VM外でのコンテナマイグレーションのための状態保存機構

朝倉 優輝[1]　光来 健一[1]

概要：近年，コンテナを提供するクラウドが普及しており，クラウドにおいては仮想マシン (VM) 内でコンテナを動作させることが多い．コンテナは負荷分散などのためにマイグレーションによって他の VM へ移動させることができる．しかし，VM の負荷が高い場合にはマイグレーション性能が低下し，マイグレーション処理がコンテナ性能にも大きな影響を与える．また，マイグレーション性能は VM による仮想化の影響も受ける．本稿では，VM 内で動作しているコンテナの軽量なマイグレーションを実現するために，VM 外でコンテナの状態を保存するシステム OVmigrate を提案する．OVmigrate は移送元ホストにおいて VM イントロスペクションを用いて VM のメモリに格納されているコンテナの状態を VM 外から取得する．OVmigrate を用いることで VM の負荷や VM による仮想化の影響を受けることなく迅速にコンテナマイグレーションを行うことが可能となる．KVMonitor と LLView を用いて OVmigrate を実装し，既存ツールとの性能比較を行った．

1. はじめに

近年，コンテナを提供するクラウドが普及しており，AI や IoT を支えるビッグデータを扱うためなどに利用されている．クラウドにおいては管理の面から仮想マシン（VM）内でコンテナを動作させていることが多い [6]．コンテナは OS によって提供される軽量な仮想環境であり，いくつかのプロセスとその実行環境のみによって構成されている．コンテナはマイグレーションと呼ばれる技術を用いて実行を継続したまま別の VM へ移動させることができる．例えば，負荷が高い VM の中のいくつかのコンテナを別の VM に移動させることで，負荷を分散することができる．

負荷分散を行う際のように，コンテナマイグレーションは VM の負荷が高い時に行わざるを得ないことも多い．このような場合には，コンテナマイグレーションが VM の負荷の影響を受け，マイグレーション性能が大きく低下する可能性がある．コンテナマイグレーション自体の負荷も高いため，マイグレーション処理が VM 内のコンテナの性能に大きな影響を与えることもある．また，コンテナを VM 内で動作させるとマイグレーション処理も VM 内で行うことになるため，コンテナマイグレーションの性能は VM による仮想化の影響も大きく受ける．

本稿では，VM 内で動作しているコンテナの軽量なマイグレーションを実現するために，VM 外でコンテナの状態を保存するシステム OVmigrate を提案する．OVmigrate は移送元ホストにおいて VM イントロスペクションを用いて VM のメモリに格納されているコンテナの状態を VM 外から取得する．保存したコンテナの状態を移送先ホストへ転送し，そのホストの VM 内に移送元のコンテナの状態を復元する．OVmigrate を用いることで，VM の負荷や VM による仮想化の影響を受けず，迅速にコンテナマイグレーションを行うことができる．また，コンテナマイグレーションの処理が VM 内のコンテナに影響を与えるのを防ぐこともできる．

我々は OVmigrate を KVMonitor [2] および LLView [1] を用いて実装した．OVmigrate は KVMonitor を用いて共有した VM のメモリから OS データを取得する．また，LLView を用いて OS のソースコードを利用してメモリ上の OS データを解析する．コンテナの状態は主にプロセスの状態によって構成されているため，OVmigrate はプロセスのメモリ情報，ファイル情報，Cgroup 情報などを保存する．実験により，OVmigrate を用いて VM 外でプロセスの状態を保存することができ，VM 内で動作させた既存ツールの CRIU [3] がその状態を用いて正常にプロセスを復元できることを確認した．また，CRIU と比較して OVmigrate では最大で 1.3 倍高速にプロセスの状態が保存でき，ばらつきも抑えられることが分かった．

以下，2 章でコンテナマイグレーションの問題点について述べる．3 章で VM 外でコンテナの状態を保存する OVmigrate を提案し，4 章でその実装について述べる．5 章で OVmigrate を用いてプロセスの状態を保存する性能について調べた実験について述べる．6 章で関連研究に触

[1] 九州工業大学
Kyushu Institute of Technology

© 1959 Information Processing Society of Japan

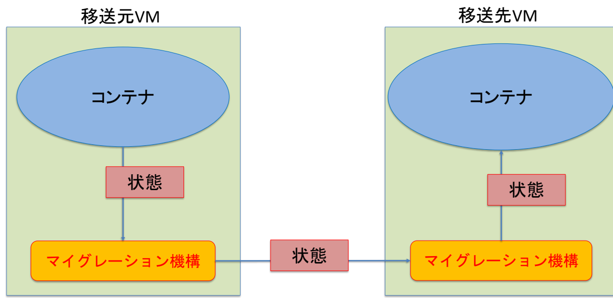

図 1　コンテナマイグレーション

れ，7 章で本稿をまとめる．

2.　コンテナマイグレーション

近年, Amazon ECS/EKS, Google GKE，Microsoft AKS などのクラウドでコンテナが提供されるようになっている．コンテナは OS によって提供される軽量な仮想環境である．VM を用いた従来の仮想化とは異なり，コンテナはハードウェアの仮想化を行わず，ホストの OS カーネルを利用して動作する．コンテナの中ではいくつかのプロセスが実行され，ファイルシステムやネットワークインタフェースなどの実行環境が提供される．このように，コンテナは少ないリソースで構成されるため，高速に起動や実行を行うことができる．クラウドでは VM 内でコンテナを動作させていることが多い [6]．これは物理ホストよりも柔軟に管理を行えるようにするためと考えられる．

VM 内で動作しているコンテナはマイグレーションと呼ばれる技術を用いて，実行を終了させることなく別の VM へ移動させることができる．コンテナマイグレーションを実行する流れを**図 1** に示す．まず，移送元 VM 上でコンテナ内部のプロセスの状態と実行環境をコンテナの状態として保存する．次に，保存されたコンテナの状態を移送先 VM へ転送し，移送先 VM 上で受信したコンテナの状態から元のコンテナを復元する．コンテナマイグレーションを用いることで，例えば，負荷の高いコンテナを別の VM に移動させることにより，VM 間で負荷を分散することが可能となる．また，VM 内の OS のアップデートを行う際にコンテナを別の VM に移動することにより，コンテナを停止させずに OS の保守や VM の再起動などを行うこともできる．

クラウドでは，1 つの VM に多くのコンテナを集約させて実行しているため，VM の負荷が高い時にその中のコンテナをマイグレーションせざるを得ないことも多い．例えば，コンテナマイグレーションによる負荷分散は一般的に，コンテナが動作している VM の負荷が高くなったことを検出した時に行われる．コンテナをマイグレーションしている間にそのコンテナが動作している VM の負荷が高くなることも考えられる．このような場合には，コンテナマイグレーションが VM の負荷の影響を大きく受け，マイグレーション性能が大きく低下する可能性がある．また，コンテナマイグレーション自体の負荷も高いため，VM の中で動作しているコンテナの性能への影響は避けられない．

VM に関しては，マイグレーションを用いた負荷分散について様々な研究が行われている．例えば，Sandpiper [4] では CPU やネットワークの使用率が 75%を超えた時に VM マイグレーションを行うことを推奨している．VM マイグレーションによって負荷の高いホストのサーバ性能が 50%，ネットワーク性能が 20%低下することから，この閾値はマイグレーションのオーバヘッドを吸収できるように決められている．しかし，ホストのリソース使用率が 75%以下に低下するため，クラウドのコスト上昇につながる．また，GCE ではホストの負荷が高い時には VM マイグレーションの速度を調節していると報告されている [5]．しかし，その場合には VM マイグレーションを迅速に行うことができないため，負荷分散などに時間がかかることになる．

また，VM 内のコンテナをマイグレーションする際にはマイグレーションの処理も VM 内で行われ，VM による仮想化の影響を受ける．実際に，ネットワーク仮想化がオーバヘッドの主な原因であり，CPU 使用率が移送元 VM で 70%，移送先 VM で 118%になるという報告がある [7]．そのため，Portkey [7] はコンテナマイグレーションの際に行われる状態転送のためのネットワークアクセスを高速化しているが，VM 内のマイグレーション機構やゲスト OS への変更が必要である．また，我々の実験 [9] によると，仮想ネットワーク以外のプロセス間通信や仮想ディスクへの書き込みでも仮想化による影響が見られている．

3.　OVmigrate

本稿では，VM 内で動作しているコンテナの軽量なマイグレーションを実現するために，VM 外でコンテナの状態を保存するシステム OVmigrate を提案する．OVmigrate のシステム構成を**図 2** に示す．2 章で述べたように，従来のコンテナマイグレーションはコンテナが動作する VM 内でマイグレーション機構を実行していた．それに対して，OVmigrate は移送元のマイグレーション機構をコンテナが動作する VM と同じホストではあるが VM の外部で実行する．移送元ホストで動作するマイグレーション機構は VM 内のコンテナの状態を保存し，VM の仮想ネットワークを用いずに移送先ホストへ転送する．移送先ホストのマイグレーション機構を VM の外部で動作させる研究も行われている [10] が，本研究では状態復元の軽量化は対象外としている．そのため，移送先ホストのマイグレーション機構は従来通り VM 内で動作させ，VM 内に新しいコンテナを作成してから受信したデータを用いて移送元 VM のコンテナの状態を復元する．

ⓒ 1959 Information Processing Society of Japan

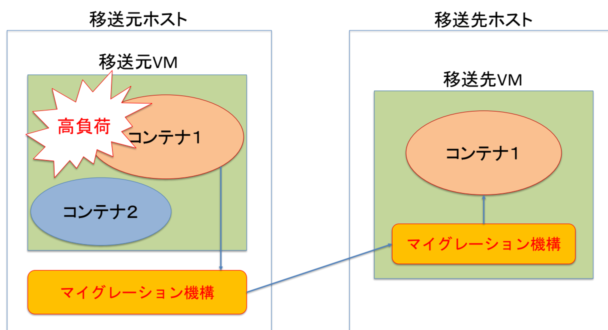

図 2　OVmigrate のシステム構成

OVmigrate はコンテナの状態保存を VM 外で実行するため，マイグレーション機構は VM による仮想化の影響を受けない．また，マイグレーション機構は VM の負荷の影響も受けない．VM には CPU やメモリなどのリソースを一定の量だけ割り当てるため，VM が高負荷になっても割り当てられた量以上のリソースを使うことはない．I/O の帯域幅についても VM ごとに制限することが可能であるため，VM が大量の I/O を発生させてもホスト全体では一定の負荷に抑えることができる．加えて，VM 内でコンテナマイグレーションに関する処理を行わないことから，VM 内で動作しているコンテナの性能に影響を与えない．一般に，VM を動作させるホストではリソースに余裕があることが多いため，マイグレーション機構は VM に割り当てたリソースを使わずに実行することができる．

OVmigrate はコンテナの状態を保存する際に主にプロセスの状態の保存を行う．コンテナはその内部で動作するプロセスとその実行環境で構成されており，コンテナの状態の大部分はプロセスの状態によって占められている．また，コンテナの実行環境に関する情報の多くもプロセスの状態に含まれている．プロセスの状態としては，プロセスに割り当てられているメモリに関する情報，プロセスが開いているファイルに関する情報，スレッドに関する情報，プロセス間の関係を表すプロセス木の情報などがある．一方，コンテナの実行環境に関する情報としては，プロセスのグループへのリソースの割り当てや制限に関する情報などがある．

OVmigrate は VM 内のプロセスの状態を VM イントロスペクション [1] を用いて取得する．VM イントロスペクションは VM のメモリ上にある OS やプロセスのデータを解析することによって，VM 外から VM 内の情報を取得する手法である．OVmigate はプロセス ID を基にカーネルメモリ上にあるプロセス構造体を見つける．そして，プロセス構造体からポインタを辿っていくことにより，プロセスに関する様々な状態を取得する．VM イントロスペクションを用いることで，VM 内で状態取得のためのプロセスを動作させたり，VM 内の OS に変更を加えたりすることなく，プロセスの状態保存を行うことができる．

OVmigrate は VMMfas [8] を用いて VM 内のプロセスの状態保存時にプロセスの実行を制御する．VMMfas は VM のカーネルメモリ上のプロセス情報を書き換えてシグナルが送られた状態に変更することにより，疑似的なシグナル送信を実現する．さらに，シグナルがプロセスに即座に配送されるようにするために，プロセススケジューラの状態を書き換えて疑似的なスケジューリングも行う．この機構を用いて，プロセスの状態保存を開始する時に STOP シグナルを疑似送信してプロセスを停止させる．これにより，プロセスの状態が変更されないようにしてから状態の保存を行う．また，プロセスの状態保存が完了した時には KILL シグナルを疑似送信してプロセスを強制終了させる．

4. 実装

我々は QEMU-KVM 4.2.0 上で動作する VM 内のゲスト OS である Linux 5.4 に対して OVmigrate を実装した．現在のところ，OVmigrate は OS の最小限の機能を用いるプロセスについてすべての状態を保存することができている．OVmigrate が対応しているプロセスの状態は，メモリ情報，ファイル情報，スレッド情報，時刻情報，プロセス木の情報，Cgroup 情報である．プロセスの保存・復元を行う既存のツールである CRIU [3] を用いて復元が行えるように，CRIU との互換フォーマットで状態を保存する．

4.1 メモリ情報の保存

プロセスのメモリ情報として，プロセスメモリのレイアウト情報や OS によってプロセスメモリに格納された実行に必要な情報を保存する．まず，プロセス ID を基に init_task 変数からプロセスリストをたどり，task_struct 構造体を探索する．この構造体にはプロセスに関する情報が格納されている．次に，task_struct 構造体からポインタをたどって mm_struct 構造体を見つける．この構造体にはプロセスメモリ全体に関する情報が格納されている．そして，この構造体からコード領域，データ領域，ヒープ領域の開始・終了アドレス，スタックのアドレスを取得する．また，プロセスに渡された引数が格納された領域と環境変数が格納された領域の開始・終了アドレス，ELF ローダによって様々な情報が格納された補助ベクタ（AUXV）の値も取得する．

プロセスの仮想メモリ領域の情報として，**図 3** のようにプロセスに割り当てられているすべての仮想メモリ領域の情報を保存する．そのために，mm_struct 構造体から vm_area_struct 構造体のリストを取得する．この構造体にはプロセスに割り当てられたそれぞれの仮想メモリ領域の情報が格納されている．そして，この構造体から仮想メモリ領域の開始・終了アドレス，アクセス権限，フラグ，状態などを取得する．ファイルがマップされた仮想メモリ領域については，ファイルのオフセットも取得する．

© 1959 Information Processing Society of Japan

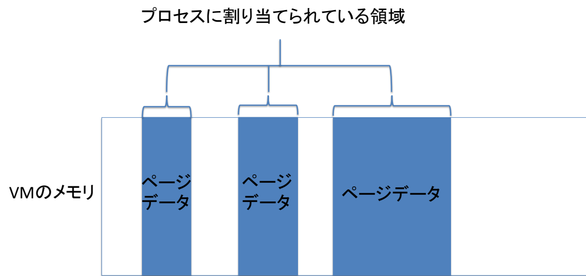

図 3 プロセスに割り当てられている仮想メモリ領域

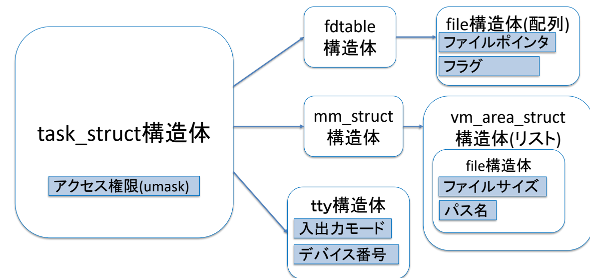

図 4 ファイル情報が格納された構造体

ページマップの情報として，仮想メモリ領域の開始アドレスと実際に割り当てられている物理ページ数の情報を保存する．開始アドレスの情報は上記で保存した仮想メモリ領域の情報と重複するが，メモリページのデータの保存と対応させるために別途，保存する．仮想メモリ領域のすべてのページに物理ページが割り当てられているわけではないため，vm_area_struct 構造体から取得した仮想メモリ領域の開始アドレスから順番に物理ページの割り当ての有無を調べる．そのために，mm_struct 構造体から取得したプロセスのページテーブルを参照し，ページテーブルエントリのページ番号が 0 の場合には保存の対象としない．また，匿名メモリページでない場合にも保存の対象としない．そして，物理ページが割り当てられており，かつ，連続している領域を 1024 ページ以下になるように分割して保存する．

ページのデータとして，プロセスに割り当てられており，かつ，ファイルがマッピングされていないページの内容を保存する．そのために，ページマップの情報の保存処理において存在することが分かったページについて，そのページの 4KB のデータを取得する．データの保存はページマップに保存した仮想メモリ領域の順番で行う．

4.2 ファイル情報の保存

プロセスがオープンしているファイルについて，ファイルディスクリプタに関する情報を保存する．まず，task_struct 構造体からポインタをたどって files_struct 構造体を見つけ，さらに fdtable 構造体を見つける．これらの構造体はプロセスによってオープンされたすべてのファイルの情報を管理している．fdtable 構造体はファイルの情報が格納されている file 構造体の配列を管理しており，そのインデックスからファイルディスクリプタ番号を，file 構造体からファイルの種類，ファイルディスクリプタに関連づけられたフラグ，ファイルポインタを取得する．このファイルポインタと，file 構造体からたどれる inode 構造体から取得したデバイス番号と inode 番号を用いて ID を算出する．

さらに，それぞれのファイルの種類ごとに詳細な情報を保存する．プロセスにロードされた実行ファイルや共有ライブラリおよびメモリマップされた通常のファイルの場合，仮想メモリ領域を管理している vm_area_struct 構造体からメモリにマッピングされているファイルを管理する file 構造体を見つける．file 構造体から inode 構造体を見つけ，ファイルのサイズや詳細な種類を取得する．また，file 構造体に含まれる path 構造体から再帰的にディレクトリをたどることでファイルのパス名を取得する．

一方，標準入出力のような端末デバイス (tty) の場合，端末デバイスに関する詳細な情報を保存する．そのために，file 構造体から tty_struct 構造体を見つける．この構造体から POSIX termios で定義されている端末デバイスの入出力モード，制御モード，ローカルモード，ライン制御，特殊文字の情報，入出力速度を取得する．また，端末デバイスの状態や端末デバイスを制御するセッションやプロセスグループの ID を取得する．さらに，tty_struct 構造体から tty_driver 構造体を見つけ，この構造体から取得したメジャー番号とマイナー番号を使って端末デバイスのデバイス番号を算出する．また，端末の幅と高さも取得する．さらに，file 構造体に含まれる inode 構造体からデバイス番号，ユーザ ID，グループ ID を取得する．

ファイルシステムの情報として，新規ファイルや新規ディレクトリの作成時に設定されるアクセス権 (umask) を保存する．そのために，task_struct 構造体から fs_struct 構造体を見つける．この構造体にはカレントディレクトリなどの情報が格納されている．この構造体から umask の値を取得する．

4.3 スレッド情報の保存

プロセスの実行状態として，スレッドの状態を保存する．レジスタ情報は CPU 依存であるため，現在のところ x86-64 にのみ対応している．停止中のプロセスのレジスタはプロセスに割り当てられたカーネルスタックに退避されているため，task_struct 構造体からカーネルスタックのアドレスを取得し，メモリからレジスタの値を取得する．FSBASE レジスタと GSBASE レジスタについては，task_struct 構造体からスレッド情報が格納された thread_struct 構造体を見つけ，格納されている値を直接取得するか格納されて

© 1959 Information Processing Society of Japan

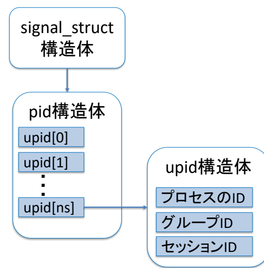

図 5　pid 構造体と upid 構造体の関係

いる値から計算する．また，XMM レジスタやセグメントディスクリプタなどの値についてもこの構造体から取得する．

4.4　時刻情報の保存

時刻情報として，モノトニック時刻とカーネルが起動してからの時間（ブートタイム）を保存する．モノトニック時刻は tk_core 変数が指す timekeeper 構造体から実時間（ウォールクロック）を取得し，同様に取得したウォールクロックからのオフセットを加えて計算する．ブートタイムは timekeeper 構造体からモノトニック時刻を取得し，モノトニック時刻からのオフセットを加えて計算する．それぞれの時刻には前回のウォールクロックの更新からの差分を加える必要がある．しかし，そのためには RDTSC 命令を実行してタイムスタンプカウンタの値を取得する必要があり，VM 外では取得することができない．現在のところ，この差分を 0 としているため，保存する時刻にはわずかな誤差が含まれる．

4.5　プロセス木の情報の保存

プロセス木の情報として，対象プロセスの ID，そのプロセスの親プロセスの ID，グループ ID，セッション ID，対象プロセスが持つスレッド数，各スレッドのスレッド ID を保存する．まず，task_struct 構造体からポインタをたどって signal_struct 構造体を見つける．この構造体にはシグナルに関連して使われるプロセスの情報が格納されている．次に，signal_struct 構造体からポインタをたどって pid 構造体を見つける．この構造体にはプロセス ID，グループ ID，セッション ID が格納されているが，それぞれの ID は階層化されたプロセスの名前空間ごとに割り当てられている．そのため，名前空間の階層を考慮して ID が格納された upid 構造体を見つける．pid 構造体と ID が格納された upid 構造体の関係を**図 5** に示す．現在のところ，1 つのスレッドのみを持つプロセスを想定し，プロセス ID と同じ値をスレッド ID として保存している．

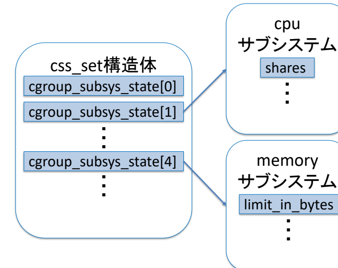

図 6　Cgroup のサブシステムの構成

4.6　Cgroup の情報の保存

Cgroup の情報として，サブシステムの名前とパスおよび，パラメータとその値を保存する．Cgroup はプロセスのグループに対してリソースの割り当てや制限を行う機能であり，コンテナでプロセスを他のコンテナから隔離するのに用いられる．Cgroup にはメモリや CPU などの各リソースごとにサブシステムが存在し，いくつかのパラメータを持つ．例えば，メモリサブシステムの limit_in_bytes パラメータには利用可能なメモリの最大値が設定されている．Cgroup のサブシステムの構成を**図 6** に示す．

OVmigrate は状態を保存する際にまず，task_struct 構造体から css_set 構造体を見つけ，kernfs のノード情報を保持している cgroup 構造体を見つける．kernfs のノードからルートに到達するまで探索することでパスを取得する．パラメータ名はサブシステムごとに用意された cgroup_subsys 構造体から取得する．例えば，メモリサブシステムの場合は memory_cgrp_subsys 変数が指す構造体がパラメータの一覧を保持している．パラメータの値は css_set 構造体からたどれる cgroup_subsys_state 構造体から取得する．

4.7　VM イントロスペクション

OVmigrate は KVmonitor [2] を用いて VM イントロスペクションを行う．KVMonitor を用いた OVmigrate のシステム構成を**図 7** に示す．まず，VM に割り当てる物理メモリをホスト上のファイルとして作成し，VM とマイグレーション機構の両方にマッピングして共有する．次に，QEMU と通信して仮想 CPU の CR3 レジスタに格納されているページテーブルのアドレスを取得する．このページテーブルを用いて，プロセスの状態が格納されているカーネルデータの仮想アドレスを物理アドレスに変換し，VM 内の情報を取得する．

VM のメモリ上のカーネルデータの解析を容易にするために，OVmigrate は LLView フレームワーク [1] を用いる．LLView では，Linux カーネルのヘッダファイルで定義されているカーネルの構造体やグローバル変数などを用いて，プロセスの状態を取得するプログラムを記述することができる．このプログラムをコンパイルして生成された LLVM

ⓒ 1959 Information Processing Society of Japan

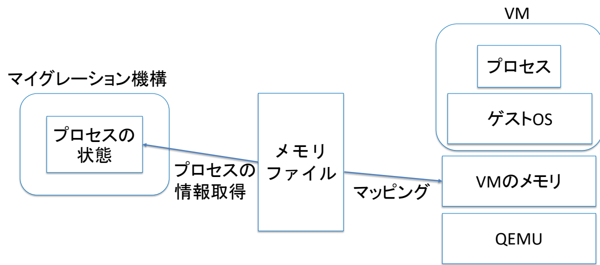

図 7 VM 外からのメモリデータの取得

```
message pagemap_entry {
        uint64 vaddr    = 1;
        uint32 nr_pages = 2;
        uint32 flags    = 3;
}
```

図 8 proto ファイルによるメッセージの定義の例

の中間表現を変換することで，必要に応じて VM のメモリにアクセスするコードを埋め込む．

4.8 プロトコルバッファを用いた状態保存

OVmigrate は CRIU 3.16 によって保存されているものと同じプロセスの状態を保存する．CRIU はプロトコルバッファ [11] を用いてプロセスの状態を保存しており，保存するプロセスの状態が proto ファイルで定義されている．OVmigrate でもこの proto ファイルを利用し，C 言語用のプロトコルバッファ [12] を用いてプロセスの状態を保存する．proto ファイルでは**図 8** のように状態の種類ごとにメッセージが定義されており，その中のフィールドで個別の状態が定義されている．

4.9 疑似的なシグナルの送信

OVmigrate は VMMfas [8] を用いて VM 内のプロセスに疑似的にシグナルを送信することにより，VM 外からプロセスの一時停止・終了を行う．task_struct 構造体から sigpending 構造体を見つけ，その中のシグナルビットマップの対応するビットを 1 にセットする．次に，task_struct 構造体から thread_info 構造体を見つけ，未処理フラグをセットする．これだけでは停止しているプロセスに対してはシグナルが配送されないため，疑似的なプロセススケジューリングを行ってプロセスが実行されるようにする．task_struct 構造体から sched_entity 構造体を見つけ，CFS スケジューラが用いる cfs_rq 構造体の中の赤黒木に追加する．また，task_struct 構造体の中のプロセスの状態を実行可能状態に変更する．

5. 実験

OVmigrate を用いて VM 外でプロセスの状態を正常に

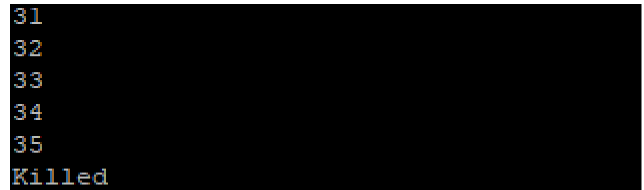
```
31
32
33
34
35
Killed
```

図 9 OVmigrate による状態保存時の表示

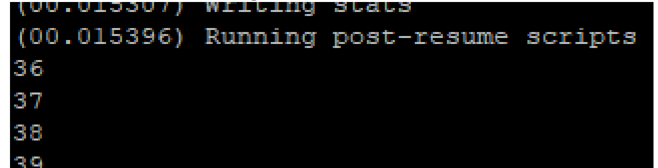
```
(00.015307) Writing stats
(00.015396) Running post-resume scripts
36
37
38
39
```

図 10 CRIU による状態復元時の表示

保存することができることを確認する実験を行った．また，VM 外でプロセスの状態を保存するのにかかる時間を測定した．比較として，VM 内で CRIU を用いてプロセスの状態を保存するのにかかる時間も測定した．さらに，プロセスの状態を保存する処理が他のプロセスに与える影響を測定した．ホストとして，Intel Core i7-10700 の CPU，64GB のメモリ，2TB の SATA HDD を搭載したマシンを用いた．このホストでは OS として Linux 5.4，仮想化ソフトウェアとして QEMU-KVM 4.2.0 を動作させた．VM には仮想 CPU を 2 個，メモリを 50GB 割り当て，ゲスト OS として Linux 5.4 を動作させた．

5.1 動作確認

VM 内で 1 秒ごとにカウンタ値を増加させながら表示するプログラムを実行し，OVmigrate を用いて VM 外でそのプロセスのすべての状態を保存した．その後，保存された状態を VM 内に転送し，CRIU を用いてプロセスを復元した．OVmigrate による状態保存時のプロセスの表示を**図 9** に示す．状態保存を開始するとカウンタ値が止まり，保存が完了するとプロセスが強制終了させられていることが分かる．また，CRIU による状態復元時の表示を**図 10** に示す．OVmigrate によって保存された状態を用いてもプロセスの復元処理が正しく実行され，状態が保存された時点からプロセスがカウントを再開したことが分かる．

5.2 状態保存性能

OVmigrate の状態保存性能を CRIU と比較するために，VM 内で動作するプロセスの状態を保存するのにかかる時間を測定した．プロセスメモリの保存に最も時間がかかるため，プロセスが使用するメモリ量を 0〜40GB に変えて測定を行った．測定結果は**図 11** のようになり，20GB のメモリを使用する場合，OVmigrate の性能は CRIU と同程度になった．プロセスが使用するメモリ量を 40GB に増やすと OVmigrate の方が 31%性能がよくなった．これは VM 内のページキャッシュがホストより少なくなり，CRIU によ

© 1959 Information Processing Society of Japan

表 1 各状態の保存時間の内訳

イメージファイル名	保存時間
mm	0.000601352
pages	91.4215943
fdinfo	0.000041351
file	0.000147304
tty	0.000219057
fs	0.000702407
inventory	0.000081407
ids	0.000021045
core	0.000195387
time	0.00003088
pstree	0.000092516
cgroup	0.04295691
seccomp	0.00001895

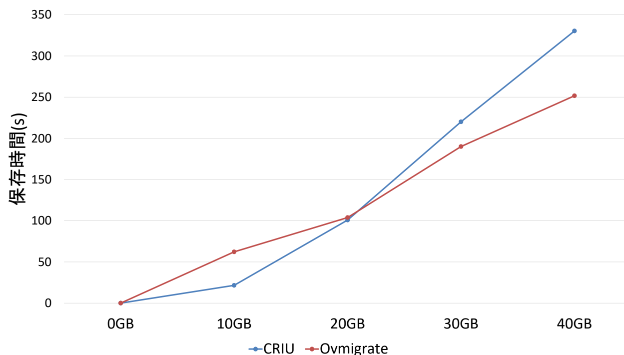

図 11 プロセスの状態保存時間の比較

るディスクアクセスが増えたためと考えられる．一方，プロセスが使用するメモリ量を 10GB に減らすと OVmigrate の方が 65%性能が低下した．これは VM 内のページキャッシュが多くなり，CRIU によるディスクアクセスが発生しなくなったためと考えられる．プロセスが最小限のメモリしか使わなかった場合，CRIU は 10 ミリ秒前後であったが，OVmigrate は 30〜70 ミリ秒であった．この差は VMI のオーバーヘッドのためだと考えられる．イメージファイルごとの保存時間の内訳を表 1 に示す。

次に，VM の負荷が状態保存性能に与える影響を調べるために，VM に負荷をかけた場合とかけなかった場合とでプロセスの保存時間を測定した．VM に負荷をかける場合には VM 内で stress-ng を実行し，CPU に負荷をかけた．プロセスが使用するメモリ量は 20GB とした．測定結果は**図 12** のようになり，OVmigrate は VM の負荷が高い場合に CRIU より 42%高速にプロセスの状態を保存できることが分かった．これは VM 外で動作する OVmigrate が VM の負荷の影響を 13%しか受けなかったためである．また，OVmigrate は VM の負荷の有無に関わらずより安定した時間で状態保存を行えることがわかった．

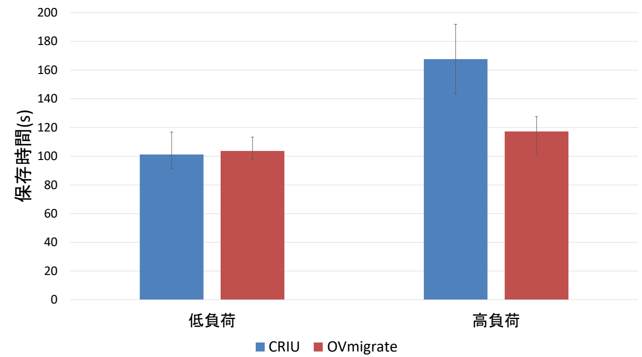

図 12 VM の負荷の状態保存時間への影響

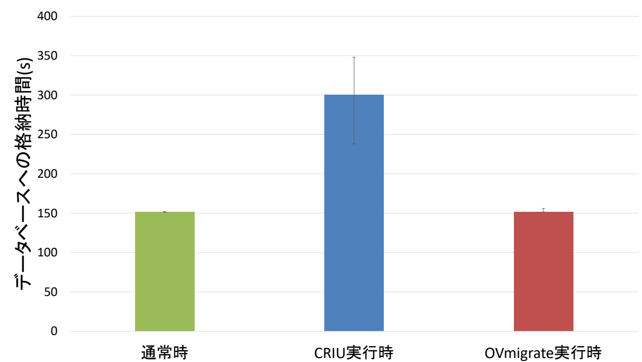

図 13 状態保存処理による他のプロセスへの影響

5.3 状態保存処理の影響

OVmigrate の状態保存処理が VM 内の他のプロセスの性能に及ぼす影響を調べるために，状態保存中に memcached に 1GB のデータを格納するのにかかる時間を測定した．比較のために，VM 内で CRIU を用いて状態保存を行っている時と状態保存処理を行っていない時についても測定した．測定結果は**図 13** のようになり，OVmigrate が VM 外で状態保存を行っている間，memcached の性能は低下しなかった．一方，VM 内で CRIU が状態保存を行っている間は，memcached の性能が大幅に低下することが分かった．これは CRIU による状態保存処理の負荷が VM に大きな影響を与えたためである．

6. 関連研究

Portkey [7] はネットワーク転送を最適化することにより，VM 内のコンテナを効率よくマイグレーションすることを可能にしている．CRIU が保存したコンテナの状態を転送する際には Portkey のカーネルモジュールを呼び出し，カーネルモジュールはゲスト OS でのネットワーク処理をバイパスしてハイパーバイザを呼び出す．ハイパーバイザが移送先ホストへコンテナの状態を転送し，CRIU はカーネルモジュール経由でハイパーバイザから状態を受け取る．ハイパーバイザで転送処理を行うことでコンテナマイグレーション中の CPU 使用率を抑えることはできているが，マイグレーションは高速化されていない．OVmigrate

© 1959 Information Processing Society of Japan

はコンテナの状態を完全に VM 外で転送するため，さらに CPU 使用率を削減し，マイグレーションの高速化も実現することができると考えられる．

mWarp [13] は同一ホストの VM 間でメモリを再配置することにより，コンテナマイグレーションにおいて時間がかかるプロセスメモリのコピーや転送を行わないようにする．mWarp では，移送元 VM 内の CRIU はシステムコール経由でハイパーコールを呼び出し，コンテナ内のプロセスのメモリ情報をハイパーバイザに通知する．移送先 VM 内の CRIU がハイパーバイザを呼び出すと，ハイパーバイザが移送元 VM のメモリを移送先 VM へ再マッピングし，転送を完了させる．しかし，mWarp では同じホスト内の VM 間でのみコンテナマイグレーションが実行可能である．OVmigrate でも移送元と移送先の VM が同一ホスト上にあれば，同様の最適化が可能であると考えられる．

VMBeam [14] はホスト VM 内で動作しているゲスト VM を対象として，mWarp と同様にゼロコピーでのマイグレーションを可能としている．VMbeam では，移送元と移送先のホスト VM 内のマイグレーション機構がハイパーバイザを呼び出し，移送元ホスト VM のメモリを移送先ホスト VM とスワップすることでゲスト VM のメモリ転送を高速に実行する．VMBeam も同一ホスト内でのみ利用可能な最適化である．

Sledge [15] は Docker コンテナの効率のよいライブマイグレーションを実現している．Sledge は移送元ホストのコンテナが使っている階層的なイメージの中の冗長なレイヤを転送しない．また，CRIU のインクリメンタル・チェックポイントを利用してメモリの差分だけを繰り返し転送する．さらに，移送先ホストで時間のかかる Docker デーモンの再ロードを行わず，管理コンテキストのロードだけを行う．これらの最適化技術は OVmigrate でも利用できる可能性がある．

7. まとめ

本稿では，VM 内で動作しているコンテナの軽量なマイグレーションを実現するために，VM 外でコンテナの状態を保存するシステム OVmigrate を提案した．OVmigrate を用いることで，VM 内の負荷や VM による仮想化がコンテナマイグレーションに及ぼす影響および，コンテナマイグレーションがコンテナ性能に及ぼす影響を小さくすることができる．OVmigrate は VM イントロスペクションを用いて VM のメモリを解析することで，VM 内のコンテナの状態を VM 外で保存する．実験により，OS の最小限の機能だけを用いるプロセスの状態を正しく保存できていることが確認できた．また，OVmigrate は VM 内で CRIU を用いる場合より最大 1.3 倍高速にプロセスの状態を保存することができ，保存時間のばらつきも抑えられることが分かった．

今後の課題は，コンテナマイグレーションに必要なすべての状態を保存できるようにすることである．対応する必要がある状態としては，OS の様々な機能を用いるプロセスの状態やコンテナエンジンが管理しているコンテナの状態がある．また，VM 外でのコンテナの状態復元機構 [10] と組み合わせることで，コンテナマイグレーションを完全に VM 外で行えるようにすることも計画している．

謝辞 本研究の一部は，JST，CREST，JPMJCR21M4 の支援を受けたものである．また，本研究成果の一部は，国立研究開発法人情報通信研究機構（NICT）の委託研究（JPJ012368C05501）により得られたものである．

参考文献

[1] Ozaki, Y., Kanamoto, S., Yamamoto, H. and Kourai, K.: Detecting System Failures with GPUs and LLVM, *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pp. 47–53 (2019).
[2] Nakamura, K. and Kourai, K.: Efficient VM Introspection in KVM and Performance Comparison with Xen, *IEEE 20th Pacific Rim International Symposium on Dependable Computing*, pp. 192–202 (2014).
[3] Team, O.: CRIU, `https://criu.org/Main_Page`.
[4] Wood, T., Shenoy, P., Venkataramani, A. and Yousif, M.: Black-box and Gray-box Strategies for Virtual Machine Migration, *4th USENIX Symposium on Networked Systems Design & Implementation* (2007).
[5] Ruprecht, A., Jones, D., Shiraev, D., Harmon, G., Spivak, M., Krebs, M., Baker-Harvey, M. and Sanderson, T.: VM Live Migration At Scale, *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 45–56 (2018).
[6] VMware, Inc.: VMware tanzu, `https://tanzu.vmware.com/application-platform`.
[7] Prakash, C., Mishra, D., Kulkarni, P. and Bellur, U.: Portkey: Hypervisor-Assisted Container Migration in Nested Cloud Environments, *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 3–17 (2022).
[8] Kiumura, K. and Kourai, K.: Xfas: Fault Recovery by Externally Controlling OS Behavior, In Proceedings of the 16th IEEE/ACM International Conference on Utility and Cloud Computing (2023).
[9] 朝倉優輝, 光来健一：VM 外で実行可能なコンテナの状態保存機構, 2022 年並列／分散／協調処理に関するサマー・ワークショップ (2022).
[10] 木本翔太, 光来健一：コンテナマイグレーションを VM 外で実行するための状態復元機構, 2024 年並列／分散／協調処理に関するサマー・ワークショップ (2024).
[11] Google: Protocol buffers, `https://developers.google.com/protocol-buffers`.
[12] Benson, D.: Protocol Buffers implementation in C, `https://github.com/protobuf-c/protobuf-c`.
[13] Sinha, P., Doddamani, S., Lu, H. and Gopalan, K.: mWarp: Accelerating Intra-Host Live Container Migration via Memory Warping, *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops* (2019).
[14] Ooba, H. and Kourai, K.: Zero-copy Migration for Lightweight Software Rejuvenation of Virtualized Systems, *Proceedings of the 6th Asia-Pacific Workshop on*

© 1959 Information Processing Society of Japan

Systems (2015).
[15] Xu, B., Wu, S., Xiao, J., Jin, H., Shi, G., Rao, J., Yi, L. and Jiang, J.: Sledge : Towards Efficient Live Migration of Docker Containers, *IEEE International Conference on Cloud Computing*, pp. 321–328 (2020).

© 1959 Information Processing Society of Japan