Vol.2018-OS-142 No.4
2018/2/27

コンテナ環境におけるジャーナリングI/Oの制御

飛松 秀三郎[1]　青田 直大[1]　Asraa Abdulrazak Ali Mardan[1]　河野 健二[1]

概要：コンテナ型仮想化では複数のコンテナ間で単一のカーネルを共有しているため，コンテナ間での性能隔離が容易ではない．特に，ファイルシステムにおけるジャーナリング I/O は，コンテナ間での分離が難しく，著しい性能干渉を起こすことが知られている．このような性能干渉が起きるのは，それぞれのコンテナに起因するジャーナリング I/O がカーネル内で集約されてしまい，ジャーナリング I/O によるディスク帯域の使用量を適切に把握することができないためである．本研究では，各コンテナが発行するシステムコールの履歴から各コンテナに起因するジャーナリング I/O の帯域消費量を推定し，この推定に基づいてジャーナリング I/O を含めてディスクの帯域制御をおこなう方法を提案する．提案方式を用いると，既存のファイルシステムやコンテナ環境を改変することなくジャーナリング I/O の制御が可能となる．実験では，ジャーナリング I/O を大量に発行するコンテナがある場合に，提案方式で適切に帯域制御が行われることを示す．

キーワード：ジャーナリング，ファイルシステム，コンテナ型仮想化，クラウド・コンピューティング

1. はじめに

近年，コンテナ型仮想化がクラウド環境で使用されるようになっている．例としては，Amazon Web Service (AWS)[1]，Google Cloud Platform[2] などが挙げられる．他の仮想化の手段としては，ハイパーバイザ型仮想化がある．ハイパーバイザ型仮想化はそれぞれの VM に対しハードウェア抽象を与えているのに対し，コンテナ型仮想化は単一 OS 上に複数の仮想ユーザ空間を提供する形を取っており，各コンテナはホスト OS のカーネルを共有して使用しているため，種々のオーバーヘッドが少ない．

しかしながら，コンテナ間でホストのカーネルを共有することにより，マルチコンテナ環境においては著しい性能低下と性能干渉が起こる問題があることがわかっている．クラウド環境では，異なる顧客が同一マシン上で仮想マシンを利用するため，リソースは公平に分配されることが望ましい．そこでこの問題を解決するために，ファイルシステムをコンテナごとに物理的に分割する手法 [3] や，仮想ストレージを独立した I/O スタックでホストする手法 [4] が提案されている．これらは性能隔離を実現し，さらに性能低下も十分に抑えることに成功している．ところが，いずれもファイルシステムに変更を加えており，導入コストが大きいという側面がある．本研究のモチベーションは，より導入コストを抑えながら性能隔離を実現したいというものである．

性能干渉の一つに，ジャーナリング I/O がカーネル内で集約されてしまうことによって，各コンテナのディスク帯域の使用量が適切に制御できないという問題がある．ジャーナリング [5] とは，ファイルシステムの一貫性を保つための仕組みである．ジャーナリングではディスクへデータを書き込む前に，ジャーナルと呼ばれる領域に書き込みデータのメタデータを書き込むのだが，ジャーナリング I/O はこのメタデータの書き込みを指す．

本研究は，以上の問題に焦点を当て，Linux カーネルには変更を加えず，各コンテナのディスク I/O の帯域消費量を矯正する手法を提案する．具体的には，システムコールのメタデータ I/O 発行量の分散が小さいことに着目し，各コンテナのシステムコールの呼び出しをモニタリングする．そして，システムコールの呼び出し回数から各コンテナに起因するジャーナリング I/O の帯域消費量を推定し，この推定を利用してディスクの帯域制御をおこなう．

実験では，提案手法を実装した場合と実装していない場合で性能比較する．具体的には，2 つのコンテナを起動し，一方のコンテナではシーケンシャルリードワークロードを動かし，一方のコンテナではシーケンシャルリードとメタデータインテンシブなライトワークロードを動かす．これにより，後者のコンテナだけが大量にジャーナリング I/O を発行する状態を作る．ディスク I/O は両者とも 50%に制

[1] 慶應義塾大学
Keio University

限した．その結果，本提案を実装していない場合では，前者と後者のジャーナリング I/O を含めたディスクの帯域消費率がそれぞれ平均で 64.1% ，35.9% にであったのに対し，実装した場合では 49.9% ，50.1% にすることができ，リソース制限本来の 50% ，50% に近づけることができた．

本論文の構成を以下に示す．第 2 章では，コンテナ環境におけるジャーナリング I/O の問題を紹介する．第 3 章にて本研究の提案手法について述べる．第 4 章にて提案手法の実装方法について述べる．第 5 章にて提案手法を実装した場合としていない場合の性能の違いを確認する実験について述べる．第 6 章では，本研究の関連研究を紹介する．第 7 章ではまとめを述べる．

2. コンテナ環境におけるジャーナリング I/O の問題点

2.1 ジャーナリング

ジャーナリングは，ファイルシステムへの更新内容をトランザクションと呼ばれる単位でアトミックにログを書き出すことによって，ファイルシステムの一貫性を保証する手法で，ext4 や NTFS など多くのファイルシステムで採用されている．具体的には，トランザクションをジャーナルと呼ばれるディスク上の領域に書き出した後，本来のファイルシステムのデータ領域に書き出す．例えば，トランザクションをジャーナル領域に書き出す最中に OS がクラッシュした場合，リカバリの際にはその書き出しが取り消すことによって一貫性を保つ．また，トランザクションをジャーナル領域に書き出し終わり，その内容をデータ領域に書き出す最中に OS がクラッシュした場合，リカバリの際にはジャーナルに書き出された内容をデータ領域にリプレイすることによって一貫性を保つ．

ext4 では，journaling block device version 2 (JBD2) と呼ばれる Linux カーネル内のモジュールによってジャーナリングがおこなわれる [6]．JBD2 スレッドは種々のユーザープロセスから発行されたライトのリクエストによるメタデータやデータの更新を メモリ上に 1 つのトランザクションとして集約する．これにより，短期間に同一のメタデータ構造を頻繁に更新するような場合にパフォーマンスの向上が見込める．トランザクションがジャーナル領域に書き出されるタイミングは 2 通り存在し，fsync や syncfs といった同期操作によって書き出される場合と，JBD2 の定期実行で書き出される場合がある．

ジャーナリングには，メタデータのみをログに書き出すメタデータジャーナリングと，データとメタデータの両方をログに書き出すデータジャーナリングがあるが，パフォーマンスの観点からメタデータジャーナリングをおこなうのが一般的となっている．

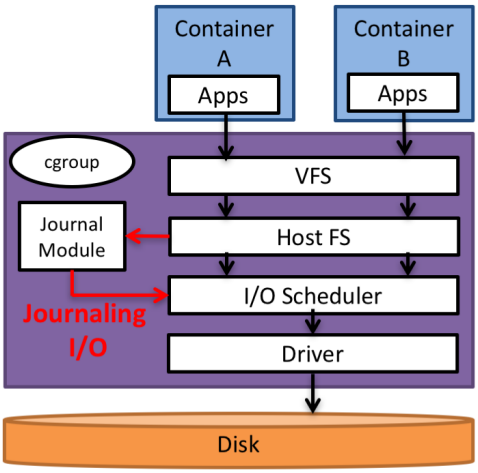


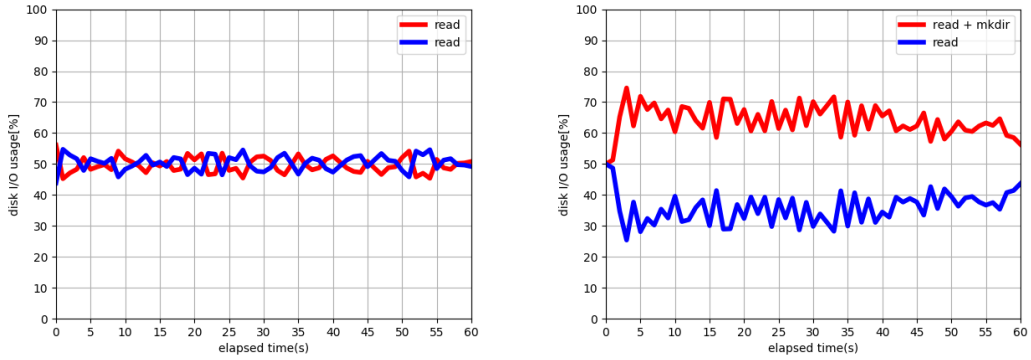
図 1: コンテナ環境における I/O の流れ

2.2 コンテナ環境における問題点

図 1 はコンテナ環境における I/O の流れを示した図である．Linux におけるコンテナ型仮想化では，カーネルが提供する Control Groups (cgroup)[7] によってディスク I/O やメモリなどの物理リソースの制限をおこなっているが，先に挙げたジャーナリングはカーネル内の JBD2 スレッドによって I/O リクエストがおこなわれるため，cgroup の制限外となってしまう．しかし，ジャーナルにはリソース制限をおこなう必要のあるコンテナ由来のメタデータも含まれているため，公平にディスクリソースを分配するにはジャーナリング I/O も正しく分配する必要がある．

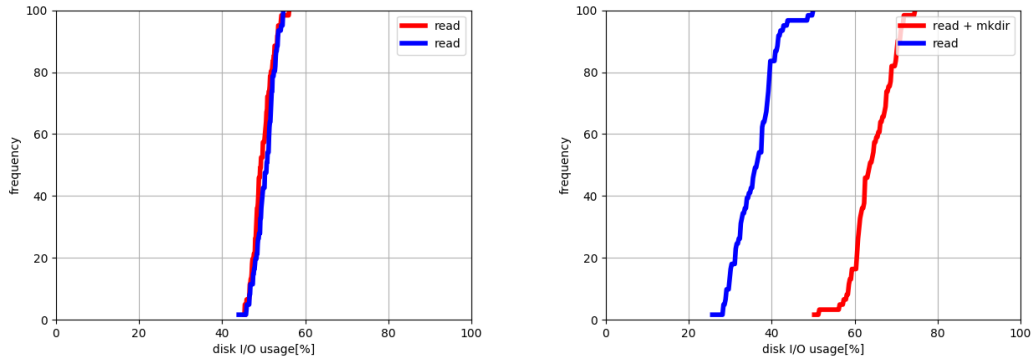
実際にジャーナリング I/O が引き起こす性能干渉を確認するために，ジャーナリング I/O が発行されないワークロードを実行する場合と，ジャーナリング I/O が発行されるワークロードを実行する場合とで比較をおこなった．実験環境は 5 章と同様である．

コンテナを 2 つ立ち上げ，(a) 両コンテナでシーケンシャルリードをおこなうワークロードを実行する実験と，(b) 一方ではシーケンシャルリードのみを実行し，もう一方ではシーケンシャルリードに加え，mkdir を 1 秒間に 2000 回おこなうワークロードを実行する実験の 2 つをおこなう．mkdir 大量に発行することによって，一方のコンテナでジャーナリング I/O が発行される状況を作る．シーケンシャルリードには Flexible I/O[8] ベンチマークを利用した．各コンテナの I/O は，cgroup の比例配分ポリシーにて両者の重みが 50:50 となるよう設定した．図 2 は各コンテナのジャーナリングを考慮した I/O の推移であり，図 3 はその累積頻度グラフである．ジャーナリング I/O が発行されていない (a) では各コンテナの I/O 比率がリソース制限の 50%，50%に近い状態となっているが， (b) ではジャーナリング I/O を発行するコンテナが 65% ほど帯域を消費し，リードのみをおこなうコンテナの帯域消費率は 35% に留まっている．



(a) リード，リード　　(b) リード，リード + メタデータライト

図 2: リードのみの場合とメタデータライト有りの場合の比較



(a) リード，リード　　(b) リード，リード + メタデータライト

図 3: I/O 比率の累積頻度グラフ

3. 提案

3.1 概要

ジャーナリング I/O は単一のトランザクションに集約されており，現状，ブロックレベルにおいてはトランザクション中のメタデータがどのプロセスによって更新されたものか，という情報を持っていない．故にカーネルに変更を加えない限り，各プロセスによって発行された正確なジャーナリング I/O 量は取得することができない．そこでシステムコールの呼び出し履歴からメタデータのジャーナリング I/O を推定をする手法を提案する．そして，その推定を利用してユーザ空間から帯域制御をおこなう．図 4 は本提案の概要を表した図である．

図内に示されたユーザ空間内で動作するモジュールによって，各コンテナからのシステムコールの呼び出しのモニタリングと，帯域制御をおこなう．システムコールの呼び出しの検知には，Linux では strace や Linux Performance Tool (perf) といったプロファイリングツールがユーザ空間内で利用できるため，これを利用する．帯域制御には cgroup の blkio サブシステム [9] で提供されるインタフェースを利用する．既存の blkio サブシステムではジャーナリング I/O が考慮されずにリソース制限されているため，推定されたジャーナリング I/O 量を利用して各コンテナに割り振るリソースの制限値を補正する．

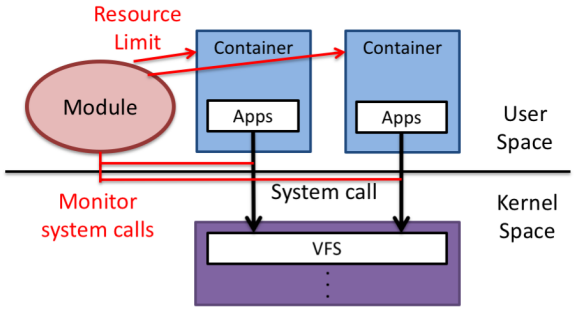


図 4: 本提案の概要

3.2 ジャーナリング I/O の推定精度

本提案のジャーナリング I/O の推定精度は，各システムコールのメタデータ I/O の呼び出しごとの偏りに依存す

る．そのため，この偏りを確認する実験をおこなう．

ext4 ファイルシステム上にて，mkdir，creat などのメタデータ I/O を発行するシステムコールを 1000 回呼び出すワークロードを実行する．1000 回としたのは，システムコール 1 回のメタデータの実際の書き込みサイズはブロックサイズである 4KiB よりも小さい場合が多いためである．これを 10 回行い，ジャーナル領域へ書き出すバイト数の平均とその分散を測定したところ，表 1 の結果を得た．

表 1: 各システムコールのメタデータ更新量

システムコール	平均 (KiB)	標準偏差 (KiB)	変動係数
mkdir	4442.4	47.5	0.011
creat	323.6	14.6	0.045
mknod	318.8	12.7	0.040
truncate	261.2	2.6	0.010
unlink	296.0	0.0	0.000
write	268.0	0.0	0.000
rmdir	327.6	4.4	0.013

変動係数が十分に小さいことから，ジャーナリング I/O の推定精度は高く望める．

3.3 帯域制御の方法

cgroup の blkio サブシステムは，ディスクの帯域制御の方法としてスロットリングと重み付け比例配分の 2 種類を提供している．スロットリングが I/O 操作数の上限を固定するものであるのに対し，重み付け比例配分は帯域が各コンテナからの I/O で消費されている時のみ，帯域制限がかかるものである．本提案では重み付け比例配分を採用し，各コンテナに対して理想の重み i をあらかじめ設定しておき，実際に割り当てる重み m を推定されたジャーナリング I/O に合わせて周期的に変更する．

N 個のコンテナが動作しているとして，コンテナ n について，ある期間に発行されたジャーナリング I/O 量を J_n，それ以外の I/O 量を C_n，帯域制限を再設定後のジャーナリング以外の見込み I/O 量を P_n とすると，$P_n = \frac{m_n}{i_n} C_n$ であり，全 I/O に対する理想配分の割合 I_n は

$$I_n = \frac{J_n + \frac{m_n}{i_n} C_n}{\sum_k (J_k + C_k)} \tag{1}$$

と表せる．よって，次の期間に割り当てる重み m_n は

$$m_n = \frac{I_n(\frac{\sum_k (J_k + C_k)}{i_n}) - J_n}{C_n} \tag{2}$$

となる．しかし，(1)(2) 式は C が理想の重み i を割り当てられた場合の値であることが前提の式となっている．そこで，直前の期間で割り当てられた重み m'_k の環境下でおこなわれたジャーナリング以外の I/O C'_k を用いて 1) 式を表現すると

$$I_n = \frac{J_n + \frac{m_n}{m'_n} C'_n}{\sum_k (J_k + \frac{i_k}{m'_k} C'_k)} \tag{3}$$

となる．また，m_n は

$$m_n = \frac{I_n(\frac{\sum_k (J_k + \frac{i_k}{m'_k} C'_k)}{i_n}) - J_n}{C'_n} \tag{4}$$

となる．(3)(4) 式は，任意の周期にておこなわれたリソース制限内の I/O 量と推定されたジャーナリング I/O 量に対して適用することができる．例外パターンとして，$C_n = 0$ である場合は $m_n = i_n$ とする．これは，m_n で制限することのできる I/O は C_n に限られ，リソース制限内にて I/O が発行されていない状況では制限をかけることができないため，理想値を設定している．

実際のクラウド環境では，アイドルなコンテナが存在するケースも多い．このような場合，I/O をおこなうコンテナの重みは全体に対して相対的に小さく設定される．しかし，先にも述べたとおり本提案では比例配分方式を採用しており，I/O をおこなうコンテナ同士でのみ帯域を分け合うため，アイドルなコンテナが存在しても，ディスクの帯域全体を利用することが可能である．

4. 実装

本提案を Linux 4.4 に実装した．システムコールのモニタリングには perf を利用し，帯域制御には cgroup の blkio サブシステムで提供されるインタフェースを利用する．

4.1 システムコールのモニタリング

perf はシステムコール perf_event_open を利用してプロファイリングをおこなうサブコマンドを複数提供している．perf-stat コマンドでは，sys_enter_mkdir などのトレースポイントを指定して，cgroup ごとに呼び出し回数をモニタリングすることができる．しかし，perf-stat ではシステムコールの引数が取得できない．例えば，open システムコールでは引数 flags に O_CREAT が設定されている場合とされていない場合で，ファイルの生成の有無が変わり，ジャーナリング I/O 量が異なる．perf-stat では以上のケースに対応できない．

そこで，システムコールの引数を記録できる perf-record コマンドに着目する．現状の perf-record ではプロセス番号は出力するものの，そのプロセスが属する cgroup 名は出力しない．そこで perf-record をもとに，cgroup ごとにシステムコールの呼び出しを検知するコマンドを実装した．現状では perf-record 同様に一度ディスクにログファイルを書き込んでいるが，この書き込みはディスク I/O が

ビジー状態である場合で全体の 100 分の 1 以下と非常に小さく，全体の性能にはほとんど影響を及ぼさないことが確認できている．

4.2 cgroup を利用した帯域制御

cgroup の blkio サブシステムでは，各 cgroup のディスク I/O を制限するインタフェースや，これまでに行われたディスク I/O の合計を IOPS と バイト単位にて取得するインタフェースを提供している．

各コンテナへの重みの設定には，cgroup の blkio サブシステムの blkio.weight を利用する．blkio.weight には 10 以上 1000 以下の整数値を設定することができ，(4) 式から算出される重みの値が 10 未満，または 1001 以上になることも考えられる．その場合には，10 未満であれば 10 を，1001 以上であれば 1000 を設定する．各コンテナ由来の I/O の取得には，cgroup が生成されてからの合計 I/O がバイト単位で記録されている blkio.io_service_bytes_recursive を利用する． これは，ジャーナリング I/O がバイト単位で推定されるためである．重みの再設定の周期は 5 秒とする．これは ext4 のジャーナリングの周期実行と同じ間隔である．周期実行に合わせることによって，重みを算出するのに使用する cgroup で観測された I/O の期間の秒数と，ジャーナリング I/O の期間の秒数を一致させることができる．

4.3 制約

4.2 節にて利用するとした cgroup ごとの I/O 量を取得するインタフェースだが，長い間に渡り fsync や syncfs が呼ばれずに，ライトが行われず続けると，ジャーナリング I/O を差し引いても正しく測定できないケースがあることが分かっている．これは頻繁にライトが起こった場合に，dirty フラグが立ったページキャッシュの割合が一定以上に達したときに swapper が root のコンテキストで ディスク I/O を行ってしまうためである．その結果，本来ライトをしたコンテナの cgroup で観測されるはずの I/O が blkio のヒエラルキーの最も上で観測されてしまうという現象が起きる．よって，現状ではライトを大量におこなうが fsync をおこなわないワークロードでは正しく動作しない．

5. 実験

ジャーナリング I/O が発行されるマルチコンテナ環境において本提案を適用した場合に，ディスクの帯域制御が正しくおこなわれていることを確認する．

実験環境の概要を表 2 にまとめた．一点，ジャーナリング I/O を測定するために，デフォルトではデータ領域と同じパーティションに配置される Ext4 ファイルシステムのジャーナル領域を別パーティションに設定するオプションを採用している．

表 2: 実験環境

項目	詳細
Linux Kernel	Linux 4.4.0
OS	ubuntu 16.04
CPU	Intel (R) Xeon (R) X5650 2.67GHz
Core	12
RAM	4GB
HDD	500GB
Container	LXC[10]
File System	Ext4, ordered mode

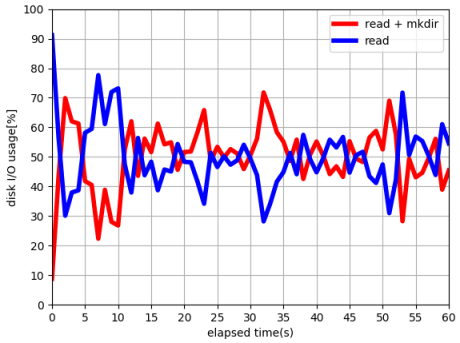


図 5: 提案手法を動作させた場合の I/O の推移

動作させるワークロードは 2 章の実験 (b) と同じで，各コンテナに設定する I/O の重みも 50:50 になるように設定した．ただし，4.3 節で紹介した問題を回避するために，mkdir を呼び出すコンテナでは 1 秒間に 1 回 fsync システムコールを呼ぶようにする．これにより，コンテナ由来の I/O を swapper に発行されることを防ぐ．図 5 は提案手法を動作させた場合の，ジャーナリング I/O を含めた各コンテナの I/O 量の推移である．動的に I/O 制限の重みを設定しているため I/O が安定していないが，I/O の比率を累積頻度グラフで表した図 6 を見ると各コンテナの I/O がほぼ同じであることが読み取れる．60 秒間の平均 I/O は，提案手法を実装していない場合ではリードのみのコンテナで 35.9%，mkdir を動作させているコンテナで 64.1% であったのに対し，提案手法を実装した場合では，49.9%，50.1% にすることができ，リソース制限本来の I/O 比率に近づけることができた．

6. 関連研究

マルチコンテナ環境において生じる性能干渉の問題を解決する研究として Multilanes[4] や IceFS[3] が挙げられる．これらは各コンテナごとにファイルシステムを分割することによって性能隔離を実現している．Multilanes ではコンテナごとにファイルシステムを単一のファイルに割り当て，その上にゲストファイルシステムをマウントし，各

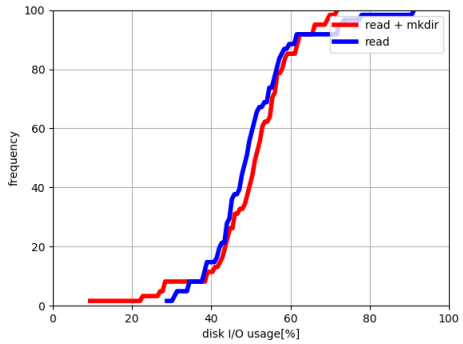


図 6: I/O 比率の累積頻度グラフ

コンテナに独立した I/O スタックを提供する．これによりコンテナ間で共有されていたホストファイルシステムのデータ構造をコンテナごとに分割し，ロックの競合を削減する．IceFS では 各コンテナごとに Cube と呼ばれるファイルシステム抽象を提供し，Cube 間では互いにデータ構造を参照できないようにすることによって，性能隔離と耐障害性を実現している．また，ジャーナリングについても，各 Cube ごとにおこない，コンテナ間の隔離を実現している．以上の研究では，コンテナ間での性能隔離を達成しているものの，既存のファイルシステムレイヤーに対して大きな変更を加えている．本研究では既存のカーネルには変更を加えずに性能隔離を実現する手法を提案した．

Split-Level I/O Scheduling[11] では， Complete Fair Queueing Scheduler (CFS)[12] をはじめとする既存の Linux の I/O スケジューラがブロックレベルの情報のみでスケジューリングをおこなうことが原因で設定された優先度通りに動作しなかったり，I/O スループットが低下したりする点を指摘している．そして，ブロックレベルの情報だけでなく，システムコールレベルとページキャッシュレベルの情報を考慮してスケジューリングをおこなうフレームワークを提案している．その中で，カーネル内のスレッドによって各プロセスのジャーナリング I/O が集約されている問題に触れている．

現在のジャーナリングは各プロセスからのジャーナルを 1 つのトランザクションという単位にシリアライズしており，あるプロセスがおこなう fsync が，他のプロセス由来のジャーナルとそれに関連するページキャッシュを含めてディスクに書き込むため同期操作であるにもかかわらず大きな遅延が生じるという問題がある．iJournaling[13] では，fsync がおこなわれた際，fsync 対象ファイルのメタデータのみを既存のジャーナル領域とは別の領域に確保し，個別にジャーナリングをおこなうことによって，それ以外のファイルの更新がディスクに書き込まれることを回避し，遅延を低減している．

7. まとめ

ファイルシステムのジャーナリング I/O がコンテナごとのリソース制限機能を提供する cgroup で捕捉できないため，ディスクの帯域制御が正しくおこなわれない．以上の問題をカーネルに改変を加えることなく解決する手法として，各コンテナのシステムコールの履歴からジャーナリング I/O を推定し，推定を考慮してディスクの帯域制御を行う手法を提案した．

実験では，2 つのコンテナを立ち上げ，両者のディスク帯域の理想配分を 50:50 に設定し，一方ではジャーナリング I/O を大量に発行し，片方ではリードのみを実行し，本提案が実装されていない場合とされた場合で比較した．本提案が実装されていない場合で，ジャーナリング I/O 込みのディスク I/O の割合がジャーナリング I/O ありのコンテナでは 64.1% ，なしのコンテナでは 35.9% のスループットであったのに対し，本提案を実装した場合では実装後と 49.9%，50.1% のスループットで，制限通りの I/O 比率を達成することを確認した．

謝辞

本研究は JSPS 科研費 JP16K00104 の補助を受けています．

参考文献

[1] : Amazon Web Services, `https://aws.amazon.com/jp/`.
[2] : google cloud platfrom, `https://cloud.google.com/`.
[3] Lu, L., Zhang, Y., Do, T., Al-Kiswany, S., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Physical Disentanglement in a Container-based File System, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Berkeley, CA, USA, USENIX Association, pp. 81–96 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2685048.2685056⟩ (2014).
[4] Kang, J., Zhang, B., Wo, T., Hu, C. and Huai, J.: MultiLanes: Providing Virtualized Storage for OS-level Virtualization on Many Cores, *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, Berkeley, CA, USA, USENIX Association, pp. 317–329 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2591305.2591337⟩ (2014).
[5] Hagmann, R.: Reimplementing the Cedar File System Using Logging and Group Commit, *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, New York, NY, USA, ACM, pp. 155–162 (online), DOI: 10.1145/41457.37518 (1987).
[6] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. and Vivier, L.: The new ext4 filesystem: current status and future plans, *Proceedings of the Linux Symposium*, Vol. 2, pp. 21–33 (2007).
[7] : Control Groups, `https://www.kernel.org/doc/`

Documentation/cgroup-v2.txt.
[8] : The flexible I/O (FIO) benchmark, https://github.com/axboe/fio.
[9] : cgroup blkio subsystem, https://www.kernel.org/doc/Documentation/cgroup-v1/blkio-controller.txt.
[10] : Linux Container (LXC), https://linuxcontainers.org/.
[11] Yang, S., Harter, T., Agrawal, N., Kowsalya, S. S., Krishnamurthy, A., Al-Kiswany, S., Kaushik, R. T., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Split-level I/O Scheduling, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 474–489 (online), DOI: 10.1145/2815400.2815421 (2015).
[12] : Complete Fairness Queueing, https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.
[13] Park, D. and Shin, D.: iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call, *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, USENIX Association, pp. 787–798 (online), available from ⟨https://www.usenix.org/conference/atc17/technical-sessions/presentation/park⟩ (2017).