コンテナランタイムのパフォーマンス比較とシステム管理における評価

伊藤 佳城[1,a)]　串田 高幸[1]

概要：近年, 仮想化の常套手段として使われているコンテナ型仮想化では, docker イメージ利用の環境構築コストの削減、異なる OS 環境の管理の容易、多様な環境への迅速な対応が利点として挙げられるが, コンテナ仮想化におけるパフォーマンスの分析は, 通信速度・起動・削除の一連の流れであるライフサイクルという点が重要視されており, リソース変化におけるメトリクスの分析・解析は行われていない. 本稿では, docker コンテナの cpu, メモリ, I/O のリソース変化によるメトリクスの変化に焦点を当てることで既存研究との差別化を行い, パフォーマンスの分析・解析を行った. 解析した結果, コンテナの cpu とメモリは少なからず関係性があることが示され, リソースが制御されている場合は, 逆にリソースを制御することで安定するのではないかという結果が得られた.

1. はじめに

コンテナ型仮想化は, コンテナエンジンを使うことでコンテナごとにアプリケーションを動かすことができる, 非常に軽量な仮想マシンのようなものである. ハイパーバイザー型仮想化と比べると, OS の起動を不要とするため起動・処理速度が VM 型よりも優れていることがメリットとして挙げられる. またコンテナは, ハードウェアのリソースの制約を受けにくいため, より高い密度で多くのプロセスを実行可能にすることが利点として挙げられている [1]. また, コンテナオーケストレーションシステムである Kubernetes の登場もコンテナ仮想化の普及に貢献してる [2]. この利点から現在はマイクロサービスや web クラスター構築に用いられている [3] しかし, コンテナ仮想化は, いまだ発展途中であるためコンテナのバージョン齟齬 [4], 稼働 OS の制約による脆弱性 [5] , コンテナ内に最適なアプリケーションの搭載数・キャパシティプランニングといった課題が挙げられる. その中でもコンテナのキャパシティ及びボトルネックとなりえる部分をコンテナの cpu 割り当て時間の変更, メモリ制御, I/O 帯域幅の制御に焦点を当て, リソース変化を行うことでその結果に基づき, 議論を行う.

本稿では, docker コンテナのパフォーマンスとシステム管理における評価について説明を行う. この論文は, 次のように構成される. 2 章では, コンテナ仮想化における関連研究の説明. 3 章は, 提案手法の説明を行う. 4 章では, 実験の手法についてのアーキテクチャと図を使って説明を行う.5 章で結果を示し,6 章で結果から得られたデータに基づいて議論・考察を行い. コンテナのパフォーマンス及びシステム管理における評価を行う. 最後に 7 章で, 最終的なまとめで締めくくる.

2. 関連研究

コンテナ仮想化における既存研究では, コンテナ自体のパフォーマンスではなくコンテナを利用した際のパフォーマンスやコンテナ仮想化と既存の仮想化との比較が多い傾向にある. 関連研究の紹介を下記に示す. コンテナオーケストラレーションシステムである Kubernetes 上のコンテナのパフォーマンス最適化のための設定チューニングこれは, コンテナイメージを使う際にコンテナにより個別に設定が必要になり, ユーザーの負担が増えてしまうのでその設定チューニングのフレームワークを提案している [6]. docker コンテナのセキュリティレベルの分析では, docker コンテナは同一カーネルを使用するという特徴があり最悪コンテナからカーネルに攻撃される恐れがあるため, セキュリティの観点からコンテナを研究している [7]. コンテナ型仮想化とハイパーバイザー型仮想化のワークロードのパフォーマンス比較では, ハイパーバイザー仮想マシンとコンテナのワークロードに重点を当てて評価が行われている [8]. コンテナをエッジコンピューティングとして利用する際の評価では, エッジコンピューティングに焦点を

[1] 東京工科大学コンピュータサイエンス学部
CDSL, TUT, Hachiouji, Tokyo 101–0062, Japan
[a)] C0117039

当てて従来のプラットフォームとのパフォーマンス比較を行っている [9]. ベース OS を攻撃した際の linux コンテナと docker コンテナの比較・評価では, ホストベースのシステムでサービスを実行した際の脆弱性の評価を行っている [10]. コンテナライフサイクルにおいて重要な役割を果たすコンテナネットワークの現状とコンテナネットワークモデルとネットワークソリューションの調査・分析では, 主流ネットワークソリューションを使ったパフォーマンス測定を行っている [11]. コンテナを利用した際の経済的な効率化の検証では, コンテナを利用することでリソースを減らすことで経済的にどう影響が出るかを調査している [12]. docker コンテナを利用したデータセンターの効率化のための動的リソース割り当てアルゴリズムの提案では, データセンターでのアプリケーション展開コストの削減に取り組んでいる [13]. ハイパーバイザー型仮想化とコンテナ型仮想化のパフォーマンスオーバーヘッドの比較では, コンテナがどのな場合でオーバーヘッドが高くなるかの調査を行っている [14]. 上記の研究では, コンテナのボトルネックとなりえる部分の究明はされていない. そのため, 本実験では, コンテナのリソース変化によるメトリクスに焦点を当てる. 既存の研究としてコンテナメトリクスに関するものはあるが, リソース変化・コンテナに負荷を与えてパフォーマンスを測定したものは数が少ない [15].

3. 提案手法

本稿のコンテナのリソース変化によるメトリクスの変化及びパフォーマンスの分析・解析の手法について説明する. コンテナ型仮想化における利点は, 起動の速さや手軽さに目が行きがちだがコンテナに負荷を与えたパフォーマンス比較は, 少ない. なので今回のコンテナメトリクスの分析には, コンテナに計算処理で負荷を加えた状態にさらにリソース変化を行うことでコンテナメトリクスの状態を確認する.

4. 実験手法・環境

本実験を行う環境を, 下記に示す.

- OS: ubuntu-18.04.2
- vCPU: 3
- memory: 3GB
- HDD: 50GB

この条件下のもと本実験は, 行っていく.

本稿の実験のアーキテクチャを図 1 に示す. このセクションでは, アーキテクチャに基づいて実装内容の説明をおこなう. 図 1 では, 水色が既存のソフトウェアを示しており, 赤色が本実験で作成するプログラム・ソフトウェアである. この章では主にこの赤色の部分についての解説を行っていく. dokcer run では, dockerfile を用いたコンテナの起動を行い, docker イメージ取得・作業ディレクトリ

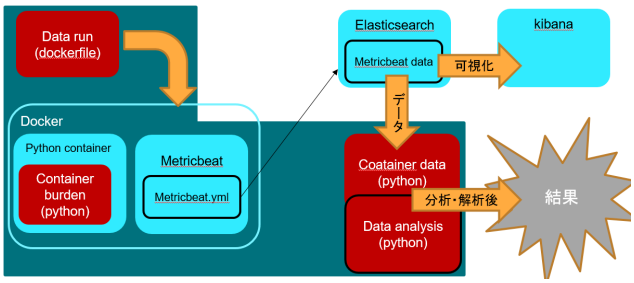

図 1 アーキテクチャの図

の変更を行い起動時に python プログラムがコンテナ内部で起動できるようになっている. docker システムでは, 上記で挙げた python プログラムである container-burden.py をもとに起動しているコンテナのデータをシステムの統計上をを収集する metricbeat を用いてオープンソースの検索・分析エンジンである elasticsearch にて, ログ解析・リアルタイムでのデータ分析を行う. その後, データの読み書きを行う python プログラムである container data を用いて elasticsearch から直接データを取得する. そのデータをもとに python プログラムである data analysis で分析・解析・評価を行う. 同時に elasticsearch のデータをログデータ監視ツールである Kibana によって可視化も行う. しかし, 本実験では, container data, data analysis は提案という形になっている. 実際のデータ取得には elasticsearch から csv ファイルを用いて行った.

4.1 実装

Docker run

Dockerfile とは, docker イメージの取得, それを用いる際に必要なパッケージ・アプリケーションのインストールの作業や構成情報をまとめることで複数の作業を一括で行うことができるファイルである. これにより構築の手間を削減, 導入に際の手順書の代わりになる利点が挙げられる. コンテナに計算をさせ負荷を与えるプログラムである container burnen をコンテナ内部で実行させるため, Dockerfile を用いたコンテナの起動を行う. 本実験で使用した Dockerfile のを下記のソースコード 1 示し, 解説を行う.

ソースコード 1 Docker run

```
1 FROM python:3.6
2 COPY container-burden.py /run/
3 WORKDIR /run
4 CMD ["python3","container-burden.py"]
```

まず, 初めに FROM でコンテナのベースイメージの指定を行う. 本実験では, python プログラムをコンテナ内部で

起動し負荷をかけるため使用するイメージは, python:3.6 となっている．その後, COPY で container-burden.py を run ディレクトリに追加, そして WORKDIR で作業ディレクトリを container-burden.py をコピーした run ディレクトリに設定. 最後に CMD でコマンド (=python container-burden.py) を実行する．

Container burden

コンテナ内部の負荷には, python で記述した簡単な計算プログラムを用いている．そのプログラムをソースコード 2 に示す．

ソースコード **2** container burden

```
1 n = 2
2 while True:
3     print(n)
4     n = n ** 5
```

while 文を使いループを起こし, その中で $\sum_{n=2}^{\infty} n^5$ という計算を行い負荷をかけている．指数関数を用いて, 膨大な値を計算させ続けることでコンテナに負荷を与える狙いがある．

Container data

elasticsearch からのデータの受け取りのプログラムを抜粋したソースコード 3 に示す．しかし, 実装部分で足りない部分があるので提案という形にする．今回のデータ取得では elasticsearch で生成された csv ファイルを使う．

ソースコード **3** container data

```
1     es = Elasticsearch(["elasticsearch-edge.
          a910.tak-cslab.org"])
2
3     JST = timezone(timedelta(hours=+9), 'JST')
4     now = datetime.now(JST)
5
6     index_name = 'ito-container' + now.strftime
          ('%Y-%m-%d')
7     es.indices.create(index=index_name, ignore
          =400)
8
9     payload = {
10        'random_number': random_num,
11        'host.hostname': gethostname(),
12        '@timestamp': now.strftime("%Y-%m-%dT%H
              :%M:%S%z")
13    }
14    print("Request:: ", json.dumps(payload,
          indent=4))
15    res = es.index(index=index_name, body=
          payload)
16    print("Response:: ", json.dumps(res, indent
          =4))
```

研究室内に設置された elasticsearch を使い, python の elasticsearch パッケージを使用してデータの受け取りを行う.python プログラムで elasticsaerch への index を作成し, metricbet で受け渡したデータのやり取りを行う．

Data analysis

docker stats, htop, scv ファイルから受け取ったデータを元に受け取ったデータの分析・解析を行う．今回は csv ファイルを元に excel で起動時間及びメトリクスのグラフ作成を行う．

5. 実験・結果

この実験では, リソース変化を行った際のコンテナメトリクスの変化を分析・評価することで, コンテナのボトルネックとなりえる部分の発見を目的としているため, コンテナのリソースである cpu, メモリ, ディスク I/O 帯域幅の変化を意図的に行う．リソース変化の手法として, docker コンテナのオプションを用いる．コンテナの cpu 割り当てには, コア番号を指定する cpu–cpuset-cpus オプション・割当容量を変更する–cpu-shares オプションを用い, メモリの容量制限には, -m オプションを使用. ディスク I/O 帯域幅の制限には, dd コマンドによる書き込み速度の計測を行う．

CPU 割り当て

ここでは, cpu の割り当て時間を 10%,50%,100%にしたコンテナをコアを別々にする．NET I/O に関しては, 特に数値の変動がなかったため今回はメモリ使用量の推移のみを示す．

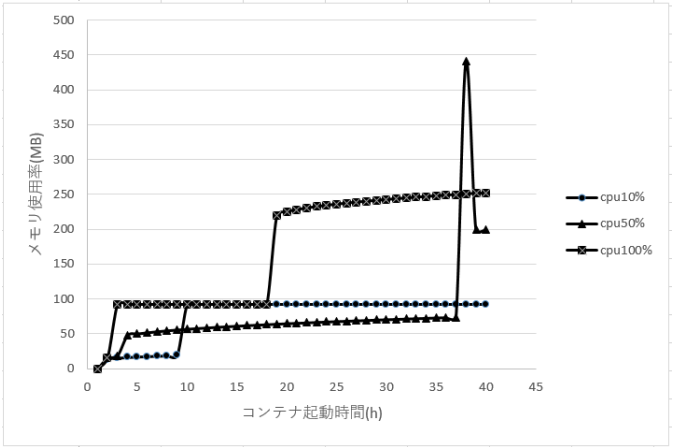

図 2 cpu 割り当て時間を設定した際のメモリ使用量推移

メモリ

ここでは, cpu の割り当て時間を 10%,50%,100%のコンテナを cpu のコアを分けたうえでメモリの使用量を 50MiB, 100MiB に制御した結果を図 3,4 に示す. ここでも NET I/O に関する数値の変動が見られなかったので省略する.

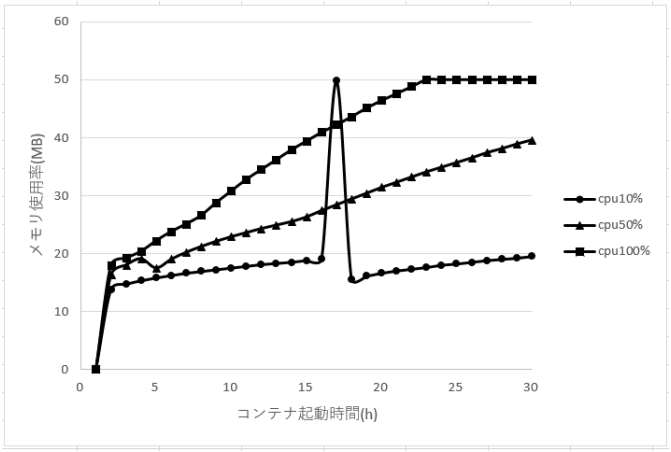

図 3 メモリ使用量を 50m に制限した際のメモリ使用量推移

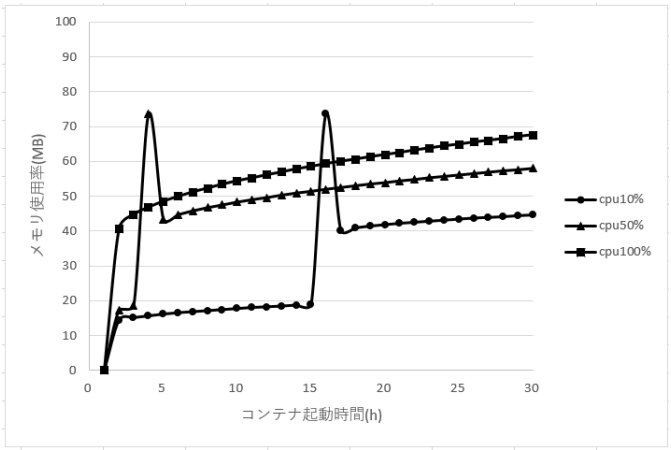

図 4 メモリ使用量を 100m に制限した際のメモリ使用量推移

ディスク I/O

cpu 割り当て時間の 10%, 50%, 100%に対し, メモリ使用率を 50MiB,100MiB に制限した上てコンテナ内部に 10M の file を作成した結果を表 1 に示す.

表 1 10M のファイル生成速度

10%cpu100m 51.8 MB/s	50%cpu100m 153 MB/s	100%cpu100m 512 MB/s
10%cpu50m 50.4 MB/s	50%cpu50m 151.8 MB/s	100%cpu50m 510.4 MB/s

6. 議論・考察

実験・結果から得られたものに対しての議論・考察を行っていく.

CPU 割り当て

cpu 割り当てに関する部分では, cpu を変更し, 変化があったのはメモリ容量率であった. 変化がなかったのは NET I/O, BLOCK I/O の部分である. 変化があったメモリ使用量に関しては, cpu の割り当てが少ないとメモリ使用量を少なく比例しているように見えるが, cpu10%よりも cpu50%の方が低くその後 50%の方が爆発的に使用量が多くなり一気に下がり安定するという推移を見せた. メモリ使用量が爆発的に増加するのは, どの cpu 割り当てのコンテナにも起こっており, その後に安定した数値を出すのも同じであった. コンテナ起動時間の 0 から 10 の間にメモリ使用量は, 100%,50%,10%の順で増加していることから cpu 割り当てが高いと計算が多く行える為, 負荷がかかる時間が早くなる. そのためメモリ使用量は, cpu に順じて高くなる. しかし, cpu50%が 400MB 以上もメモリを瞬間的に使用した推移が見られたため, 負荷に対しリソースが適さないと安定性はなくなると考えられる.

メモリ

メモリ制御では, メモリ使用量を 50MB に制御した際には, cpu10%のコンテナが安定したメモリ使用率を示さず, 急に上限に達した後使用率が減るという推移を見せた. 100MB に制御した際は, cpu10%, cpu50%のコンテナどちらとも 73.6MB まで使用率が急上昇し, その後安定した推移を見せた. どちらもメモリ使用量が急上昇した際の値が同じであった為, ある一定のリソースがない場合のコンテナは, 安定して動くことができず結果として, cpu%割り当て時間が少ないほどメモリの使用の推移が安定しないことが示された. しかし, cpu50%の推移が 100MB よりも 50MB の方が安定しているため, 限られたリソースで動かす際には, 制御を行うことで逆に安定する可能性もあるともいえる.

ディスク I/O

10M のファイル生成にかかる速度を測定したところ, メモリ容量の制限には関係なく近い数値が表れた. その後, 複数回書き込みを行っても cpu10%だと 50 60MB/s,cpu50%だと 150 160MB/s, cpu100%だと 510 530MB/s の範囲が躊躇に表れた. コンテナに計算による負荷をかけていてもファイル生成速度には影響がないことが分かり, 結果とし

て, 書き込み速度には, メモリ使用率は関係性はなく cpu 利用率が高いほど速度は速くなる結果となった.

7. 結論

コンテナのメトリクスは, ある程度のパフォーマンスは cpu やメモリに左右されることがないことが分かった. しかし, コンテナの cpu とメモリは少なからず関係性があることが示された. コンテナの負荷量に対して, メモリ使用率が爆発的に増加したりする結果となり, 安定的なコンテナの挙動を常に維持するためには, リソースを与えればよいのではなく負荷に応じて制御を行う必要性もあるのではないかと考える.

8. おわりに

本実験では, コンテナのリソース変化を行いそのうえでコンテナメトリクスの変化がどうなるかを測定し, その評価を行った. コンテナの cpu とメモリは少なからず関係性があることが示され, リソースが制御されている場合は, 逆にリソースを制御することで安定するのではないかという結果が得られた. NET I/O, BLOCK I/O については, 躊躇な値を取得するような実験は行うことができなかった. 今後の課題として NET I/O, BLOCK I/O の負荷を与えるもので実験, リソース制御による安定性の提示が必要となるため. 次回以降の実験では, 引き続きこの部分に留意して実験を行っていく.

参考文献

[1] Anderson, C.: Docker [Software engineering], *IEEE Software*, Vol. 32, No. 3, pp. 102–c3 (online), DOI: 10.1109/MS.2015.62 (2015).

[2] Bernstein, D.: Containers and Cloud: From LXC to Docker to Kubernetes, *IEEE Cloud Computing*, Vol. 1, No. 3, pp. 81–84 (online), DOI: 10.1109/MCC.2014.51 (2014).

[3] Julian, S., Shuey, M. and Cook, S.: Containers in Research: Initial Experiences with Lightweight Infrastructure, *Proceedings of the XSEDE16 Conference on Diversity, Big Data, and Science at Scale*, XSEDE16, New York, NY, USA, ACM, pp. 25:1–25:6 (online), DOI: 10.1145/2949550.2949562 (2016).

[4] Merkel, D.: Docker: Lightweight Linux Containers for Consistent Development and Deployment, *Linux J.*, Vol. 2014, No. 239 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2600239.2600241⟩ (2014).

[5] Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: The True Cost of Containing: A gVisor Case Study, *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, USENIX Association, (online), available from ⟨https://www.usenix.org/conference/hotcloud19/presentation/young⟩ (2019).

[6] Chiba, T., Nakazawa, R., Horii, H., Suneja, S. and Seelam, S.: ConfAdvisor: A Performance-centric Configuration Tuning Framework for Containers on Kubernetes, *2019 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 168–178 (online), DOI: 10.1109/IC2E.2019.00031 (2019).

[7] Bui, T.: Analysis of Docker Security, *CoRR*, Vol. abs/1501.02967 (online), available from ⟨http://arxiv.org/abs/1501.02967⟩ (2015).

[8] Felter, W., Ferreira, A., Rajamony, R. and Rubio, J.: An updated performance comparison of virtual machines and Linux containers, *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 171–172 (online), DOI: 10.1109/ISPASS.2015.7095802 (2015).

[9] Ismail, B. I., Mostajeran Goortani, E., Ab Karim, M. B., Ming Tat, W., Setapa, S., Luke, J. Y. and Hong Hoe, O.: Evaluation of Docker as Edge computing platform, *2015 IEEE Conference on Open Systems (ICOS)*, pp. 130–135 (online), DOI: 10.1109/ICOS.2015.7377291 (2015).

[10] Mohallel, A. A., Bass, J. M. and Dehghantaha, A.: Experimenting with docker: Linux container and base OS attack surfaces, *2016 International Conference on Information Society (i-Society)*, pp. 17–21 (online), DOI: 10.1109/i-Society.2016.7854163 (2016).

[11] Zeng, H., Wang, B., Deng, W. and Zhang, W.: Measurement and Evaluation for Docker Container Networking, *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pp. 105–108 (online), DOI: 10.1109/CyberC.2017.78 (2017).

[12] Kumar, K. and Kurhekar, M.: Economically Efficient Virtualization over Cloud Using Docker Containers, *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pp. 95–100 (online), DOI: 10.1109/CCEM.2016.025 (2016).

[13] Guan, X., Wan, X., Choi, B., Song, S. and Zhu, J.: Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers, *IEEE Communications Letters*, Vol. 21, No. 3, pp. 504–507 (online), DOI: 10.1109/LCOMM.2016.2644658 (2017).

[14] Li, Z., Kihl, M., Lu, Q. and Andersson, J. A.: Performance Overhead Comparison between Hypervisor and Container Based Virtualization, *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pp. 955–962 (online), DOI: 10.1109/AINA.2017.79 (2017).

[15] Casalicchio, E. and Perciballi, V.: Measuring Docker Performance: What a Mess!!!, *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, ICPE '17 Companion, New York, NY, USA, ACM, pp. 11–16 (online), DOI: 10.1145/3053600.3053605 (2017).