仮想化環境におけるクラウド・アプリケーションのための カーネル内実行フレームワーク

大須賀 敦俊[1]　縣 直道[1]　窪田 貴文[1]　河野 健二[1]

概要：仮想化を用いたクラウド環境では，ひとつの仮想マシン (VM) でひとつのアプリケーションを動作させる形態が一般的になっている．この特徴を活用し，クラウド環境に特化した OS の設計・開発が行われている．これらのクラウド特化型 OS は，ハイパーバイザと重複した機能をゲスト OS から取り除いた，軽量かつ省メモリな実装となっている．これらの OS は新規の設計となっているため，1) 既存 OS と同等の機能を利用できるとは限らないという機能性，および 2) 既存 OS とは別個に保守が必要になるという保守性の面において不十分である．本論文では，既存の OS に対する改変を最小限に留めつつ，クラウド特化型 OS と同等の性能を実現する方法として，既存アプリケーションを既存のカーネル内で実行できるようにするフレームワークを提案する．この方法では，プロセスによるオーバーヘッドを削減することで，クラウド特化型 OS に近い軽量化を行うことができる．同時に，1) 実装において既存 OS の機能をそのまま利用できること，2) 既存 OS コミュニティによりコア機能が保守されることを期待できる．Linux を対象にフレームワークの実現を行った．本フレームワークの有用性を示すため，LevelDB というキーバリューストアをカーネル内で実行したところ，プロセスとして実行した場合と比較して，スループットが最大 1.46 倍となった．

キーワード：クラウド・コンピューティング，仮想化，ライブラリ OS

1. はじめに

ハイパーバイザによる仮想マシン (VM) は，クラウド環境の基盤技術としてよく用いられている．VM 間の隔離はハイパーバイザによって実現されており，ある VM の不具合が同一ハイパーバイザ上の他の VM に影響を及ぼすことがない．さらにクラウド環境では，ひとつの VM でひとつのアプリケーションを動作させる形態が一般的となっており，VM 内で動作するゲスト・オペレーティングシステム (OS) は，既存 OS が提供するプロセス間の隔離を提供する必要がない．その結果，ゲスト OS 自体が特定のアプリケーション専用となるため，アプリケーションからカーネルを保護する必要もなくなる．

この特徴に着目し，クラウド環境に特化した OS の設計・開発が行われている [1], [2], [3], [4], [5]．これらのクラウド特化型 OS は，クラウド環境に特化した新規の設計となっており，ハイパーバイザと重複する機能を取り除くことで軽量化を行なっている．しかしながら，従来のクラウド特化型 OS では， 1) 既存 OS と同等の機能を利用できるとは限らないという機能性の面で不十分であり，2) 既存 OS とは別個に保守が必要になるという保守性の面において不十分である．

本論文では既存 OS に対する改変を最小限に留めつつ，クラウド特化型 OS と同等の性能を実現する手法を提案する．提案手法では，ゲスト OS ではプロセスとゲスト OS 間の隔離が不要であることに着目し，既存のアプリケーションを既存の OS カーネル内で実行できるようにする．具体的には，既存アプリケーションをローダブル・カーネル・モジュール (LKM) として動作させる．

本手法では，クラウド特化型の OS と同等の軽量化を実現しつつ，高い機能性と保守性を達成することができる．まず，カーネル内で動作する設計となっているため，プロセスが不要となり，クラウド特化型 OS と同等の軽量化が期待できる．次に，既存 OS をベースとした実装となっているため，提案手法の実装に既存 OS の機能を用いることができ，高い機能性の達成が容易である．最後に，ソースコードのほとんどの部分が既存 OS と共通となるため，共通部分の保守は既存 OS の開発・保守コミュニティによって行われることとなり，高い保守性を保つことができる．

ただし，原理上既存 OS の機能のほとんどを使用できる

1 慶應義塾大学
Keio University

ⓒ 2017 Information Processing Society of Japan

といっても，既存 OS カーネルの内部 API とユーザアプリケーション向けの API には違いがあるため，これを吸収する必要がある．例えば，ユーザアプリケーションはシステムコールを用いてカーネルの機能を呼び出すことができるものの，LKM 内からシステムコールを用いることはできない．本フレームワークでは，この問題を標準ライブラリのレベルで解決する．

提案フレームワークの標準ライブラリは，システムコール相当の処理をカーネルのカーネル内部から利用可能な関数を用いて実装する．Linux 4.8 向けの実装では既存の標準ライブラリをベースにしており，システムコールハンドラを利用するようにする，Virtual File System (VFS) の関数を利用するようにする，カーネルの内部 API を使用するようにする，これらの組み合わせで実装するという 4 通りの変更を加えている．

提案フレームワークを Linux 4.8 を対象に実装し，その有用性を確認するため，LevelDB 1.19 というキーバリューストアをカーネル内で動作するように変更した．その結果，Linux 上でプロセスとして動作する LevelDB と比較して，スループットが最大 1.46 倍に向上した．また，LevelDB [6] のコード行数が 2 万行に対し，コードの変更は 100 行以下に抑えられた．

本論文の構成は，次のとおりである．2 章では，関連研究について説明する．3 章では，提案フレームワークの設計について述べる．4 章では，提案フレームワークの実装について説明する．5 章では，提案フレームワークの有用性を評価する．6 章では，本論文のまとめを述べる．

2. 関連研究

既存のクラウド特化 OS としては，Unikernels [1]，OSv [2]，EbbRT [3] などがある．これらのクラウド特化 OS では，クラウドの性質を活用したパフォーマンスの向上を実現している．しかしながら，OS 自体が新規に設計されているため，機能に関しては汎用 OS と比較して限定的であり，また，OS 自体の保守が必要になる．提案フレームワークでは，ハイパーバイザと重複する機能の排除によるパフォーマンス向上，汎用 OS の提供する大半の機能の提供，フレームワーク自体の保守の易化を同時に実現することを目指す．

OSv は，クラウド環境に特化した OS であり，単一アプリケーションの動作が行なえれば十分であるということ，ハイパーバイザにより VM 間が隔離されることを利用してプロセスによる保護を提供しない．これにより，プロセスのオーバーヘッドを削減している．OSv では Linux API の多くを提供しており，既存アプリケーションをほぼ変更なしに実行できる．また，ゼロコピーのネットワーク API を追加で提供しており，それを用いるようにアプリケーションを変更することでより高いパフォーマンスを期待できる．OSv はクラウド環境に特化した新規の設計となっており，例えばスピンロックの排除によって Lock Holder Preemption [7] の問題を解決している．しかし，新規に OS を設計しているため，OS 自体の保守を既存 OS とは別に行う必要があり，保守性に問題がある．

Unikernels, EbbRT は，クラウド環境に特化した OS として，アプリケーションにライブラリ OS をリンクする設計を採用している．この設計ではプロセスによる保護を提供しないため，プロセスのオーバーヘッドを削減できる．加えて，OS はそのアプリケーション専用のライブラリとなるため，アプリケーションに特化した最適化が容易である．しかし，OS を新規に開発する必要があるため，既存の OS と同等の機能を提供することが難しい．実際，Unikernels の実装の 1 つである Mirage では既存 OS との API 互換性は提供しておらず，アプリケーションを Mirage 向けに実装する必要がある．EbbRT は，この設計の OS の開発を容易にするためのフレームワークを提供することでこの問題の緩和をっているものの，一般的な実行環境と異なりイベント駆動かつノンプリエンプティブであるため，既存アプリケーションを変更なしに動作させることは容易ではない．また，この設計では新規に OS を設計しているため，OS の保守を既存 OS とは別に行う必要があり，保守性を担保することが難しい．

ClickOS [4] は，ネットワーク処理のアプリケーションを Xen ハイパーバイザ上で高速に実行するための OS である．本論文で提案するフレームワークでは，ネットワーク処理に限らず，一般的なアプリケーションを対象とする．

Arrakis [8]，IX [9] では，コントロールプレーンとデータプレーンを分離することで，少ない OS の関与でアプリケーションからデバイスを使用できる OS アーキテクチャ提案している．コントロールプレーンとデータプレーンの分離には，Arrakis はデバイスの仮想化機能を，IX は CPU の仮想化機能を用いている．Arrakis と IX はデータセンタのアプリケーションを対象としてプロセスを提供するのに対し，本論文の提案ではクラウド環境のアプリケーションを対象としてプロセスを提供せず，アプリケーションをカーネル内で動作させる．アプリケーションをカーネル内で動作させ，OS をバイパスしてデバイスを使用することで OS の関与を減らすことができる．

3. フレームワークの設計

提案フレームワークでは，アプリケーションを既存 OS のカーネル内で実行する．具体的には，既存アプリケーションを既存 OS のローダブル・カーネル・モジュール (LKM) として実行できるようにする．これにより，1) プロセスによるオーバーヘッドを回避しつつも，2) 既存 OS の機能を利用でき，かつ 3) 既存 OS コミュニティによるコア機能の保守を活用することができる．既存アプリケー

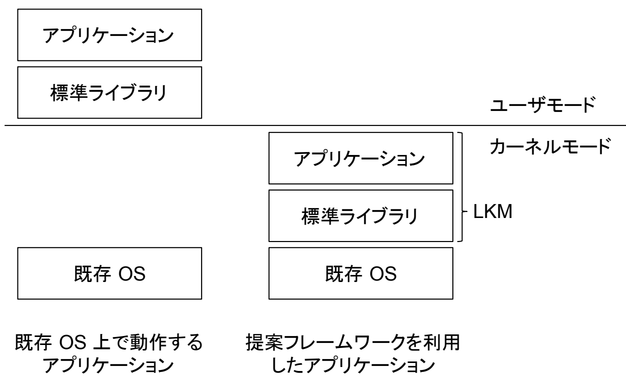

図 1 既存 OS 上のアプリケーションと提案フレームワークを利用したアプリケーション

ションを LKM として実行するために，提案フレームワークは LKM 向けの標準ライブラリを提供する．提案フレームワークを利用したアプリケーションは，フレームワークが提供する標準ライブラリとリンクされ，LKM として動作する (図 **1**).

3.1 カーネル内実行によるオーバーヘッドの排除

クラウド環境においては，ハイパーバイザにより VM 間が隔離され，かつ 1 つの VM 上では 1 つのアプリケーションを実行できれば十分であるため，プロセスによる保護は必要でない．そこで，アプリケーションをカーネル内で実行することでプロセスを利用しないようにする．プロセスを利用する場合，ユーザスペースとカーネルスペース内でのメモリコピーが発生するが，カーネル内で実行することでこれを排除できる．

また，プロセスを用いてアプリケーションを実行する場合，アプリケーションからカーネルの機能を使用する際には多くの場合システムコールを用いる．しかし，システムコールは関数呼び出しと比較してオーバーヘッドが大きい．アプリケーションをカーネル内で実行する場合，カーネルのあらゆる機能を関数呼び出しによって利用できるため，システムコールによるオーバーヘッドを削減できる．

3.2 既存 OS の機能の利用

既存 OS をベースにクラウド環境への特化を行うようにすることで，アプリケーションとフレームワークの実装では既存 OS のほぼ全ての機能を利用できることが期待される．例えばメモリ管理やデバイスドライバ，ファイルシステムのような OS の基本的な機能を再実装する必要がない．これは，アプリケーションやフレームワークの実装コストを下げるために重要である．

ただし，既存 OS がユーザアプリケーション向けに提供するの機能の多くははユーザアプリケーションから利用できれば十分であり，全てを一般的にカーネル内から利用できるようにする必要はない．したがって，対象の既存 OS がこれらの機能の全てをカーネル内から利用しやすい形で実装しているとは限らない．こういった機能の利用に関しては制限がある可能性がある．

3.3 保守性の向上

既存 OS をベースに実装するため，提案フレームワークの保守においては，ベースとなる既存 OS の部分に関しては既存 OS コミュニティの保守を活用することができる．ベースの既存 OS を新しいバージョンに更新し，それによる既存 OS の変更に合わせてライブラリやアプリケーションを変更することで全体の保守を行うことができる．

既存 OS に含まれない部分に関しては既存 OS とは別に保守が必要になる．しかし，既存 OS 自体のコード量と比較すると提案フレームワークのコード量は極めて少ない．したがって，提供できる機能を考慮すると，新規に OS を開発する場合と比較して保守コストは低いといえる．

また，提案フレームワークではプログラミング言語の標準ライブラリを提供するため，これの保守は必要である．しかし，これに関しても既存の標準ライブラリを改変して実装することで，差分のみの保守で全体の保守が行える．これらのライブラリの差分も，既存 OS やライブラリ自体のコード量と比較して極めて少ない．

3.4 ライブラリレベルでの API 互換性の確保

既存アプリケーションを LKM として実行できるようにするために必要な移植コストを下げるために，既存 OS と互換性の高い API をカーネル内で使用できるようにする．具体的には，LKM 向けの標準ライブラリを提供し，アプリケーションをそれにリンクするようにする．この API の互換性が十分であれば，既存アプリケーションをリビルドするだけで移植を行うことができる．

LKM 向け標準ライブラリにおいて既存 OS との互換 API を提供するためには，既存 OS が LKM 向けに公開している API から互換 API への変換が必要になる．例えば，標準ライブラリ内でシステムコールを用いてカーネルの機能を呼び出す部分を，カーネルの内部 API を呼び出すように変更する．この方法では，既存 OS への変更なしでカーネルの内部 API と互換 API の間の変換を行うことができる．

多くの標準ライブラリは既存 OS の大半のシステムコールのラッパーを実装しているため，アプリケーションはそれを介してシステムコールを発行することが一般的である．したがって，多くの既存アプリケーションを，ソースコードへの変更なしに互換 API を使用するように変更できる．ただし，移植対象のアプリケーションが標準ライブラリを介さず直接システムコールを発行する場合にはアプリケーションへの変更が必要になる．

この方法ではフレームワーク提供の標準ライブラリをリンクする以外に特殊なことをする必要がないため，コンパイラツールチェーンは既存の LKM 用のものをそのまま使用できる．

4. 実装

提案フレームワークを Linux 4.8 を対象に実装した．LKM の仕組みは Linux がすでに提供しているため，提案フレームワークは，LKM から Linux API を利用するための標準ライブラリを提供すればよい．C，C++ のアプリケーションを対象とし，それぞれの言語の標準ライブラリを提供する．

4.1 標準ライブラリ

C，C++ 言語の標準ライブラリとして広く用いられている，musl libc [10] と libc++ [11] を LKM 向けに移植した．カーネルの内部 API から ユーザアプリケーション向け API への変換は libc 内で行うようにした．

この変換のため libc に加えた変更は次の 4 通りである．

(1) システムコールのハンドラを呼び出すようにする
(2) Virtual File System (VFS) レイヤの関数を使用するようにする
(3) カーネルの内部 API を使用するようにする
(4) 上の 3 つの組み合わせで実装する

(1) システムコールのハンドラを使用する場合は，標準ライブラリのシステムコールラッパーを，システムコールハンドラを使用するように書き換える．この方法は 4 つのうち最も簡単であるものの，システムコールハンドラは，LKM 内から呼び出されることを想定した実装にはなっていないため，これらを用いる場合に関しては実装の確認と挙動のテストを行う．システムコールハンドラを使用できない場合の例としては，current タスク構造体の files メンバのように，カーネルスレッドから使用できないメンバを使用する場合がある．

(2) Virtual File System (VFS) レイヤの関数を使用する場合は，標準ライブラリのシステムコールラッパーを，VFS レイヤの関数を使用するように書き換える．この場合，元のシステムコールではファイルディスクリプタを使用してファイルを指定するが，VFS レイヤではファイル構造体を用いるため，この変換を行う必要がある．

(3) カーネルの内部 API を使用するようにする場合は，標準ライブラリのシステムコールラッパーを，カーネルの内部 API を使用するように書き換える．この場合，引数ならびに返り値の型や意味の違いを吸収する必要がある．

(4) 3 つの組み合わせで実装する場合は，標準ライブラリのシステムコールラッパーを呼び出したとき，元のシステムコール相当の処理が行われるように実装する．

Linux が x64 向けに提供するシステムコール 322 個のうち，表 1 に示した 30 個のシステムコールのみ実装済みである．表 1 に示したとおり，いくつかのライブラリ関数に関してはシステムコールを用いない形で実装している．

表 1　API の変換方法の分類と各場合に該当する関数

分類	関数
(1) システムコールハンドラを用いる	sys_stat
	sys_lstat
	sys_rmdir
	sys_unlink
	sys_access
	sys_rename
	sys_nanosleep
	sys_clock_gettime
	sys_gettimeofday
	sys_set_robust_list
	sys_sched_getaffinity
	sys_mkdir
	sys_sysinfo
	sys_getrlimit
	sys_prlimit64
	sys_getrusage
	sys_futex
(2) VFS の関数を用いる	sys_open
	sys_close
	sys_pread64
	sys_pwrite64
	sys_read
	sys_write
	sys_fsync
	sys_fdatasync
	sys_lseek
	sys_readv
	sys_writev
	sys_getdents
	sys_fcntl [*1]
(3) カーネルの内部 API を用いる	sched_yield
	geteuid
(4) 組み合わせで実装する	malloc
	pthread

また，標準ライブラリ内にはカーネルのシンボルと衝突する関数や，名前の衝突する型が存在するため，これらをリネームする必要がある．

libc に対する変更は，7000 行程度の追加と 5000 行程度の削除であった．このうちの多くはカーネル内のシンボル，型名と衝突するものをリネームする変更であり，プログラムにより自動的に行われた．

4.2 制限

現状，Linux API の全てを提供できていない，アプリケーション内で透過的に浮動小数点数を使用できないという 2 つの問題がある．

*1 FUTEX_PRIVATE フラグを立てる必要がある

1つめは，現状の実装は Linux API の限られたサブセットのみを実装していることによる．mmap や vfork といったシステムコールや，ネットワーク関連のシステムコールはアプリケーションの実装によく用いられる．しかしながら，現状の実装はこれらをアプリケーションに提供していない．また，疑似ファイルシステムに関しても動作を確認していない．

現状提供できていないシステムコールのうちの多くは先に分類した方法で容易に実装可能である．しかし，一部のシステムコールはアプリケーションの動作に必要となることが多いにもかかわらず，実装が容易ではない．API Importance [12] は，Linux API に対して定義される API の重要度の指標である．これは，Linux システムをランダムに選んだとき，そのシステム上で動作するアプリケーションのうち1つ以上がその API を必要とする確率である．mmap と vfork は API Importance が 100% であるにも関わらず，実装が容易ではない．

まず，mmap ではファイルを仮想アドレス空間にマッピングできる必要がある．しかし，LKM で確保できるメモリはカーネル空間内のページでありページキャッシュを使用できない．また，カーネル空間のメモリはページングできないため，物理メモリ以上の大きさのファイルを mmap できない．これらの課題を解決するためにカーネルのメモリサブシステムを変更すると，保守性に問題が生じる．この問題の回避方法としては，mmap の場合のみ特定のプロセスの仮想アドレス空間を使用する方法がある．

次に，vfork は子プロセスを生成するためのシステムコールである．本フレームワークのアプリケーションはプロセスを使用しないものとしたため，プロセスを生成する動作を実現することは難しい．ただし，アドレス空間を複数に区切り，vfork した際には同一アドレス空間内の別の区間を用いるようにすることで，vfork 相当の処理を行うことができる．

2つめは，Linux では，カーネル内での浮動小数点数の使用に制限があることによる．具体的には，浮動小数点数を使用する前に kernel_fpu_begin 関数を呼び出し，終わりに kernel_fpu_end 関数を呼び出す必要がある．この区間ではプリエンプションが無効になるため，実質的に浮動小数点数の使用に制限が生じる．ただし，この挙動はカーネルに変更を加えることで変更できると考える．

5. 実験

提案フレームワークが既存アプリケーションに適用可能であること，またそれによりパフォーマンスが向上することを確認するため，LevelDB [6] というキーバリューストアを Linux の LKM として動作させる．評価環境を表 2 にまとめた．

表 2　評価環境

項目	詳細
Linux Kernel	4.8
LevelDB	1.19
musl libc	1.1.15
libc++	commit d842fe
CPU	Intel Core i7-3820QM
RAM	16 GB
SSD	768 GB

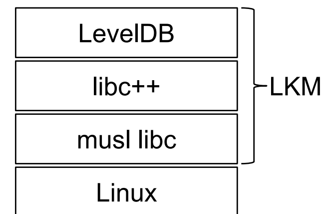

図 2　提案フレームワークを利用した LevelDB のアーキテクチャ

表 3　LevelDB に対する変更行数とその理由

理由	追加行数	削除行数
固定小数点数への変更	48	7
カーネル内と重複する型名の変更	28	28
mmap から pread への変更	0	79
合計	76	114

5.1 LevelDB

LevelDB [6] は，ユーザプログラム向けのキー・バリュー・ストアライブラリである．提案フレームワークを利用し，LevelDB をカーネル内で動作するように変更した．LevelDB に対し必要な変更は，コンパイルオプションの変更とコードへの変更に分けられる．

コンパイルオプションへの変更としては，リンクする標準ライブラリをフレームワーク提供の標準ライブラリへ変更すること，一般的な LKM 向けのコンパイルオプションを追加することが必要である．LevelDB は C++ で実装されているため，動作には C 言語と C++ 言語の標準ライブラリが必要であり，これらをフレームワーク提供のものに変更する．LKM 向けのコンパイルオプションの例としては，メモリモデルをカーネル向けにするオプションなどがある．提案フレームワークを適用した LevelDB は，**図 2** のように LKM 向けの標準ライブラリとリンクされ，LKM として動作する．

コードへの変更としては，表 3 のものがある．なお，追加，削除したコード行数を合計しても約 200 行であり，LevelDB のコード行数が約 20000 行であるのに対して 1/100 以下である．

Yahoo! Cloud Serving Benchmark (YCSB) [13] を用いて，ユーザプロセスとして動作する LevelDB と，カーネル内で動作する LevelDB のスループットを比較する．YCSB

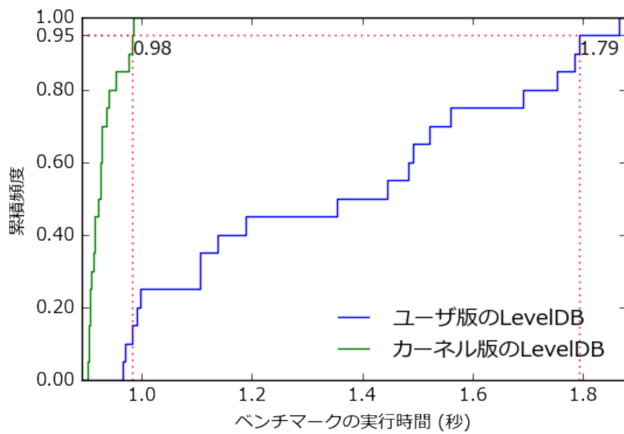

図 3 リードが 50%，ライトが 50%のワークロードにおけるベンチマーク実行時間

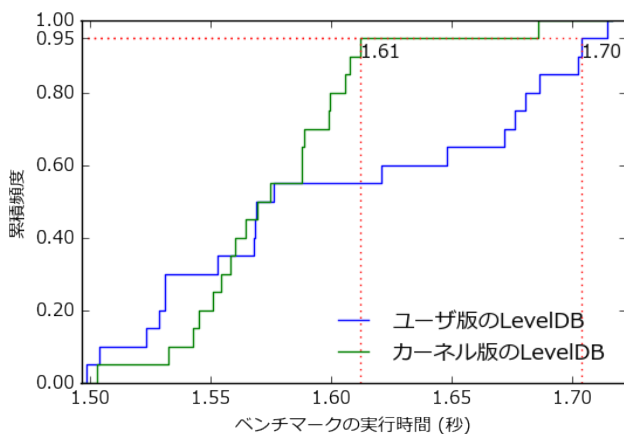

図 4 リードが 100%のワークロードにおけるベンチマーク実行時間

から次の 2 種類のワークロードを用いて測定を行なった．

(1) リードが 50%，ライトが 50%のワークロード

(2) リードが 100%のワークロード

いずれも操作の合計回数は 10 万回とし，それぞれ 20 回測定した．このときのベンチマーク実行時間の累積頻度を**図 3**，**図 4** に示す．

まず，(1) リードが 50%，ライトが 50%のワークロードでは，図 3 の 95 パーセンタイルの実行時間を比較するとカーネル版は 0.98 秒であるのに対し，ユーザプロセス版は 1.79 秒であった．また，平均実行時間はそれぞれ 1.35 秒，0.92 秒であり，カーネル版のスループットはユーザプロセス版の 1.46 倍であった．このとき，カーネル版のスループットの標準偏差は小さくなり，ユーザプロセス版の 1/10 であった．次に，(2) リードが 100%のワークロードでは，図 4 の 95 パーセンタイルの実行時間を比較するとカーネル版は 1.61 秒であるのに対し，ユーザプロセス版は 1.70 秒であった．また，平均実行時間はそれぞれ 1.58 秒，1.60 秒であり，カーネル版のスループットはユーザプロセス版の 1.02 倍であった．このとき，カーネル版のスループットの標準偏差は (1) の場合と同様に小さくなり，ユーザプロセス版の 1/6 であった．

いずれの場合でもカーネル版のスループットはユーザプロセス版を上回った．しかし，スループットの向上は (2) リードが 100%のワークロードよりも，(1) リードが 50%，ライトが 50%のワークロードのほうが大きく，ライトよりもリードのほうがスループットの向上が小さいことがわかる．リードのスループットの向上が小さい原因は，現状の提案フレームワークの実装は mmap を提供していないため，元々の LevelDB で mmap を使用していた部分に pread を使用しているためである．ライトに関しては，元々の LevelDB においても pwrite を使用しており mmap を使用しておらず，オーバーヘッドの削減によってパフォーマンスが向上している．

6. まとめ

クラウド環境の特徴に着目したアプリケーションのパフォーマンス向上，既存 OS と同等の機能の提供，実行基盤自体の高い保守性を同時に達成する方法として，既存のアプリケーションを既存の OS カーネル内で実行するためのフレームワークを提案した．この方法では，まず，カーネル内で動作する設計によりプロセスのオーバーヘッドを排除し，クラウド特化型 OS と同等の軽量化が期待できる．次に，既存 OS をベースに実装することで提案手法の実装に既存 OS の機能を用いることができ，高い機能性の達成が容易である．最後に，実行環境のほとんどを占める既存 OS 部分の保守が既存 OS コミュニティによって行われるため，高い保守性を保つことができる．

提案フレームワークを Linux に対して実装し，LevelDB というキー・バリュー・ストアをカーネル内で動作するようにした．カーネル版の LevelDB のスループットは，ユーザプロセス版の LevelDB の最大 1.46 倍となった．また，LevelDB に対して行う必要のあった変更行数は，LevelDB のコード行数の 1/100 以下であった．

今後の課題として，より多くの API を提供することがある．アプリケーションの動作に必要な API をはじめから提供できれば，リンクする標準ライブラリを切り替えるだけでカーネル内での実行が可能になる．次に，アプリケーションに最適化した API の提供が考えられる．例えば，ゼロコピー API を提供することで，パフォーマンスの向上を期待できる．

謝辞

本研究は JST，CREST の支援を受けたものである．

参考文献

[1] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S. and Crowcroft, J.: Unikernels: Library Operating Systems for the Cloud, *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, (ASPLOS '13), New York, NY, USA, ACM, pp. 461–472 (online), DOI: 10.1145/2451116.2451167 (2013).

[2] Kivity, A., Laor, D., Costa, G., Enberg, P., Har' El, N., Marti, D. and Zolotarov, V.: OSv—Optimizing the Operating System for Virtual Machines, *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, USENIX Association, pp. 61–72 (online), available from ⟨https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity⟩ (2014).

[3] Schatzberg, D., Cadden, J., Dong, H., Krieger, O. and Appavoo, J.: EbbRT: A Framework for Building Per-application Library Operating Systems, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, (OSDI'16), Berkeley, CA, USA, USENIX Association, pp. 671–688 (online), available from ⟨http://dl.acm.org/citation.cfm?id=3026877.3026929⟩ (2016).

[4] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R. and Huici, F.: ClickOS and the Art of Network Function Virtualization, *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, (NSDI'14), Berkeley, CA, USA, USENIX Association, pp. 459–473 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2616448.2616491⟩ (2014).

[5] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E. and Wisniewski, R. W.: Libra: A Library Operating System for a Jvm in a Virtualized Execution Environment, *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, New York, NY, USA, ACM, pp. 44–54 (online), DOI: 10.1145/1254810.1254817 (2007).

[6] Google: LevelDB, `https://github.com/google/leveldb`.

[7] Friebel, T.: How to deal with lock-holder preemption, Xen Summit North America (2008).

[8] Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T. and Roscoe, T.: Arrakis: The Operating System is the Control Plane, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, (OSDI'14), Berkeley, CA, USA, USENIX Association, pp. 1–16 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2685048.2685050⟩ (2014).

[9] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C. and Bugnion, E.: IX: A Protected Dataplane Operating System for High Throughput and Low Latency, *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, (OSDI'14), Berkeley, CA, USA, USENIX Association, pp. 49–65 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2685048.2685053⟩ (2014).

[10] Felker, R. et al.: musl libc, `https://www.musl-libc.org/`.

[11] Group, L. D.: "libc++" C++ Standard Library, `http://libcxx.llvm.org/`.

[12] Tsai, C.-C., Jain, B., Abdul, N. A. and Porter, D. E.: A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting, *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, ACM, pp. 16:1–16:16 (online), DOI: 10.1145/2901318.2901341 (2016).

[13] Cooper, B.: Yahoo! Cloud Serving Benchmark, `https://github.com/brianfrankcooper/YCSB`.