Vol.2021-OS-151 No.10
2021/3/2

遅延レイヤ取得による高互換コンテナ起動高速化手法

五反田 正太郎[1]　品川 高廣[1]

概要：実行環境の多様化や DevOps 等の近年の開発手法の普及により, アプリケーションの開発・管理・運用を統一的かつ迅速に行う手法として, コンテナの活用が進んでいる. コンテナイメージは, そうしたコンテナのニーズを満たす要である一方, レジストリからイメージを取得する際に発生する時間がアプリのデプロイ・起動の遅延を増大させる最大のボトルネックともなっている. これに対し, コンテナイメージの遅延取得と呼ばれる手法はイメージの機能や汎用性を損なわずに起動の高速化を達成できる手法であり研究が進んでいる. しかし既存のイメージ遅延取得手法はアプリケーションや既存レジストリ, コンテナランタイム等, 既存のコンテナのエコシステムとの間に互換性の問題があり, 導入や運用における障壁も大きい. 本研究では, コンテナイメージの仕様及びイメージ管理のデファクトスタンダードであるオーバーレイファイルシステムを最大限活かし, 高い互換性を持つイメージ遅延取得によるコンテナ起動高速化手法を提案する.

1. はじめに

コンテナは, アプリケーションの実行基盤としてクラウドからラップトップ,IoT 機器等幅広い領域で活用が進んでいる. そうしたコンテナの人気の一つの要因と成るのが Docker に端を発するコンテナイメージによるアプリケーション管理の容易さにある. コンテナ型仮想化は一般には OS の提供する機能により実現される, プロセス単位での軽量な仮想化であるが, コンテナイメージはゲストのファイルシステムや環境変数等, アプリケーションの実行に必要な「環境」を統一されたフォーマットでパッケージ化させることで, 環境構築やデプロイの手間を減らし, アプリケーション開発・運用の効率を向上させている.

コンテナイメージはアプリケーション実行の為の環境を提供する要でもある一方, そのサイズの大きさがデプロイ・起動の大きなボトルネックとなっている. Hater らが 2016 年に行った DockerHub の主要イメージを用いた調査により, コンテナイメージをイメージレジストリから取得しローカルに展開する時間, すなわちイメージの pull にかかる時間は起動時間の 76%を占める事が示されている [1]. コンテナイメージはゲスト毎にファイルシステムを提供する役割を担っており, Ubuntu や CentOS 等のディストリビューションや言語ランタイムの為のファイル等, 大量のデータを含んでいる事がその主な原因である.

コンテナイメージは 1 度 pull すればイメージを再利用することが可能であり, イメージの pull にかかる時間は償却可能という考えもある. しかし, コンテナの利用背景上 pull にかかる時間は大きな問題となる. まず, コンテナを用いる様な昨今の開発においてはアジャイル開発や継続的デリバリー等の手法が取り入れられられることも多く, 頻繁にイメージの更新と pull が発生する. データセンターでの活用を考えると, 多くのノードでそれぞれ pull を行う必要もある他, サーバーの更新やオートスケール等により, イメージの存在しないノードでの起動も多い. 前者は短いメンテナンス期間で以降を完了させる必要がある他, 後者では負荷の急上昇に対応するため, 迅速な起動が求められる.

一方, ラップトップ等で日々の開発に使うソフトウェアをコンテナとして使う場合は, 必ずしも最新版のイメージに追従する必要は無いが, 今や多くのアプリがコンテナ化されており, それぞれイメージの pull が発生するとなると, その影響も大きい. また, コンテナイメージはときに数 GB となることもありストレージを圧迫する. そのため, 使い終えたイメージを削除し再度必要となる場合に pull するケースもあり, この問題はストレージ容量の小さいラップトップやシングルボードコンピュータでは頻繁に発生する. また, 昨今のインターネット回線の帯域の逼迫により, 回線次第ではたった一つのイメージの取得に数時間かかる場合もあり, 開発を長時間ストップさせる要因となる.

こうした背景もあり, コンテナのイメージ取得にかかる時間はアプリケーションのデプロイ・起動にかかる時間のボトルネックとなっており, その解消のためこれまでに多くの手法が考案されている. Alpine Linux 等の小型のコンテナ特化ディストリビューションをベースとする方法は Docker 公式も推奨する方法である [2] ほか, 自動でファイ

[1] 東京大学
The University of Tokyo

ⓒ 2021 Information Processing Society of Japan

ルを削除してイメージサイズを縮小する方法もある [3]. しかし, 前者は独自のライブラリやパッケージシステムを採用しており, 互換性から既存アプリに適用できない場合も多く, 後者はファイルを削除することにより機能性を悪化させ, 最悪のケースではアプリが正しく動作しない場合もある.

イメージの小型化に対し,Harter らの提案した Slacker [1] を始めとするイメージの遅延取得をベースとしたコンテナ起動高速化手法は, ディストリビューションを限定せず, ファイルを削除して機能性を下げることもない手法である. イメージ遅延取得の関連研究は多く存在するものの, 動作環境を限定するもの [1, 4–7], 対応アプリを限定するもの [8], 特殊なフォーマットを用いているため利用可能なレジストリを限定するもの [9–11] など, いずれも互換性に問題がある. コンテナは幅広いユースケースにおいて利用されているため, 現実的に広く問題解決を行うには, 高い互換性を持った手法が求められる.

本研究では, 起動時に必要な最小限のファイルのみを最初に取得することで起動を高速化しつつ，各種仕様の範囲内で不必要なファイルの取得を回避する仕組みを導入することで，互換性の高いコンテナ起動高速化手法を実現する. 遅延取得の単位は既存研究 [8] 同様レイヤとすることで, 既存のイメージやレジストリとの互換性を保つ. 起動時に取得する起動レイヤの作成は，ファイルアクセスの動的トレースを元におこない，静的解析と比べて汎用性の高い手法を実現する．一方，既存研究では遅延取得の実現に共有ライブラリの置き換えやオーバーレイファイルシステムの未定義動作などを利用しているため，対応できないアプリケーションが存在していたのに対し, 本研究ではオーバーレイファイルシステムの実装に対し最小限の変更を加えることで, アプリケーションとの互換性を保ちつつレイヤの遅延取得を実現することを可能にしている.

実験では, 幅広いアプリケーションの起動時間と, ベンチマークによるファイルシステムのオーバーヘッドを評価し, 少ないオーバーヘッドで,1Gbps 環境では 4.9 倍,100Mbps では 10 倍,10Mbps では 15 倍の起動時間の改善という結果を得た.

本稿の構成を述べる. 2 章では研究背景としてイメージを中心としたコンテナを整理しつつ, 既存手法における問題点について整理する. 3 章では高い互換性を保ちつつレイヤの遅延取得によりコンテナ起動を高速化するシステムを提案する. 4 章で提案手法の評価を行った上,5 章でまとめとする.

2. コンテナとイメージ

コンテナは元来は OS により提供される機能から成るプロセス単位での仮想化・分離機構である. しかし Docker に端を発する現在のコンテナでは, 分離機構に加えイメージやネットワーク等, アプリケーションの管理に必要な周辺事項についても機能の整備や共通仕様の策定が進められており, それによりクラウドからラップトップ,IoT 等, 幅広い環境で統一的かつ簡易にアプリケーションの管理を可能としている.

2.1 レイヤとオーバーレイファイルシステム

コンテナを活用する際, 各アプリケーションはコンテナイメージとしてパッケージ化され管理される.

技術的にはコンテナイメージは, ゲストのルートファイルシステムを構成するファイル群とアプリケーション実行の為に必要な実行コマンドや環境変数等のメタデータで構成されるが, 統一的かつ簡易な管理を可能とするため, イメージの構造は仕様により定められている [12].

イメージの仕様のうち, 特に特徴的なのがレイヤである. イメージはレイヤをファイル単位でのファイルシステムの差分として，1 つ以上のレイヤの集合から構成される．また，差分を適用し, イメージを 1 つのファイルシステムとしてゲストに提供するためには, 複数の選択肢が存在するが, オーバーレイファイルシステムという仕組みが用いられる事が標準的となっている.

オーバーレイファイルシステムは,ext4 等の通常のファイルシステムの複数のディレクトリおよびそこに含まれるファイルを積層・透過する様にマージする機能を持ち, ファイルに変更を加えた際は, 最上位の書き込み可能レイヤ（ディレクトリ）にファイルをコピーしてから変更を加える等, ファイル単位での CoW の機能を持ち, レイヤの仕様のベースともなっている.

オーバーレイファシステムは概念的にはユニオンファイルシステムと呼ばれる事もあるディレクトリを透過的にマージする機能をもつ特殊なファイルシステムだが, 現在のコンテナでは,Linux カーネルのモジュールとしての実装である overlayfs と, FUSE での実装である fuse-overlayfs の 2 つが，Docker やその互換ランタイムである Podman で採用されている.

2.2 イメージのデプロイとレジストリ

コンテナイメージは, その構成のみならず配布方法についても仕様により統一化が図られている [13]. イメージ配布の仕様は, コンテナイメージの保管・配布を行うコンテナイメージレジストリの API について定めている. コンテナイメージのレジストリは,DockerHub 等の SaaS に加え, OSS として多くの実装が開発され, セルフホストも可能となっている. レジストリには Web UI やセキュリティチェック機能を備えるものや P2P によりイメージ配布を最適化したものなど様々な実装が存在するが, 共通の仕様に準拠しているためそれぞれのユースケースにあった柔軟な選択が可能となっている.

ⓒ 2021 Information Processing Society of Japan

2.3 イメージ取得を伴うコンテナ起動高速化

イメージ遅延取得によるコンテナ起動高速化手法には多くの関連研究がある．この手法の先駆けである Slacker を初め, 当初の手法は NFS 等の専用サーバを必要するものやイメージの分散協調管理を行う等, クラスタ内部での高速化に効果を限定しており, その適用範囲は限定的だった．これに対し, 近年の手法はクラスタを前提とせずレジストリを直接利用可能な手法が複数提案されているものの, 十分な選択可能性を持つとは言えない．

Li らが提案した DADI [11] はレジストリの仕様に従っているもの, 独自のフォーマットや圧縮形式を用いて遅延取得を行う方法であり, あくまで BLOB の配布サーバとしてレジストリを利用しているに過ぎず, 作成したイメージは従来のコンテナランタイムで動作させる事もできない．

Google の開発した CRFS [9] と呼ばれるファイルシステムは,stargz 形式という形式を用いることでファイル単位でのレイヤの遅延取得を可能としたが, オーバーヘッドにより通常のコンテナよりも起動が遅くなる事例が示されており, 起動高速化効果を期待することは難しい．stargz 形式を改良しつつイメージの互換性を持つ estargz 形式 [10] というものも存在するが, その活用にはレジストリの仕様としてはオプショナルの API が必須であるほか, イメージの仕様に定めれれてないデータ構造に依存しているため, レジストリに関する複数の研究で提示されている様な最適化によりその機能を消失する可能性が極めて高い．

独自のフォーマットを用いずレイヤを遅延取得の単位として用いる FogDocker という手法もある [8]．この手法は, 既存イメージのレイヤ群の上に, 起動に必要なファイルとイメージ中のファイルリストを含むレイヤを作成することで, 起動時にはこのレイヤのみ取得し起動の高速化を図るという手法である．しかし，遅延取得の管理のために共有ライブラリによる open や fopen 等の C のライブラリコールのフックを利用しているため, C 言語でも静的リンクされたバイナリや Go や Rust 等独自に生成されたバイナリのアプリケーションでは適用できない上, Linux カーネルモジュールの未定義動作に依存するという問題もある．

3. 提案手法

コンテナは, イメージの取得が起動時間のボトルネックとなっているため, いかにして起動時間を短縮するかが問題となると同時に, コンテナは様々な場面や実装と共に活用が進んでいるため, 現実的に適用可能な方法とするためには, 既存の仕様やデファクトスタンダードとの互換性が重要である．

そこで本研究では, 既存手法の FogDocker と同じく仕様により定義されているイメージ・レイヤ単位でイメージの遅延取得を行うことで, レジストリの制限なく適用可能であると同時に, レイヤ管理のデファクトスタンダードをベースに極力変更を少なく独自のファイルシステムを開発することで, アプリケーションの制限なく適用可能なコンテナ起動高速化手法を提案する．

本章では, まずはじめに提案システムの概要を示した上で, レイヤ単位での遅延取得の互換性の問題を洗い出して解決策を提示する他, 手法の効果と適用可能性を高める為に行った様々な工夫を解説する．

3.1 システム概要

まずはじめに, 提案するシステムの概要を解説する．本研究の提案システムは, 遅延取得に最適化されたイメージを生成するためのシステム（図 1）及びレイヤ遅延取得を可能にしたコンテナ実行システム（図 2）の 2 つのシステムから構成される．

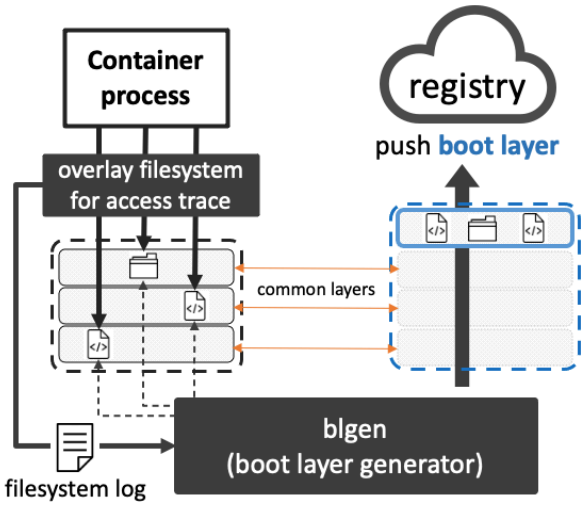

図 1 起動レイヤ生成システム

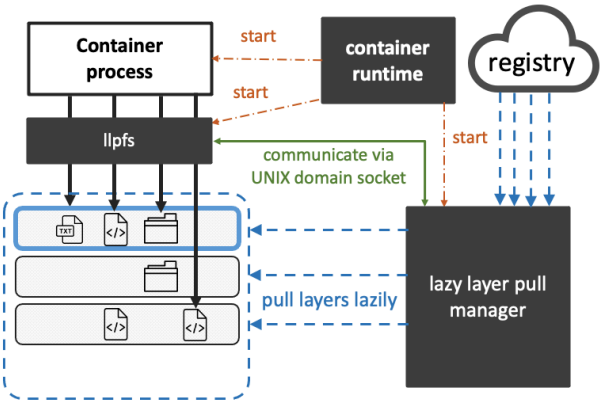

図 2 レイヤ遅延取得によるコンテナ実行システム

3.1.1 起動レイヤ生成システム

本提案手法では, 起動時に必要なファイルから構成されるレイヤ（起動レイヤ）のみを取得し, その他のレイヤは遅延取得することで, コンテナイメージの pull にかかる時間を隠蔽する．そのため, 起動レイヤ生成システムを構築し

た．起動に必要となるファイルの情報は, イメージの静的解析により得ることが困難であるため, オーバーレイファイルシステムの既存の実装に改変を加え実行時のログを出力できるようにして, 動的なトレースを行い取得する設計を採用した．なお, この際の実装は簡単のため FUSE 実装のオーバーレイファイルシステムであり, Docker およびその互換ランタイムである Podman でもサポートされている fuse-overlayfs をベースとした．

また, 起動レイヤ生成用のソフトウェア blgen (Boot Layer GENerator) も実装した．blgen は, ファイルシステムのログとベースとなるイメージから起動用レイヤと遅延起動に最適化されたイメージを生成し, それらをレジストリにアップロードする．この際, 基本的にファイルは各属性を保つようにしてコピーされるが, ファイルの所有者および所有グループに関しては, ユーザーのマッピングによる差異を吸収するような実装となっている．また, 起動レイヤは既存のイメージのレイヤの上に重ねる形で遅延取得に最適化されたイメージを構成する．そのため, 起動レイヤのみをレジストリにアップする実装となっており, レジストリにある既存のイメージの資産を活用でき, 追加でかかるコストを最小限に抑えられる．

3.1.2 レイヤ遅延取得によるコンテナ実行システム

起動レイヤを加えた遅延起動に最適化されたイメージを用いてコンテナ起動高速化を行うため, fuse-overlayfs にレイヤ遅延取得機能を追加したファイルシステム llpfs（Lazy Layer Pull File System) を実装した．より柔軟な実装とするため llpfs は直接レジストリとの通信やレイヤの取得を行わず, レイヤが必要になると UNIX ドメインソケット経由で外部の遅延レイヤ取得マネージャにリクエストを発行して完了の通知が来るまでファイルアクセスをブロックする設計となっている．なお, 本提案の実装ではこの遅延レイヤ取得マネージャは Go 言語で書かれているコンテナランタイムの goroutine として実装されている．

また, 通常コンテナを実行する際は、実行に先立ちイメージを取得してしまうため、そうしたイメージの取得を回避しつつ、llpfs 等の実装を用いてコンテナの実行を行うよう, 既存のコンテナランタイムにも実装を加えた．このコンテナランタイムとしては理論上どの実装も使えると考えているが, 今回は簡単のため Docker ではなく, その互換ランタイムである Podman をベースに実装を行った．

3.2 レイヤ単位遅延取得の問題及び解決策

提案システムの実装を進める上で問題となるのは，ターゲットとなる元のイメージ中のアクセスされたファイルのみから起動レイヤを構成するだけでは起動の高速化と互換性を両立できないという転である．

問題となるケースは主に２つある．１つは存在しないパスへのアクセス，もう一つはディレクトリエントリの読み込みである．存在しないパスへのアクセスは, 例えばコマンド実行の際のパスの探索や新規ファイルの作成の際に発生する．このケースでは, そもそも元のイメージに該当ファイルが存在しないため, ファイルが起動レイヤにファイルが存在しない．しかし, オーバーレイファイルシステムの仕組み上, より下層のレイヤにファイルが存在する可能性もあるため，１つのレイヤにファイルが存在しないだけでそのファイルが存在しないという事は判定できない．また, 各レイヤは tar.gz 形式で配布されるため, ファイルの不在を確認するためには, 全レイヤを pull する必要があり, 結果としてコンテナの起動が高速化されない．

ディレクトリエントリの読み込みでも, オーバーレイファイルシステムおよびレイヤの仕様に起因して同様に全レイヤを pull する必要がある．オーバーレイファイルシステムでは通常のファイルシステムと異なりディレクトリのエントリを指す実データが存在せず, 各レイヤの該当ディレクトリのマージとして扱われる．そのため, ディレクトリ単体を起動レイヤに含めるだけでは起動レイヤにディレクトリエントリの情報が含まれないうえ，該当ディレクトリ以下のファイルを再帰的にコピーしたとしても, その他のファイルがディレクトリに含まれない事を判断するために, 結果的に全レイヤが必要となる．

これらの問題を回避するためには, ファイルの不在を示す情報及び, 該当ディレクトリの全エントリ情報を何らかの方法で全レイヤを取得せずにオーバーレイファイルシステムに伝えられればよい．これらの情報は, ファイルの中身を含まずデータサイズは大きな問題とならないため, いかにして互換性を保ちつつ実現するかが重要となる．

当初我々は，レイヤの仕様に含まれるホワイトアウトファイルというファイルの削除を示す特殊なファイルを用いてファイルの不在を示しつつ, エントリを読み込むディレクトリは, 子孫を再帰的にコピーした上で, レイヤの仕様に含まれるディレクトリの不透過化の機能を活用して対処する手法を実装した．この方法は,Nginx や redis の様に利用ファイルが一定でかつ, エントリを読み込むディレクトリに含まれるファイルは全て利用するという様なアプリケーションに対しては十分効果的に動作した．しかし, ランダムな名前のファイルを生成するアプリでは，１ファイルずつの削除を示すホワイトアウトファイルでは対応できない他, ディレクトリに含まれるファイルが多い場合, 起動レイヤが十分小さくならず, 起動が高速化されない例も多く見られた．

そこで我々は, 全ファイルのメタデータを含めることでこの問題に対処した．メタデータを含める方法としては,JSON 等の形で１ファイルにまとめるという事も考えられるが, この方法では, パースして全データをメモリ上に保持しておくか, 必要となる毎に JSON をパースする必要が生じ, 性能上のオーバーヘッドが懸念される．そこで, 本

研究ではファイルシステムとして同じディレクトリ構造を保ったままメタデータのみをコピーし, レイヤの仕様上アプリケーション側からは確認できないように.wh..wh. をプレフィックスに持つ専用のパス (.wh..wh.llpfs) 以下に含めるという手法を採用した（図 3）. なお, アクセスされることがトレース結果から判明しているファイルについては, データの増大を避けるため重複排除する.

この方法を採用するとにより, あるパスへのメタデータアクセスは.wh..wh.llpfs/をプレフィックスとして付加する程度の少ないオーバーヘッドでのアクセスを可能とした.

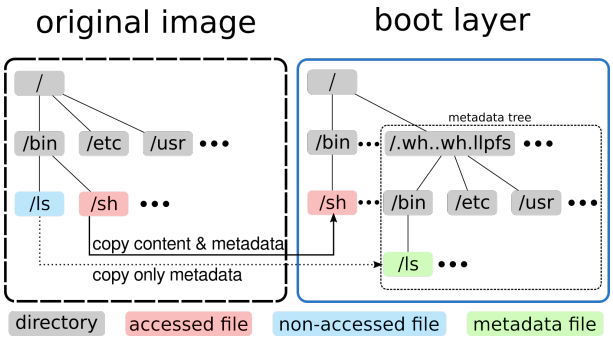

図 3 メタデータを含む起動レイヤ生成概要

メタデータのコピーの際に問題となるのが, ファイルサイズである. ファイルの所有者やパーミッション等は, コンテナの実行時におけるユーザーのマッピングを考慮し変換することを除き直接コピー可能だが, 素朴な実装では元のファイルと同じ量のデータをファイルに含めなければファイルサイズを変更できない.

そこで, 我々はレイヤの形式である tar.gz にも含む事が可能である, スパースファイルを用いることでデータ量の増大を防ぎつつファイルサイズをコピーする実装を検討した. しかし, 実装に用いたライブラリの関係でスパースファイルを tar に含める事ができなかったため, 空ファイルではなく, ファイルサイズを文字列として含めたファイルに文字列以外のメタデータをコピーし、ファイルサイズが必要なときのみ文字列をパースするようにした。

4. 実験

本研究の提案手法の評価として, アプリケーションの起動時間の評価及び提案によるオーバーヘッドの計測実験を行った. なお, 起動時間評価は表 1 に示す実機 1 で行い, オーバーヘッド評価は実機 2 で行った.

4.1 実アプリケーションのコンテナ起動時間評価

本研究の実用的な効果を評価するため,8 つの実アプリケーションを用いて起動時間の評価を行った. DockerHub [14] で公開されているイメージから 7 つの著名なアプリケーション（ubuntu, python, golang, nginx, redis, jupyter/dataseice-notebook, texlive/texlive) を選定したほか, Heroku に代表される SaaS で用いられる Web フレームワーク向けの最適なイメージ合成ツールである buildpack [15] の提供する Spring Boot のサンプルアプリケーションを対象として選定した.

評価の対象となる起動時間については, アプリケーション毎に実用に即して適切な評価を行えるように, それぞれ個別に評価方法を決定した. まず, nginx,redis,jupyter,buildpack の常駐型のアプリケーションに関しては, 最初のリクエスト応答までの時間を起動時間とした. また, texlive 及び golang は繰り返すアプリケーションとして開発の初期段階を想定し,hello world に類する単純なソースのビルドを起動時間とした. python,ubuntu(bash) も同様に hello world 相当の単純な処理をバッチ実行する時間を起動時間とした.

これらのアプリケーションのベースイメージ及び, 起動時のファイルアクセスのログを元に生成した起動レイヤサイズを表 2 に示す. 起動レイヤのサイズは元のコンテナイメージと比べて最大で 3.7%に縮小した（ubuntu). buildpack の様な特化型のイメージビルドツールを用いた際は割合は大きくなるものの, 最大でも 3 割台のデータしか必要でない事が分かった.

表 2 生成イメージ及びデータサイズ

対象アプリ名	ベースサイズ	起動レイヤサイズ
ubntu(batch)	29MB	1.0MB(3.7%)
nginx	53MB	5.4MB(10%)
redis	38MB	9.0MB(23%)
buildpack	159MB	60MB(38%)
python	355MB	17MB(5.0%)
golang	322MB	30MB(9.5%)
texlive	2252MB	109MB(4.7%)
jupyter	1486MB	116MB(7.8%)

なお, イメージの pull はレジストリの負荷や帯域の影響を大きく受けるため,QEMU/KVM によりコンテナ動作用 VM（4vCPU, 8GBRAM）とレジストリ用 VM（2vCPU, 8GB RAM）を立て,DockerHub の registry コンテナを用いてレジストリをセルフホストすることで, 安定した評価を可能とした. また,VM 間の通信帯域を, 無制限,1Gbps, 500Mbps, 100Mbps, 50Mbps, 10Mbps, 5Mbps に制限し, 帯域による影響も評価した.

まず, ケーススタディとして図 4 に示した Nginx コンテナの起動時間を帯域別に確認する. 公式の提供する Nginx イメージを通常のランタイムで動作させた際の起動時間 (baseline) と比較し, 通常のイメージに起動レイヤを重ねたイメージを用いて遅延起動を有効にしてコンテナを起動させた際の起動時間 (proposal) がどの帯域においても起動時間が短縮できている事が分かる. なお, この傾向は割合の差はあれ全てのアプリケーションで確認できた.

ⓒ 2021 Information Processing Society of Japan

表 1　実験環境

項目	実機 1	実機 2
CPU 型番	Intel(R) Core(TM) i7-9700	Intel(R) Xeon(R) E5-2603 v4
CPU 基本動作周波数	3.00GHz	1.70GHz
物理コア数 (論理コア数)	8(8)	6(6)
RAM	32GB	16G
Disk	Samsung SSD 970 EVO Plus 500GB	Samsung SSD 950 EVO Plus 256GB
OS	Ubuntu18.04(Linux 5.3.0)	Ubuntu20.04(Linux 5.4.0)

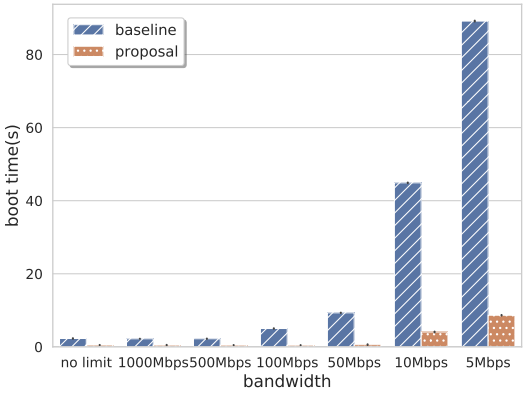

図 4　Nginx コンテナの起動時間

続いて, 図 5 に示すアプリケーションおよび帯域別の起動時間の改善率を確認する. 改善率は 通常の起動時間/提案手法による起動時間により算出している. 実験結果から, 提案手法が帯域に依らず性能向上を示していることを確認できる他, 全体として帯域が狭まるほど改善率が向上する傾向も確認できる.

これは, 帯域が広く起動時間自体が短い場合, その他のコンテナの初期化にかかる時間の割合が大きい一方, 帯域が狭まるにつれ, ダウンロードにかかる時間が支配的になるためと考えられる.

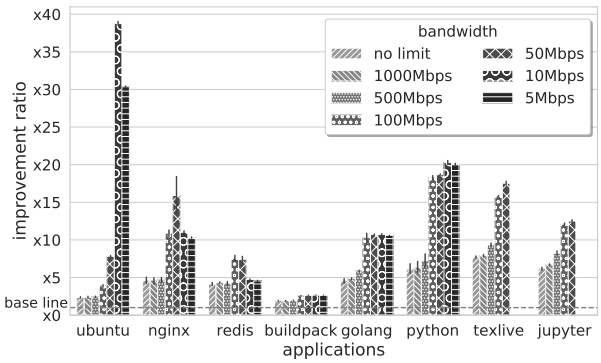

図 5　アプリケーション・帯域別起動時間改善率
通常の起動時間/提案手法による起動時間

4.2　ファイルシステムのオーバーヘッド評価

続いて, 本研究の提案手法におけるオーバーヘッドを評価する為,sysbench を用いて性能評価を行った. 実験は, 表 1 の実機 2 に示すマシンを使用し, DockerHub の提供する MySQL のイメージを予めローカルに取得した状態で各オーバーレイファイルシステムを用いてコンテナを動かし, 同一マシン上で sysbench の oltp_read_write ワークロードをそれぞれ 5 回ずつ実行した. sysbench はバージョン 1.0.20 を使用し, 基本的にはデフォルトの設定を用いて実験を行った. ただし, スレッド数に関しては, 性能がほぼ飽和するように, コア数 (6)×4 の 24 スレッドで動かすように修正した. 評価対象のオーバーレイファイルシステムとしては, 本研究の提案実装とそのベース実装である fuse-overlayfs に加え, 参考として Linux カーネルのモジュールとしての実装である overlayfs を用いて評価を行った.

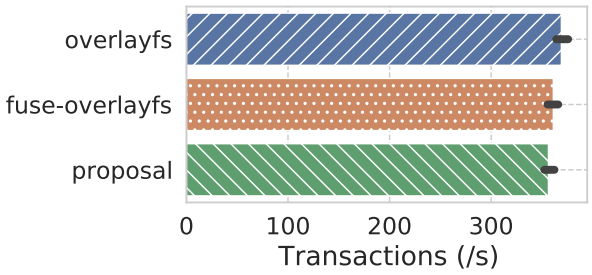

図 6　sysbench によるトランザクション数スループット

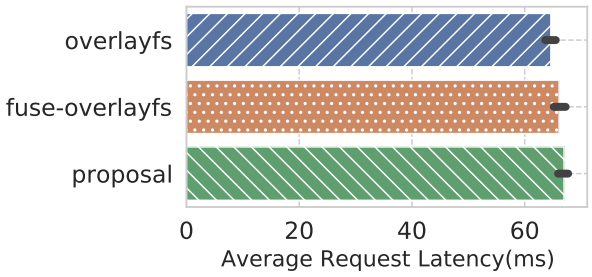

図 7　sysbench による平均リクエストレイテンシ

実験結果として, トランザクション数スループットを図 6 に, 平均リクエストレイテンシを図 7 に示す. ベース実装

である fuse-overlayfs と比較し提案手法ではスループットについては 1.3%の低下, 平均リクエストレイテンシについては 1ms 以下で 1.3%の増大と, オーバーヘッドは低く抑えられることが分かった.

5. 関連研究

本論文では, イメージ取得時の通信帯域や解凍処理を主なボトルネックとして解決を行ったが, コンテナイメージレジストリでの処理がボトルネックとなる場合もある. Anwar ら [16] はレジストリのアクセス傾向解析を行い, その知見を元にした効率的なキャッシュを導入により各リクエストへのレスポンスを高速化した.

また, レジストリに保存されるデータ量は日々増大することから, そのストレージコストも莫大なものとなっており, その削減の為の研究も進んでいる. とりわけ, レイヤーの分解, 再構築等によりレイヤー間でのレイヤー重複を削減するという手法に基づく研究は複数行われており [17–19], ストレージの重複による容量圧迫の問題を解決するために今後普及が見込まれる.

これらのコンテナイメージレジストリを対象とした研究は, 既存のコンテナの仕様をベースとしているため, コンテナ起動に対し専用のクラスタや独自のフォーマットのイメージを用いる既存研究とは両立可能ではない. 一方で, 本研究は仕様の範囲内での手法であるためレジストリに関する研究とは修正なしに両立可能である.

6. まとめ

コンテナのイメージ取得を伴う起動レイテンシの問題がある. それを解決する手法としてイメージの遅延取得は, イメージの小型化等と比べアプリケーションの機能を落とさないものの, 動作環境やアプリ, レジストリ等と互換性の問題があり, その適用には制限があった. そこで, 本研究ではオーバーレイファイルシステムをベースにレイヤ単位でのイメージ遅延取得を行うことで, 高い互換性を持つ手法を提案した.

実験では, 多様な実アプリケーションの起動時間評価およびファイルシステムのオーバーヘッドの評価を行い, オーバーヘッドは少なく抑えつつ, 1Gbps 環境では 4.9 倍,100Mbps では 10 倍,10Mbps では 15 倍の起動時間の改善という結果を示し, 互換性を保ちつつも高い起動時間削減効果を達成できることを確認した.

参考文献

[1] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: *Proceedings of 14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA.
[2] : Best practices for writing Dockerfiles. `https://docs.docker.com/develop/develop-images/dockerfile_best-practices/`.
[3] : DockerSlim. `https://dockersl.im/`.
[4] Hardi, N., Blomer, J., Ganis, G. and Popescu, R.: Making containers lazy with Docker and CernVM-FS, *Journal of Physics: Conference Series*, Vol. 1085, No. 3 (2018).
[5] Spillane, R. P., Wang, W., Lu, L., Austruy, M., Rivera, R. and Karamanolis, C.: Exo-clones: Better Container Runtime Image Management across the Clouds, *Proceedings of 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO (2016).
[6] Du, L., Wo, T., Yang, R. and Hu, C.: *Proceedings of 2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*.
[7] Zheng, C., Rupprecht, L., Tarasov, V., Thain, D., Mohamed, M., Skourtis, D., Warke, A. S. and Hildebrand, D.: *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA.
[8] Civolani, L., Pierre, G. and Bellavista, P.: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, New York, NY, USA.
[9] : CRFS. `https://github.com/google/crfs`.
[10] : Stargz Snapshotter. `https://github.com/containerd/stargz-snapshotter`.
[11] Li, H., Yuan, Y., Du, R., Ma, K., Liu, L. and Hsu, W.: DADI: Block-Level Image Service for Agile and Elastic Application Deployment, *Proceedings of 2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, pp. 727–740 (2020).
[12] : Open Container Initiative Image Format Specification. `https://github.com/opencontainers/image-spec`.
[13] : Open Container Initiative Distribution Specification. `https://github.com/opencontainers/distribution-spec`.
[14] : Docker Hub. `https://hub.docker.com/`.
[15] : Cloud Native Buildpacks. `https://buildpacks.io`.
[16] Anwar, A., Mohamed, M., Tarasov, V., Littley, M., Rupprecht, L., Cheng, Y., Zhao, N., Skourtis, D., Warke, A. S., Ludwig, H., Hildebrand, D. and Butt, A. R.: Improving Docker Registry Design Based on Production Workload Analysis, *Proceedings of 16th USENIX Conference on File and Storage Technologies (FAST 18)* (2018).
[17] Littley, M., Anwar, A., Fayyaz, H., Fayyaz, Z., Tarasov, V., Rupprecht, L., Skourtis, D., Mohamed, M., Ludwig, H., Cheng, Y. and Butt, A. R.: Bolt: Towards a Scalable Docker Registry via Hyperconvergence, *Proceedings of 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)* (2019).
[18] Zhao, N., Albahar, H., Abraham, S., Chen, K., Tarasov, V., Skourtis, D., Rupprecht, L., Anwar, A. and Butt, A. R.: DupHunter: Flexible High-Performance Deduplication for Docker Registries, *Proceedings of 2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020).
[19] Skourtis, D., Rupprecht, L., Tarasov, V. and Megiddo, N.: Carving Perfect Layers out of Docker Images, *Proceedings of 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA (2019).

ⓒ 2021 Information Processing Society of Japan