様々な負荷の Web アプリケーション Docker コンテナが混在するクラウド環境性能評価

榎　美紀† 　堀井　洋† 　小野寺　民也†

† IBM Research - Tokyo
E-mail: 　† enomiki@jp.ibm.com

あらまし　クラウド環境を提供するデータセンターでは, 実際は全体的に低い CPU 使用率のサーバーが多く存在する状況に陥りやすい．そこでリソースをできるだけ使い切るため，CPU をオーバーコミットした状態でクラウドのインスタンスを提供している．しかしながら，すべてのインスタンスが常に同じリソースを使い続ける場合は，クラウド内のリソース使用率が安定するかもしれないが，季節や時間帯により，あるインスタンスのアプリケーションの稼働率が急激にあがるような時もある．そのように一部のインスタンスの負荷が急激に大きくなると，サーバーのリソースを大幅に使用することになり，オーバーコミットしていた他のインスタンスも影響を受けて性能が劣化してしまう可能性が生じる．そこで本研究では，CPU をオーバーコミットしたクラウド環境にて，一部のインスタンスにて高負荷のワークロードが実行された時の，各インスタンスの性能への影響度を評価する．また，他のインスタンスの負荷の影響をおさえるため，クラウドインスタンスのリアルタイムなリソース制御システムを提案する．これにより，高負荷のワークロードのスループットを高く維持しながらも，出来るだけ他インスタンスへの影響を小さくするようなクラウド環境が実現できることを示す.

キーワード　クラウド，Docker，性能評価，オーバーコミット

Performance Evaluation of various Workloads' Web Application on Docker Container Environment

Miki ENOKI† 　Hiroshi HORII† 　and 　Tamiya ONODERA‡

† IBM Research - Tokyo
E-mail: 　† enomiki@jp.ibm.com

1. はじめに

クラウドコンピューティングは IT サービス構築基盤の主流の一つとなっており, 多くの IT 開発者がクラウド環境を利用してアプリケーションを開発している．このようにコンピュータの基盤環境をインターネット経由のサービスとして提供するものは Infrastructure as a Service (IaaS) とよばれ，SaaS や PaaS に並ぶサービス提供形態となっている．

データセンターでクラウド環境を提供するベンダーにとっては，ユーザーが想定する性能要件を満たしながら，ホストサーバーのリソースをうまく使い切る状態でサーバーを稼働させることが理想的である．しかしながら，一般的にユーザーは必要なリソースを多めに見積もってインスタンスを確保するため，実際にはホストサーバー全体の 40%以下の CPU リソースしか通常稼働して利用されていないという状況に陥りやすい[1]．そのため， CPU リソースをオーバーコミットした状態でユーザーにインスタンスを割当てることも多い．

すべてのインスタンスが常に同じリソースを使い続ける場合は，それによりクラウドのサーバーリソースをうまく使うことができるかもしれないが，季節や時間帯により，あるインスタンスのアプリケーションの稼働率が急激にあがるような時もある．例えば，インターネットショッピングを稼働しているアプリケーションでは，クリスマスや年末のシーズンになると，急激にユーザーのアクセス数が増加し，普段使用しているリソースの数倍のリソースを消費する[2]．実際，Google のデータセンターのクラウドアプリケーションは，全体の約 20%のインスタンスがアクティブにリソースを消費し，他のインスタンスはほとんどサーバーのリソースを消費していないというパレートの法則が観測されている[3].

クラウド環境を利用するユーザーは様々な用途で

アプリケーションを稼働させるため，そのように，一部のインスタンスのワークロード負荷が急激に大きくなると，サーバーのリソースを大幅に使用することになり，同じ物理サーバー上に配置された他のインスタンスも影響を受けて性能が劣化してしまう可能性が生じる．

そこで本研究では，CPU をオーバーコミットしたクラウド環境にて，一部のインスタンスにて高負荷のワークロードが実行された時の，各インスタンスの性能の変化を評価する．また，他のインスタンスの負荷の影響を少なくするため，CPU リソース割当ての優先度やホスト OS のタスクの割当てを動的に変更するクラウドインスタンスのリアルタイムなリソース制御システムを提案する．これにより，高負荷のワークロードのスループットを高く維持しながらも，出来るだけ他インスタンスへの影響を小さくするようなクラウド環境が実現できることを示す．

2. 本実験におけるクラウド環境

本章では本論文で性能評価を行うクラウド環境と，クラウドインスタンスのワークロードの検証パターンについて述べる．

2.1. Docker コンテナ

クラウド上の環境は仮想化されてユーザーに提供されるが，コンテナ型の仮想化技術を実現する Docker [4] はコンテナ管理の手軽さと軽量さにより，近年広く普及しているオープンソースである．

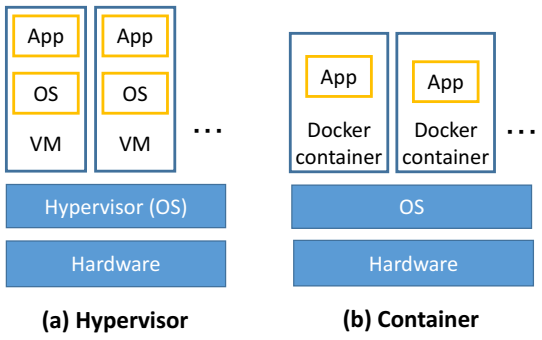

図 1 ハイパーバイザーとコンテナのスタックの違い

Figure1 Stack of hypervisor and container.

Docker コンテナのスタックは VM のハイパーバイザーのスタックと，図 1 に示すような違いがある．ハイパーバイザーでは仮想化した任意の OS をインストールして稼働させるが，コンテナ環境はホスト OS のカーネル空間を共有して仮想化環境を作るため，一般的に OS はホスト OS と同一となる．そのためコンテナ上のアプリケーションはオーバーヘッドが小さいという利点がある[5]．本論文では，クラウド上のインスタンスを Docker コンテナで提供する．

2.2. クラウドインスタンスのワークロードのシナリオ

本論文にて性能を評価するクラウドインスタンスのワークロードのパターンを以下に示す．各ユーザーは 1Docker コンテナのインスタンスを稼働する．稼働するアプリケーションは，ベンチマークアプリケーションの DayTrader3 を用いる[6]．Java EE アプリケーションサーバー上で稼働するデイトレードを模した Web アプリケーションである．

クライアントからのアクセス数によりアプリケーションへの負荷を調整する．図 2 に示すように，Docker 上でベンチマークを動かし，8 インスタンスが 16 論理 CPU を共有して稼働する．データベースサーバーは別サーバーの RAM 上で稼働し，データベースの処理がボトルネックとなることを回避している．DayTrader のアプリケーションへのクライアントからの負荷により，インスタンスのリソース使用率が変化するような実験環境となっている．

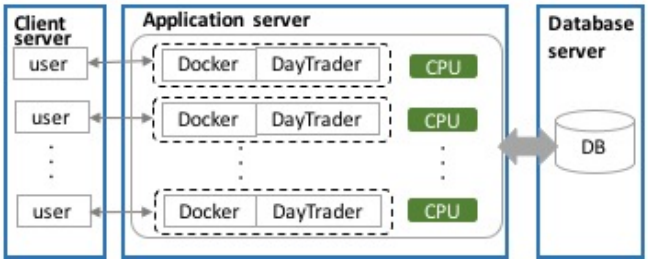

図 2 実験構成

Figure2 Experimental configuration

[Scenario 1]

すべてのインスタンスにおいてアプリケーションの CPU 使用頻度が低く，マシンリソースに余裕がある状態

[Scenario 2]

1 インスタンスのアプリケーションのみ使用頻度が高くなり，負荷が高くなった状態

[Scenario 3]

2 インスタンスのアプリケーションの使用頻度が高くなり，負荷が高くなった状態

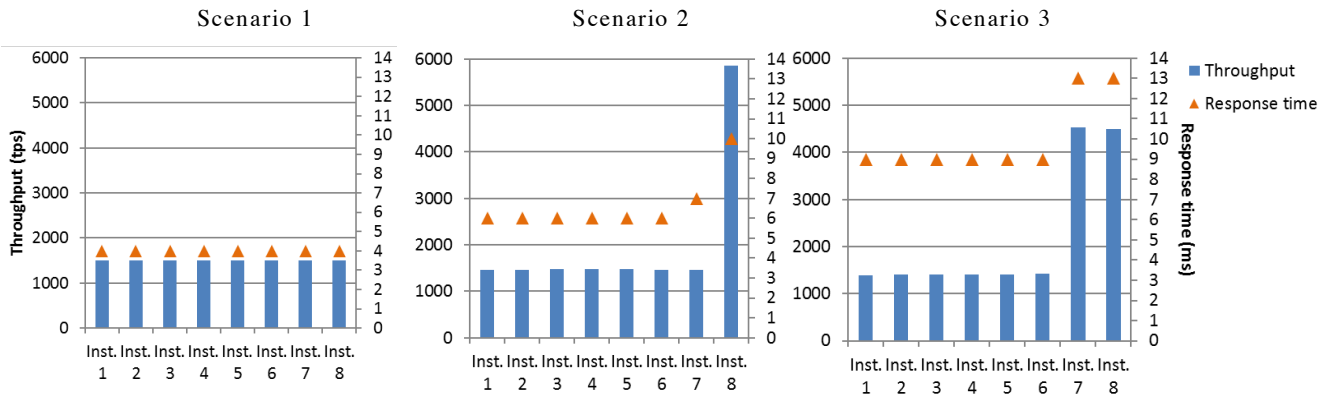

図 3　各 Scenario のスループットと応答時間

Figure 3　Throughput and response time of each Scenario

以上の 3 シナリオで各インスタンスのベンチマーク性能がどのように変化するかを検証する．8 インスタンス中 2 インスタンスのみ負荷を高くしているのは，前述のデータセンターの典型的な使用パターンであるパレートの法則に従っている．従って全体の約 2 割のインスタンスまでを負荷を高くしている．

3. 性能評価

前章にて述べたシナリオのベンチマーク性能を評価する．

3.1. 測定環境

構成は図 2 に示した通りである．クライアントサーバーは，　2 x CPU Xeon X5670 (2.93GHz, 6 cores) with 32 GB of RAM, OS　Red Hat Linux 5.5，アプリケーションサーバーは，2 x CPU Xeon E5-2680 (2.70GHz, 8cores) with 32 GB of RAM, OS Ubuntu 14.04，データベースサーバーは，2 x CPU Xeon X6550 (2.0GHz, 8 cores) with 16 GB of RAM, OS　Red Hat Linux 5.5 である．アプリケーションサーバーは 16 論理 CPU(8core)のみ使用するため，他の core は off にする．アプリケーション上で稼働する Docker は v1.3.2, DayTrader ベンチマークは IBM WebSphere Application Server Liberty Profile v8.5.5.3 with Java (IBM J9 VM JRE 1.7.0)にインストールする．データベースには DB2 UDB v9.7 を使用する．クライアントサーバーでは JMeter v2.9 を用いてクライアントスレッドを並列稼働して負荷をかける．各シナリオにて一定のクライアントスレッド数で 10 分間測定した時の平均スループットと応答時間を測定する．

3.2. 測定結果

各 Scenario における，各インスタンスの平均スループットと応答時間を図 3 に示す．

まず，Scenario 1 では各インスタンスの Docker 上で稼働する DayTrader に対して目標スループットを JMeter にて 1500tps を指定し，クライアントスレッド数は 32 である．目標スループット 1500tps 程度の状態で，各インスタンスの平均応答時間は 4ms であった．サーバー全体の平均 CPU 使用率は約 66%であり，CPU リソースにはまだ余裕がある状態である．

次に，Scenario 2 では Instance8 のアプリケーションに対してのみ，目標スループットを JMeter にて 20000tps を指定して高負荷環境をつくっている．サーバー全体の平均 CPU 使用率は約 82%まで上昇し，Instance8 のスループットは 5858tps であった．他のインスタンスは 1500tps 前後を維持できているが，リソース圧迫の影響を受け，Scenario 1 の結果では 4ms であった応答時間が，平均して 6.1ms と長くなった．

続いて，2 インスタンスのアプリケーション負荷を高くした Scenario3 の結果では，サーバー全体の平均 CPU 使用率は約 87%となり，Instance7,8 のスループットは平均して 4500tps であった．他のインスタンスは 1412tps となり，1500tps を下回る結果となり，平均応答時間も 9ms と倍以上の長さとなってしまい，高負荷のインスタンスのリソース圧迫の影響が Scenario2 よりもさらに大きくなったことが分かる．

以上により，CPU をオーバーコミットした状態にて，高負荷のインスタンスが存在すると，全体の CPU リソースが圧迫されて，他のインスタンスの性能を劣化させてしまうことが分かった．

高負荷となったインスタンスのクライアントリクエストに出来るだけ沢山応答することは大切であるが，それにより全体のリソースを消費しつくし，他のインスタンスへの性能を劣化させてしまうことは望ましくない．そこで次章にて，出来るだけ他インスタンスの影響を及ぼさずに高負荷インスタンスのリクエスト処

理を実行するための，リアルタイムなリソース制御システムを提案する．

4. リアルタイムなクラウドインスタンスのリソース制御

Scenario 3 の構成を例にして説明，効果を測定する．

4.1. CPU の優先割り当て制御

図 3 に示したように，一部のインスタンスのアプリケーションが高負荷になると，他のインスタンス性能に影響を及ぼし，性能を劣化させてしまうことがある．そのため，できるだけ性能劣化をさけながらも， CPU の空いているリソースは高負荷のアプリケーションの処理に利用して，サーバー全体としてはリソースを十分に活用している環境を作り出すことが理想的であると考える．

本論文では，各インスタンスにあらかじめ特定の CPU を明示的に割当てておき，その CPU リソースを使い切る程アプリケーションの負荷が高くなった場合に，CPU 割当てを解除するようなリアルタイム制御を行う．これにより，各インスタンスは割当てられた CPU を優先的に使う一方で，CPU 割当てが解除されたインスタンスは，インスタンスが持つ CPU リソースをすべて使用する．

CPU の明示的な割当ては taskset で該当のインスタンスプロセスに動的に設定するか，Docker のコンテナは cgroup を利用しているため，cgroup の CPU リソースにて設定しても良い．本論文の Scenario 3 の構成では taskset を用いて，Instance 1-6 に 2CPU ずつを割当て，高負荷の Instance 7, 8 には指定せずに 16CPU を利用できるような制御となる．

4.2. CPU の優先割当ての効果の測定

図 4 に CPU 割当て制御後の結果を示す．サーバー全体の平均 CPU 使用率は約 86%となり 3.2 節の結果を同程度にリソースを使っている．Instance 7,8 の平均応答時間は 26ms と長くなったが，平均スループットは 4712tps と，3.2 節と同程度の性能となった．Instance 1, 3, 4, 5 の平均応答時間は制御前の 9ms から 5.4ms まで短くなった．スループットも 1483tps まであがるようになった．これらのインスタンスに CPU を明示的に割当てたことにより，高負荷インスタンスの影響をうけずに従来の性能とほぼ同じ性能を出せている．

しかしながら，Instance 2 の結果は，平均応答時間が 22ms まで長くなり，スループットも 974tps までさがってしまった．この要因としては，ホスト OS 側でのハードウェア割込み処理のログを検証したところ，外部サーバーとのネットワーク IO 処理の割込み処理タスクが，Instance 2 に割当てた CPU に集中していたためと考えられる．

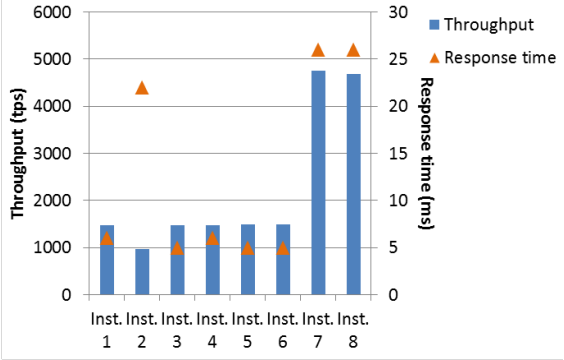

図 4　CPU の優先割当て制御の結果

Figure 4　Result of CPU prioritization control

本来，ハードウェアの割込み処理(IRQ, Interrupt ReQuest)は Linux ディストリビューションに標準実装されているデーモンである，irqbalance により，各 CPU に分散される．irqbalance を有効にしていると，１０秒毎に各 CPU の負荷状態に応じて，各 CPU への IRQ 割込み処理が再配置実施される．これにより低負荷の CPU に割込み処理が割当てられるはずだが，実験ではすべての CPU が 80%以上の使用率になっており，割当ての再配置が行われなくなっていた．

4.3. ホスト OS のハードウェア割込みの制御

前述のように，CPU を各インスタンスに割当てるとこで，高負荷のインスタンスの影響をできるだけ避けることができるが，ハードウェア割込みタスクが特定の CPU に集中した場合，そこに割当てられたインスタンスの性能を下げてしまう可能性がある．CPU を割当てていない時は，全インスタンスは全 CPU リソースを共用するため，ハードウェア割込みタスクの影響は平滑化されている．

そこで，ハードウェア割込みタスクを，明示的に特定のインスタンスに割当てていない CPU 間でのみ共有させる．割込み処理を担当する CPU は，該当の IRQ の smp_affinity に割当てたい CPU 番号を明示的に指定することで制御できる．本論文では，10 秒ごとにネットワーク IO の IRQ を，インスタンスに明示的に割当てられていない CPU 間で順番に指定する．

4.4. ホスト OS のハードウェア割込み制御の効果の測定

図 5 にハードウェア割込み制御の結果を示す．サーバー全体の平均 CPU 使用率は約 78%，Instance 7,8 の平均スループットは 4000tps，平均応答時間は 31ms と 4.2 の結果よりも，IRQ の影響をうけて性能は下がったものの，他のインスタンスは平均応答時間 5ms，スループット 1489tps となり，高負荷インスタンスの影響

をうけずに稼働した.

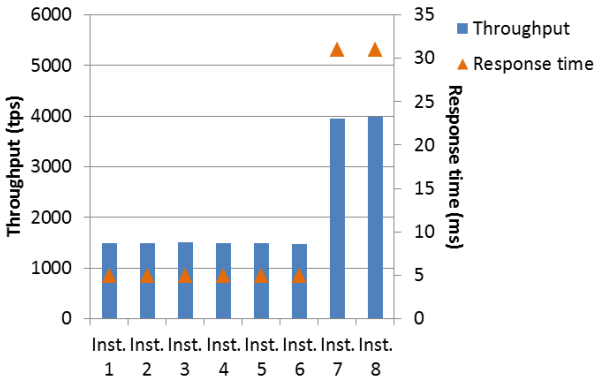

図 5　ハードウェア割込み制御の結果

Figure 5　Result of hardware interruption control.

以上により，CPU をオーバーコミットした環境にて，一部の高負荷なインスタンスの影響をできるだけ他のインスタンスが受けないようにしながら，全体としては CPU リソースをつかいきれるような，リアルタイムなリソース制御を提案し，効果を検証した．高負荷でないインスタンスの性能を担保する一方，高負荷インスタンスの性能は，リソース制御する前よりは劣化してしまうが，これはトレードオフになると考えられる．

5. まとめと今後の課題

CPU をオーバーコミットしたクラウド環境にて，一部のインスタンスにて高負荷のワークロードが実行された時の，各インスタンスの性能の変化を評価する．また，他のインスタンスの負荷の影響を少なくするため，CPU リソース割当ての優先度やホスト OS のハードウェア割込み処理の割当てを動的に変更するクラウドインスタンスのリアルタイムなリソース制御システムを提案する．これにより，高負荷のワークロードのスループットを高く維持しながらも，出来るだけ他インスタンスへの影響を小さくするようなクラウド環境が実現できることを示した．

本論文では，リソース制御を人手で設定を変更して効果を確認した．今後は，CPU リソースの割当てとハードウェア割込み処理を制御するための手順を定式化し，クラウドのインスタンスの使用状況やアプリケーション性能を監視して，自動でリソースを制御できるようにしたい．

参　考　文　献

[1] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. Intel Science and Technology Center for Cloud Computing Technical Report ISTC–CC–TR–12–101, Carnegie Mellon University, Pittsburgh, PA, USA, Apr. 2012

[2] Cloud success secret: Flexible capacity planning https://www.ibm.com/developerworks/cloud/library/cl-capacityplan/

[3] Sheng Di, Derrick Kondo, Franck Cappello. Characterizing Cloud Applications on a Google Data Center (ICPP'13)

[4] Docker https://www.docker.com/

[5] Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio, “An Updated Performance Comparison of Virtual Machines and Linux Containers”, IBM Research report http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

[6] DayTrader3 http://geronimo.apache.org/GMOxDOC22/daytrader-a-more-complex-application.html