コンテナのリソース使用量を VM 外部から監視するために，Ciel は VM イントロスペクション [6] と呼ばれる技術を用いる．VM イントロスペクションは VM の外部から VM 内部の情報を取得する手法である．例えば，VM のメモリを解析することで OS が管理している CPU 使用時間，メモリ使用量，ディスクアクセス量などの情報を取得することができる．その際に，取得する OS データの仮想アドレスを VM の物理メモリアドレスに変換して，VM のメモリにアクセスする．また，VM の仮想ディスク上のファイルシステムを解析することで，VM 内で使われているファイルにアクセスすることもできる．

図 3 に Ciel におけるオートスケールの様子を示す．Ciel では，VM 内コンテナを用いてきめ細かいオートスケールを行うオートスケーラが動作する．Ciel のオートスケーラは VM 外部から VM イントロスペクションを用いて VM 内の各コンテナのリソース使用量を監視する．負荷の高いコンテナが見つかれば，そのコンテナが動作するのに必要十分な大きさの VM を新たに起動する．新しい VM ではスケールアウトを行う必要のあるコンテナだけを起動し，元の VM で動作していたそれ以外のコンテナは起動しない．スケールアウト後は，ロードバランサが各 VM 内のコンテナにリクエストを分散させる．

このようにサービス単位でのスケールアウトを行うことにより，Ciel では新たに起動する VM への課金を最小限に抑えることができる．VM には動作させるコンテナが使用するリソースだけを割り当てればよく，VM への課金は基本的に割り当てられたリソース量に比例して決まるためである．また，スケールアウトする必要のないサービスを起動しないため，スケールアウトにかかる時間を必要最小限に抑えることができる．迅速に起動する必要のあるサービスの起動処理が不要なサービスの起動処理と競合することもない．

サービスの負荷の監視はまず VM 全体で行い，VM の負荷が高い場合に VM 内で動作している各コンテナに対して行う．一つの理由は，VM のリソース監視に比べて VM 内コンテナのリソース監視のほうが，VM イントロスペクションを必要とすることからオーバヘッドが大きいためである．もう一つの理由は，各コンテナの負荷が高くない場合でも VM 全体の負荷は高い場合があるためである．例えば，VM 内でコンテナが 3 つ動作している場合，それぞれの CPU 使用率が 30%だとしても VM 全体の CPU 使用率は 90%となる．Ciel のオートスケーラは VM の負荷上昇を検知すると，各コンテナのリソース使用量から負荷の原因となっているコンテナを特定してスケールアウトを行う．上の例のように，どのコンテナの負荷もあまり高くない場合には，リソース使用量の変化などからスケールアウトさせるのが最も効果的だと考えられるコンテナを選択する．

最近は VM の代わりにコンテナを提供するクラウドも増えており，VM を用いずにコンテナのみを用いてサービスを提供する方法も考えられる．各コンテナで 1 つのサービスのみを動作させれば，クラウド側から各サービスのリソース使用量を容易に取得することができ，サービス単位のスケールアウトを行うことができる．しかし，VM をコンテナと併用したほうがよい理由が少なくとも 3 つ存在する．第一に，VM のほうがセキュリティがより強固である．第二に，VM 間ではリソース分離がより厳密に行える．第三に，現状ではコンテナのマイグレーションを安定して行うことができない．これらの理由から，Ciel では従来の VM を用いるクラウドを想定し，VM 内でコンテナを活用する．

4 実装

図 4 に Ciel における各ホストのシステム構成を示す．現在の実装では，Xen [3] を用いて VM を動作させ，VM の中で Docker [14] を用いてコンテナを動作させる．オートスケーラはドメイン 0 と呼ばれる特権 VM で動作させる．仮想化ソフトウェアとして KVM [10] などを用いたり，コンテナソフトウェアとして LXD [5] などを用いたりすることも可能である．

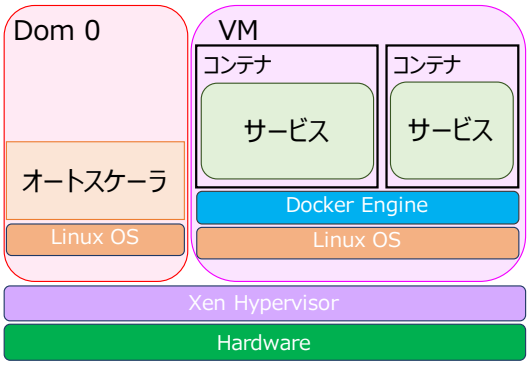

図 4　各ホストのシステム構成

4.1 リソース使用量の監視

Docker では Linux の cgroup を用いてリソース制御を行っているため，Ciel は VM イントロスペクションを用いて VM 内の cgroup の階層構造を解析することにより情報を取得する．まず，監視するリソースに対応する cgroup のサブシステムを探す．例えば，CPU アカウンティングサブシステム，メモリサブシステム，ブロック I/O サブシステムなどがある．次に，各 Docker コンテナの情報が格納されている docker グループを探す．コンテナはコンテナ ID で識別する．VM の仮想ディスクの中に格納されているコンテナのコンフィグを解析することで，コンテナ ID とコンテナ名を対応づけることもできる．そして，コンテナごとに cgroup のサブシステム情報を取得する．

CPU アカウンティング（cpuacct）サブシステムの場合，サブシステム情報の中の CPU アカウンティング情報を用いて CPU ごとの使用時間の合計をナノ秒単位で取得する．この情報は VM 内コンテナの中では，cgroup ファイルシステムの cpuacct.usage 疑似ファイルから取得することができる．定期的にコンテナの CPU 使用時間を取得してその増分を計算することにより，CPU 使用率を算出する．

メモリ（memory）サブシステムの場合，サブシステム情報の中のメモリ cgroup の情報を用いて CPU ごとのメモリ統計情報を取得する．取得できる情報は，RSS と呼ばれる匿名メモリとスワップキャッシュの合計のバイト数と，ページキャッシュのバイト数である．匿名メモリはコンテナ内のプロセスが使用して

いるメモリであり，スワップキャッシュはスワップアウトされた後，まだメモリ上に保持されているメモリである．ページキャッシュはプロセスがファイルを読み書きする際に OS 内に確保されるメモリである．これらの情報は VM 内コンテナの中では，cgroup ファイルシステムの memory.stat 疑似ファイルから取得することができる．

ブロック I/O（blkio）サブシステムの場合，サブシステム情報の中のブロック cgroup の情報を用いて CPU ごとのディスクアクセス量をバイト単位で取得する．この情報は VM 内コンテナの中では，cgroup ファイルシステムの blkio.throttle.io_service_bytes 疑似ファイルから取得することができる．定期的にコンテナのディスクアクセス量を取得してその増分を計算することにより，消費ディスク帯域を算出する．

一方，VM のリソース使用量は libvirt API を用いて取得する．定期的に VM の CPU 使用時間をナノ秒単位で取得してその増分を計算することにより，VM の CPU 使用率を算出する．

4.2 リソース監視機能の開発

オートスケーラの中でコンテナのリソース使用量を監視する機能は，VM 内で動作している Linux カーネルのソースコードをできるだけ利用して作成した．ヘッダファイルはそのまま利用し，cgroup の情報を取得するコードは必要な部分だけを一部修正して利用した．しかし，そのままでは VM 内の情報を取得することができないため，VM のメモリにアクセスして情報を取得するように変更する必要がある．このような変更を利用したソースコード全体について行うのは非常に大変な作業になる．

そこで，Ciel では LLView [25] を用いてコンパイル時にプログラム変換を行うことにより，VM イントロスペクションを行えるようにした．LLView はソースコードを LLVM コンパイラでコンパイルし，中間表現の中の load 命令を変換する．LLView は GPU プログラム用に開発されていたが，それを VM 用に修正した．具体的には，リソース監視機能が VM のメモリにアクセスする際に，VM のメモリの必要な部分をマッピングしてから，Linux カーネルの仮想アドレスをマッピング先のメモリアドレスに変換するようにした．

例えば，リソース監視機能をコンパイルして次のような中間表現が得られたとする．

```
%1 = load i64, i64* %jiffies
%2 = udiv i64 %1, 250
```

この中間表現は Linux カーネル内の 64 ビットのグローバル変数 jiffies の値をローカル変数%1 に読み込み，250 で割って%2 に格納している．LLView を用いるとこの中間表現は次のように変換される．

```
%1 = bitcast i64* %jiffies  to i8*
%2 = call i8* @g_map(i8* %1)
%3 = bitcast  i8* %2 to i64*
%4 = load i64, i64* %3
%5 = udiv i64 %4, 250
```

この中間表現では jiffies のアドレスを 8 ビット整数のポインタにキャストして%1 に格納し，それを引数として g_map 関数を呼び出してメモリマッピングおよびアドレス変換を実行する．g_map 関数から返されたアドレスを元の 64 ビット整数のポインタにキャストし，そのアドレスにあるデータを%4 にロードする．

LLView はアセンブリ言語で書かれたプログラムの変換に対応していないため，等価な C プログラムに置き換える必要があった．例えば，ビットマップの指定したビットの値を調べる test_bit 関数などである．

4.3 VM のスケールアウト

Ciel で VM のスケールアウトを行う際には，テンプレートから新たに VM を作成する．ただし，事前に VM を作成しておいたり，スケールインを行ったりしてシステムに VM がプールされている場合にはその VM を再利用することもできる．テンプレートには Docker を動作させるための最小限のシステムだけをインストールしておく．そのために，Barge [1] のような Docker ホスト用軽量 OS を使うことができる．コンテナイメージについては，VM 作成後に必要

なものだけをインストールする．

テンプレートはVMの構成情報が書かれたコンフィグとディスクイメージからなる．Cielで用いたlibvirt[19]ではVMのコンフィグはXML形式となっているため，libxml2[21]を用いて構文解析を行う．そして，同一物理ホスト上で重複が許されないUUIDやMACアドレス，VM名，イメージファイル名を変更する．また，スケールアウトするサービスに必要十分なサイズのVMとなるように，仮想CPU数やメモリ容量を変更する．作成したVMのコンフィグはlibvirt APIを用いてシステムに登録する．

VMのディスクイメージはテンプレートのイメージファイルを複製することによって作成する．ディスクサイズが大きくなると複製に時間がかかるため，テンプレートのイメージファイルはスパースファイルを用いて作成しておく．スパースファイルはデータが書き込まれたブロックだけ実体を持つ特殊なファイルである．スパースファイルを使用することで，実データを持つブロックのみが複製されるため，ディスクイメージの複製時間を短縮することができる．

スケールアウトによりVMの台数が増えると，Cielは Linux Virtual Server (LVS)[23]と呼ばれるロードバランサを用いてVM内コンテナにリクエストを分散させる．LVSに設定した仮想IPアドレスへのパケットが追加したVM内のコンテナにも転送されるように，コンテナのIPアドレスをLVSに登録する．転送方式にはDirect Server Return (DSR)と呼ばれる方式を用い，リクエストが転送されたコンテナはクライアントに直接，レスポンスを返すことで高速化を図る．

5 実験

Cielの有用性を確かめるための実験を行った．実験にはIntel Xeon E3-1225 v5のCPU，12GBのメモリ，1TBのディスクを搭載したマシンでXen 4.6.0動作させ，1個の仮想CPU，1GBのメモリ，50GBの仮想ディスクを持つVMを用いた．VM内ではUbuntu 16.04とDocker 17.05.0-ceを動作させた．

5.1 VM内コンテナの性能

VM内でコンテナを使用することによるオーバヘッドを調べるために，VM内コンテナの中でUnixBench[13]を実行して性能測定を行った．ストレージドライバにはAUFS[18]，OverlayFS，ZFS[12]，Device Mapperの4種類を用いた．比較として，VM上でコンテナを用いない場合の性能も測定した．UnixBenchは10回実行し，平均を算出した．

測定結果を図5に示す．このグラフではコンテナを用いない場合の性能を1として正規化している．整数演算，プログラム実行，プロセス生成，シェルスクリプト実行についてはVM内コンテナのオーバヘッドは小さいことがわかった．一方，システムコール，浮動小数点演算，パイプ，コンテキストスイッチについてはオーバヘッドが大きかった．これはコンテナによる仮想化の影響のためだと考えられる．

ファイルコピーではストレージドライバによって性能差が大きく，AUFSとZFSでオーバヘッドが大きかった．特にZFSに関しては約75 %の性能低下を引き起こした．これはZFSがデータブロックを読み出す際にブロックポインタをたどる回数が多く，同一の処理に対してより多くのI/Oが必要になるためだと考えられる．また，シェルスクリプト実行ではAUFSだけが15〜20 %の性能低下を引き起こした．

実験結果より，ストレージドライバにOverlayFSまたはDevice Mapperを使用した場合の性能低下は平均10%程度であり，許容できるオーバベッドであることが分かった．そこで，以下の実験ではOverlayFSを用いた．

5.2 VM内コンテナのリソース監視

VM内コンテナのリソース使用量を監視するために，VM内で3つのコンテナを用いてApacheウェブサーバ[2]，nginxウェブサーバ[17]，WildFlyアプリケーションサーバ[20]を動作させた．Apache上ではBottleフレームワーク[8]を用いて作成したウェブアプリケーションを動作させた．このウェブアプリケーションは指定された画像にモザイク処理を行う．このウェブアプリケーションに負荷をかけた時に，VMイントロスペクションを用いて各コンテナのCPU使用

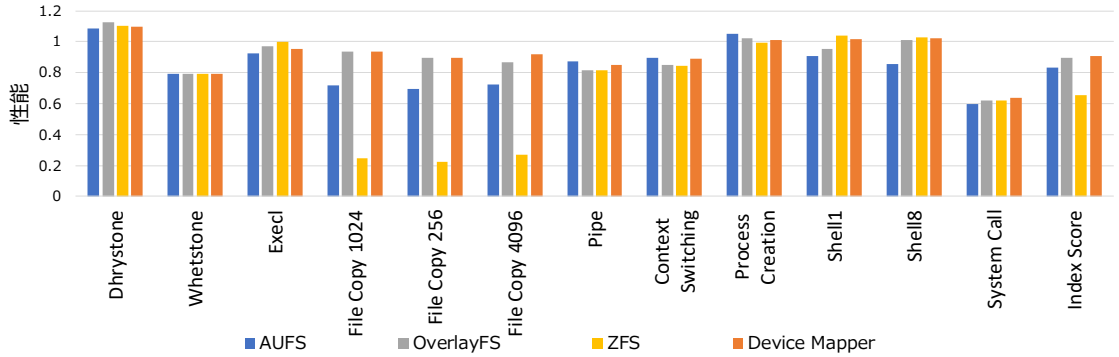

図 5　VM 内コンテナの性能

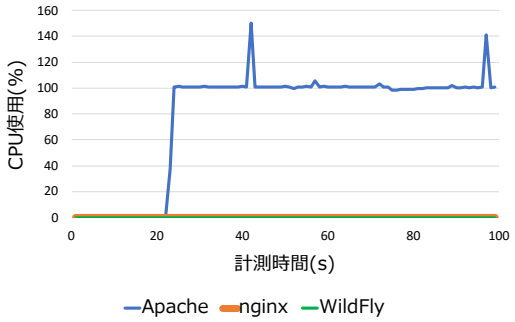

図 6　VM 内コンテナの CPU 使用率

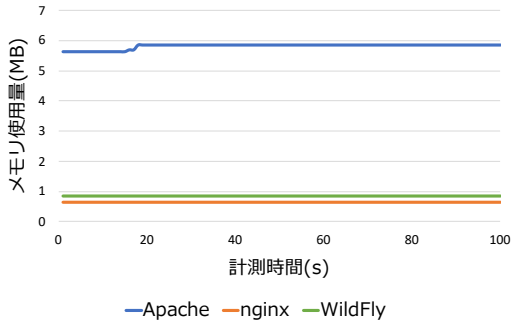

図 7　VM 内コンテナのメモリ使用量

率，メモリ使用量，消費ディスク帯域を測定した．

図 6，図 7，図 8 にそれぞれの測定結果を示す．Apache の CPU 使用率だけが 100%に増加していることが分かる．また，メモリ使用量についても Apache だけが多かった．消費ディスク帯域の結果からは，Apache が最初だけディスクアクセスを行ったことが分かる．その後はページキャッシュにアクセスしたと考えられる．

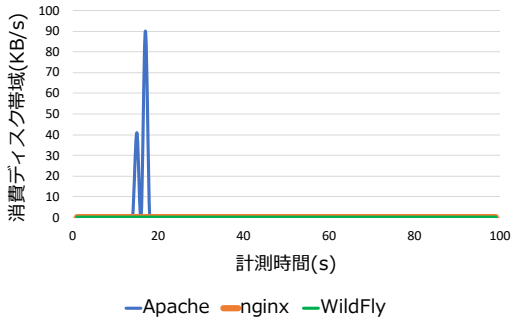

図 8　VM 内コンテナの消費ディスク帯域

5.3　スケールアウトにかかる時間

5.2 節の実験において，Apache の CPU 使用率が 80%を超えた時にスケールアウトを行い，スケールアウトにかかる時間を測定した．スケールアウト時にはテンプレートから VM を作成してコンテナを起動し，そのコンテナ内で Apache を起動した．この実験では，テンプレートのディスクイメージに Ubuntu 16.04 をインストールし，Apache のコンテナイメージはあらかじめインストールしておいた．比較のために，VM 単位でスケールアウトを行う従来システムについてもスケールアウトにかかる時間を測定した．従来システムでは元々の VM と同様に，3 つのコンテナイメージをインストールしたディスクイメージを用意した．

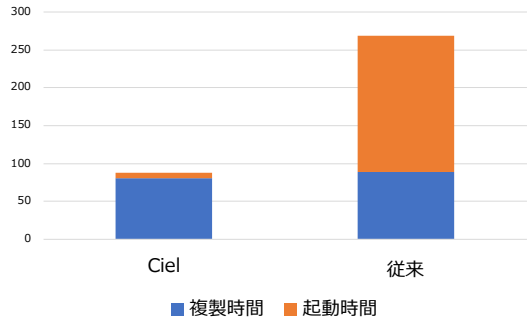

図 9　スケールアウト完了時間

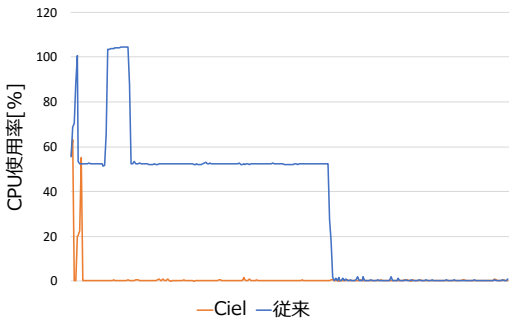

図 10　スケールアウト中の CPU 使用率

スケールアウトにかかった時間を図 9 に示す．Ciel では従来システムの 33%の時間でスケールアウトが完了した．内訳を見ると VM の起動時間が大幅に削減されており，Ciel では 7 秒しかかかっていなかった．これはスケールアウトする必要のないサービスを起動しなかったためであり，特に WildFly の有無の影響が大きかった．VM を作成せずにプールされた VM を使う場合には，スケールアウトにかかる時間は起動時間のみとなり，Ciel では従来システムの 4%の時間で完了することになる．ディスクイメージの複製時間については，インストールされているサービスが少ない Ciel のほうが若干高速になった．これはスパースファイルを用いたため，複製するディスクブロック数が少なかったことによる．巨大なデータベースを使うサーバがある場合にはこの差がより顕著になると考えられる．

5.4　スケールアウト時のリソース使用量

5.3 節の Apache のスケールアウトの際に，新しい VM を起動してからサービスの起動が完了するまでの VM 全体での CPU 使用率とメモリ使用量を測定した．図 10 と図 11 に測定結果を示す．スケールアウトする必要のない nginx や WildFly も起動されてしまう従来システムと比べて，Ciel では VM の CPU 使用時間が 3 分から 7 秒に減少した．CPU 使用率も低く抑えられた．また，Ciel ではメモリ使用量も 41%に抑えられた．これらは主に WildFly の起動に CPU とメモリが必要とされるためである．

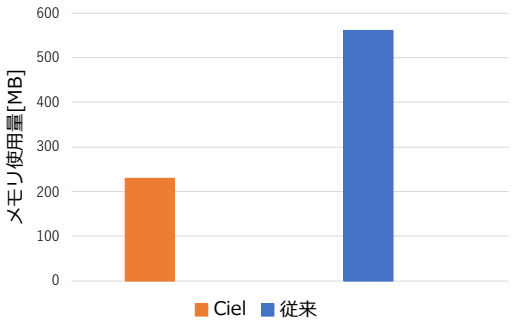

図 11　スケールアウト中のメモリ使用量

5.5　スケールアウト後のリソース使用量

5.3 節の Apache のスケールアウトが終了した後，LVS を用いて 2 台の VM 内のコンテナにリクエストを分散させた時の CPU 使用率の変化を測定した．負荷分散方式にはラウンドロビンを用いた．ベンチマークには httperf[16] を用い，ウェブアプリケーションに 1 秒間に 5 個のリクエストを送信した．図 12 に測定結果を示す．170 秒付近でスケールアウトが完了し，新しい VM のコンテナの中で Apache が動作し始めた．その後は，各 VM 内のコンテナにリクエスト処理が分散し，それぞれの CPU 使用率は 60%と 30%程度となった．CPU 使用率が同程度にならない原因は現在調査中である．

6　関連研究

Picocenter [22] は，長期間実行されるがほとんど使われないサービスをクラウドで効率よく動作させるために，VM 内コンテナを用いて実行を行う．サービ

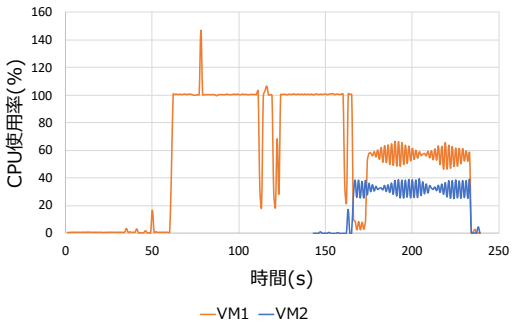

図 12　スケールアウト後の CPU 使用率

スが使用されず，コンテナが非アクティブになった場合は，コンテナごとストレージにスワップアウトし，再びサービスへの要求が来た時にコンテナをスワップインする．VM 内コンテナを用いてサービスを実行する点で Ciel に似ているが，本研究では各サービスのリソース使用量を VM の外部から正確に計測し，負荷の高いサービスだけをスケールアウトできるようにするためにコンテナを使用している．

VCRecovery [24] は，障害対策として VM 内コンテナを用いて待機系のサービスを動作させる．ウォームスタンバイの場合には，運用系の VM と待機系のコンテナを一対一に対応させて動作させる．コールドスタンバイの場合には，障害発生時に待機系の VM 内でコンテナを起動する．これにより待機系の VM にかかるコストを最小限に抑えつつ，迅速な障害復旧を実現している．VCRecovery では従来の VM 内のサービス全体をそのままコンテナ内で動作させるのに対して，Ciel ではサービスごとに異なるコンテナを用いる．VCRecovery では VM 内コンテナの負荷が高まった時にコンテナマイグレーションを行うため，Ciel と相補的に用いることができる．

FlexCapsule [11] は，VM 内で軽量な VM を用いてサービスを実行する．クラウドにおいて VM 構成のきめ細かい最適化を動的に行うことにより，コストを削減することが可能である．例えば，VM マイグレーションにより負荷の低いサービスを少数の VM に集約したり，負荷の高いサービスを新たに起動した VM に移動させたりすることができる．FlexCapsule では，ネストした仮想化 [4] を用いて VM の中で VM を動作させ，ライブラリ OS を用いることで軽量化している．しかし，VM 内で VM を動作させるオーバヘッドが依然として大きい．Ciel では VM 内でコンテナを動作させることでオーバヘッドを抑えることができる．

V-Met [15] は，監視対象 VM が動作している仮想化システム全体を VM 内で動作させることにより，信頼できない仮想化システムの外側で侵入検知システム (IDS) を安全に動作させる．V-Met では，VM イントロスペクションを用いて VM の外部からまず仮想化システムを解析し，さらにその中の監視対象 VM の OS データを解析する．Ciel では VM 内の OS データを解析するだけで VM 内コンテナを監視することができる．また，セキュリティに必要な情報ではなく，リソース使用量に関する情報を収集する点も異なる．

Docker コンテナの中で Docker コンテナを動作させる Docker in Docker と呼ばれる手法も利用されている．dind [9] は Docker コンテナの中で Docker Engine を動作させることにより，ユーザが独自のコンテナを用いることができる．ホストの Docker Engine を共有することにより，コンテナ内でコンテナを起動する手法もある．これらの手法は主に開発用途で用いられている．

cAdvisor [7] は Docker コンテナのモニタリングツールであり，cAdvisor 自体もコンテナで動作する．cAdvisor はコンテナの CPU 使用率やメモリ使用量を取得することができる．Ciel において VM イントロスペクションを用いて取得した VM 内コンテナの情報を標準インタフェースで提供できるようにすることで，VM 外部で cAdvisor を動作させることも可能になると考えられる．

7　まとめ

本稿では，VM 内コンテナを用いることで，サービス単位でのより柔軟なオートスケールを行うシステム Ciel を提案した．Ciel では VM 内で動作している各サービスをコンテナを用いて実行することにより，各サービスのリソース使用量を正確に計測することができる．VM 内コンテナのリソース使用量は VM イントロスペクションを用いて VM 内の OS データ

を取得することで計測する，負荷の原因となっているサービスのみをスケールアウトすることで，新たに起動する VM のコストを削減し，スケールアウトにかかる時間を短縮することができる．

今後の課題は，様々なリソースの使用量を考慮してオートスケールのポリシーを作成することである．また，負荷の高いサービスを動作させるのに必要なリソース量を見積もることができるようにし，必要十分な大きさの VM を起動してスケールアウトが行えるようにする必要がある．さらに，サービスの負荷が低くなった時に VM の数を減らし，複数の VM で動作しているコンテナを集約するスケールインにも対応していくことを計画している．

参 考 文 献

[1] A.I.: Barge: Yet Another Lightweight Linux Distribution for Docker Containers, https://github.com/bargees/barge-os.

[2] Apache Software Foundation: Apache HTTP Server Project, https://httpd.apache.org/.

[3] 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, *Proc. Symp. Operating Systems Principles*, 2003, pp. 164–177.

[4] Ben-Yehuda, M., Day, M. D., Dubitzky, Z., Factor, M., Har'El, N., Gordon, A., Liguori, A., Wasserman, O., and Yassour, B.-A.: The Turtles Project: Design and Implementation of Nested Virtualization, *Proc. Symp. Operating Systems Design and Implementation*, 2010.

[5] Canonical Ltd.: Linux Containers, https://linuxcontainers.org/.

[6] Garfinkel, T., Rosenblum, M., et al.: A Virtual Machine Introspection Based Architecture for Intrusion Detection., *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[7] Google, Inc.: cAdvisor: Analyzes Resource Usage and Performance Characteristics of Running Containers, https://github.com/google/cadvisor.

[8] Hellkamp, M.: Bottle: Python Web Framework, https://bottlepy.org/.

[9] J. Petazzoni, d.: Docker in Docker, https://github.com/jpetazzo/dind.

[10] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A.: kvm: the Linux Virtual Machine Monitor, *Proc. Ottawa Linux Symp.*, 2007, pp. 225–230.

[11] Kourai, K. and Sannomiya, K.: Seamless and Secure Application Consolidation for Optimizing Instance Deployment in Clouds, *Proc. Int. Conf. Cloud Computing Technology and Science*, 2016, pp. 318–325.

[12] Lawrence Livermore National Laboratory: ZFS on Linux, https://zfsonlinux.org/.

[13] Lucas, K.: Byte-UnixBench, https://github.com/kdlucas/byte-unixbench.

[14] Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment, *Linux J.*, Vol. 2014, No. 239(2014).

[15] Miyama, S. and Kourai, K.: Secure IDS Offloading with Nested Virtualization and Deep VM Introspection, *Proc. European Symp. Research in Computer Security*, 2017, pp. 305–323.

[16] Mosberger, D. and Jin, T.: httperf—A Tool for Measuring Web Server Performance, *ACM SIGMETRICS Performance Evaluation Review*, Vol. 26, No. 3(1998), pp. 31–37.

[17] NGINX, Inc.: NGINX: High Performance Load Balancer, Web Server, &Reverse Proxy, https://www.nginx.com/.

[18] Okajima, J. R.: Aufs4 – Advanced Multi Layered Unification Filesystem Version 4.x, https://aufs.sourceforge.net/.

[19] Red Hat, Inc.: libvirt: The virtualization API, https://libvirt.org/.

[20] Red Hat, Inc.: WildFly, https://www.wildfly.org/.

[21] Veillard, D.: Libxml2: The XML C parser and toolkit of Gnome, https://xmlsoft.org/.

[22] Zhang, L., Litton, J., Cangialosi, F., Benson, T., Levin, D., and Mislove, A.: Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments, *Proc. European Conf. Computer Systems*, 2016.

[23] Zhang, W.: Linux Virtual Server for Scalable Network Services, *Proc. Ottawa Linux Symp.*, 2000.

[24] 森川智紀, 光来健一: クラウドにおける VM 内コンテナを用いた低コストで迅速な自動障害復旧, 第 *142* 回 *OS* 研究会, 2018.

[25] 尾崎雄一, 山本裕明, 光来健一: GPU を用いた OS レベルでの障害検知, 第 *142* 回 *OS* 研究会, 2018.