Vol.2021-EMB-58 No.8
2021/11/18

コンテナ型仮想化における低レベルランタイムの性能評価

西村惇[1,a)]　李奕驍[1]　松下瑛佑[1]　松原豊[1]　高田広章[1]

概要：コンテナ型仮想化は OS レベルで仮想化を行うため，VM 型仮想化に比べて資源効率や処理速度で優れており，自動車や航空機などの組込みシステムで利用が検討されている．コンテナ作成時に使用する低レベルランタイムは性能やセキュリティを大きく左右するため，詳細かつ網羅的な評価が求められているが，従来の研究では対象ランタイムが少なく，組込みシステムで使用される ARM 環境での評価が不足している．そこで本研究では対象ランタイムを runc，crun，gVisor，Kata Containers，Inclavare Containers の 5 つとし，x86_64 と ARM の 2 つの環境で，コンテナ管理，CPU 実行，メモリアクセス，ファイル I/O，システムコール，ネットワークの 6 項目について性能評価を行った．

キーワード：コンテナ型仮想化，Docker，TouchStone，runc，crun，gVisor，Kata Containers，Inclavare Containers

Performance Evaluation of Low-Level Runtimes in container-based virtualization

ATSUSHI NISHIMURA[1,a)]　LI YIXIAO[1]　EISUKE MATSUSHITA[1]
YUTAKA MATSUBARA[1]　HIROAKI TAKADA[1]

1. はじめに

近年, VM 型仮想化に代わる技術としてコンテナ型仮想化が注目を集めている．ハードウェアレベルで仮想化を行う VM 型仮想化とは異なり，コンテナ型仮想化は OS レベルで仮想化を行う．そのため資源効率や処理速度において優れており，自動車や航空機などの組込みシステムの分野でも利用が検討されている．資源効率や処理速度の観点では，限定された機能のみを実装したカーネルを使用するユニカーネル技術も考えられる．しかし，組込みシステムでは既存の組込みアプリをできる限り修正せずに動作させる必要がある．その点で，現時点ではコンテナ技術の方が有用であるため，本研究ではコンテナ技術を対象とする．

コンテナ環境を実現する際に使用されるコンテナエンジンはコマンドラインインターフェース，コンテナ管理プロセス，低レベルランタイムの主に 3 つの要素で構成されている．とくに低レベルランタイムはコンテナの作成を担当し，コンテナの性能やセキュリティを大きく左右するため，詳細かつ網羅的な評価が求められている．しかし従来の研究[1][2][3]では対象ランタイムを runc と gVisor, runc と Kata Containers のように 2 種類程度に限定しながら評価を行っている．そのため，コンテナの性能に関して詳細な評価は行っていても，対象ランタイムの数が少なく，網羅的な評価は行っていない．さらに従来の研究[1][2][3][4]では実行環境として x86_64 プロセッサを使用しており，組込みシステムの分野で使用される ARM プロセッサでの評価を行っていない．そのため ARM 環境での詳細かつ網羅的なランタイムの評価は現時点で存在しない．これらの課題を解決するべく，本研究では一般的に PC などで利用される x86_64 プロセッサと，組込みシステムで利用される ARM プロセッサの 2 つの環境を使用し，様々な低レベルランタイムの性能評価と比較を行った．対象ランタイムは runc，crun，gVisor，Kata Containers，Inclavare Containers の 5 つとした．他に有名な低レベルランタイムとして，コンテナ技術とユニカーネル技術を組み合わせた Nabla Containers が存在する．しかし，Nabla Containers では性能評価で使用するベンチマークを専用に作り直す必要があり，手間がかかる．さらに 2 年以上更新されていない．そのため，本研究では対象としない．

性能評価ではコンテナ管理, CPU 実行, メモリアクセス，ファイル I/O，システムコール，ネットワークの 6 つの項目についてベンチマークを実行して評価を行った．

2. 基本知識

2.1 コンテナエンジン

コンテナエンジンはコマンドラインインターフェース（CLI），コンテナ管理プロセス，低レベルランタイムの主に 3 つの要素で構成されており，利用者から受け取った指示に従い，コンテナの作成や管理を行う．CLI は利用者にコンテナエンジンの操作を行うコマンドを提供する．コンテナ管理プロセスは CLI から指示を受け取り，コンテナやコンテナの素となるコンテナイメージ，ネットワークの管理を行う．低レベルランタイムは namespace，cgroup などの Linux カーネルの機能を組み合わせてコンテナを作成する．

1 名古屋大学
Nagoya University, Nagoya, Aichi 464-8601, Japan
a) nishimura.atsushi@d.mbox.nagoya-u.ac.jp

実際にコンテナの作成を担当するため，使用する低レベルランタイムによってコンテナの性能やセキュリティが大きく異なる．第 3 節にて，本研究で対象とする各低レベルランタイムの特徴を説明する．

コンテナエンジンの中で最も有名なのが Docker[5]である．基本的に Docker も上述した 3 つの要素で成り立っているが，コンテナ管理プロセスはさらに dockerd と高レベルランタイムに分割される．dockerd は CLI からの指示の受け取りとコンテナイメージやネットワークの管理を担当する．一方で高レベルランタイムはコンテナの管理を担当し，低レベルランタイムのバイナリを実行して，コンテナの作成を行う．Docker でデフォルトの高レベルランタイムは containerd であるが，他にも CRI-O などの高レベルランタイムが存在する．高レベルランタイムは低レベルランタイムとまとめてコンテナランタイムと称される場合がある．

2.2 TouchStone

低レベルランタイムはコンテナの性能やセキュリティを大きく左右するため評価の需要が高まっているが，各項目についてベンチマークをわざわざ用意して評価を行うのは手間と時間を要する．この問題を解決するべく Lennart らによって TouchStone[6]と呼ばれるコンテナランタイム評価ツールが開発された．TouchStone は指定されたコマンドを入力するだけで，利用者が自らベンチマークを用意せずとも，9 つの評価項目からランタイムの性能を評価し比較できる．

TouchStone は高レベルランタイムと低レベルランタイムの各組み合わせについてコンテナを作成し，その中で指定したベンチマークを実行する．実行結果は html と json 形式で出力され，html ファイルではグラフによって各ランタイムの差を一目で確認できる．さらに GitHub 上で MIT ライセンスにて公開されており，自由に改変してベンチマークの追加や変更ができる．TouchStone には上述した利点がある一方で，更新が 2019 年 7 月以降行われておらず，最新のランタイムに対応できない場合がある．さらに知名度が低く資料が少ないため，初学者には導入が難しい．TouchStone を利用する際にはこれらの点に注意する必要がある．

Lennart らの研究[1]では TouchStone を使用して低レベルランタイムの性能評価を行っている．ただし，対象ランタイムが runc と gVisor の 2 種類のみで少なく，コンテナランタイムを比較する際に重要なネットワークやシステムコールなどの項目を評価していない．本研究では TouchStone の対象ランタイムとして crun，Kata Containers，Inclavare Containers，ベンチマークとしてシステムコールの項目を新たに追加して，これらの問題を解決している．

3. 低レベルランタイム

3.1 runc

本研究で対象のランタイムの中で最初に開発されたのが runc[7]である．runc は元々 Docker の一部であったが，コンテナ型仮想化の標準化の流れにより，2017 年に低レベルランタイムとして分割された．

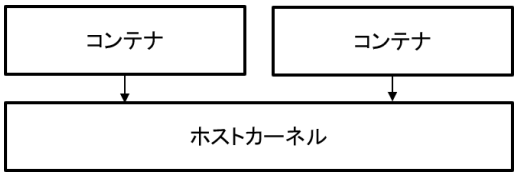

図 1　runc のシステムコール処理

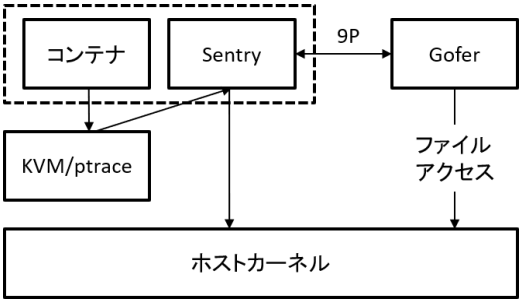

図 2　gVisor のシステムコール処理

runc のシステムコール処理の様子を図 1 に示す．runc はホストカーネルを共有する形式でコンテナを作成する．そして，図中の矢印が示すようにコンテナが発行したシステムコールをホストカーネルで処理する．そのため，悪意を持ったコンテナがホストカーネルを介して他のコンテナに影響を及ぼしかねない．具体的な事例として runc に関する脆弱性（CVE-2019-5736）が公開されている[8]．この脆弱性を悪用したコンテナは，ホスト上に存在する runc のバイナリを上書きし，ホスト上でコードを実行できる root 権限を獲得する恐れがある．この脆弱性を修正するパッチは同日に配布されたが，runc は他のランタイムと異なり，セキュリティを強化する要素がなく，安全な低レベルランタイムとは言えない．

3.2 crun

crun[9]は runc よりも優れた性能をもつ低レベルランタイムを実現するべく，2019 年に RedHat 社によって開発された．本研究で対象とする他のランタイムは Go 言語を使用して開発されている中，crun は C 言語を使用して開発されているため高速に動作し，メモリ使用量が少ない．crun のアーキテクチャとセキュリティに関する情報は現在 GitHub 上で公開されていない．

3.3 gVisor

gVisor[10]は runc の脆弱性を改善するべく，セキュリティを重視した低レベルランタイムとして，2018 年に Google 社によって開発された．gVisor は低レベルランタイムである runsc を含んだプロジェクトの名前であるが，本研究では後述の Sentry や Gofer など，runsc 以外の要素も考慮して評価を行っているため，runc の比較対象を gVisor としている．

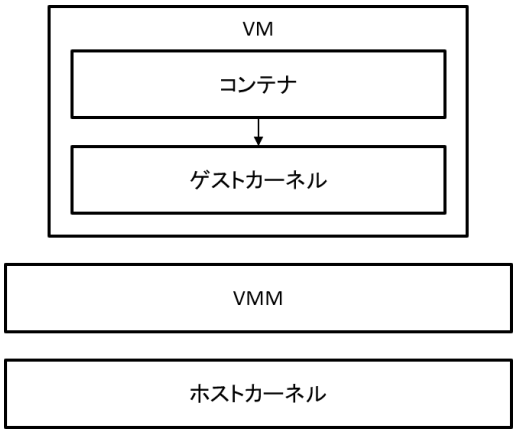

図 3　Kata Containers のシステムコール処理

gVisor のシステムコール処理の様子を図 2 に示す．gVisor は runsc によってコンテナを作成し，ユーザー空間のカーネルである Sentry とまとめてサンドボックス空間を形成する．図中の矢印が示すようにコンテナが発行したシステムコールは直接ホストカーネルに渡さず, Sentry を通して処理する．その際に ptrace によるシステムコール追跡のオーバーヘッドが大きく，runc よりも性能が劣化する．ただしオプションにて ptrace を使用しない KVM モード[11]を提供している．ファイル操作は Sentry が 9P プロトコルによって，仮想ファイルシステムである Gofer を呼び出して行う. コンテナとホストカーネルの間に Sentry や Gofer などの要素を介することで，ホストカーネルへの侵害を軽減できるため，セキュリティ面で優れている．

3.4　Kata Containers

Kata Containers[12]は gVisor 同様に runc の脆弱性を改善するべく 2017 年に OpenStack Foundation によって開発された．Kata Containers も低レベルランタイムである kata-runtime を含んだプロジェクトの名前であるが, gVisor と同様の理由により本研究では Kata Containers としている．

Kata Containers のシステムコール処理の様子を図 3 に示す. Kata Containers は kata-runtime によって VM 内部にコンテナを作成する．図中の矢印が示すように，コンテナが発行したシステムコールはホストカーネルではなくゲストカーネルで処理する．また，ゲストカーネルはセキュリティに関する設定が少なくオーバーヘッドが小さいため，ホストカーネルよりも処理が高速である. VM 内部で処理を完結させるため，悪意を持ったコンテナがホストカーネルを侵害する心配がなく，セキュリティ面で優れている．

3.5　Inclavare Containers

Inclavare Containers[13]は安全なコンテナの実現を目的として, 2020 年 5 月に Alibaba 社によって開発された. Inclavare Containers も低レベルランタイムである rune を含んだプロジェクトの名前であるが, gVisor や Kata Containers と同様の理由により本研究では Inclavare Containers としている．

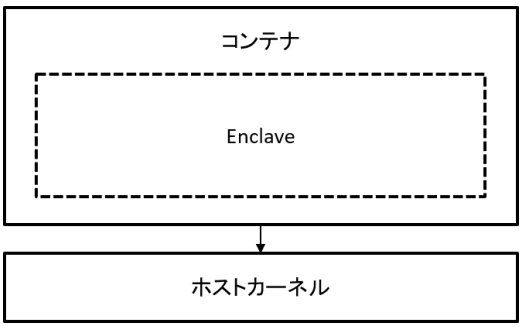

図 4　Inclavare Containers のシステムコール処理

Inclavare Containers のシステムコール処理の様子を図 4 に示す. Inclavare Containers は rune によってコンテナを作成する．そして，ハードウェア志向の TEE 機能を用いたメモリ暗号化により，重要なデータやコードをコンテナ外部から隔離している．コンテナが発行したシステムコールは図中の矢印が示すようにホストカーネルで処理する．gVisor や Kata Containers のようにホストカーネルを侵害するリスクを減らす手法ではなく，Enclave 領域にデータやコードを格納してコンテナ内部から脆弱性を排除する手法をとることで，セキュリティ面で優れている．

4.　性能評価

4.1　評価方法

x86_64 プロセッサと ARM プロセッサの 2 つの環境で評価を行った．実行環境の詳細をそれぞれ表 1，2 に示す．実行マシンとして，x86_64 環境では HP 社のノート PC，ARM 環境では LABISTS 社の Raspberry Pi 4 を使用した．メモリとストレージの大きさは 8GB と 512GB と両環境で同じだが, x86_64 環境では PC 内蔵の SSD，ARM 環境では microSD を使用した．OS は x86_64 環境と ARM 環境の両方で Ubuntu を使用した．高レベルランタイムは CRI-O を使用した．理由としては 2.2 節で述べた通り, TouchStone が一部のランタイムに対応できない場合があり，本研究においてもログ取得時のエラーで containerd を使用できなかったためである．ただし ARM 環境では x86_64 環境と同じバージョンを使用できなかったため，古いバージョンの CRI-O を使用した．

対象とする低レベルランタイムとバージョンを表 1，2 に示す．ランタイムは大きく分けて，デファクトスタンダードである runc, 性能志向の crun, セキュリティ志向の gVisor, Kata Containers，Inclavare Containers に分類される．Inclavare Containers は ARM 環境には対応しておらず，x86_64 環境のみ評価を行った．gVisor のみ TouchStone 開発時期以降のバージョンを TouchStone で使用できなかったため，x86_64 環境では古いバージョンを使用した．ARM 環境では該当バージョンが存在しなかったため，最新バージョンを使用した．

評価項目はコンテナ管理，CPU 実行，メモリアクセス，ファイル I/O, システムコール, ネットワークの 6 つとした．

表 1　x86_64 実行環境

実行マシン	HP EliteBook Folio G1
CPU	Intel® Core™ m5-6Y54
コア数	2
メモリ	8GB
ストレージ	SSD 512GB
OS	Ubuntu18.04
Linux Kernel	5.4.0-74-generic
Docker	19.03.13
CNI	0.3.1
CRI-O	1.17.4
runc	1.0.0-rc92+dev
crun	0.12.1.455-3fcf
gVisor	release-20190529.1
Kata Containers	1.12.0-alpha1
Inclavare Containers	1.0.0-rc10+dev

表 2　ARM 実行環境

実行マシン	Raspberry Pi 4 Model B
CPU	Cortex-A72
コア数	4
メモリ	8GB
ストレージ	microSD 512GB
OS	Ubuntu20.04
Linux Kernel	5.4.0-1036-raspi
Docker	20.10.5
CNI	0.8.7
CRI-O	1.15.4
runc	1.0.0-rc93
crun	0.18.42-a793
gVisor	release-20210315.0
Kata Containers	1.11.2

表 3　x86_64 環境における測定結果（デフォルト設定）

測定項目		runc	crun	gVisor	Kata Containers	Inclavare Containers
コンテナ管理	作成（秒）	0.5132	0.4598	0.4271	2.0451	0.5834
	実行（秒）	0.0308	0.0454	0.0559	0.1626	0.0459
	破棄（秒）	0.4252	0.4685	0.4785	1.0010	0.4713
CPU 実行	実行時間（秒）	40.2966	40.4075	40.7683	41.3706	40.4075
メモリアクセス	アクセス時間（秒）	20.4192	20.4605	21.0100	21.3678	20.4703
ファイル I/O	SeqRead（秒）	0.5987	0.6212	7.6309	27.0362	0.6277
	RndRead（秒）	0.0520	0.0533	0.7113	2.0767	0.0547
	SeqWrite（秒）	8.8933	8.9155	13.7048	35.5674	8.8925
	RndWrite（秒）	12.0627	12.0548	17.1254	15.1615	12.0698
システムコール	Score	164.4	164.7	4.0	164.7	164.8
ネットワーク	Bandwidth（Gbits/秒）	31.26	32.09	7.45	21.08	31.57

表 4　ARM 環境における測定結果（デフォルト設定）

測定項目		runc	crun	gVisor	Kata Containers
コンテナ管理	作成（秒）	1.1083	1.1024		4.3405
	実行（秒）	0.0564	0.1341		0.6452
	破棄（秒）	1.5024	2.2360		5.1838
CPU 実行	実行時間（秒）	17.5674	17.5576	17.6661	17.5687
メモリアクセス	アクセス時間（秒）	20.9402	20.0844	21.6486	21.3990
ファイル I/O	SeqRead（秒）	1.7555	1.7737	18.4973	307.9154
	RndRead（秒）	0.1601	0.1616	1.4726	23.5222
	SeqWrite（秒）	66.9723	66.8767	73.6165	322.7174
	RndWrite（秒）	56.1584	56.0781	71.9587	92.0697
システムコール	Score	136.4	137.9	1.0	355.1
ネットワーク	Bandwidth（Gbits/秒）	6.32	6.16	1.11	2.78

各項目について，コンテナ内部でベンチマークを実行して評価を行った．また，本研究はあくまで各ランタイム間の傾向を調査することを目的としているため，x86_64 環境と ARM 環境での数値差については言及しない．

4.2　コンテナ管理

TouchStone を使ってコンテナの作成・実行・破棄を 10 回行い，平均時間を計測した．コンテナイメージは busybox[14] を使用した．

x86_64 環境と ARM 環境の実行結果を表 3, 4 に示す．ARM

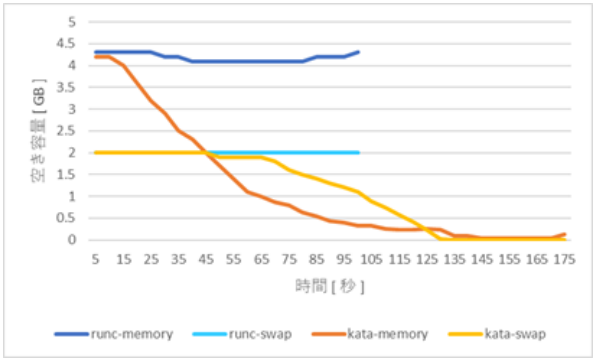

図 5　コンテナ 50 個作成・破棄時の空き容量の推移

環境については gVisor のライフサイクルを測定する際にコンテナの停止と破棄ができない現象が発生したため，表 4 に gVisor の実行結果を記載していない．runc, crun, gVisor, Inclavare Containers が全ての項目で時間差が小さく拮抗し，Kata Containers のみ他のランタイムに大きく劣ることが分かった．とくにコンテナ実行時に他のランタイムと大きく差が開き，x86_64 環境では最大で 5.3 倍，ARM 環境では最大で 11 倍の時間を要した．この結果から，各ランタイム間の傾向は x86_64 環境と ARM 環境の両方で共通することが分かった．Kata Containers の性能劣化はコンテナ作成時に VM を使用することが原因だと考えられる．

4.2.1　コンテナのメモリ使用量の計測

runc と Kata Containers を対象に追加実験を行った．x86_64 環境でコンテナを 50 個だけ作成・破棄する際に，free コマンドによってメモリ空き容量を計測した．

結果を図 5 に示す．runc は 100 秒程度でコンテナ 50 個の作成・破棄が完了した．その間，memory の空き容量はわずかに減少するが，swap は使用されず，メモリ全体に負担がかかっていないことが分かった．一方で，Kata Containers はコンテナ 50 個作成の途中で memory と swap の両方を使い切り，コンテナを作成できなくなった．

この結果について，Kata Containers ではコンテナを作成する際に VM を使用するため，他のランタイムよりもメモリ使用量が大きいことが原因だと考えられる．そのため，一度に大量のコンテナを作成する場合には，他のランタイムよりもホストのメモリが圧迫され，コンテナを作成できなくなることに注意しなければならない．

4.3　CPU 実行

TouchStone を使ってコンテナを作成し，その中で sysbench[15]の cpu オプションを実行した．cpu では指定数字以下の素数を計算する時間を計測する．本研究では関連研究[1]と同様に数字を 20000 に指定した．

x86_64 環境と ARM 環境の実行結果を表 3，4 に示す．ランタイム間の最大時間差は x86_64 環境で約 1.1 秒，ARM 環境で約 0.1 秒であり，全てのランタイムについて時間差が小さく拮抗することが分かった．この結果から，各ランタイム間の傾向は x86_64 環境と ARM 環境の両方で共通することが分かった．また，CPU を使用した処理を行う際に各ランタイムで特有のオーバーヘッドがないことが分かった．

4.4　メモリアクセス

TouchStone を使ってコンテナを作成し，その中で sysbench の memory オプションを実行した．memory ではメモリに対する連続書き込みにかかった時間を計測する．メモリ書き込みの合計サイズである memory-total-size と 1 回あたりの書き込みサイズである memory-block-size を指定する．書き込み回数は memory-total-size を memory-block-size で割った値となる．本研究では，関連研究[1]と同様に memory-total-size を 100GB，memory-block-size を 1MB に指定した．

x86_64 環境と ARM 環境の実行結果を表 3，4 に示す．ランタイム間の最大時間差が x86_64 環境で約 0.9 秒，ARM 環境で約 1.6 秒であり，全てのランタイムについて時間差が小さく拮抗することが分かった．この結果から，各ランタイム間の傾向は x86_64 環境と ARM 環境の両方で共通することが分かった．また，メモリアクセスを行う際に各ランタイムで特有のオーバーヘッドがないことが分かった．

4.5　ファイル I/O

TouchStone を使ってコンテナを作成し，その中で sysbench の fileio オプションを実行した．fileio では，ファイルの連続読み込み（SeqRead），ランダム読み込み（RndRead），連続書き込み（SeqWrite），ランダム書き込み（RndWrite）にかかった時間を計測する．あらかじめ 16MB のファイルを 128 個用意し，ブロックサイズとして 16KB を指定して読み込みと書き込みを行った．ファイルサイズ，ファイルの個数，ブロックサイズは全てデフォルトの数値を採用した．

x86_64 環境と ARM 環境の実行結果を表 3, 4 に示す．runc, crun，Inclavare Containers が全ての項目について時間差が小さく拮抗し，続いて gVisor，最後に大きく差が開いて Kata Containers が続くことが分かった．gVisor と Kata Containers はとくにファイル読み込み時に他のランタイムと大きく差が開いた．gVisor は x86_64 環境の RndRead で他のランタイムとの差が最も開き，最大で 14 倍の時間を要した．Kata Containers は ARM 環境の SeqRead で他のランタイムとの差が最も開き，最大で 175 倍の時間を要した．各ランタイム間の傾向は x86_64 環境と ARM 環境の両方で共通することが分かった．ただし，ARM 環境の Kata Containers の実行時間は他のランタイムに比べて極端に大きくなった．

関連研究[2]で述べられている通り，gVisor の性能劣化はファイルアクセス時に Gofer を経由することが原因だと考えられる．Kata Containers の性能劣化は，ホストとコンテナ間でファイル共有を行う際に使用する 9pfs が原因だと考えられる．ただし，Kata Containers ではオプションとして 9pfs の代わりに virtio-fs[16]を使用できる．Kata Containers で virtio-fs を使用した場合の性能評価について 4.5.1 節で述べる．ARM 環境の Kata Containers の実行時間が極端に大きい

表 5　x86_64 環境における Kata Containers の実行時間

	9pfs	virtio-fs
SeqRead（秒）	27.0362	2.1030
RndRead（秒）	2.0767	0.8587
SeqWrite（秒）	35.5674	23.5220
RndWrite（秒）	15.1615	14.8495

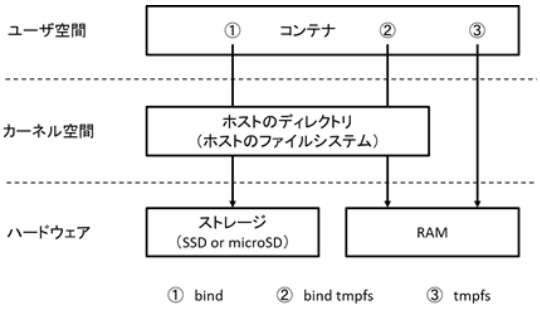

図 6　マウントの相関図

結果に対する追加実験について 4.5.2 節で述べる．

4.5.1　Kata Containers の virtio-fs の評価

Kata Containers を対象に追加実験を行った．x86_64 環境において，Kata Containers のオプションである virtio-fs を使用して先程と同様の測定を行った．

実行結果を表 5 に示す．virtio-fs を使用することで全ての項目の実行時間が小さくなることが分かった．とくに読み込み速度が大きく改善され，SeqRead は実行時間が約 25 秒小さくなり，gVisor に逆転することが分かった．ただし，ARM 環境では virtio-fs に対応しておらず，利用する際には注意が必要である．

4.5.2　書き込みに使用するストレージの影響の調査

Kata Containers を対象に追加実験を行った．本研究では x86_64 環境は SSD, ARM 環境は microSD を使用しており，ストレージの差が実行結果に影響している可能性がある．そこでストレージの影響を最小化するために，高速に処理が可能な RAM に書き込み先を変更して再度実験を行った．マウントの相関図を図 6 に示す．Docker のオプションである bind マウント[17]を使用し，ホストのディレクトリをコンテナ内にマウントした．その後，マウントしたディレクトリ内で SeqWrite を行う時間を計測した．この時，ホストのディレクトリについて tmpfs を使用せずに SSD（ARM 環境では microSD）上で書き込みを行う場合（bind），tmpfs を使用して RAM 上で書き込みを行う場合（bind tmpfs）の 2 つについて測定を行った．ただし，Docker でデフォルトのストレージドライバである overlay2 は tmpfs マウントしたホストのディレクトリをコンテナ内にマウントできなかったため，代わりに devicemapper を使用した．

実行結果を表 6 に示す．x86_64 環境と ARM 環境の両方で bind と bind tmpfs の実行時間の差が小さいことが分かった．とくに ARM 環境では，bind と bind tmpfs の実行時間が 300 秒を超えているのに対して時間差は約 12 秒であり，実行時間に対して時間差が小さいことが分かる．この結果から，書き込み場所が異なっていても実行時間に大きな差はなく，ARM 環境での Kata Containers の極端な性能劣化が microSD によるものではないことが分かった．

表 6　mount 方法ごとの実行時間

	runc	Kata Containers
x86_64　bind（秒）		13.3829
x86_64　bind tmpfs（秒）	0.7617	10.4779
x86_64　tmpfs（秒）	0.7647	0.2864
ARM　bind（秒）		322.8506
ARM　bind tmpfs（秒）	5.1588	310.4519
ARM　tmpfs（秒）	5.4796	1.1984

runc と Kata Containers を対象にさらに追加実験を行った．マウントの相関図を図 6 に示す．先程と同様にホストのディレクトリを tmpfs マウントした後にコンテナ内にマウントする場合（bind tmpfs），Docker のオプションである tmpfs マウント[18]を使用してコンテナのディレクトリを RAM 上に配置する場合（tmpfs）の 2 つで測定を行った．

実行結果を表 6 に示す．x86_64 環境と ARM 環境の両方で，bind tmpfs では Kata Containers，tmpfs では runc の実行時間が大きいことが分かった．よって，bind tmpfs と tmpfs で runc と Kata Containers の性能が逆転することが分かった．

実行結果について，Kata Containers のゲストカーネルがコンテナのワークロードに必要な処理のみを実装しており，ホストカーネルよりも処理が高速であることが原因だと考えられる．bind tmpfs では Kata Containers がホストのディレクトリを参照する際に 9pfs を使用する処理が遅く，カーネルの性能差は実行時間にさほど影響しない．一方で，tmpfs では RAM を直接参照できるため，bind tmpfs でボトルネックであった 9pfs の処理がなくなる．そのため tmpfs では，ゲストカーネルを使用した Kata Containers の方が実行時間は小さくなり，性能の逆転に繋がったと考えられる．

4.6　システムコール

TouchStone を使ってコンテナを作成し，その中で unixbench[19]の syscall オプションを実行した．syscall では一定時間内の close，getpid，getuid，umask の呼び出し回数によって Score を計測する．Score では，George（CPU：SuperSPARC，メモリ：128MB，OS：Solaris2.3）と呼ばれるサーバーの性能を 10 とした場合の相対値が算出される．

x86_64 環境と ARM 環境の実行結果を表 3，4 に示す．実行結果から，各ランタイム間の傾向が x86_64 環境と ARM 環境で異なることが分かった．x86_64 環境では runc, crun, Kata Containers, Inclavare Containers の Score 差が小さく拮抗

し，gVisor のみ Score が他のランタイムの約 41 分の 1 と極端に劣る結果となった．一方で，ARM 環境では runc，crun の Score 差が小さく拮抗し，gVisor は Score が他のランタイムの約 137 分の 1 と極端に劣り，Kata Containers は約 2.6 倍と極端に優れる結果となった．

gVisor の性能劣化はシステムコールを Sentry に渡す際に使用する ptrace が原因だと考えられる．システムコールを発行する度に ptrace によるオーバーヘッドがかかり，一定時間内で呼び出すシステムコールの回数が減少し，Score が小さい値になっている．ただし，gVisor ではオプションとして ptrace を行わない KVM モード[11]を使用できる．gVisor で KVM モードを使用した場合の性能評価について 4.6.1 節で述べる．ARM 環境の Kata Containers の性能向上は使用するゲストカーネルが原因だと考えられる．Kata Containers のゲストカーネルを変更した場合の性能評価について 4.6.2 節で述べる．

4.6.1 gVisor の KVM モードの評価

gVisor を対象に追加実験を行った．x86_64 環境において，gVisor のオプションである KVM モードを使用して先程と同様の測定を行った．

KVM モードを使用すると Score が 75.2 まで向上した．ただし，この Score は他のランタイムの半分程度であり，KVM モードを考慮しても gVisor のシステムコールの処理性能は低い．また ARM 環境では KVM モードを使用できないため，注意する必要がある．

4.6.2 Kata Containers のゲストカーネルの調査

Kata Containers を対象に追加実験を行った．今回，x86_64 環境と ARM 環境では別の方法で Kata Containers をインストールしており，使用するゲストカーネルが違っていた．そこで同じゲストカーネルを使用するべく，x86_64 環境でも ARM 環境と同じ snap を用いたインストール方法を試した．しかし，x86_64 環境で snap を用いてインストールする場合，実験時点で Kata Containers のバージョンが 2.11 となっていた．TouchStone でバージョン 2 以上の Kata Containers を使用したところ，コンテナの作成時にエラーが発生した．同様に，Docker でもバージョン 2 以上の Kata Containers をサポートしていない[20]．そこで，今回は別の方法[21]でインストールしたゲストカーネルを使用し，先程と同様の測定を行った．

実行結果として，Score が 462.7 まで向上した．追加実験では先程と同じ Kata Containers を使用し，ゲストカーネルのみ別の方法でインストールしたものに変更したところ Score が向上した．このことから Kata Containers では使用するゲストカーネルがシステムコールの処理性能に大きく影響することが分かった．

4.7 ネットワーク

Docker を使ってコンテナを作成し，その中で iperf[22]を実行した．iperf ではクライアント側からサーバー側の IP アドレスを指定した後に，一定時間だけトラフィックを流し，その間の帯域を計測する．本研究では同一 PC 上で，コンテナをサーバー側，ホスト環境をクライアント側として実行した．また指定秒数としてデフォルトの数値である 10 秒を採用した．今回，コンテナ作成時に TouchStone ではなく Docker を使用している理由は，TouchStone では開発時に IP アドレスを埋め込む必要があり，後から柔軟に IP アドレスを変更することができないからである．

x86_64 環境と ARM 環境の実行結果を表 3, 4 に示す．runc，crun，Inclavare Containers の Bandwidth 差が小さく拮抗し，大きく差が開いて Kata Containers, 最後に gVisor が続く結果となった．gVisor は x86_64 環境で他のランタイムの約 4 分の 1 の Bandwidth，ARM 環境で約 6 分の 1 の Bandwidth となった．Kata Containers は x86_64 環境で他のランタイムの約 3 分の 2 の Bandwidth, ARM 環境で約 7 分の 3 の Bandwidth となった．この結果から，各ランタイム間の傾向は x86_64 環境と ARM 環境の両方で共通することが分かった．

gVisor の性能劣化は，ホストのネットワークスタックを使用せずに Sentry 内部のネットワークスタックを使用することが原因だと考えられる[23]．ただし，gVisor ではオプションとしてホストのネットワークスタック[24]を使用できる．gVisor でホストのネットワークスタックを使用した場合の性能評価について 4.7.1 節で述べる．Kata Containers の性能劣化は，VMM を使用することが原因だと考えられる．コンテナがホストと通信を行う際に，仮想ネットワークインターフェースである veth を使用する．そしてコンテナの veth とホストに存在する Docker の veth の 2 つで veth pair を形成して通信する．Kata Containers では VMM を使用することが原因で，コンテナの veth と Docker の veth の間に直接 veth pair を形成できない．そこで VMM を使用しないランタイムとの差分を埋めるために TC Filter を使用しており，この処理が Bandwidth の低下に繋がったと考えられる．

4.7.1 gVisor のホストのネットワークスタックの評価

gVisor を対象に追加実験を行った．gVisor のオプションであるホストのネットワークスタックを使用して先程と同様の測定を行った．

ホストのネットワークスタックを使用すると，x86_64 環境では 16.86Gbits/秒，ARM 環境では 3.03Gbits/秒にまで Bandwidth が向上した．この際に，x86_64 環境では他のランタイムに劣るが，ARM 環境では Kata Containers に 0.25Gbits/秒だけ逆転することが分かった．

5. 結論

低レベルランタイム間の性能比較についてまとめると，runc，crun が全ての項目で優れており，続いて Inclavare Containers, 最後に大きく差が開いて gVisor と Kata Containers が続く結果となった．gVisor や Kata Containers のようなセキュリティを重視したランタイムでは，Sentry，Gofer や VM

などのセキュリティを強化する要素がボトルネックとなり，ファイル I/O，システムコール，ネットワークの項目で他のランタイムと大きく差が生じた．オプションを指定することで若干の性能向上は見られたが，runc や crun のような性能重視の低レベルランタイムには依然として及ばなかった．

x86_64 環境と ARM 環境のランタイムの傾向の比較についてまとめると，x86_64 環境と ARM 環境では基本的に大きな違いはなかった．システムコールの項目では ARM 環境が x86_64 環境に比べて，Kata Containers の Score が極端に大きくなった．しかし，使用するゲストカーネルを変更することで x86_64 環境でも Score が向上し，各ランタイム間の傾向はほとんど変わらない結果となった．また性能評価とは別に，ARM 環境ではランタイムの対応が x86_64 環境に比べて進んでいないため，利用を検討する際には注意が必要である．とくに Inclavare Containers を使用できない点，gVisor や Kata Containers の性能向上のオプションを指定できない点について頭に入れておく必要がある．

最後に今後の展望について述べる．本研究で使用した TouchStone は最新のランタイムに対応できない場合があることや文献が少ないことにより初学者には導入が難しい．さらに，組込みシステムで重要となる最悪実行時間が出力されない．これらの問題を解決するべく，多くの利用者に馴染みのある Docker を利用し，TouchStone に代わるコンテナランタイム評価ツールを開発したいと考えている．

参考文献

[1] Lennart Espe, Anshul Jindal, Vladimir Podolskiy, Michael Gerndt. Performance Evaluation of Container Runtimes. 10th International Conference on Cloud Computing and Services Science. 2019. p.273-281.
[2] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, The True Cost of Containing: A gVisor Case Study, USENIX conference on Hot Topics in Cloud Ccomputing. 2019.
[3] Rakesh Kumar, B Thangaraju. Performance Analysis Between RunC and Kata Container Runtime. IEEE Industrial Electronics Society. 2021, vol. 2, p.153-168.
[4] Gabriele Minì, Davide Antonino Giorgio, OS-level virtualization with Linux containers: process isolation mechanisms and performance analysis of last generation container runtimes. https://webthesis.biblio.polito.it/18660/1/tesi.pdf, (参照 2021-10-14).
[5] “Get Started with Docker”. https://www.docker.com/, (参照 2021-06-24).
[6] “touchstone”. https://github.com/lnsp/touchstone, (参照 2021-06-28).
[7] “runc”. https://github.com/opencontainers/runc, (参照 2021-06-28).
[8] “runc の権限昇格の脆弱性（CVE-2019-5736）に関する注意喚起”. https://www.jpcert.or.jp/at/2019/at190007.html, (参照 2021-06-28).
[9] “crun”. https://github.com/containers/crun, (参照 2021-06-28).
[10] “gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere”. https://gvisor.dev/, (参照 2021-06-28).
[11] “Changing Platforms”. https://gvisor.dev/docs/user_guide/platforms/, (参照 2021-07-03).
[12] “The speed of containers, the security of VMs”. https://katacontainers.io/, (参照 2021-06-28).
[13] “An open source enclave container runtime and security architecture for confidential computing scenarios”. https://inclavare-containers.io/en/, (参照 2021-06-28).
[14] “busy box”. https://hub.docker.com/_/busybox, (参照 2021-07-14).
[15] “sysbench”. https://github.com/akopytov/sysbench, (参照 2021-06-28).
[16] “Kata Containers with virtio-fs”. https://github.com/kata-containers/documentation/blob/master/how-to/how-to-use-virtio-fs-with-kata.md, (参照 2021-07-23).
[17] “Use bind mounts”. https://docs.docker.com/storage/bind-mounts/, (参照 2021-10-26).
[18] “Use tmpfs mounts”. https://docs.docker.com/storage/tmpfs/, (参照 2021-10-26).
[19] “byte-unixbench”. https://github.com/kdlucas/byte-unixbench. (参照 2021-07-01).
[20] “Kata Containers snap package”. https://github.com/kata-containers/documentation/blob/master/install/snap-installation-guide.md, (参照 2021-08-22).
[21] “Run Kata Containers on a Service VM”. https://projectacrn.github.io/2.1/tutorials/run_kata_containers.html, (参照 2021-08-22).
[22] “iPerf - The ultimate speed test tool for TCP, UDP and SCTP”. https://iperf.fr/iperf-download.php, (参照 2021-07-03).
[23] “gVisor Networking Security”. https://gvisor.dev/blog/2020/04/02/gvisor-networking-security/. (参照 2021-07-19).
[24] “Networking”. https://gvisor.dev/docs/user_guide/networking/. (参照 2021-07-13).