

# 高頻度な再配置を想定したコンテナマイグレーション機構の実現

永井 陽太<sup>1,a)</sup> 松原 克弥<sup>1,b)</sup>

**概要:** 筆者らは、教室 PC の余剰計算資源を活用するコンテナ型クラウドシステムの実現を目指している。本システムでは、授業での利用やクラウド管理者の意図しないシャットダウンによって、利用できる計算資源が動的かつ頻繁に変化することが想定される。この課題に対しては、実行中のコンテナを動的にマイグレーションすることによって再配置を行うことが有効である。しかし、任意ノードへの再配置を想定した既存コンテナマイグレーション機構は、インスタンス状態の保存や転送にかかるオーバーヘッドが大きい。本研究では、コンテナマイグレーションの際、遷移元ノードにチェックポイント状態を一定期間保持しておき、再び再配置が必要となった際の遷移先ノードとして優先的にそのノードを選択し、さらに、過去のチェックポイントとの差分のみを抽出して転送することでネットワーク転送量を削減する「残身型コンテナマイグレーション機構」を提案する。本稿では、OCI 標準コンテナランタイムである runC を対象として、提案する残身型コンテナマイグレーション機構の実現手法について述べる。

**キーワード:** クラウドコンピューティング, コンテナ型仮想化, マイグレーション

## 1. はじめに

本研究では、教室 PC の余剰計算資源を活用した学内向けコンテナ型クラウドシステム (以下、余剰計算資源クラウド) の実現を目指している [1]。教室 PC は、大学関係者であれば誰でも利用可能な状況にある。そのため、授業利用やクラウド管理者の意図しない教室 PC のシャットダウンによって、クラウドシステムの計算資源は動的かつ頻繁に変化することが想定される。この課題に対しては、クラウド上のコンテナを、より計算資源に余裕がある教室 PC へマイグレーションする方法が有効である。しかし、任意ノードへの再配置を想定した既存のコンテナマイグレーション機構では、コンテナのチェックポイント作成、転送にかかるオーバーヘッドが大きい。

そこで、コンテナマイグレーションの際、遷移元ノードにチェックポイントを一定期間保持しておき、再び再配置が必要となった際の遷移先ノードとして優先的にそのノードを選択する。さらに、過去のチェックポイントとの差分のみを抽出して転送することでネットワーク転送量を削減する「残身型コンテナマイグレーション機構」を提案す

る。本稿では、OCI(Open Container Initiative)[2] 標準コンテナランタイムである runC[3] と、Linux プロセスマイグレーションツールである CRIU[4] を対象として、提案する残身型コンテナマイグレーション機構の実現手法について述べる。また、残身型コンテナマイグレーション機構実現のために実装した、コンテナの差分 CP 作成機能とコンテナのレストア時メモリ変更追跡機能について述べた後、その有用性を示すための実験結果を示す。

## 2. 演習授業向け学内クラウドシステムの構築

筆者らは、PBL などの演習授業における学内での利用を想定したオンプレミス・クラウドの構築を行っている。本クラウドシステムの目的は、ソフトウェア開発などの演習を行っている学生に対して、自由に利用可能なコンテナを提供することである。

本クラウドシステムのシステム構成を図 1 に示す。本クラウドシステムの特徴は、学内に存在する限られた計算資源を有効活用するために、軽量なコンテナ型の仮想計算機環境を提供することである。コンテナ型のクラウドシステムにおけるオーケストレーション・エンジンとして広く導入されている Kubernetes を採用し、本システム用に調達したマイクロサーバだけでなく、既存オンプレミス・クラウド上の VM (仮想マシン)、さらには、利用されていない

<sup>1</sup> 公立はこだて未来大学 システム情報科学研究科  
Graduate School of Systems Information Science, Future University Hakodate

a) g2118026@fun.ac.jp

b) matsui@fun.ac.jp

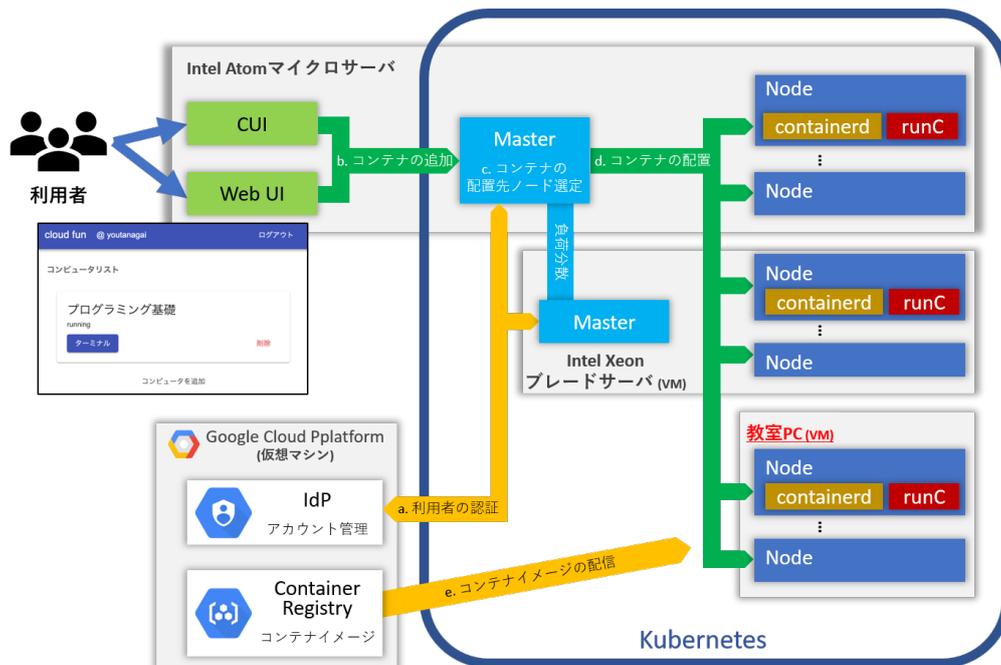


図 1 学内クラウドシステムのシステム構成

教室 PC をクラウドの計算資源として提供する機能を実現する。特に、教室 PC は、近年の BYOD(Bring Own Your Device) の普及により、稼働率が低下しており、利用されていない余剰計算資源の活用が求められている。筆者らが所属する大学の教室 PC の稼働状況を計測したところ、1 日あたりの平均稼働時間が 59 分しかないことが判明している\*1。

本クラウドシステムの利用者は、システムが提供する認証をパス (図 1 中の a.) した後、Web UI やコマンドライン CUI を介してコンテナの作成リクエストを発行する。発行されたリクエストを受け取った Master ノードは、ベアメタルサーバや VM、教室 PC などの異なる H/W プラットフォームのなかから利用可能な計算資源を特定し、コンテナランタイムである containerd や runC を介してコンテナのインスタンスを作成する (図 1 中の d.)。runC は、OCI(Open Container Initiative) というコンテナ仕様規格標準のリファレンス実装で、コンテナの実行に必要なすべてのファイルが含まれている rootfs というディレクトリとコンテナの実行に必要な設定が記述されている config.json を含むコンテナ実行環境 Bundle を生成する。また、runC は、コンテナやプロセスのマイグレーションツールとの連携機能を有しており、実行中のコンテナを別の計算ノードに移動させることが可能である。

### 3. 教室 PC の余剰計算機活用における課題

本研究では、学内クラウドシステムの計算資源として、

教室 PC の余剰計算資源を活用することを目指している。教室 PC の余剰計算資源活用における課題について、以下にまとめる。

#### 3.1 余剰計算資源の動的な増減

教室 PC の余剰計算資源を活用したクラウドシステムを構築する場合、授業利用やクラウド管理者の意図しないシャットダウンによって、頻繁にクラウドシステムの計算資源が増減すると考える。教室 PC が授業利用される場合、バックグラウンドでクラウドシステムのコンテナが動作していることで、教室 PC の動作が遅くなり本来の用途に悪影響を与えると考える。また、授業利用により、余剰計算資源が圧迫されることでコンテナの動作にも悪影響を与える。教室 PC は大学関係者であれば誰でも利用可能な環境にある。たとえば、教室 PC がクラウドシステムの計算資源として利用されていることを大学中に周知させたとしても、教室 PC をシャットダウンしてしまう可能性は存在する。教室 PC がシャットダウンされると、その教室 PC 上で稼働していたクラウドシステムのコンテナにアクセスできなくなってしまう。

#### 3.2 計算資源の動的な増減への対処

これらの課題に対しては、コンテナマイグレーションによるコンテナの再配置が有効である。授業利用によって余剰計算資源が圧迫された場合、余剰計算資源に余力がある教室 PC にコンテナをマイグレーションすることで、授業やコンテナへの悪影響を回避可能である。また、教室 PC がシャットダウンされた場合、シャットダウン処理中にコ

\*1 48 台の PC を対象として、2017 年 10 月 16 日から 11 月 15 日までの平日のみの稼働時間を計測

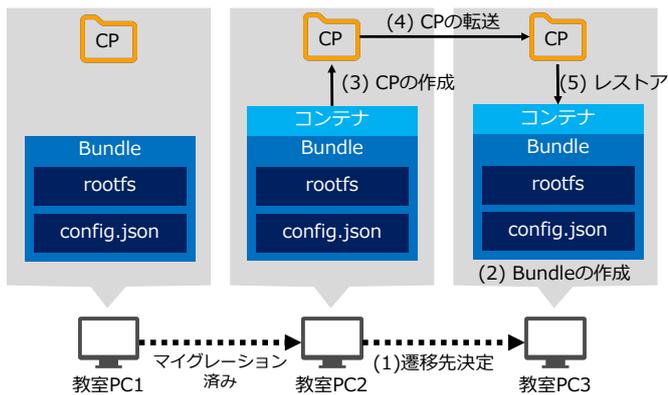


図 2 マイグレーション手順

コンテナを他の教室 PC にマイグレーションすることで、コンテナへアクセスできなくなる課題を回避可能である。

### 3.3 頻繁なマイグレーションによるクラウドの性能低下

節 3.1, 3.2 から、余剰計算資源を活用したクラウドでは頻繁にマイグレーションが行われると考える。コンテナのマイグレーションに時間がかかればかかるほど、コンテナにアクセスできない時間も長くなる。また、コンテナの計算以外に余剰計算資源が利用される時間も長くなる。つまり、頻繁にマイグレーションが行われる余剰計算資源を活用したクラウドシステムにおいては、コンテナのマイグレーションにかかる時間を削減する必要がある。

### 3.4 既存コンテナマイグレーションの手順

runC によって作成されたコンテナを例に、コンテナのマイグレーション手順を図 2 に示す。以降、チェックポイント操作によって作成されるディレクトリのことを CP と表記する。

- (1) 任意のノードを遷移先として決定
  - (2) 遷移先ノード上に Bundle を作成
  - (3) 遷移元ノード上で動作しているコンテナの CP 作成
  - (4) 遷移先ノードへ CP を転送
  - (5) 遷移先ノード上で CP をもとにコンテナをレストア
- マイグレーションを行うたびに、(2) の手順で Bundle を作成する必要がある。これは無駄な手順であると考えられる。また、(3) の手順で生成される CP のサイズが大きければ大きいほど (4) の手順で CP の転送に時間がかかってしまう。

## 4. 残身型コンテナマイグレーション機構の提案

コンテナ作成時やマイグレーション時に作成される Bundle を一定期間保持しておき、マイグレーションの際に再利用することで、図 2 における (2)Bundle の作成という手

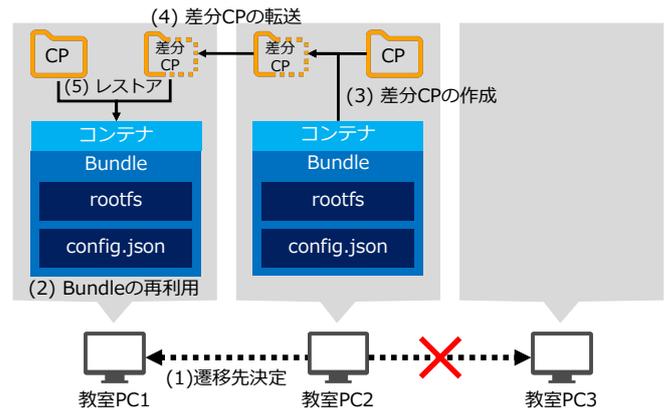


図 3 残身型コンテナマイグレーション機構におけるマイグレーション手順

順を省略可能と考える。また、CRIU のもつ差分チェックポイント (以下、差分 CP) 作成機能を利用し、CP のサイズを削減することで、図 2 における (4)CP の転送という手順の所要時間を削減可能と考える。そこで、Bundle やコンテナマイグレーションの際に作成された CP をノード上に一定期間保持しておき、再度コンテナのマイグレーションを行う際に、Bundle と過去の CP を保持しているノードを遷移先として決定することで、図 2 における (2)Bundle の作成と (4)CP の転送にかかる時間を削減する残身型コンテナマイグレーション機構を提案する。

### 4.1 残身型コンテナマイグレーション機構におけるコンテナマイグレーション手順

残身型コンテナマイグレーション機構におけるコンテナマイグレーション手順を図 3 に示す。

- (1) Bundle と過去の CP を保持しているノードを遷移先として決定
- (2) Bundle を再利用
- (3) 差分 CP の作成
- (4) 差分 CP の転送
- (5) 差分 CP と過去の CP を利用したレストア (このとき、メモリ変更追跡を開始)

### 4.2 残身型マイグレーションの実現に必要な機能

上述した残身型コンテナマイグレーション機構の実現には以下、4 つの仕組みが必要である。

- (1) Bundle と CP (残身) とコンテナを管理する仕組み
  - (2) Bundle や CP, ノードの負荷情報を元に遷移先ノードを決定する仕組み
  - (3) メモリ変更差分のみを CP として抽出する仕組み
  - (4) 複数の差分 CP を 1 つの CP として統合する仕組み
- それぞれの仕組みについて詳しく説明する。

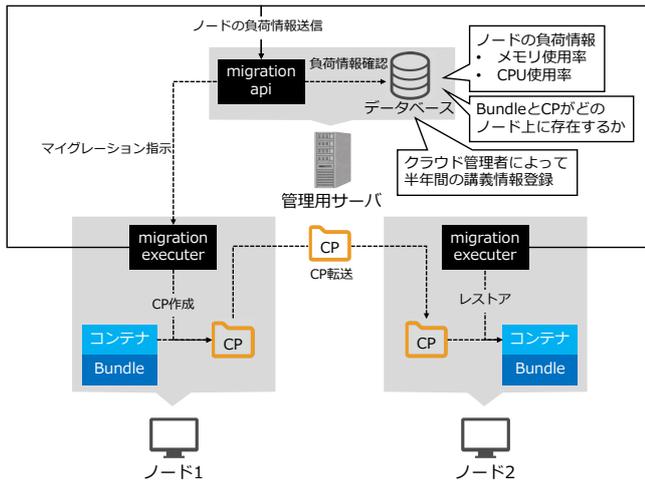


図 4 残身とコンテナ管理の仕組み

#### 4.2.1 残身とコンテナ管理

残身とコンテナの管理は図 4 に示すアーキテクチャで行う。各ノード上に残身 (Bundle と CP) とコンテナの管理を行う migration-executer を配置する。コンテナのマイグレーションを行う際に、コンテナに紐づく残身の情報を管理用 API サーバ (migration-api) に対して送信する。CP の管理情報を受け取った管理用 API サーバは、ノードとコンテナと CP を紐付ける情報をデータベース上に登録する。migration-executer は、後述する遷移先ノード決定のために、ノードの負荷情報 (CPU 使用率とメモリ使用率) を migration-api に対して一定間隔で送信し続ける。

#### 4.2.2 遷移先ノード決定ポリシー

遷移先ノードの決定は、式 1 によって求める。

$$k(C + M) - l(P + B) + mL \quad (1)$$

ここで、 $k$ ,  $l$ ,  $m$  はそれぞれ、ノードの負荷状態、残身の保有状態、講義の予定状態に対する係数である。 $C$  はノードの cpu 使用率であり、 $0 \leq C \leq 1$  とする。 $M$  はノードのメモリ使用率であり、 $0 \leq M \leq 1$  とする。 $P$  はどれだけ直近の差分 CP を持っているかを表している変数であり、CP の総世代数を  $N$ 、ノード上に存在する CP の世代数を  $M$  とした時、 $P$  は式 2 によって求められる。

$$P = M/N \quad (2)$$

$B$  は対処の Bundle を保持していれば 1、保持していなければ 0 となる変数である。 $L$  は、ノードが授業によって利用されるまでの期間を表す変数である。24 時間以内に講義がなければ、 $L$  は最大値 1 を取る。ノードが講義で利用されている場合、 $L$  は最小値 0 を取る。この時間割情報は、クラウド管理者が、項 4.2.1 で述べた migration-api に対して半年に 1 回、講義情報を登録する。

#### 4.2.3 メモリ変更差分抽出

コンテナのメモリ変更差分を抽出するためには、コンテ

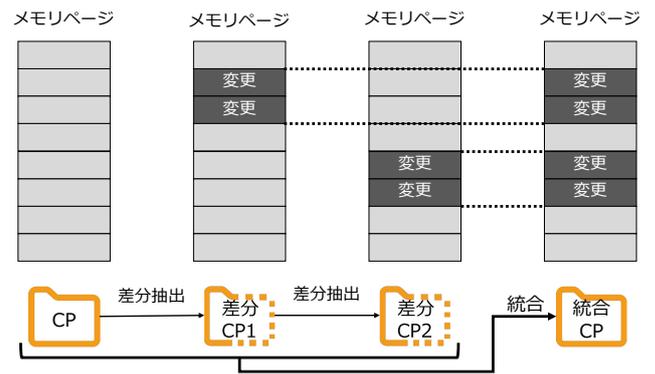


図 5 差分 CP 統合の仕組み

ナレストア時にコンテナのメモリ変更追跡を開始し、変更されたメモリページのみ抽出する機能が必要である。

#### 4.2.4 複数の差分 CP 統合

残身型コンテナマイグレーション機構では、マイグレーションを行うたびに 1 つのコンテナに紐づく差分 CP が増えていく。なぜならば、差分 CP からの差分も差分 CP として作成されるからである。コンテナマイグレーションを行う際、差分 CP が複数ある場合すべての差分 CP を遷移先に転送する必要がある。これでは、転送する CP のサイズを削減するという利点が無くなってしまふ。そこで、コンテナに紐づく差分 CP の数が一定の値を超えた時、図 5 の用に 1 つの CP として統合する。

### 5. 実装

項 4.2.3 にて述べた機能の実装について詳しく述べる。

#### 5.1 差分 CP 作成の実現

差分 CP 作成の実現は、runC を改造することによって行う。CRIU は、プロセスの CP 作成時に `-track-mem` というオプションを指定することで、Linux カーネルが持つ Memory Change Tracking の機能呼び出すことが可能である。この機能によって、プロセスレベルでメモリ変更を追跡することが可能である。そこで、runC のレベルでコンテナの CP 作成時に、CRIU のメモリ変更追跡機能呼び出せるように runC を改造した。

runC は Go 言語によって実装されている。runC は CRIU を呼び出すに `CriuOpts` という構造体を利用する。この `CriuOpts` という構造体には、CRIU が持つ様々なオプションをメンバとして持っている。しかし、`CriuOpts` には `-track-mem` を呼び出すためのメンバが存在していない。そのため、`CriuOpts` に CRIU の `-track-mem` オプションと対応した `TrackMem` というメンバを追加した。

runC を利用してコンテナの CP を作成する際は、runC の `Checkpoint` というサブコマンドを利用する。この `Checkpoint` サブコマンドには `CriuOpts` と対応したオプション

が指定可能となっている。そこで、Checkpoint サブコマンドへ、CriuOpts に追加した TrackMem メンバと対応する `-track-mem` オプションを追加した。

以上の改造を行うことで、runC の Checkpoint サブコマンドに `-track-mem` オプションを追加し、CRIU の `-track-mem` オプションを呼び出すことを可能にした。これによって差分 CP の作成を実現した。

## 5.2 レストア時メモリ変更追跡の実現

CRIU は、CP 作成時に Linux カーネルが持つ Memory Change Tracking の機能呼び出すことが可能であるが、レストア時に呼び出すことはできない。CRIU は CP 作成時に `task_reset_dirty_trck()` という関数を呼び出すことで、Memory Change Tracking を呼び出している。そこで、レストア時にも同様の関数を呼び出すことで、レストアしたコンテナの Memory Change Tracking を有効にし、コンテナのメモリ変更追跡を行う。

runC から、CRIU に追加したレストア時メモリ変更追跡機能呼び出すために、節 5.1 にて CriuOpts に追加した TrackMem を再利用した。また、runC によるコンテナのレストアは Restore サブコマンドによって実現されている。そこで、runC の Restore サブコマンドに `-track-mem` オプションを追加することで、CRIU に実装したレストア時メモリ変更追跡機能呼び出せるよう runC を改造した。これらの改造によって、レストア時メモリ変更追跡を実現した。

## 6. 実験と評価

実装した差分 CP 作成機能と、レストア時メモリ変更追跡機能について、その性能を評価するため、実験を行った。それぞれの実験の内容と結果を述べる。

### 6.1 差分 CP 作成のオーバーヘッド

差分 CP 作成のために必要なメモリ変更追跡によって、CP 作成にどれほどのオーバーヘッドが発生するのか計測を行った。実験の対象となるコンテナは、内部で `stress` というコマンドを実行するコンテナである。

#### 6.1.1 差分 CP 作成のオーバーヘッド計測手順

差分 CP 作成時に発生するオーバーヘッドの計測は、以下の手順で行った。

- (1) 対象コンテナの CP を作成する
- (2) (1) の手順を 100 回繰り返し、CP 作成にかかる時間を計測する
- (3) 対象コンテナの CP を作成する (この時、メモリ変更追跡を開始する)
- (4) (3) で作成した CP との差分 CP を作成する
- (5) (3), (4) の手順を 100 回繰り返し、差分 CP 作成にかかる時間を計測する

実験環境は、表 1 の通りである。

表 1 実験で利用した計算機のスペック

OS	Ubuntu 18.04.1 LTS
メモリ	8GB
CPU	Intel Core i7 @ 2.2GHz

### 6.1.2 差分 CP 作成のオーバーヘッド計測の結果

項 6.1.1 にて述べた手順によって表 2 のような結果が得られた。CP 作成を 100 回行った結果、その平均所要時間は 5.56 秒となった。差分 CP 作成を 100 回行った結果、その平均所要時間は 5.64 秒となった。CP 作成と差分 CP 作成の平均所要時間の差は 0.08 秒となった。これは CP 作成にかかる時間の約 1.48% である。

表 2 CP 作成と差分 CP 作成の平均所要時間

	平均所要時間 (秒)
CP 作成	5.56
差分 CP 作成	5.64

## 6.2 レストア時メモリ変更追跡のオーバーヘッド計測

CP からコンテナをレストアする際に、メモリ変更追跡を有効にすることで、どれほどのオーバーヘッドが発生するのか計測を行った。実験の対象となるコンテナは、内部で `stress` というコマンドを実行するコンテナである。

### 6.2.1 レストア時メモリ変更追跡のオーバーヘッド計測手順

レストア時メモリ変更追跡によって発生するオーバーヘッドの計測は、以下の手順で行った。

- (1) 対象コンテナの CP を作成する
- (2) (1) の手順で作成した CP をもとに、メモリ変更追跡を無効にしてレストアを行う
- (3) (2) の手順を 100 回繰り返し、レストアにかかる時間を計測する
- (4) (1) の手順で作成した CP をもとに、メモリ変更追跡を有効にしてレストアを行う
- (5) (4) の手順を 100 回繰り返し、レストアにかかる時間を計測する

実験環境は表 1 の通りである。

### 6.2.2 レストア時メモリ変更追跡によるオーバーヘッド計測の結果

項 6.2.1 にて述べた手順によって表 3 のような結果が得られた。メモリ変更追跡を無効にした場合、100 回のレストアにかかる時間の平均は 3.93 秒となった。メモリ変更追跡を有効にした場合、100 回のレストアにかかる時間の平均は 4.32 秒となった。約 0.39 秒のオーバーヘッドが発生しており、これはメモリ変更追跡を無効にしたレストアにかかる時間の約 10% である。

表 3 メモリ変更有効と無効で 100 回レストアした平均の所要時間

メモリ変更追跡の状態	平均所要時間 (秒)
無効	3.93
有効	4.32

### 6.3 評価・考察

項 6.1.2 にて述べた結果より、差分 CP 作成によって約 1.48% のオーバーヘッドの発生を確認した。1.48% のオーバーヘッドは、差分 CP のサイズが差分でない CP のサイズに比べて小さくなるというメリットに対して許容できるオーバーヘッドであると考えられる。

項 6.2.2 にて述べた結果より、レストア時にメモリ変更追跡を行うことで、約 10% のオーバーヘッドの発生を確認した。10% のオーバーヘッドは、決して小さいとは言えない。そのため、レストア時にメモリ変更追跡を行うことで実現可能な差分 CP による CP サイズ削減によって、どれほど CP の転送にかかる時間を削減できるのか計測し、その相殺分を調査する必要がある。

## 7. おわりに

### 7.1 まとめ

本稿では、教室 PC の余剰計算資源を活用したコンテナ型オンプレミス・クラウドの実現に向けて、技術的な課題とその解決手法について検討した。クラウドシステムの計算資源が動的かつ頻繁に増減するため、頻繁にマイグレーションが発生し、クラウドの性能が低下する課題に対して、Bundle の再利用やマイグレーション時に作成する過去の CP からの差分のみを抽出し転送することで、マイグレーションにかかる時間を削減する残身型コンテナマイグレーション機構を提案した。また、実装の第 1 段階として OCI 標準のコンテナランタイムである runC と、プロセスのマイグレーションを実現する CRIU を改造することで、コンテナの差分 CP 作成とコンテナレストア時にメモリ変更追跡を有効にする機能の実装を行った。その後、差分 CP 作成とレストア時メモリ変更追跡によるオーバーヘッドの計測を行った。その結果、差分 CP 作成のためのオーバーヘッドが 1.48%、レストア時メモリ変更追跡によるオーバーヘッドが 10% 発生することがわかった。これらのオーバーヘッドは、これらの機能によって実現できる CP の転送時間削減との相殺分を評価する必要がある。

### 7.2 今後の課題

今後の課題として、項 4.2.1 にて述べた、式 1 の係数  $k$ ,  $l$ ,  $m$  を決定する。また、項 4.2.2, 項 4.2.4 にて述べた、残身とコンテナ管理の仕組みと差分 CP を統合する仕組みを実現する。すべての実装が終了したのち、実環境にてクラウドシステムの運用を行い、提案した残身型コンテナマイグレーション機構の有用性を調査する。

## 参考文献

- [1] 永井陽太, 松原克弥, 学内向けオープンクラウドにおける計算資源の動的な増減への対応手法に関する検討, 研究報告インターネットと運用技術 (IOT), 2018-IOT-40(5), 1-6, (2018-02-26).
- [2] OPEN CONTAINER INITIATIVE : HOME(online), 入手先 (<https://www.opencontainers.org>) (2019.02.02).
- [3] GitHub, Inc : opencontainers/runc(online), 入手先 (<https://github.com/opencontainers/runc>) (2019.02.02).
- [4] CRIU Project : CRIU Main Page(online), 入手先 ([https://criu.org/Main\\_Page](https://criu.org/Main_Page)) (2019.02.02).
- [5] The Kubernetes Authors : Production-Grade Container Orchestration, 入手先 (<https://kubernetes.io>) (2019.02.04).