異なるコンテナにおける同一ライブラリのファイルサイズ比較による完備率の算出

山口 舜[1]　飯島 貴政[2]　串田 高幸[1]

概要：コンテナに割り当てられた CPU とメモリ容量は他のコンテナで使用できず, 非効率なノードの追加が行われることがある. そのため既存の手法としてコンテナで実行しているアプリケーションを異なるコンテナにコピーする. 1 つのコンテナで複数のアプリケーションを実行し, 余剰した CPU とメモリ容量を活用する手法がある. しかし, この手法では Python のアプリケーションを異なるコンテナで実行するため, 依存するライブラリがインストールされているか判断できないことが課題となる. 本研究では完備率という異なるコンテナでアプリケーションを実行するためのライブラリが, どの程度インストールされているか判断する指標を提案する. 完備率は異なるコンテナで実行したいアプリケーションに依存したライブラリを取得する. 取得したライブラリのファイルサイズが何%コピー先のコンテナに含まれているかを算出した値である. 実験はアプリケーションの実行に必要なライブラリのインストール時間と算出した完備率を比較し評価を行う. 実験結果は完備率とインストール時間に約-0.94 という高い相関関係を示すことができた. そのため完備率は異なるコンテナのアプリケーションを実行する際, 必要なライブラリのインストール時間の指標として使用することができる.

1. はじめに

背景

マイクロサービスはコンテナ技術を用いて運用することができる [1]. このマイクロサービスはサービスごとにユーザーからのアクセス数や一つのトランザクションで CPU 使用量とメモリ使用量が異なる. そのためサービスごとに CPU 使用量とメモリ使用量で偏りが出来る. これにより一つのサービスで CPU 使用量とメモリ使用量を独占しないよう, コンテナごとに CPU とメモリを割り当てる必要がある. しかし, コンテナに割り当てた CPU とメモリは別のコンテナで使用できない. また, どのサービスのコンテナの割り当てを優先したらよいかわからないため, アクティブなサービスはすべて等しく重要となり, サービスに割り当てた CPU が枯渇した際にノードを追加しなくてはならない. 例として図 1 のように 8 [vCPU] 使用できるノード 1 に upload コンテナ一つと search コンテナ一つが動いている場合, 残りの使用できる vCPU は 3 [vCPU] である. そのため upload サービスにアクセスが増加すると, upload コンテナをスケールする際に使用していないノード 2 を稼働させなければならず, 非効率なノード追加が行われる.

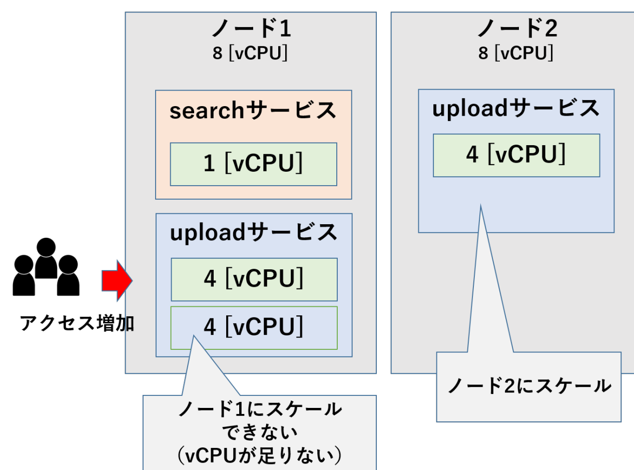

図 1　リソース枯渇によるノード追加

これらを解決する研究として「マイクロサービスにおけるメトリクスによるサービスの優先順位および計算リソースの共助モデル」という研究がある [2]. この研究では各サービスで CPU 使用率と通信を行うサービス数を取得し, これらの情報から優先度を算出する. また一つのコンテナで別のコンテナの処理を行う方法を提案しており, コンテナ一つで二つのアプリケーションの処理を行うことができる [3]. この優先度と一つのコンテナで二つのアプリケー

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[2] 東京工科大学大学院 バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

© 2021 Cloud and Distributed Systems Laboratory

ションの処理を行う方法を用いて, 優先度が高いサービスを優先度が低いサービスで処理を補うことにより, リソースを効率よく使用するという研究である. 図 2 にコンテナ内で行う処理を別のコンテナの余剰リソース (CPU の処理量とメモリ容量) を持つコンテナを用いてリクエストを分散する例を示す. この例ではコンテナ A にリクエストが大量に来ることで割り当てた CPU 使用率が 100%になり, コンテナ B は割り当てた CPU 使用率が 20%であったとする. この場合コンテナ B にプログラム A をコピーし, コンテナ B でコンテナ A の処理を行うことでリクエストを分散できる. これにより新たにノードを追加することなくリクエストを処理できるため, ノードにおける CPU の処理量とメモリ容量を効率よく使用できる.

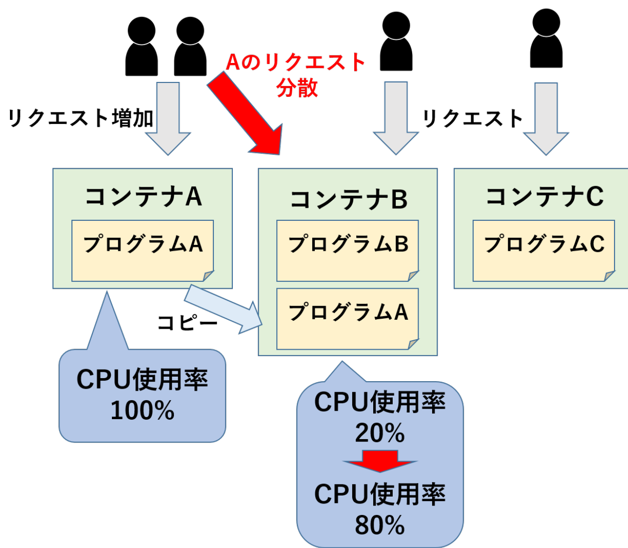

図 2 異なるコンテナでリクエスト分散

異なるコンテナで処理を補うためには, 処理するアプリケーションに依存する実行環境をそろえる必要がある. 実行環境とはコンテナで実行する Python で作成されたアプリケーションでは, Python 標準ライブラリは Python のバージョンとコンテナベースイメージに依存する [4][5]. また, Python のアプリケーションを実行するためには標準ライブラリだけではなく, 外部からインストールして使用する外部ライブラリがある. そのためコンテナベースイメージに同じものを使用している場合, 同じ外部ライブラリをインストールする必要がある. 図 3 のようにコンテナ A のアプリケーションである Python ファイル A をコンテナ B で動かすためには Flask をインストールする必要がある. またコンテナ C で Python ファイル A を実行するには Flask がもともとインストールされているため, Python ファイル A を実行するために新たにライブラリをインストールする必要がない. このようにコンテナの実行環境ができるだけ等しいコンテナでプログラムを実行することにより, 少ないコンテナ実行環境の変更でプログラムを実行できる.

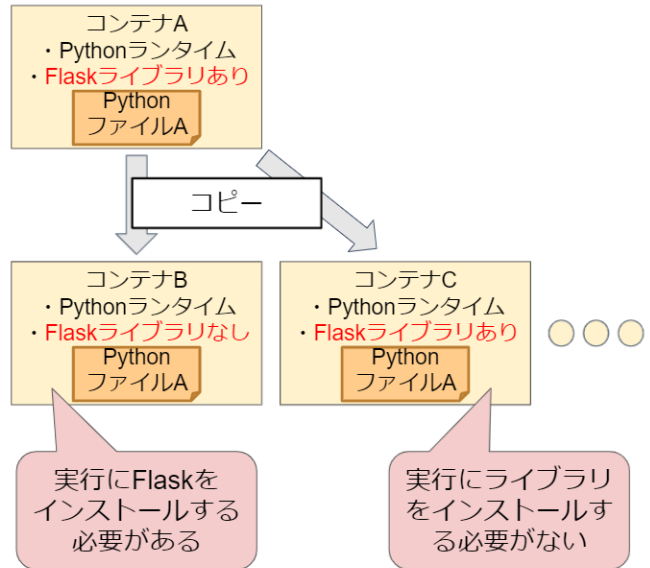

図 3 コンテナ内実行環境の違い

課題

異なるコンテナでアプリケーションを実行するための実行環境, ここではアプリケーションを実行するのに必要なライブラリがコンテナにインストールされているかを示す指標が存在しない. そのため複数のコンテナの中で, どのコンテナの実行環境が類似しているか判断できないことが課題である. また, コンテナで実行しているアプリケーションを異なるコンテナにコピーした際, アプリケーションを実行できるか判断できない. 実行できない場合どの程度のライブラリがアプリケーションを実行するために必要なのか判断できない. 例として図 4 のようにコンテナ A で実行できるアプリケーションである Python ファイル A をコンテナ B またはコンテナ C, D, E, F にコピーして実行できるかわからない. これはアプリケーションを実行するために必要なライブラリが完備されているか判断できないためである. また実行できなかった場合コンテナ B とコンテナ C のどちらの実行環境が似ているか判断する指標が存在しない. そのためコンテナ B とコンテナ C のどちらのコンテナが実行に必要なライブラリをインストールする数が少なく Python ファイル A を実行できるのか判断できないことを課題とする.

各章の概要

第 2 章では本研究の関連研究について述べる. 第 3 章では第 1 章で述べた課題を解決するシステムの提案方式とユースケースについて述べる. 第 4 章では提案するシステムの実装と実験環境について述べる. 第 5 章では提案するシステムの評価と分析について述べる. 第 6 章では提案, 実験, 評価に関する議論について述べる. 第 7 章では本研究の結果から得られた成果について述べる.

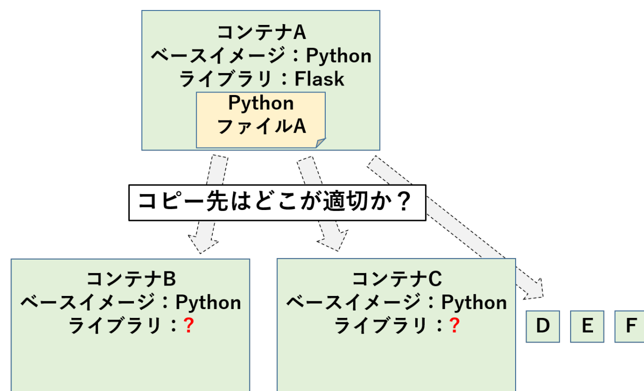

図 4 コンテナ実行環境のブラックボックス

2. 関連研究

マルチクラウドにおける科学ワークフローのためのコンテナベースの実行環境管理として Skyport を提案する論文がある [6]. この論文では課題として新しいバージョンのソフトウェアのインストールは多くの場合, 労働集約的であり通常システム管理者による介入が必要になる. そのため通常ベースイメージへのツールの自動インストールによって作成されるか, 依存関係とともに既にインストールされているツールのセットで事前構成されたイメージから開始する. このため Docker Linux コンテナを利用して, 既存のすべてのワークフロープラットフォームに固有のソフトウェアデプロイメントの問題とリソース使用率の非効率性を解決する. 実験結果から複雑なワークフローの実行に必要な環境の提供に関連する複雑さを大幅に軽減した. しかし, 既に作成されたコンテナへのソフトウェアデプロイについて述べられていない.

プラットフォームに依存しない実装アルゴリズム用の Docker ベースのパッケージングシステムとして AlgoRun を提案する論文がある [7]. この論文では Docker テクノロジーを使用し, 実装されたアルゴリズム専用のパッケージングシステムである AlgoRun を使用することで, 使用可能なアルゴリズムの使いやすいソフトウェア実装に対するバイオインフォマティクスと, 時間の経過に伴うソースコードの変更と依存関係の問題を解決する [8]. この解決方法はソフトウェア実行環境全体をパッケージ化したアルゴリズムになっているため, ソフトウェアの依存関係の一般的な問題や時間の経過に伴う計算の再現性の欠如を排除することができる. しかし, このパッケージ化されたソフトウェアの変更や違うソフトウェアへの変更をしたときの実行環境の依存関係については述べられていない.

3. 提案方式

本研究で述べた課題を解決する提案方式とユースケースシナリオについて説明する.

提案方式

課題の解決方法として異なるコンテナでアプリケーションを実行するために必要なライブラリが, どの程度もとからインストールされているのかの指標として完備率という新たな指標を作成する. 完備率の算出式を以下の式 (1)(2)(3) に示す. 表 1 に式 (1)(2)(3) で使用する変数について示す.

表 1 使用した変数

変数	説明
C	完備率
m	コピー元ライブラリのファイル数
l	コピー先ライブラリのファイル数
n	コピー元とコピー先の同一ライブラリのファイル数
$libS$	コピー元ライブラリのファイルサイズ
$libE$	コピー元とコピー先の同一ライブラリのファイルサイズ

アプリケーションのコピー元のコンテナに m 個のライブラリがあるとき, 一つのライブラリのファイルサイズを $libS_1$, $libS_2$, $libS_3$, $\cdots \mathrm{libS}_m$ とする. 全てのライブラリのファイルサイズを足した値を $libS$ とし, 式 (1) に示す.

$$libS = \sum_{k=1}^{m} libS_k \qquad (1)$$

アプリケーションのコピー先のコンテナに l 個のライブラリがあるとき, 一つのライブラリのファイルサイズを $libD_1$, $libD_2$, $libD_3$, $\cdots \mathrm{libD}_l$ とする. $libS_1$, $libS_2$, $libS_3$, $\cdots \mathrm{libS}_m$ のライブラリの中に $libD_1$, $libD_2$, $libD_3$, $\cdots \mathrm{libD}_l$ のライブラリが n 個あるとき, そのライブラリのファイルサイズを $libE_1$, $libE_2$, $libE_3$, $\cdots \mathrm{libE}_n$ とする. 全てのライブラリのファイルサイズを足した値を $libE$ とし, 式 (2) に示す.

$$libE = \sum_{k=1}^{n} libE_k \qquad (2)$$

完備率算出式を式 (3) に示す.

$$C = \frac{libE}{libS} \qquad (3)$$

この完備率を用いることにより, どのコンテナが必要なライブラリを多くインストールされている実行環境であるか数値から判断できる. これは完備率が高ければ高いほどアプリケーションのコピー先のコンテナでアプリケーションを実行するために必要なライブラリのインストールするデータサイズが少なく, アプリケーションの実行に必要なファイルが多くインストールされていることを示す. 比較するコンテナの条件を以下に定める.

- 同一ベースイメージの使用
- コピー元に Python で作成されたアプリケーションの使用

比較するコンテナの条件により, Python の標準ライブラリ

© 2021 Cloud and Distributed Systems Laboratory

はベースイメージに依存するため, 完備率算出に外部ライブラリのみの比較を行う.

完備率の算出方法は異なるコンテナ2つから同一なライブラリを比較することで求め, 同一なライブラリのデータサイズがアプリケーションの実行に必要なライブラリのうちどの程度インストールされているか, 完備率算出式 (3) を用いて 0 ≦完備率≦ 1.0 の値で算出する. また比較するライブラリはバージョンも等しいときのみ同一のものとして定める. これはバージョン違いによりアプリケーションが実行できない可能性があるためである. これよりライブラリの比較にはバージョンを含めた文字列で比較する. 2つのコンテナから取得したライブラリをもとに算出する完備率の算出式は式（3）となる. 図5に2つのコンテナの完備率算出の例を示す. アプリケーションのコピー元のライブラリの中でアプリケーションのコピー先にある同一ライブラリは fastapi, chardet の2つとなる. よって式（3）の分子は式（2）より, 2つのライブラリのファイルサイズを足した $52254 + 178743 = 230997$ となる. アプリケーションのコピー元のライブラリは fastapi, chardet, MarkupSafe, clickFlask の4つとなる. よって式（3）の分母は式（1）より, 4つのライブラリのファイルサイズを足した $52254 + 178743 + 18330 + 97404 = 346731$ となる. よって式（3）より $230997 \div 346731 = 0.6662$ となり完備率は約 0.67 となる. この完備率が 0.5 を超えていることからアプリケーションに必要なライブラリのインストールするファイルサイズのうち, 半分以上がもとから入っていることがわかる.

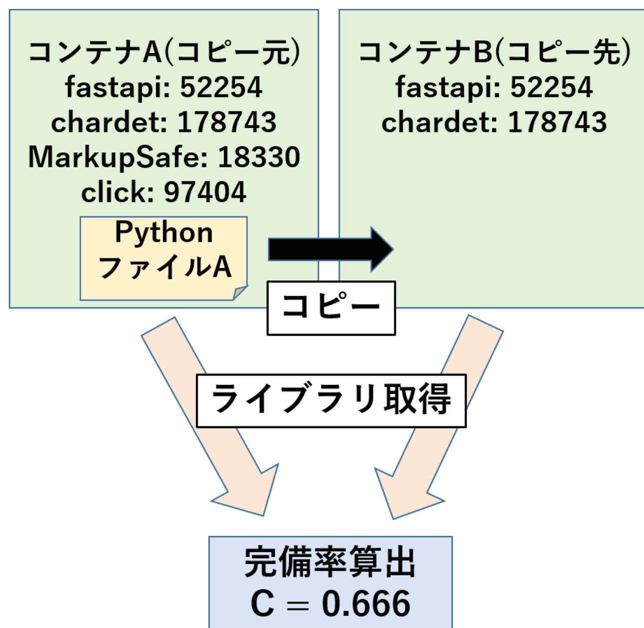

図 5 ライブラリサイズ比較例

提案する完備率をビルドされているすべてのコンテナで算出することで, もっとも同一ライブラリの多いコンテナを見つけることができる. また算出した完備率によりアプリケーションが実行できるか判断できる. 比較元と比較先のコンテナ内にあるライブラリが一致しているときは完備率が1となり, 異なるコンテナでアプリケーションが実行できると判断できる. また比較元と比較先のコンテナ内にあるライブラリが異なるとき, 完備率が0以上1未満となり, 足りないライブラリを比較先のコンテナにインストールすることでアプリケーションを実行できると判断できる. また, 完備率算出の際にアプリケーション実行に足りないライブラリを示し, どのライブラリインストールすればアプリケーションを実行できるか判断することができる.

この提案の完備率算出のタイミングは新たにコンテナが立てられたときと, コンテナが更新されたときに行う. コンテナはステートレスなため, 一度コンテナが立つとコンテナの中に入り変更されない限り完備率に変更が起こらないためである. しかし, ソフトウェア開発技法としてアジャイル開発などでコンテナが途中で更新させる可能性がある [9]. このためコンテナの更新の際に新たに完備率算出を行い, 完備率を更新する必要がある.

ユースケース・シナリオ

ユースケースとして論文掲載サービスである doktor-v2[*1] というマイクロサービスを用いる. doktor-v2 には paper, front, front-admin, thumbnail, author のサービスがあり, それぞれ別のコンテナで動いている. そのため完備率を用いて類似しているコンテナを見つけ, 類似しているコンテナで異なるコンテナのアプリケーションを実行することにより, リソース効率よくマイクロサービスを運用できる. これはコンテナで実行しているアプリケーションに依存するライブラリがコンテナごとに異なる. そのため類似しているコンテナで異なるコンテナのアプリケーションを実行することで, 実行に必要なライブラリのインストールによるコンテナの変更量を少なく, 一つのアプリケーションへのリクエストを分散することができる. 図6のように paper コンテナ, front コンテナ, front-admin コンテナ, thumbnail コンテナ, author コンテナがあるとき, paper コンテナの割り当てられた CPU 使用率が 100%になり, 処理を補いたい場合がある. この場合提案手法により完備率を算出したとき, 比較元を paper コンテナ, 比較先を front コンテナとして完備率を算出したとき完備率が 0.737, 比較先が front-admin コンテナのとき 0.748, 比較先が author コンテナ 0.895 となる. また thumbnail コンテナはまだ一つのサービスとしてコンテナで独立していなかったため算出できなかった. 算出した完備率により paper コンテナと完備率の高い author コンテナが類似していることがわかり, author コンテナで処理を補うことで author コ

[*1] https://github.com/cdsl-research/doktor-v2 (参照 2021 年 11 月 21 日)

ンテナを少ない変更で paper コンテナのアプリケーションである main.py の実行ができるようになる．

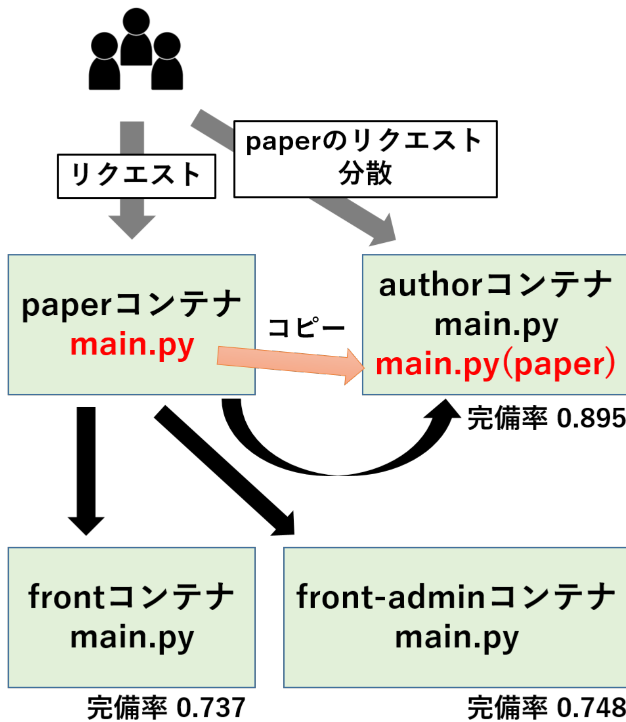

図 6 doktor-v2 への完備率使用

4. 実装と実験方法

本研究で述べた提案方式をもとに開発したソフトウェアの実装と実験方法について説明する．

実装

実装は本研究の提案内容を行うソフトウェアとして ComCal という名前のソフトウェアを新たに作成する．このソフトウェアの機能を以下に示す．

(1) アプリケーションのコピー元とコピー先の二つのコンテナからライブラリを取得する
(2) コピー元のコンテナのライブラリのなかでコピー先にある同じライブラリを取得する
(3) コピー元のコンテナにある全ライブラリのデータサイズを取得する
(4) (2) 取得した同じライブラリのデータサイズを取得する
(5) (4) で取得したデータサイズを (3) で取得したデータサイズで割り完備率とする

この (1) から (5) の処理をアプリケーションのコピー元のコンテナを 1 つ定め，同じマシン上に立てられたすべてのコンテナと完備率計算を行う．これによりもっともライブラリをインストールする際のデータサイズが小さいコンテナを求める．この実行のフローチャートを図 7 に示す．

ComCal の実装には Python 3.8.10 を用いて二つの Python プログラムより作成している．二つの Python プログラムのうち一つ目のプログラムは comcal.py という名前で，二つ目のプログラムは getlibsize.py という名前である．(1) の処理を行うため，まず comcal.py にアプリケーションのコピー元とコピー先のコンテナ名またはコンテナ ID を実行時の引数として送ることでライブラリを取得するコンテナを取得する．取得したコンテナそれぞれに docker-py ライブラリを使用し，コンテナの中で pip freeze コマンドを実行することによりライブラリを取得する．次に (2) の処理を行うため，取得したライブラリを set 型に置き換え，集合演算を行うことにより同じライブラリを求め，足りないインストールする必要のあるライブラリを requirements.txt に書き込む．(3) と (4) では comcal.py からコンテナの中で getlibsize.py を実行しコンテナの中から処理を行う．コンテナの中でライブラリそれぞれに対し pypi から requests ライブラリを用いて json 形式で受け取り，受け取った中からライブラリをインストールする際のデータサイズを取得する．取得したそれぞれのライブラリのデータサイズを用いて (3) ではアプリケーションのコピー元のライブラリの全データサイズを足すことで求める．また，(4) では (2) で取得したライブラリでデータを足すことで求める．最後に (5) では (4) で取得したデータサイズを (3) で取得したデータサイズで割ることで完備率を求める．

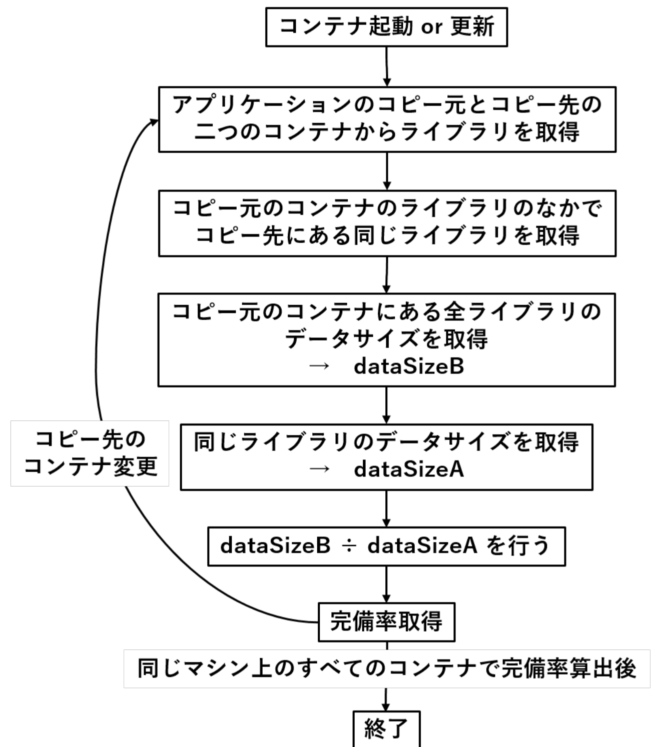

図 7 ComCal 処理のフローチャート

実験環境

実験環境は仮想マシン（VM）上に Docker をインストー

ルした環境を用いる [10][11]. VM の作成に ESXi を用いる [12]. VM の構成情報を下記に示す.

- VM 構成情報
  OS Ubuntu-20.04.2
  vCPU 4 コア
  RAM 8GB
  HDD 50GB
  Docker 20.10.8
  Python 3.8.10

またVM に実装で作成した ComCal を入れて実験する. これらの実験に用いる環境を図 8 に示す. Docker を用いることで実験で使用するコンテナを作成し, ComCal を使用することで完備率の算出を行う.

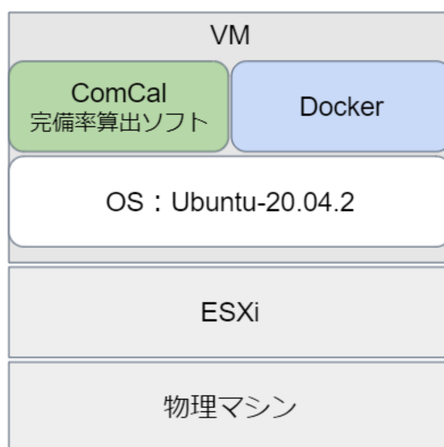

図 8 実験環境

実験方法

実験では GitHub と Docker Hub からイメージを参照しコンテナを用意する. 用意するコンテナはベースイメージに同じものを使用しているものの中で星の数が多いものから順に使用し, 星がないものはダウンロード数が多いものから順に 50 個使用する. 使用したコンテナの一例として hivdb/codfreq[*2], labd/docker-python-lint[*3], nodecustombase/nodealpine16-npm7_22_0-couchbase[*4], joserochabh/sisopapi[*5]を使用する. この用意したコンテナで完備率を算出する. また完備率を算出したコンテナにアプリケーションのコピー元のコンテナの足りないライブラリのインストールを行い, 完備率がインストール時間が相関関係にあるか評価する. このライブラリのインストールはインターネットを経由してインストールを行うため, 同じライブラリをインストールしてもインストール時間が毎回異なる. このインストールの誤差を少なくするため, 同じライブラリのインストールを 100 回行い, 平均インストール時間を使用する.

この実験を行うにあたり複数のコンテナで 100 回ずつライブラリのインストールを行うため, 評価プログラムとして hyouka.py と hyouka.sh の二つのプログラムを作成した. hyouka.py プログラムではアプリケーションのコピー先である比較先のコンテナのイメージを実行する際の引数として受け取り, 受け取ったイメージをもとにコンテナを作成する. 作成したコンテナに ComCal で作成した requirements.txt に記したアプリケーションの実行に必要なライブラリをインストールする. このインストールの際に time ライブラリを用いることでライブラリのインストール時間を測定する. インストール後, 作成したライブラリを削除する. この作成, インストール, 削除を 100 回繰り返すことで 100 回の同じライブラリのインストール時間を測定する. この hyouka.py に完備率の比較先として GitHub と Docker Hub から参照した Docker イメージを渡すコマンドを hyouka.sh に書くことにより評価実験を行った.

*2 https://github.com/hivdb/codfreq (参照 2021 年 11 月 29 日)
*3 https://github.com/labd/docker-python-lint (参照 2021 年 11 月 29 日)
*4 https://hub.docker.com/r/nodecustombase/nodealpine16-npm7_22_0-couchbase (参照 2021 年 11 月 29 日)
*5 https://hub.docker.com/r/joserochabh/sisopapi (参照 2021 年 11 月 29 日)

5. 評価と分析

同じマイクロサービスでの実験

実験の完備率算出のコンテナに GitHub から doktor-v2 のマイクロサービスを用いる. doktor-v2 のサービスは paper, front, front-admin, thumbnail, author がある. しかし, thumbnail サービスはコンテナとして実装されていないので paper, front, front-admin, author サービスを用いる. doktor-v2 のサービスの中で author コンテナを完備率の比較元コンテナ (アプリケーションのコピー元のコンテナ) に指定し, 完備率の比較先コンテナ (アプリケーションのコピー先のコンテナ) に doktor-v2 のサービスの front-admin コンテナ, paper コンテナ, front コンテナと doktor サービスの中の website コンテナで完備率を算出する. また完備率算出の際に比較先に足りないライブラリをインストールする時間を求める. 実験の結果, front-admin コンテナに足りないライブラリのインストール 100 回のインストール時間をの頻度グラフを図 9 に示す.

図 9 よりインストール時間の分布は 2.5 秒以上 3 秒未満に 43 回, 3 秒以上 3.5 秒未満に 53 回と 2.5 秒以上 3.5 秒未満にピークがある. しかし, インストール時間が 5 秒以上 6 秒未満に 3 回あることから, 通常 2.5 秒以上 3.5 秒未満にインストールできるはずだが, 3%で通常の時間の倍の時間がかかることがある. これはライブラリのインストールにインターネットを経由しているため, つながりやすいときとつながりにくいときで差が出るためである. 別の理由として実験環境に VM を用いているため, 同じサーバーの他の VM でインストール作業を行うと通信帯域に影響が起

© 2021 Cloud and Distributed Systems Laboratory

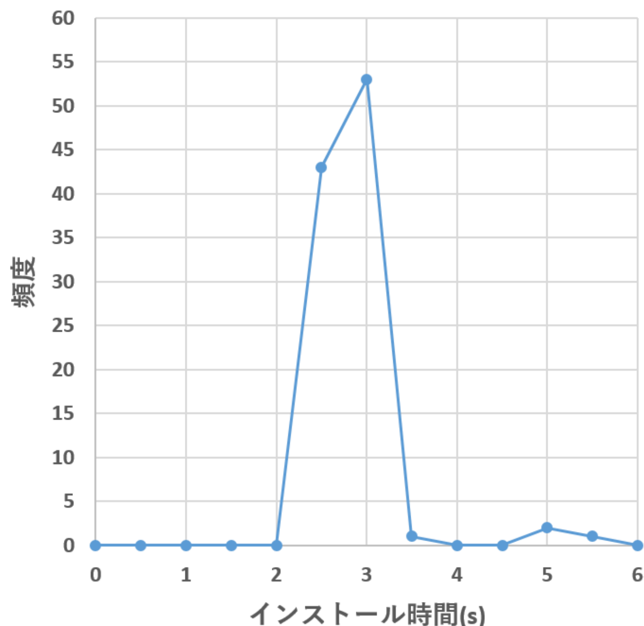

図 9　doktor-v2 の front-admin コンテナへの足りないライブラリインストール時間の頻度図

き, 通信速度が遅くなることがある.

また, author コンテナと doktor の 4 つのコンテナとのインストール時間と完備率の関係を図 10 のグラフに示す.

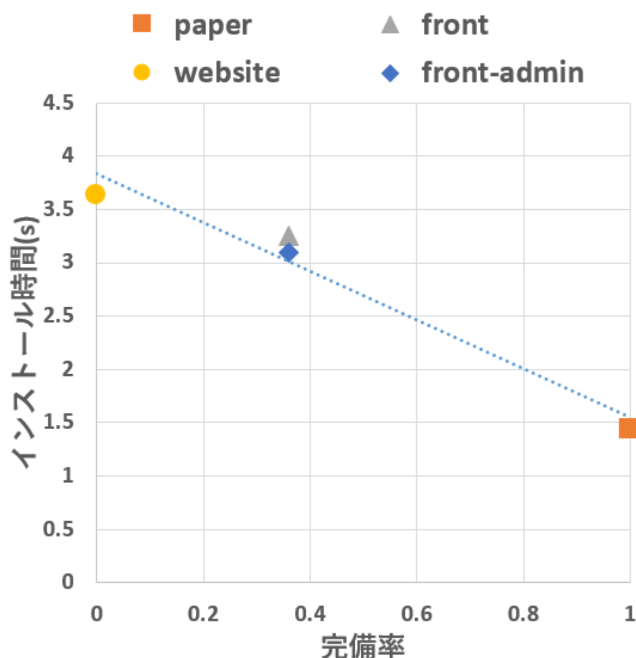

図 10　doktor-v2 の author コンテナと doktor コンテナとのインストール時間と完備率

図 10 より近似曲線を見ると, 完備率とインストール時間に一次関数の関係があることがわかる. この関係を相関係数で算出すると約-0.97 という値になり, 相関が高いことがわかる. しかし, 同じマイクロサービスだけで取ったデータではコンテナの数が少なくデータの数が少ないため比較数を増やす必要がある.

GitHub と DockerHub のコンテナ 50 個での実験

多くのコンテナで完備率を算出するため GitHub と DockerHub からイメージを参照し実験を行う. 比較元に doktor-v2 の paper コンテナを用い, 比較先のコンテナに GitHub と DockerHub にあるイメージでベースイメージに同じ python:3.9-alpine を使用しているイメージを使用して完備率を算出した. また足りないライブラリをそれぞれのコンテナで 100 回インストールし, インストール時間と完備率の関係を図 11 のグラフに示す.

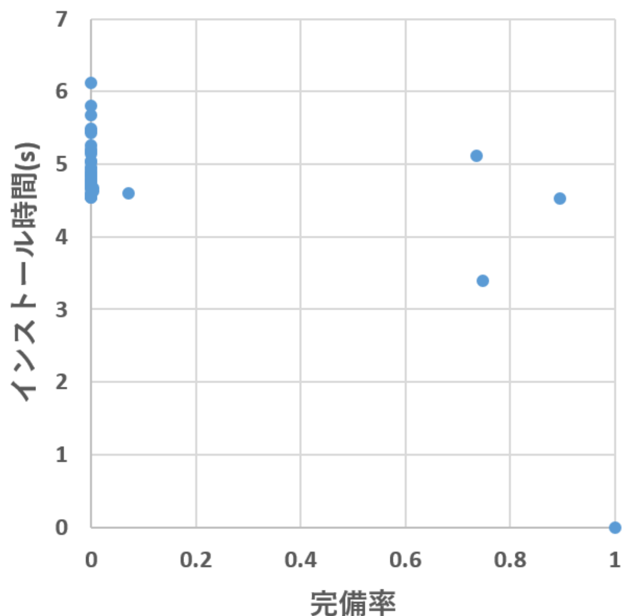

図 11　doktor-v2 の paper コンテナと GitHub, DockerHub のコンテナ 50 個のインストール時間と完備率

図 11 の関係を相関係数で算出すると約-0.63 と先ほどと比べるとあまり高い相関は得られなかった. しかし約-0.63 は相関があると言え, 本提案の完備率からインストール時間によるコンテナの類似度を測ることができる. また, 図 11 のグラフから GitHub と DockerHub のコンテナでは完備率に 0 が多く, 完備率が高いコンテナは同じ doktor のコンテナだけだった. これは GitHub と DockerHub のコンテナでは作られた時期が大きく違うため, 本提案である完備率算出のバージョン一致がほとんど起きなかったためである. 他の要因として, GitHub と DockerHub では様々な人が作成したものを使用しているため, 人によってよく使うライブラリが異なるため, 無数にある Python のライブラリでは全く違う人が同じライブラリ構成でアプリケーションを作成することがないためである. これらの結果から GitHub と DockerHub からイメージを参照で行った結果, 同じライブラリを使用しているコンテナが少なく, 完備率とインストール時間の関係についてのデータが十分に取

© 2021 Cloud and Distributed Systems Laboratory

れなかった.

ランダムにライブラリをインストールしたコンテナ 50 個での実験

同じベースイメージで作成したコンテナにコピー元のライブラリをランダムにインストールし実験を行う．この実験では GitHub と DockerHub のコンテナでは完備率に 0 が多かったため示せなかった完備率とインストール時間との関係を示す．足りないライブラリをランダムにすることで, 0 から 1 までの様々な完備率を示すコンテナを作成することができる．比較元に doktor-v2 の front-admin コンテナを使用し, front-admin コンテナのランダムなライブラリの組み合わせを 50 種類用意し, ベースイメージのコンテナにそれぞれ 100 回ずつインストールを行った．doktor-v2 の front-admin コンテナのライブラリ一覧表を表 2 に示す．

表 2 doktor-v2 の front-admin コンテナのライブラリ一覧

ライブラリ名	バージョン
python-multipart	0.0.5
yarl	1.6.3
MarkupSafe	2.0.1
idna	3.2
async-timeout	3.0.1
fastapi	0.68.1
asgiref	3.4.1
click	8.0.1
attrs	21.2.0
six	1.16.0
multidict	5.1.0
Jinja2	3.0.1
starlette	0.14.2
chardet	4.0.0
h11	0.12.0
aiohttp	3.7.4.post0
uvicorn	0.15.0
typing-extensions	3.10.0.2

比較元のコンテナに doktor-v2 の front-admin コンテナを用いてベースイメージである python:3.9-alpine にライブラリをインストールし, インストール時間と完備率の関係を図 12 のグラフに示す．

図 12 より線形曲線を見ると完備率とインストール時間に一次関数の関係があることがわかる．この関係を相関係数で算出すると約-0.94 という値になり, 相関が高いことがわかる．また 1 を完備率で引くことで近似すると比例の関係があることがわかる．これにより完備率が高ければ高いほどインストール時間が短くなる傾向がある．

また, 同じ実験からインストール時間とライブラリインストール数の関係のグラフを図 13 に示す．

図 13 より近似曲線を見るとインストール時間とライブラリインストール数に比例関係があることがわかる．この関係を相関係数で算出すると約 0.90 という値になり, 相関が高いことがわかる．しかし, 今回提案した完備率のほうがライブラリインストール数との相関係数を比較したとき,

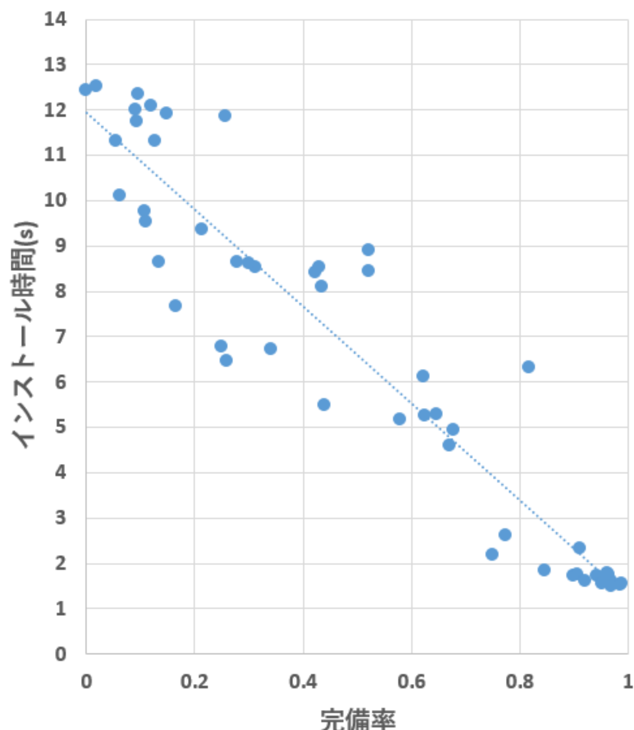

図 12 doktor-v2 の front-admin コンテナのインストール時間と完備率

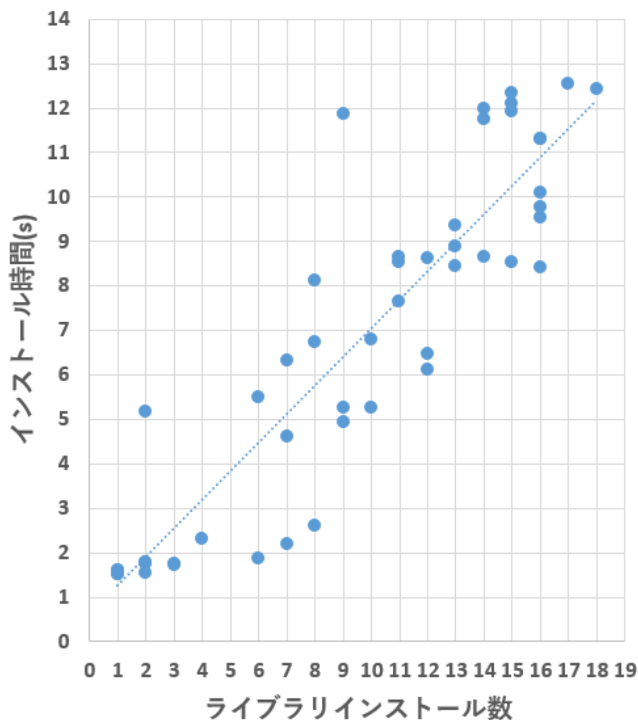

図 13 doktor-v2 の front-admin コンテナのインストール時間とライブラリインストール数

約 0.04 ポイント高いことからライブラリのファイルサイズで完備率を図ることが, ライブラリ数で測るより正確な値を算出できていることがわかる.

6. 議論

Python のライブラリの中でライブラリのインストール時に, C/C++のソースコードをコンパイルする必要がある場合がある. コンパイル自体はインストールスクリプトが自動的に行ってくれるが, コンパイラは事前に準備して おかないとそのライブラリをインストールすることができない. そのため同じベースイメージを使用したコンテナでアプリケーションをコピーし, 実行しようとしてもコピー先に gcc がインストールされていない場合実行できない. そのため, ライブラリの比較以外にも C/C++コンパイラがあるかどうかを比較する必要がある. また, C/C++コンパイラを必要とするライブラリかどうかライブラリをインストールし, できなかった場合にエラーメッセージを見ることによりライブラリを使用する際にどのコンパイラが必要か知ることができ, アプリケーションの実行に必要なライブラリのようにコンパイラもインストールすることで解決できる.

アプリケーションをコピー先のコンテナで実行する場合, アプリケーションとアプリケーションに依存するライブラリをコンテナにインストールする必要がある. このライブラリのインストール時間は本研究で提案する完備率によりどのコンテナがインストールするデータサイズが小さく, インストール時間が短いかがわかる. しかし, 本研究では完備率やインストールするデータサイズからインストール時間を特定することができない. これはライブラリのインストールにインターネットを経由し, 通信速度が常時一定ではないからである. そのためインストール時間を一定にするためにはインターネットを経由しないでインストールすることで解決できる. インターネットを経由しないでインストールする方法として, コンテナ作成時に使用したライブラリをインストールできるレポジトリに保存し, レポジトリからライブラリをインストールすることでインターネットを経由しないでインストールすることができる. これにより通信速度に左右されずにライブラリをインストールすることができるようになる. また, 通信速度が一定になり通信速度に左右されずにライブラリをインストールするすることができることにより, インストールするデータサイズからインストール時間を求められる.

本研究ではインストールするデータサイズを見ているため, インストール後に実際にライブラリを使用するための展開後のファイルサイズを見ていない. しかし, ライブラリのインストール時間には展開した後のファイルサイズにも影響する. このため展開後のファイルサイズも完備率の算出式に入れることにより, よりライブラリをインストールし, 実際にライブラリを使用できるようになる時間を正確に算出できる. また, ライブラリをインストールして使用できるようにするには, ライブラリの展開後のファイルサイズだけではなく, ライブラリ展開後のファイル数にも影響がある. そのため, ライブラリをインストールし, 展開した後のファイル数も算出式に入れることにより正確なインストール時間を算出できる.

実験のコンテナのデータの集め方について, 今回はインターネット上に上げられている様々な人が作成したイメージファイルを使用した. しかし, 図 11 のグラフから分かるように, 作成者が違うと完備率が低く, 差が出にくいため完備率があまり効果を示さなかった. そのため, 別の組織や人が作成したコンテナで完備率を用いられるようにするには, ライブラリの共通点だけでは足りないことがわかる. この解決方法として Dockerfile の一行目からベースイメージに共通した DockerHub のイメージを使用しているかで比較を行う. また Dockerfile からインストールされたコンパイラの比較を行うことで, コンテナで使用できるプログラム言語を取得することができる. これらの比較方法を完備率の算出に追加することで完備率が低く, 差が出にくいという問題を解決することができる. これによりライブラリに同じものを使用していないコンテナも, コンテナが類似しているかを知ることができるため, コンテナの変更が少なく異なるコンテナのアプリケーションを実行することができる.

7. おわりに

本研究では Python のアプリケーションを異なるコンテナで実行するため, 依存するライブラリがインストールされているか判断できないことが課題となる. そのため完備率という異なるコンテナでアプリケーションを実行するためのライブラリが, どの程度インストールされているか判断する指標の提案を行った. 実験はアプリケーションの実行に必要なライブラリのインストール時間と算出した完備率を比較し評価を行った. 実験結果は完備率とインストール時間に約-0.94 という高い相関関係を示すことができた. そのため完備率は異なるコンテナのアプリケーションを実行する際, 必要なライブラリのインストール時間の指標として使用することができる.

参考文献

[1] Jaramillo, D., Nguyen, D. V. and Smart, R.: Leveraging microservices architecture by using Docker technology, *SoutheastCon 2016*, IEEE, pp. 1–5 (2016).

[2] 飯島貴政, 串田高幸: マイクロサービスにおけるメトリクスによるサービスの優先順位および計算リソースの共助モデル, CDSL-TR-060, 東京工科大学 コンピュータサイエンス学部 クラウド・分散システム研究室 (2021.Sep.4).

[3] 野木空良, 飯島貴政, 串田高幸: レスポンスタイムの維持のための異なるファイル名とハッシュ値を用いたプ

© 2021 Cloud and Distributed Systems Laboratory

ログラムのコピー，CDSL-TR-062, 東京工科大学 コンピュータサイエンス学部 クラウド・分散システム研究室 (2021.Sep.27).

[4] Oliphant, T. E.: Python for scientific computing, *Computing in science & engineering*, Vol. 9, No. 3, pp. 10–20 (2007).

[5] Zerouali, A., Mens, T. and De Roover, C.: On the usage of JavaScript, Python and Ruby packages in Docker Hub images, *Science of Computer Programming*, Vol. 207, p. 102653 (2021).

[6] Gerlach, W., Tang, W., Keegan, K., Harrison, T., Wilke, A., Bischof, J., D'Souza, M., Devoid, S., Murphy-Olson, D., Desai, N. et al.: Skyport-container-based execution environment management for multi-cloud scientific workflows, *2014 5th International Workshop on Data-Intensive Computing in the Clouds*, IEEE, pp. 25–32 (2014).

[7] Hosny, A., Vera-Licona, P., Laubenbacher, R. and Favre, T.: AlgoRun: a Docker-based packaging system for platform-agnostic implemented algorithms, *Bioinformatics*, Vol. 32, No. 15, pp. 2396–2398 (2016).

[8] Baxevanis, A. D., Bader, G. D. and Wishart, D. S.: *Bioinformatics*, John Wiley & Sons (2020).

[9] Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J.: Agile software development methods: Review and analysis, *arXiv preprint arXiv:1709.08439* (2017).

[10] Goldberg, R. P.: Survey of virtual machine research, *Computer*, Vol. 7, No. 6, pp. 34–45 (1974).

[11] Potdar, A. M., Narayan, D., Kengond, S. and Mulla, M. M.: Performance evaluation of docker container and virtual machine, *Procedia Computer Science*, Vol. 171, pp. 1419–1428 (2020).

[12] Walters, J. P., Younge, A. J., Kang, D. I., Yao, K. T., Kang, M., Crago, S. P. and Fox, G. C.: GPU passthrough performance: A comparison of KVM, Xen, VMWare ESXi, and LXC for CUDA and OpenCL applications, *2014 IEEE 7th international conference on cloud computing*, IEEE, pp. 636–643 (2014).

© 2021 Cloud and Distributed Systems Laboratory