Vol.2022-ARC-248 No.32
Vol.2022-SLDM-198 No.32
Vol.2022-EMB-59 No.32
2022/3/10

柔軟なCPUリソースアカウントのための cgroupの拡張手法

松下瑛佑[1]　松原豊[1]　高田広章[1]

概要：複数のコンテナがホストのリソースを共有するマルチテナント環境では，コンテナに割り当てるリソースの管理が必要不可欠である．コンテナのリソース管理に用いられる cgroup は，コントローラによって cgroup にリソースを割当て，cgroup に所属するプロセスが使用したリソースを計測し，制限を行う．プロセスがワークロードを生成し，このワークロードを他の cgroup に所属するプロセスが処理した場合，このワークロードの処理に費やされるリソースは，ワークロードを生成したプロセスの所属する cgroup にはアカウントされず，ワークロードを処理するプロセスの所属する cgroup にアカウントされる．所属する cgroup 外へワークロードを生成し，このワークロードを他の cgroup に所属するプロセスに処理させることにより，ホストに同居するコンテナ間の性能分離は困難となる．本稿では，CPU リソースに着目し，所属する cgroup 外へワークロードがホストに同居する他のコンテナの性能へ与える影響を低減するために，柔軟な CPU ソースアカウントのための cgroup の拡張手法を提案する．提案手法は，ワークロードを生成したプロセスが所属する cgroup と，ワークロードを処理するプロセスが所属する cgroup 間で，CPU リソースのアカウント先を変更する．提案手法の実現により，所属する cgroup 外へ生成されるワークロードが他のコンテナの性能へ与える影響を低減できる．本稿では，提案手法の設計，実装，評価について述べる．

1. はじめに

複数のコンテナが同居するマルチテナント環境では，cgroup[1], [2] によるリソース制限が成立しなければ，コンテナ間の性能分離は困難となる．本稿ではコンテナ間の性能分離は，コンテナがホストに同居する他のコンテナの性能に影響を与えないことを指す．プロセスがワークロードを生成し，このワークロードを他の cgroup に所属するプロセスが処理した場合，このワークロードの処理に費やされるリソースは，ワークロードを生成したプロセスの所属する cgroup にはアカウントされず，ワークロードを処理するプロセスの所属する cgroup にアカウントされる．所属する cgroup 外へワークロードを生成し，このワークロードを他の cgroup に所属するプロセスに処理させ，他の cgroup のリソースを消費することによって，コンテナ間の性能分離は困難となる [3]．このため，所属する cgroup 外へ生成されるワークロードに対処することは重要である．

本稿では，所属する cgroup 外へ生成されるワークロードが，ホストに同居する他のコンテナの性能へ与える影響を低減するために，柔軟な CPU リソースアカウントのための cgroup の拡張手法（以降，提案手法）を提案する．提案手法は，ワークロードを生成したプロセスが所属する cgroup と，ワークロードを処理するプロセスが所属する cgroup 間で，CPU リソースのアカウント先を柔軟に変更することができる．提案手法を利用することによって，ホストに複数のコンテナが同居する環境において，所属する cgroup 外へ生成されるワークロードが他のコンテナの性能へ与える影響を低減できる．

本稿の貢献は以下の通りである．

- 所属する cgroup 外へ生成されるワークロードがホストに同居するコンテナに対する影響を明らかにした．
- 所属する cgroup 外へ生成されるワークロードがホストに同居するコンテナに対する影響を低減するための 2 種類のアプローチを検討した．
- 検討したアプローチ 2 を設計，実装，評価した．

2. リソースを管理するためのスコープ

コンテナのリソース管理には cgroup が用いられる．コンテナごとに cgroup を作成し，作成した cgroup にコンテナのプロセスを所属させ，作成した cgroup に，各コントローラを適用することによってコンテナのリソースを管理する．CPU コントローラは，cgroup の CPU リソースを管理するために用いられる．CPU コントローラは Linux のプロセススケジューラを利用し，cgroup に対して CPU 帯

[1] 名古屋大学 大学院情報学研究科

域幅制御を行う．CPU 帯域幅制御には，単位時間である period と，単位時間当たりの許容消費量である quota が用いられる [4]．cgroup に所属するプロセスは，単位時間である period に，単位時間当たりの許容消費量である quota の CPU 時間を使うことができる．

コンテナごとに作成された cgroup は，コンテナのリソースを管理するためのスコープとして考えることができる．cgroup は，リソースを管理するためのスコープである cgroup に所属するプロセスが使用したリソースを計測し，計測したリソースに基づいてリソースを制限する．

スコープである cgroup 外へワークロードを生成すると，このワークロードを処理するために費やされるリソースはワークロードを生成したプロセスが所属する cgroup にはアカウントされず，そのワークロードを処理したプロセスが所属する cgroup にアカウントされる．所属する cgroup 外へワークロードを生成し，このワークロードを他の cgroup に所属するプロセスに処理させることにより，cgroup というスコープによるリソース管理に，以下の問題が生じる．

(問題 1) cgroup に割り当てられた以上のリソースの消費
他の cgroup に所属しているプロセスにワークロードを生成し，処理を行わせることによって，そのワークロードを処理するプロセスが所属する cgroup に割り当てられたリソースを消費することができる．また，他の cgroup に所属しているプロセスにワークロードを生成して，処理を行わせることは，ワークロードを生成したプロセスが所属する cgroup に割り当てられた以上のリソースの消費につながる．

(問題 2) ホストに同居するコンテナの性能低下
他の cgroup に所属しているプロセスにワークロードを生成し，処理を行わせることによって，ホストのリソースを消費することができる．これによって，ホストに同居するコンテナの本来使用できるリソースが減少し，性能の低下が引き起こされる．

ホストに複数のコンテナが同居する環境で，攻撃者が 1 つのコンテナを操作可能な場合，これらの問題を悪用することによって，ホストや，ホストに同居しているコンテナに対する様々な攻撃を引き起こすことができる [3]．このため，所属する cgroup 外へ生成されるワークロードに対処することは重要である．

3. 所属する cgroup 外へ生成されるワークロードが及ぼす影響の定量的評価

3.1 評価項目と評価で対象とするワークロード

所属する cgroup 外へ生成されるワークロードが及ぼす影響を明らかにするために，以下の評価を行った．

(評価 1) CPU 使用率の評価
ワークロードの処理にかかる CPU 使用率を評価する．

(評価 2) ホストに同居するコンテナの性能評価
ワークロードの処理にかかる CPU 負荷がホストに同居するコンテナの性能に与える影響を評価する．

評価には，複数のコンテナがホストのリソースを共有するマルチテナント環境を用いる．評価に用いる環境を表 1 に示す．評価では，CPU 周波数の自動的な切り替えは行わず，Intel Turbo Boost は無効とした．

表 1 評価環境

OS	Ubuntu 20.04 LTS (Linux 5.4.124, 64bit)
CPU	Intel(R) Core(TM) i7-6600U @ 1.00GHz (4 コア)
メモリ	16 GB
CRI ランタイム	containerd v1.5.0 [5]
OCI ランタイム	runc v1.0.2 [6]
containerd CLI	nerdctl v0.14.0 [7]
Go [8]	v1.17.1

文献 [3] で述べられているコンテナのロギングを利用したワークロードを用いる．文献 [3] では，所属する cgroup 外へワークロードを生成する手法として，コンテナのロギングが利用されている．コンテナの標準出力，標準エラー出力に大量に出力を行わせ，このログを収集するプロセスに CPU 負荷をかける．この手法は，コンテナと，コンテナのログを収集するプロセスが所属する cgroup が異なることを利用している．

コンテナのログを収集するプロセスとして，nerdctl を用いる．nerdctl のプロセスはコンテナごとに存在する．コンテナのロギングはパイプに標準出力と標準エラー出力をバインドし，それぞれのパイプから出力を読み取ることで実現されている．nerdctl はデフォルトでは，パイプから標準出力と標準エラー出力を直接読み取る binary ロギング [9] を行う．コンテナの標準出力，標準エラー出力に大量に出力を行わせることで，nerdctl のプロセスに CPU 負荷をかける．containerd によって作成されるコンテナは ID を付与され，管理される．各コンテナのプロセスは cgroup: /default/containerID 以下に所属し，nerdctl のプロセスは cgroup: /system.slice/containerd.service 以下に所属する．

3.2 CPU 使用率の評価

測定手法

1 つの CPU 上に，単位時間として 100 ms，単位時間当たりの許容消費量として 50 ms，33 ms，25 ms，20 ms を割り当てたコンテナを 1 個作成する．本測定では，上記で述べた文献 [3] の手法を利用し，作成したコンテナに大量の出力を行わせる以下の 3 つのコマンドをそれぞれ実行させる．本測定の構成図を図 1 に示す．

(コマンド 1) yes

(コマンド 2) while true; do lsmod; done

(コマンド 3) while true; do echo 1; done

コマンド 1（yes）は'y'，または引数で指定された文字列を出力し続ける．コマンド 2（while true; do lsmod;

Vol.2022-ARC-248 No.32
Vol.2022-SLDM-198 No.32
Vol.2022-EMB-59 No.32
2022/3/10

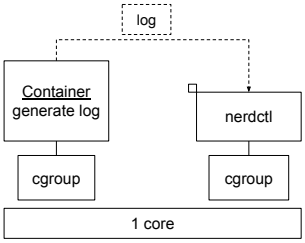

図 1 構成図（所属する cgroup 外へ生成されるワークロードの処理にかかる CPU 使用率の評価）

表 2 cgroup: /system.slice/containerd.service の CPU 使用率

単位時間当たりの許容消費量	コマンド 1	コマンド 2	コマンド 3
50 ms	99.36 %	70.45 %	77.78 %
33 ms	96.08 %	62.21 %	64.54 %
25 ms	95.81 %	47.57 %	51.04 %
20 ms	96.04 %	38.43 %	40.79 %

done）はロード済みのカーネルモジュールの一覧を出力し続ける．コマンド 3（while true; do echo 1; done）は'1' を出力し続ける．コンテナがそれぞれのコマンドを実行した場合の，nerdctl のプロセスが所属する cgroup: /system.slice/containerd.service の CPU 使用率を 30 秒間測定し，CPU 使用率の平均値を算出する．

測定結果と考察

測定結果を表 2 に示す．測定結果より，コンテナの標準出力，標準エラー出力に大量に出力を行わせることにより，これらのログを収集する nerdctl のプロセスが所属する cgroup: /system.slice/containerd.service に CPU 負荷をかけられることがわかる．コマンド 1 は，与えられた許容消費量には依存せず cgroup: /system.slice/containerd.service に CPU 負荷をかけられること，コマンド 2, 3 は，与えられた許容消費量にしたがって cgroup: /system.slice/containerd.service に CPU 負荷をかけられることがわかる．

コマンド 1 は，一度のコマンドの実行によって大量の出力が得られるのに対して，コマンド 2，3 は，コマンドを何度も実行することによって，出力を得る．また，コマンド 1，2，3 を実行するためにかかる許容消費量の消費は少ない．これらの理由から，コマンド 1 は与えられた許容消費量には依存せずに出力が得られ，コマンド 2，3 は，与えられた許容消費量にしたがって出力が得られる．

3.3 ホストに同居するコンテナの性能評価

3.2 節で測定した所属する cgroup 外へ生成されるワークロードによる CPU 負荷が，ホストに同居するコンテナの性能に与える影響を測定する．

測定手法

1 つの CPU 上に，コンテナを N 個（N = 1, 2, 3, 4, 5）作成し，それぞれのコンテナに 100 ms の単位時間と，単位時間当たりの許容消費量として単位時間 / N ms を与える．また，各コンテナに与える share[10] の値を 1024 とし，各コンテナへの CPU リソースの割り当てを等しくする．share とは，cgroup に対する CPU リソースの割り当ての比である．本測定の構成図を図 2 に示す．これによって，それぞれのコンテナは単位時間に与えられた許容消費量をすべて使い切ることができる．N 個のコンテナのうち，1 個のコンテナは runc によって作成し，sysbench[11] の CPU ベンチマークを実行する．runc によって作成した sysbench を実行するコンテナがベンチマーカとなる．sysbench のバージョンは v1.0.20 を用いる．また，そのほかの N-1 個のコンテナは nerdctl によって作成し，各コンテナは以下のコマンド 1，2，3，4 を実行する．nerdctl によって立ち上げられた N-1 個のコマンド 1，2，3，4 を実行するコンテナのログを収集する nerdctl のプロセスも，N 個のコンテナと同一の CPU で動作させる．

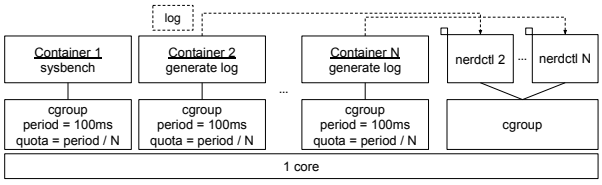

図 2 構成図（ホストに同居するコンテナの性能評価）

(コマンド 1) dd if=/dev/zero of=/dev/null

(コマンド 2) dd if=/dev/zero of=/dev/null & yes

(コマンド 3) dd if=/dev/zero of=/dev/null & while true; do lsmod; done

(コマンド 4) dd if=/dev/zero of=/dev/null & while true; do echo 1; done

コマンド 1 は CPU に負荷をかけ，与えられた許容消費量を使い切る．コマンド 2，3，4 は，コマンド 1 の効果に加え，大量の出力を発生することにより，3.2 節で示した通り，ログを収集するプロセスに CPU 負荷をかけられる．

N-1 個のコンテナがそれぞれのコマンドを実行している場合の，ベンチマーカであるコンテナが実行している sysbench の CPU ベンチマークの events per second の値を得ることにより，ホストに同居するコンテナの性能に与える影響を測定する．コマンド 1 はベースラインとして機能する．本測定における event は，10000 までの素数を見つけるループを実行した回数である．なお，sysbench の CPU ベンチマークは 1 スレッドで 10 秒間実行し，得られた events per second の値の 10 回平均を算出した．

測定結果と考察

測定結果を表 3 に示し，コマンド 1 の値をベースラインとしたコマンド 2，3，4 の値の比を表 4 に示す．測定結果より，コマンド 2，3，4 を実行した場合，コマンド 1 に比べ，events per sec の値が低下していることがわかる．所

Vol.2022-ARC-248 No.32
Vol.2022-SLDM-198 No.32
Vol.2022-EMB-59 No.32
2022/3/10

表 3 sysbench の events per second の値

N	コマンド 1	コマンド 2	コマンド 3	コマンド 4
1	328.35			
2	161.80	81.46	125.74	125.76
3	107.71	54.47	76.67	78.27
4	80.82	40.54	56.22	56.20
5	64.63	32.38	43.11	44.56

表 4 コマンド 1 に対する sysbench の events per second の値の比

N	コマンド 2	コマンド 3	コマンド 4
2	0.50	0.78	0.78
3	0.51	0.71	0.73
4	0.50	0.68	0.70
5	0.50	0.67	0.69

属する cgroup 外へワークロードを生成することによって，同一ホストに同居するコンテナの性能をコマンド 2 で 49〜50%，コマンド 3 で 22〜33%，コマンド 4 で 22〜31%低下させることができる．

コマンド 2，3，4 を実行すると，大量の出力が発生し，nerdctl が CPU リソースを消費するため，cgroup: /system.slice/containerd.service に負荷がかかる．したがって，N 個のコンテナが所属する cgroup: /default と，nerdctl が所属する cgroup: /system.slice/containerd.service が CPU リソースを奪い合う．このため，cgroup: /default 以下に所属する sysbench を実行するコンテナが本来使える CPU リソースが減少する．コマンド 2 とコマンド 1 の値の比が，0.5 程度となる理由は，本測定では，スケジューラが各 cgroup に対して均等に CPU リソースを与える設定を用いており，コマンド 2 の場合は，cgroup: /default と cgroup/system.slice に半分ずつ CPU リソースが渡り，ベースラインの場合と比べて cgroup: /default に与えられる CPU リソースが半分になるためである．また，この理由によって，コマンド 3，4 とコマンド 1 の値の比は，N の値が増加するにつれて，0.5 に近づくと推察される．

3.4 まとめ

3.2 節と，3.3 節で，所属する cgroup 外へ生成されるワークロードを処理するために費やされる CPU リソースと，このワークロードがホストに同居する他のコンテナの性能に与える影響を評価した結果について述べた．これらの評価結果から，本評価で用いたコンテナのロギングを利用したワークロードが引き起こす CPU 負荷によって，ホストに同居する他のコンテナの性能は最大で 50%低下する．

4. 所属する cgroup 外へのワークロードが及ぼす影響を低減するアプローチ

所属する cgroup 外へのワークロードが及ぼす影響を低減するアプローチとして，以下の 2 つが考えられる．

(アプローチ 1) 異なる cgroup に所属するプロセス間の処理の依存関係の解消

本アプローチは，所属する cgroup 外へ生成されるワークロードが発生する原因をなくすものである．異なる cgroup に所属するプロセス間に処理の依存関係が発生しなければ，所属する cgroup 外へワークロードは生成されない．リソースを管理するスコープである cgroup 内で，これに所属するプロセスの処理がすべて完結すればよい．これを実現するには，異なる cgroup に所属するプロセス間に処理の依存関係が発生しないように，cgroup にプロセスを所属させることが必要である．この依存関係を解消するための cgroup の構成について 5 章で述べる．

(アプローチ 2) 所属する cgroup 外へ生成されたワークロードを処理するために費やされるリソースのアカウント先の変更

本アプローチは，所属する cgroup 外へ生成されるワークロードが及ぼす影響を低減するものである．異なる cgroup に所属するプロセス間の処理の依存関係が存在する場合に，それらの cgroup 間で受け渡されるワークロードを特定し，ワークロードを処理するために費やされるリソースを計測する．ワークロードを処理した cgroup からその分を差し引いて，ワークロードの生成元の cgroup にアカウントし直す．この仕組みによって，cgroup 外へ生成されるワークロードが及ぼす影響を低減する．

5. 異なる cgroup に所属するプロセス間の処理の依存関係を解消する cgroup の構成

4 章で述べたアプローチ 1 の実現方法について述べる．ワークロードを生成するプロセスと，このワークロードを処理するプロセスが所属する cgroup が 1 対 1 の関係にある場合，異なる cgroup に所属するプロセス間の処理の依存関係を解消する cgroup の構成として，以下の方式 A と B が考えられる．以下の方式は，cgroup は，cgroup に所属するプロセスが使用したリソースが，そのプロセスの所属している cgroup にアカウントされること，cgroup はツリー階層をとり，子の cgroup が使ったリソースはその親の cgroup にもアカウントされることを利用している．

(方式 A) ワークロードを生成するプロセスとこのワークロードを処理するプロセスを同一の cgroup に所属させる

図 3 に本方式の全体像を示す．

(方式 B) 共通の親 cgroup 下に 2 つの子 cgroup を作成し，作成した 2 つの子 cgroup にそれぞれワークロードを生成するプロセスとワークロードを処理するプロセスを所属させる

図 4 に本方式の全体像を示す．

Vol.2022-ARC-248 No.32
Vol.2022-SLDM-198 No.32
Vol.2022-EMB-59 No.32
2022/3/10

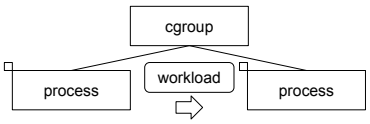

図 3 方式 A

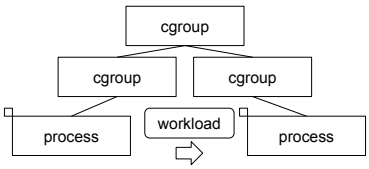

図 4 方式 B

ワークロードを生成するプロセスとこのワークロードを処理するプロセスが所属する cgroup が N 対 1（N は 1 以上の整数）の関係にある場合，ワークロードを処理するプロセスをワークロードを生成する特定の cgroup に所属させることは難しい．したがって，ワークロードを生成するプロセスごとに，これを処理するプロセス，またはスレッドを作成することにより，方式 A と方式 B の適用が検討できる．ワークロードを生成するプロセスごとに，これを処理するプロセス，またはスレッドを作成することが難しい場合は，cgroup の構成によって，異なる cgroup に所属するプロセス間の処理の依存関係を解消することが難しい．

cgroup v1 は各コントローラで cgroup のツリー階層を持つため，他のコントローラによる cgroup の設定を考慮する必要がなく，方式 A を比較的適用しやすい．一方，cgroup v2 は全コントローラで 1 つのツリー階層を共有するため，ワークロードを生成するプロセスとこのワークロードを処理するプロセスが所属する cgroup 間で設定の差異が吸収できない場合があり，方式 A を適用しずらい場合がある．例えば，cgroup 間で設定の差異の吸収が難しいコントローラとして，デバイスへのアクセスを許可，拒否を決定する devices コントローラがある．

方式 B のデメリットとして，cgroup の階層が深くなることによりスケジューリングにオーバヘッドが生じ，cgroup に所属するプロセスの性能が低下する点 [12] や，作成した 2 つの子 cgroup に対するリソース分配が難しい点がある．

ワークロードの処理時のみ，一方のプロセスをもう片方のプロセスの所属する cgroup へ移動させるという方式も考えられる．しかし，cgroup 間のプロセスの移動はコストが高いため，できるだけ cgroup 間のプロセスの移動は避けるべきである．[2]．このため，この方式は適用が難しい．

本章では，異なる cgroup に所属するプロセス間の処理の依存関係を解消する cgroup の構成について述べた．しかし，異なる cgroup に所属するプロセス間の処理の依存関係を完全に解消することは難しいと考えられる．例えば，異なる cgroup に所属するプロセスであるカーネルスレッドや，サービスプロセス，デーモンプロセスに処理が依存するプロセスが存在する．これらのすべての依存関係を考慮し，cgroup を構成するには，OS やアプリケーションの設計を見直す必要があり，それらの修正には限界がある．

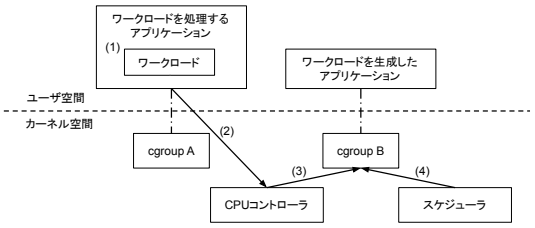

図 5 提案手法の処理流れ

6. 提案手法

6.1 考え方

5 章で述べた通り，4 章で述べたアプローチ 1 を実現する cgroup の構成には限界がある．異なる cgroup に所属するプロセス間の処理の依存関係の解消が困難である場合に，4 章で述べたアプローチ 2 は有効であると考えられる．本稿ではアプローチ 2 を実現する．現状の cgroup の仕組みでは，cgroup に所属するプロセスの使用したリソースが，そのプロセスの所属している cgroup にアカウントされる．したがって，所属する cgroup 外へ生成されるワークロードを処理するために費やされるリソースを，ワークロードを生成したプロセスが所属する cgroup と，ワークロードを処理するプロセスが所属する cgroup 間で，アカウント先を変更することはできない．

そこで，提案手法は，アカウントすべきリソースを受け渡すためのインターフェースを cgroup に作成する．また，ワークロードの処理に費やされるリソースを計測し，作成したインターフェースを利用することで，計測したリソースを処理をワークロードを生成したプロセスの所属する cgroup へアカウントする．これにより，ワークロードを生成したプロセスが所属する cgroup と，ワークロードを処理するプロセスが所属する cgroup 間で，リソースのアカウント先を柔軟に変更できる．なお，提案手法は，リソースとして CPU リソースに着目した．提案手法は，ワークロードの処理に費やされる CPU 時間を計測し，CPU 時間のアカウント先を cgroup 間で柔軟に変更できる．

6.2 処理流れ

提案手法はワークロードを処理するアプリケーション，CPU コントローラ，スケジューラ上に実現される．提案手法の処理流れを図 5 に示し，以下で説明する．

(1) ワークロードを処理するアプリケーションがワークロードを処理するために費やした CPU 時間を計測する

(2) ワークロードを処理するアプリケーションが (1) で計測した CPU 時間とこれをアカウントすべき cgroup の情報を CPU コントローラに伝達する
(3) CPU コントローラが (2) で受け取った CPU 時間とこれをアカウントすべき cgroup の情報をもとに，その cgroup に CPU 時間を保存する
(4) スケジューラが次の単位時間においてその cgroup に割り当てられる許容消費量から (3) で保存されている CPU 時間を減算する

6.3 実装

6.1 節と，6.2 節で述べた提案手法を，ワークロードを処理するアプリケーションとして nerdctl，CPU コントローラとして cgroup v1 の CPU コントローラ，スケジューラとして Completely Fair Scheduler（以降，CFS）を用いて実現した．以下で実現方式について述べる．

6.3.1 CPU 時間の計測方式

nerdctl において，ワークロード（標準出力と標準エラー出力のロギング）を処理しているスレッドの utime と stime の和をワークロード処理前後の時点で引くことで，ワークロードを処理するために費やされた CPU 時間を得る．utime と stime はそれぞれ，プロセスがユーザ，カーネルモードで実行した CPU 時間である．ワークロードの処理に費やされた正しい CPU 時間を得るため，ワークロードを実行する専用のスレッドを作成し，ワークロードを実行する．nerdctl のプロセスがコンテナごとに存在することを利用して，ワークロードを生成したプロセスが所属する cgroup を識別し，計測した CPU 時間を保持する．

6.3.2 cgroup への CPU 時間のアカウント方式

CPU コントローラを拡張することにより，各 cgroup に cpu.cfs_borrowing_us という仮想ファイルを作成する．cpu.cfs_borrowing_us に値を書き込むことによって，当該 cgroup のために他の cgroup に所属するプロセスが使用した CPU 時間を CPU コントローラに伝達する．コンテナ作成時に，コンテナに対して readonly で sysfs をマウントすることにより，コンテナからは cpu.cfs_borrowing_us に値を書き込むことはできない．これによって，コンテナからの cpu.cfs_borrowing_us に対する不正な値の書き込みを防ぐ．また，この書き込みには，管理者権限が必要である．

CFS を拡張し，各 cgroup で borrowing という変数を保持する．borrowing の値は，当該 cgroup のために他の cgroup に所属するプロセスが使用した CPU 時間の総和である．CPU コントローラは cpu.cfs_borrowing_us に書き込まれた値を当該 cgroup の borrowing に加算することによってその値をアカウントする．

6.3.3 cgroup に割り当てられる許容消費量の減算方式

CFS を拡張し，各単位時間の開始時に，cgroup の当該単位時間における許容消費量から borrowing の値を減算する．borrowing の値が当該単位時間における許容消費量よりも大きい場合，borrowing の値から当該単位時間における許容消費量を減算し，当該単位時間における許容消費量を 0 とする．一方，borrowing の値が当該単位時間における許容消費量よりも小さい場合，当該単位時間における許容消費量から borrowing の値を減算し，borrowing の値を 0 とする．

7. 評価

7.1 評価内容と評価環境

6.3 節で実装した提案手法について，有用性と性能に与える影響を明らかにするために，以下の評価を行った．評価に用いる環境は表 1 と同様である．

(評価 1) CPU 使用率の評価
提案手法を実装したアプリケーションと，コンテナの CPU 使用率を測定する．

(評価 2) ホストに同居するコンテナの性能評価
ワークロードの処理にかかる CPU 負荷が，ホストに同居するコンテナの性能に与える影響を明らかにする．

7.2 CPU 使用率の評価

単位時間として 100 ms，単位時間当たりの許容消費量として 50 ms，33 ms，25 ms，20 ms を割り当てたコンテナを 1 つ作成する．本測定では，3.2 節と同様に，文献 [3] の手法を利用し，作成したコンテナに大量の出力を行わせる以下の 3 つのコマンドをそれぞれ実行させる．

(コマンド 1) yes
(コマンド 2) while true; do lsmod; done
(コマンド 3) while true; do echo 1; done

本測定では以下の項目を 30 秒間測定し，平均値を算出する．

(項目 A) nerdctl のプロセスの CPU 使用率
(項目 B) nerdctl のプロセスのうち，提案手法によってコンテナにアカウントしたプロセスの CPU 使用率
(項目 C) コマンドを実行するコンテナの CPU 使用率

コマンド 1，2，3 を実行したときの結果をそれぞれ表 5，表 6，表 7 に示す．それぞれの表で，項目 B と項目 C の和が，単位時間当たりの許容消費量と単位時間の比程度であることがわかる．これは，例えば，単位時間として 100 ms，単位時間当たりの許容消費量として 50 ms をコンテナに与えた場合，そのコンテナの CPU 使用率は 50%になるためである．また，項目 A と項目 B の差から，nerdctl の処理のうち，コンテナにアカウントしていない部分があることがわかる．

表 5，表 6，表 7 で項目 B と項目 C の和が，単位時間当たりの許容消費量と単位時間の比程度であることから，提案手法によって，ワークロードにかかる CPU 時間を，ワークロードを生成したコンテナにアカウントできていること

がわかる．項目 A と項目 B の差は，nerdctl のプロセスのうち，標準出力と標準エラー出力を読み取り，ログを処理するスレッド以外の CPU 使用率であると考えられる．

7.3 ホストに同居するコンテナの性能評価

測定手法は 3.3 節で述べたものと同様である．測定結果を表 8 に示し，コマンド 1 の値をベースラインとしたコマンド 2，3，4 の値の比を表 9 に示す．提案手法を実装していない環境での測定結果である表 4 と，提案手法を実装した環境での測定結果である表 9 を比較すると，提案手法を実装した環境の方が，コマンド 2，3，4 のそれぞれで，コマンド 1 からの値の乖離が小さい．

所属する cgroup 外へワークロードが生成された場合でも，提案手法によって，このワークロードの処理にかかる CPU 時間を，ワークロードを生成したプロセスが所属する cgroup にアカウントすることで，同一ホストに同居するコンテナの性能に与える影響をコマンド 2 で 5〜30%，コマンド 3 で 7〜22%，コマンド 4 で 8〜18%に抑えられることがわかる．表 3 と表 8 でベースラインの events per sec の値を比較すると，events per sec の値の差は最大で 0.46 と小さく，提案手法を実装することによるオーバヘッドは小さいと考えられる．コマンド 2 とコマンド 1 の値の比，コマンド 3 とコマンド 1 の値の比，コマンド 4 とコマンド 1 の値の比が 1 から乖離している原因は，7.2 節で述べた通り，nerdctl の処理のうち，コンテナにアカウントしていない部分の CPU 使用率である項目 A と項目 B の差によるためであると考えられる．また，これらの値の比が，N の値の増加にしたがって，1 からの乖離が大きくなる原因は，項目 A と項目 B の差と，コマンドを実行するコンテナの数（N-1）の積が，N の値の増加にともなって増加しているためであると考えられる．つまり，3.3 節で述べた通り，cgroup: default と，cgroup: /system.slice/containerd.service で CPU リソースの奪い合いが生じ，cgroup: /default 以下に所属している sysbench を実行するコンテナの本来使える CPU リソースが減少していることが推察される．このため，nerdctl の処理にかかる CPU リソースをすべて，コンテナにアカウントすれば，ベースラインからの値の乖離をさらに小さくできると考えられる．

8. 関連研究

オペレーティングシステム（以降，OS）にとって，システムのリソースの管理は必要不可欠である．そして，リソースの管理を行ううえで，リソースを管理するためのスコープの構成は重要である．リソースを管理するためのスコープの構成が十分に考慮されていなければ，本来アカウントされるべきスコープにリソースがアカウントされない．これは，スコープに対するリソースの不公平な割り当てや，スコープ間の不十分な分離を引き起こす．リソースを管理するためのスコープの構成に関する研究，リソースのアカウントに関する研究，リソースを管理するためのスコープ外へのワークロードの生成に関する研究について以下で述べる．

8.1 リソースを管理するためのスコープの構成に関する研究

文献 [13] では，スレッドやプロセスの単位で，これらが使ったリソースを管理するためのスコープを構成するのではなく，スレッドやプロセスを束ねて Resource Container という単位でリソースを管理することが述べられている．Resource Container という単位でリソースを管理することによって，アプリケーションを意識した，より効率的なリソースの管理を行うことができる．OS の視点に立って，スレッドやプロセスという単位を，リソースを管理するためのスコープと捉えるのは，ボトムアップ的な考え方である．一方，アプリケーションの視点に立って，アプリケーションを構成するためのスレッドやプロセスを束ねてスコープと捉えるのは，トップダウン的な考え方である．cgroup と Resource Container によるリソース管理に対する考え方は，どちらもアプリケーションの視点に立っており，トップダウン的で共通している．Resource Container や cgroup は，リソースを管理するためのスコープを構成し，そのスコープに属するスレッドやプロセスの使ったリソースを，そのスコープに対してアカウントする．また，Resource Container や cgroup はスケジューリング対象でもあり，Resource Container や cgroup の情報に基づいてに所属するスレッドやプロセスをスケジューリングする．文献 [14] は，Xen において，Dom（Xen における VM の単位であり，ドメインと呼ばれる）の保護スコープが，リソースを管理するためのスコープとして誤って使用されていると主張している．リソースを管理するためのスコープが誤って使用されることによって，Dom U（ゲストドメイン）から Dom 0（ゲストドメインを管理するためのドメイン）にオフロードされる IO 処理などに費やされるリソースが各 Dom U にアカウントされない．そこで，特定の Dom U をサーブする IRQ スレッドやカーネルスレッドを，DomU の vCPU に紐付ける VASE（vCPU as a container）を提案している．文献 [15] は，Xen と KVM を利用する環境において，セキュリティの向上のために IDS を動作させる場合に，IDS の処理に費やされるリソースが考慮されていないとして，新たなリソースを管理するためのスコープが必要だと主張している．そこで，Resource Cages を提案し，IDS と VM を束ねてリソースを管理するためのスコープとしている．

本稿では，リソースを管理するためのスコープを再構成するのではなく，スコープ間でリソースを受け渡すための手法を提案している．

表 5 コマンド 1 を実行したときの項目 A，B，C

単位時間当たりの許容消費量	項目 A	項目 B	項目 C	項目 B + 項目 C	項目 A - 項目 B
50 ms	52.57 %	46.62 %	4.61 %	51.23 %	5.95 %
33 ms	36.94 %	32.51 %	2.11 %	34.62 %	4.43 %
25 ms	28.55 %	24.96 %	1.60 %	26.56 %	3.59 %
20 ms	23.14 %	19.97 %	1.56 %	21.53 %	3.17 %

表 6 コマンド 2 を実行したときの項目 A，B，C

単位時間当たりの許容消費量	項目 A	項目 B	項目 C	項目 B + 項目 C	項目 A - 項目 B
50 ms	17.38 %	12.54 %	37.44 %	49.98 %	4.84 %
33 ms	11.62 %	7.71 %	24.46 %	32.17 %	3.91 %
25 ms	8.85 %	5.93 %	19.21 %	25.14 %	2.92 %
20 ms	7.52 %	4.71 %	15.22 %	19.93 %	2.81 %

8.2 リソースのアカウントに関する研究

文献 [16], [17], [18] は，ネットワークスタックの処理にかかる CPU 時間が，パケットを受信，または送信したコンテナの所属する cgroup にアカウントされないことに着目している．この問題によって，同一ホストに同居するコンテナの性能低下が生じる．この問題の原因は，softirq がプロセスコンテキストで動くため，割り込みが入ったタイミングで CPU で実行されているプロセスが所属する cgroup に，処理に費やされた CPU 時間が誤ってアカウントされてしまうことと，ksoftirqd に softirq の処理が委譲され，ksoftirqd の処理に費やされるリソースがルート cgroup にアカウントされることである．Iron[16] は，これらの処理に費やされる CPU 時間を計測し，各コンテナにアカウントする仕組みである．文献 [18] は，ユーザ空間で動くネットワークスタックを実装した Snap を提案している．Snap は，独自に開発した Linux カーネルインターフェースを用いて，ネットワークスタックの処理に費やしたリソースを，各コンテナにアカウントしている．

文献 [16], [18] の研究は，カーネルに対してリソースのアカウントの仕組みの変更を行っており，リソースを管理するためのスコープ間でのリソースの受け渡しは考慮されていない．

8.3 リソースを管理するためのスコープ外へのワークロードの生成に関する研究

文献 [3] は，リソースを管理するためのスコープを超えたワークロードや，このワークロードを処理するために費やされるリソースが考慮されていないことに着目している．文献 [19] は，コンテナ型仮想化といった OS のカーネルを共有する仮想化に対して，カーネルを共有することによる脆弱性を突いた攻撃手法を提案している．この攻撃では，カーネルの変数，データ構造に対して DoS を行っている．このようにリソースを管理するためのスコープが十分に考慮されていない場合，スコープを超えたワークロードの生成によって，リソースの制限や，アカウントは難しくなる．

本稿では，リソースを管理するためのスコープを超えたワークロードの生成という問題に対して，ワークロードを処理するためのリソースを，スコープ間で受け渡すための手法を提案している．

9. おわりに

所属する cgroup 外へ生成されるワークロードが，ホストに同居する他のコンテナの性能へ与える影響を低減するために，柔軟な CPU リソースアカウントのための cgroup の拡張手法を提案した．提案手法は，ワークロードを生成したプロセスが所属する cgroup と，ワークロードを処理するプロセスが所属する cgroup 間で，CPU リソースのアカウント先を柔軟に変更することができる．提案手法の実現により，ホストに複数のコンテナが同居する環境において，所属する cgroup 外へ生成されるワークロードが他のコンテナの性能へ与える影響を低減できることを示した．今後の課題として，CPU 時間の計測方式の改良がある．

参考文献

[1] The Linux Kernel documentation: Control Groups version 1, (online), available from ⟨https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/index.html⟩ (accessed 2022-01-13).
[2] The Linux Kernel documentation: Control Group v2, (online), available from ⟨https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html⟩ (accessed 2022-01-13).
[3] Gao, X., Gu, Z., Li, Z., Jamjoom, H. and Wang, C.: Houdini's Escape: Breaking the Resource Rein of Linux Control Groups, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, New York, NY, USA, Association for Computing Machinery, p. 1073–1086 (2019).
[4] The Linux Kernel documentation: CFS Bandwidth Control, (online), available from ⟨https://www.kernel.org/doc/html/v5.4/scheduler/sched-bwc.html⟩ (ac-

表 7 コマンド 3 を実行したときの項目 A，B，C

単位時間当たりの許容消費量	項目 A	項目 B	項目 C	項目 B + 項目 C	項目 A - 項目 B
50 ms	24.99 %	20.88 %	28.71 %	49.59 %	4.11 %
33 ms	16.97 %	13.54 %	19.34 %	32.88 %	3.43 %
25 ms	13.29 %	10.32 %	13.86 %	24.18 %	2.97 %
20 ms	10.70 %	8.03 %	11.80 %	19.83 %	2.67 %

表 8 sysbench の events per second の値

N	コマンド 1	コマンド 2	コマンド 3	コマンド 4
1	328.35			
2	161.41	153.26	149.69	148.51
3	107.25	92.99	89.24	93.36
4	80.36	60.70	63.70	67.40
5	64.36	45.19	49.94	52.73

表 9 コマンド 1 に対する sysbench の events per second の値の比

N	コマンド 2	コマンド 3	コマンド 4
2	0.95	0.93	0.92
3	0.87	0.83	0.87
4	0.76	0.79	0.84
5	0.70	0.78	0.82

cessed 2022-01-13).

[5] containerd: containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability, (online), available from ⟨https://containerd.io/⟩ (accessed 2022-01-13).

[6] Open Container Initiative: opencontainers/runc, (online), available from ⟨https://github.com/opencontainers/runc⟩ (accessed 2022-01-13).

[7] containerd: containerd/nerdctl, (online), available from ⟨https://github.com/containerd/nerdctl⟩ (accessed 2022-01-13).

[8] Google: The Go Programming Language, (online), available from ⟨https://go.dev/⟩ (accessed 2022-01-13).

[9] containerd: containerd/runtime/v2/README.md, (online), available from ⟨https://github.com/containerd/containerd/tree/main/runtime/v2#logging⟩ (accessed 2022-01-13).

[10] The Linux Kernel documentation: CFS Scheduler, (online), available from ⟨https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html⟩ (accessed 2022-01-13).

[11] akopytov: akopytov/sysbench, (online), available from ⟨https://github.com/akopytov/sysbench⟩ (accessed 2022-01-13).

[12] Linux Plumbers Conference 2020: CFS flat runqueue v2, (online), available from ⟨https://linuxplumbersconf.org/event/7/contributions/762/⟩ (accessed 2022-01-13).

[13] Banga, G. and Mogul, J. C.: Resource Containers: A New Facility for Resource Management in Server Systems, *3rd Symposium on Operating Systems Design and Implementation (OSDI 99)*, New Orleans, LA, USENIX Association (1999).

[14] Liu, L., Wang, H., Wang, A., Xiao, M., Cheng, Y. and Chen, S.: VCPU as a Container: Towards Accurate CPU Allocation for VMs, *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2019, New York, NY, USA, Association for Computing Machinery, p. 193–206 (2019).

[15] Kourai, K., Arai, S., Nakamura, K., Okazaki, S. and Chiba, S.: Resource Cages: A New Abstraction of the Hypervisor for Performance Isolation Considering IDS Offloading, *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 170–177 (2017).

[16] Khalid, J., Rozner, E., Felter, W., Xu, C., Rajamani, K., Ferreira, A. and Akella, A.: Iron: Isolating Network-Based CPU in Container Environments, *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, USA, USENIX Association, p. 313–328 (2018).

[17] Kim, E., Lee, K. and Yoo, C.: On the Resource Management of Kubernetes, *2021 International Conference on Information Networking (ICOIN)*, pp. 154–158 (2021).

[18] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., Kidd, N., Kononov, R., Kumar, G., Mauer, C., Musick, E., Olson, L., Rubow, E., Ryan, M., Springborn, K., Turner, P., Valancius, V., Wang, X. and Vahdat, A.: Snap: A Microkernel Approach to Host Networking, *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, New York, NY, USA, Association for Computing Machinery, p. 399–413 (2019).

[19] Yang, N., Shen, W., Li, J., Yang, Y., Lu, K., Xiao, J., Zhou, T., Qin, C., Yu, W., Ma, J. and Ren, K.: Demons in the Shared Kernel: Abstract Resource Attacks Against OS-Level Virtualization, *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, New York, NY, USA, Association for Computing Machinery, p. 764–778 (2021).