OSvのご紹介 in iijlab セミナー

Takuya ASADA <syuu@cloudius-systems>
Cloudius Systems

自己紹介

- Software Engineer at Cloudius Systems
- FreeBSD developer (bhyve, network stack..)

Cloudius Systemsについて

- OSvの開発母体（フルタイムデベロッパで開発）
- Office：Herzliya, Israel
- CTO : Avi Kivity → Linux KVMのパパ
- 他の開発者：元RedHat(KVM), Parallels(Virtuozzo, OpenVZ) etc..
- イスラエルの主な人物は元Qumranet（RedHatに買収）
- 半数の開発者がイスラエル以外の国からリモート開発で参加
- 18名・9ヶ国（イスラエル在住は9名）

サイダー
ラシャイヤ
Set Zaynab
Nabatieh
スール
ナハリヤ
Nahariya
ツファット
Safed
Nawa
ハイファ
Haifa
ナザレ
Nazareth
ティベリア
Tiberias
Dael
スワ
Nesher
イルビド
ダルアー
ハデラ
Hadera
アフラ
Afula
Al Hosn
Ajloun
ネタニヤ
Netanya
マフラク
テルアビブ
Tel Aviv-Yafo
ジャラシュ
アンマン
Or Yehuda
西岸地区
West Bank
リション
レジオン
Rishon
LeTsiyon
Jericho
マーダバー
Madaba
ガザ
Gaza
Kiryat Gat
エルサレム
Jerusalem
Al Jiza
Al Zumayla
死海
ガザ
Gaza Strip
Yata
ベエルシェバ
Be'er Sheva
Suwaqa Al
Gharbiya
アラド
Arad
カラク
Al Qatrana
Rafa
イスラエル
ハサー
Ramat Negev
Regional
Council
ミツペ・ラモン
Mitzpe Ramon
Petra
マーン
Quweira
Google
利用規約
65
35
25
15
5
40
M5

Nacra 5.5

The first wwv
Cloudius Systems
Feb 2014
zero

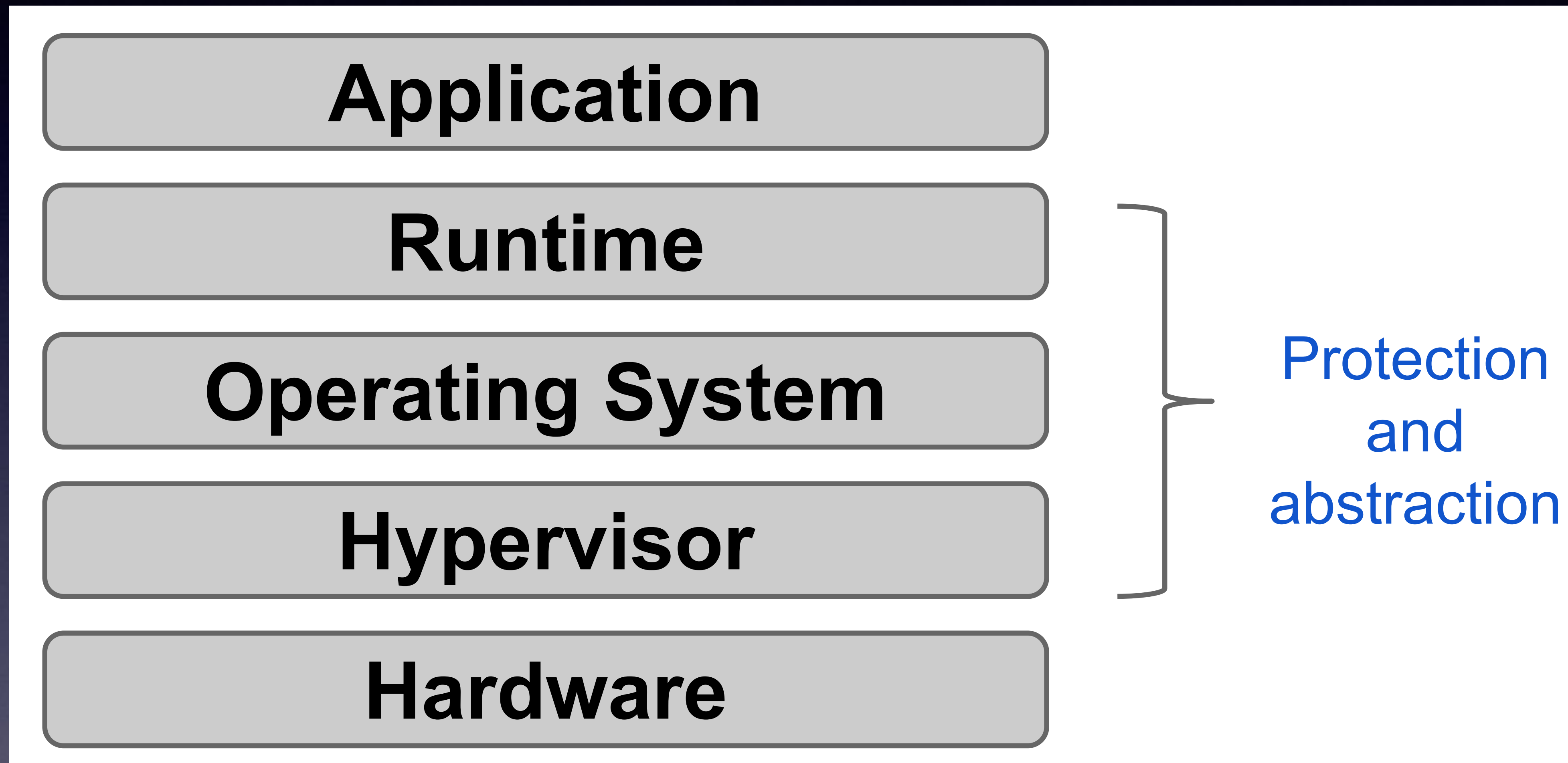
zero

OSvの概要

OSvとは？

- OSvは単一のアプリケーションをハイパーバイザ・IaaSでLinuxOSなしに実行するための新しい仕組み
- より効率よく高い性能で実行
- よりシンプルに管理しやすく
- **オープンソース（BSDライセンス）、コミュニティでの開発**
- http://osv.io/
- https://github.com/cloudius-systems/osv

標準的なIaaSスタック



- 単一のアプリケーションを実行するワークロードではフルサイズのゲストOS＋フル仮想化はオーバヘッド

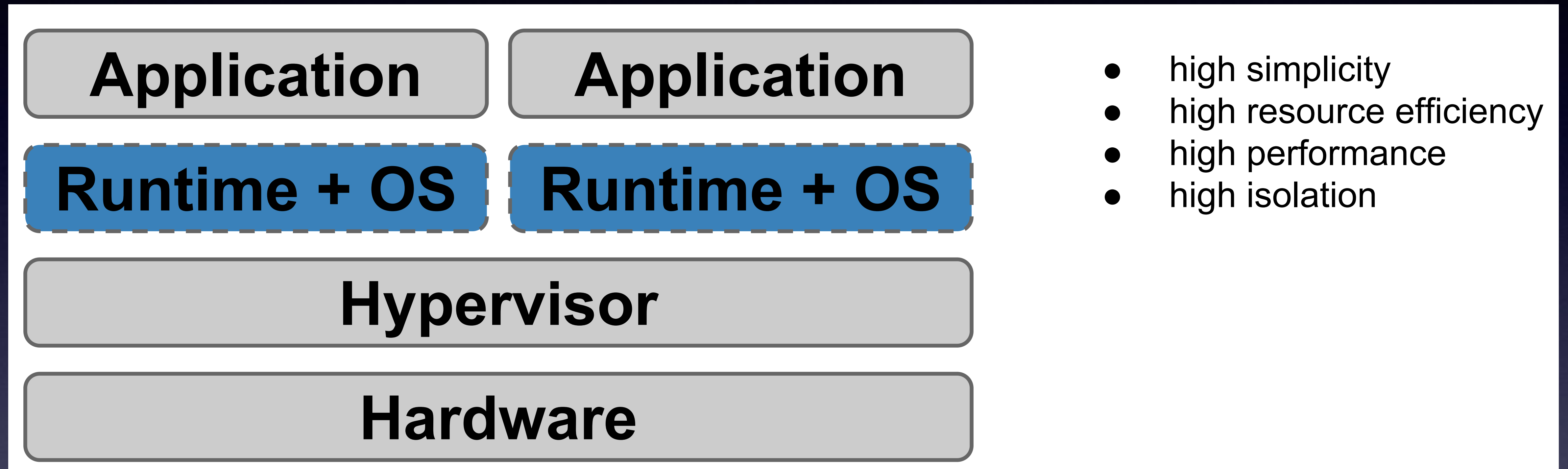
コンテナ技術

Application | Application
Runtime | Runtime
Operating System
Hardware

- high simplicity
- high resource efficiency
- high performance

- 実行環境をシンプルにする事が可能
- パフォーマンスも高い

ライブラリOS＝OSv



- コンテナと比較してisolationが高い

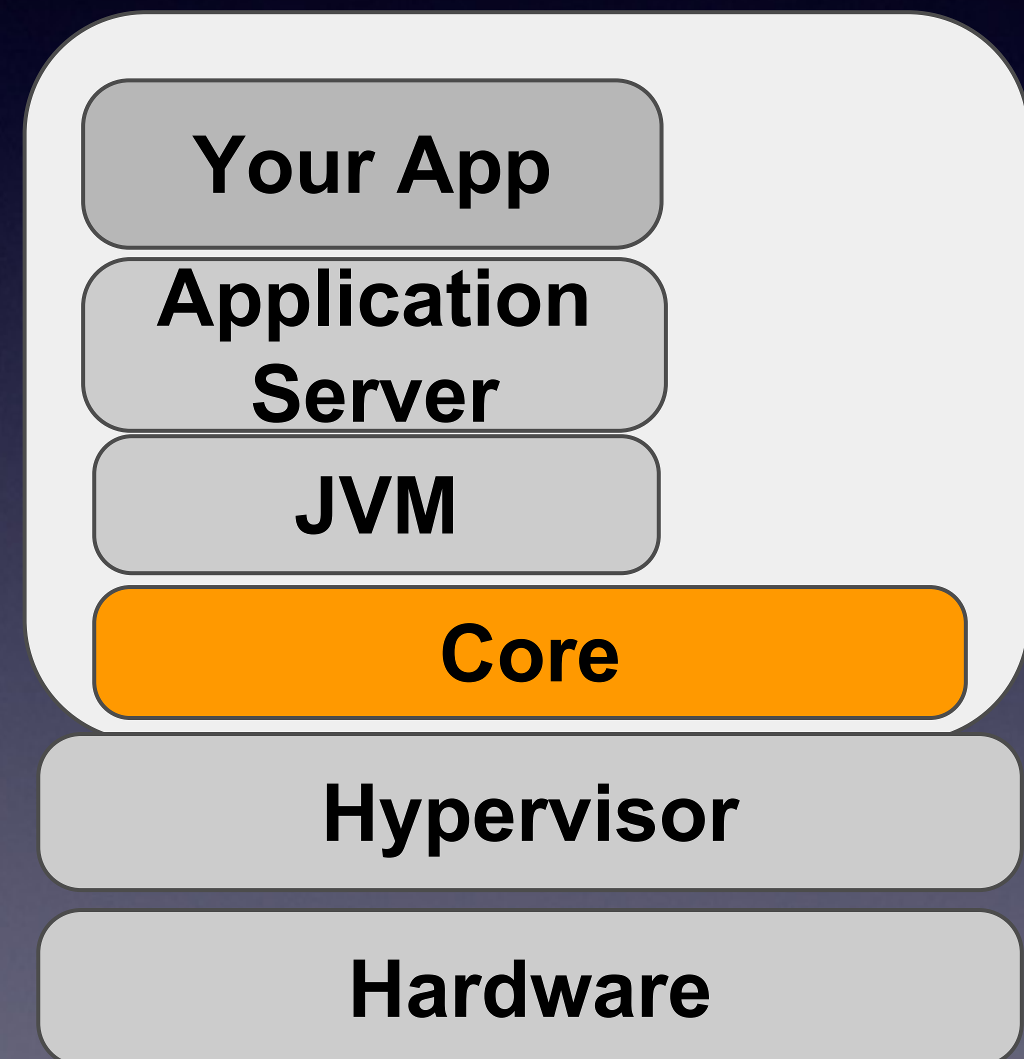
標準的なIaaSスタック：機能の重複、オーバヘッド

- ハイパーバイザ・OSの双方がハードウェア抽象化、メモリ保護、リソース管理、セキュリティ、Isolationなどの機能を提供
- OS・ランタイムの双方がメモリ保護、セキュリティなどの機能を提供
- 機能が重複しており無駄が多い

Your App
Application Server
JVM
Operating System
Hypervisor
Hardware

OSv：
重複部分の削除

- 重複していた多くの機能を排除
- ハイパーバイザに従来のOSの役割を負ってもらい、OSvはその上でプロセスのように単一のアプリケーションを実行



OSvのコンセプト

- 1アプリ＝1インスタンス→シングルプロセス
- メモリ保護や特権モードによるプロテクションは行わない
    - 単一メモリ空間、単一モード（カーネルモード
- Linux互換 libc APIを提供、バイナリ互換（一部）
- REST APIによりネットワーク経由で制御

動作環境

- ハイパーバイザ
  - KVM
  - Xen
  - VMware
  - VirtualBox
- IaaS
  - Amazon EC2
  - Google Compute Engine

対応アーキテクチャ

- x86_64（32bit非サポート）
- aarch64

対応アプリ（Java）

- OpenJDK7,8
  - Tomcat
  - Cassandra
  - Jetty
  - Solr
  - OpenDaylight
  - Gitblit
  - Clojure
- JRuby（Ruby on Railsなど）
- Ringo.JS
- Jython
- Erjang
- Scala
- Quercus（PHPエンジン、Wordpressなど）
- minecraft-server
- Oracle NoSQLDB

対応アプリ（Java以外）

- Ruby
  - WEBRick
  - Ruby on Rails
  - Publify（Railsベースのブログエンジン）
- mruby
- lua
- Node.js

何が動くの？
（ネイティブアプリ）

- Redis
- memcached
- haproxy
- MySQL
- LevelDB
- SQLite
- twemproxy

アプリケーションに提供される機能

- Linux互換API
- SMP・マルチスレッド
- TCP/IP（v4 only）
- ramfs・ZFS

OSvを便利にする機能

OSvをプログラムから操作「REST API」

- REST API経由でOSvに任意の操作を実行
- 従来のOS：コマンド実行やファイルの編集で設定を変更
  (手動が基本、シェルスクリプトなどで自動化)
  OSv：APIで設定を変更
  (自動化が基本、CLIはオプション）
- HTTPSでクライアント証明書を用いたセキュリティ設定が可能

OSvを対話的に操作「Lua CLI」

- 簡易的なシェル機能を実現
- 全ての機能をREST API上に実装
- OSvの中でも、リモートホストのLinuxマシンでも動作→SSH代わりに使用可能

初期化スクリプトを
サーバからダウンロード＆実行
「Cloud Init」

- ネットワーク上からYAMLをダウンロードしてきて、ダウンロードしたファイルに記述されているREST APIを順次実行

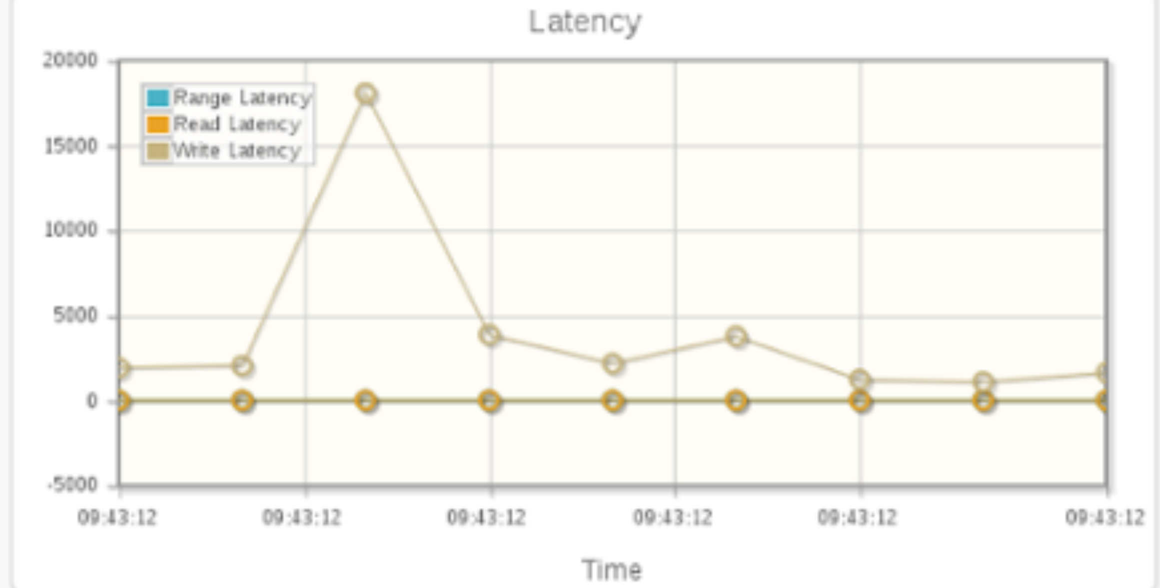
OSv GUI

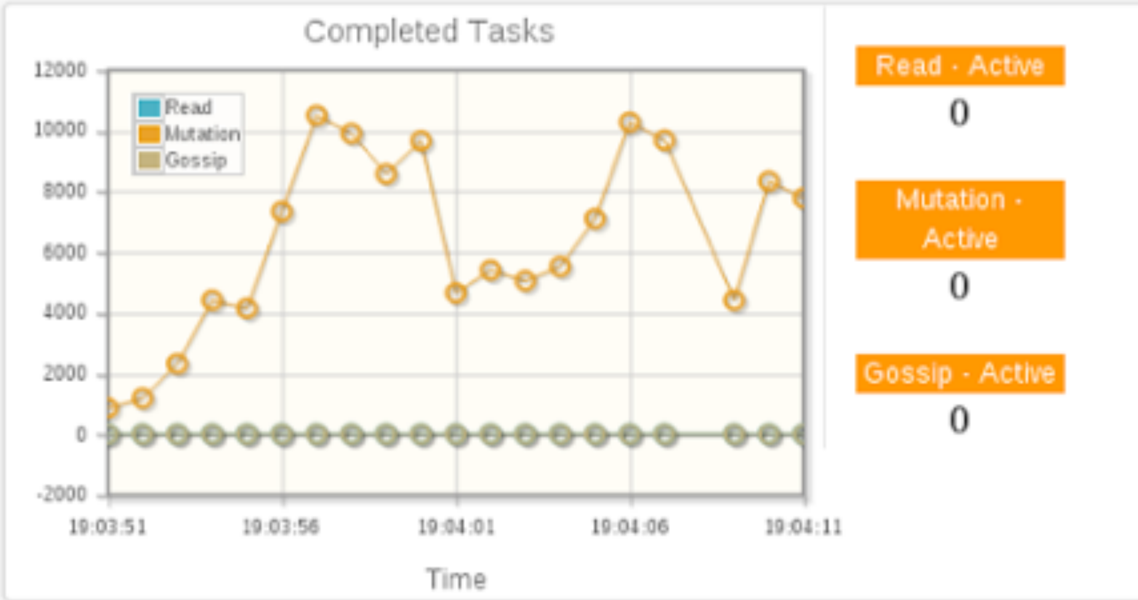
- WebベースのGUI
- OSの負荷、JVMのリソース情報、アプリ固有のstatisticsなどの統計情報を表示
  - アプリ固有情報は、まずCassandra, memcached, Redisの３つが対象

OSv | A CLOUDIUS SYSTEMS PROJECT | osv.local

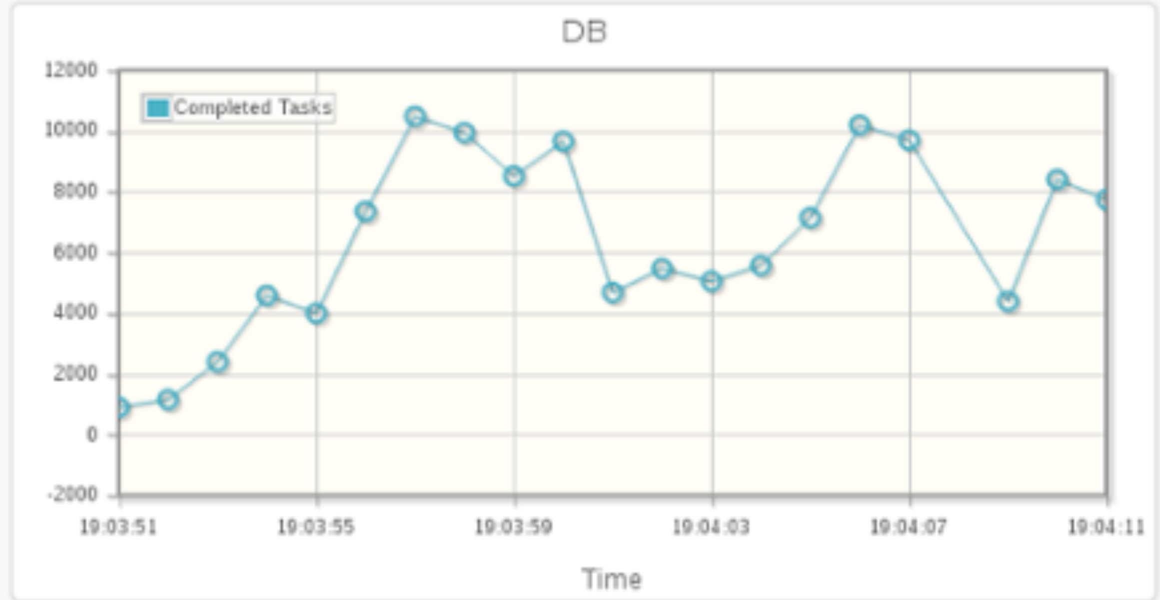
Main | Threads | Profiler | JVM | Cassandra

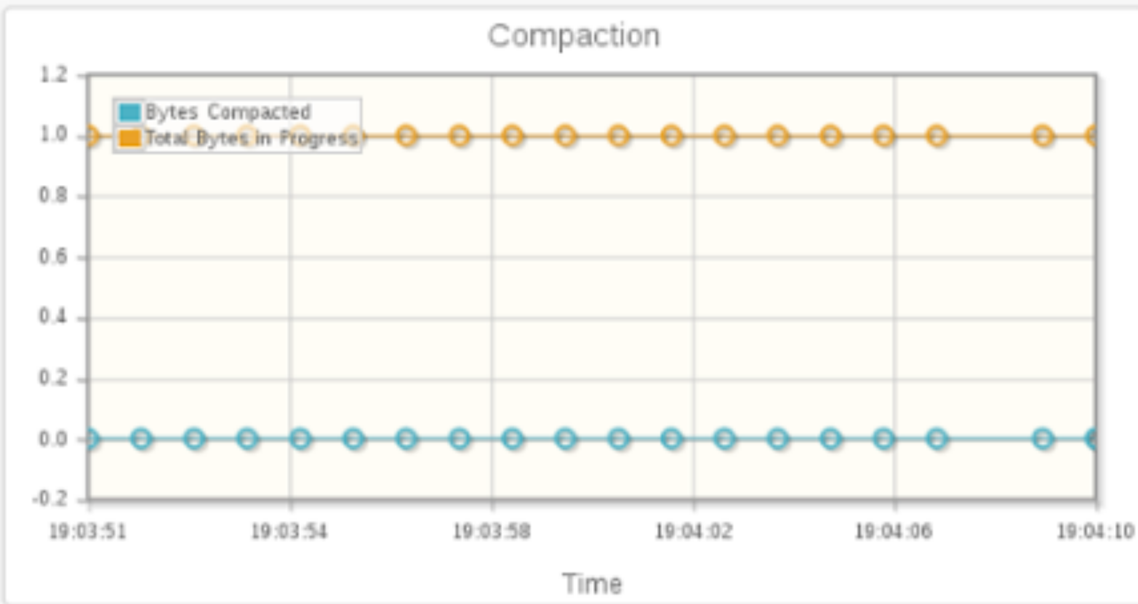
Live Nodes	• 127.0.0.1	
Load Map	127.0.0.1	40.93 KB





Read - Active
0
Mutation - Active
0
Gossip - Active
0





簡易デプロイツール「Capstan」

- 色々なアプリがインストールされたOSvのVMイメージを、コンパイルなどの難しい作業なしに手軽に実行するツール
- Linux, Mac, Windowsで動作
- VirtualBox, VMware, KVMなどに対応

クラウドへのデプロイ

- Amazon EC2
  - AMIを配布中
- Google Compute Engine
  - capstanからアップロード可能

OSvのフットプリント
と性能

フットプリント（ディスクイメージサイズ）

- mruby = 14MB
- Ruby = 48MB
- OpenJDK = 77MB

フットプリント
（最低メモリ使用量）

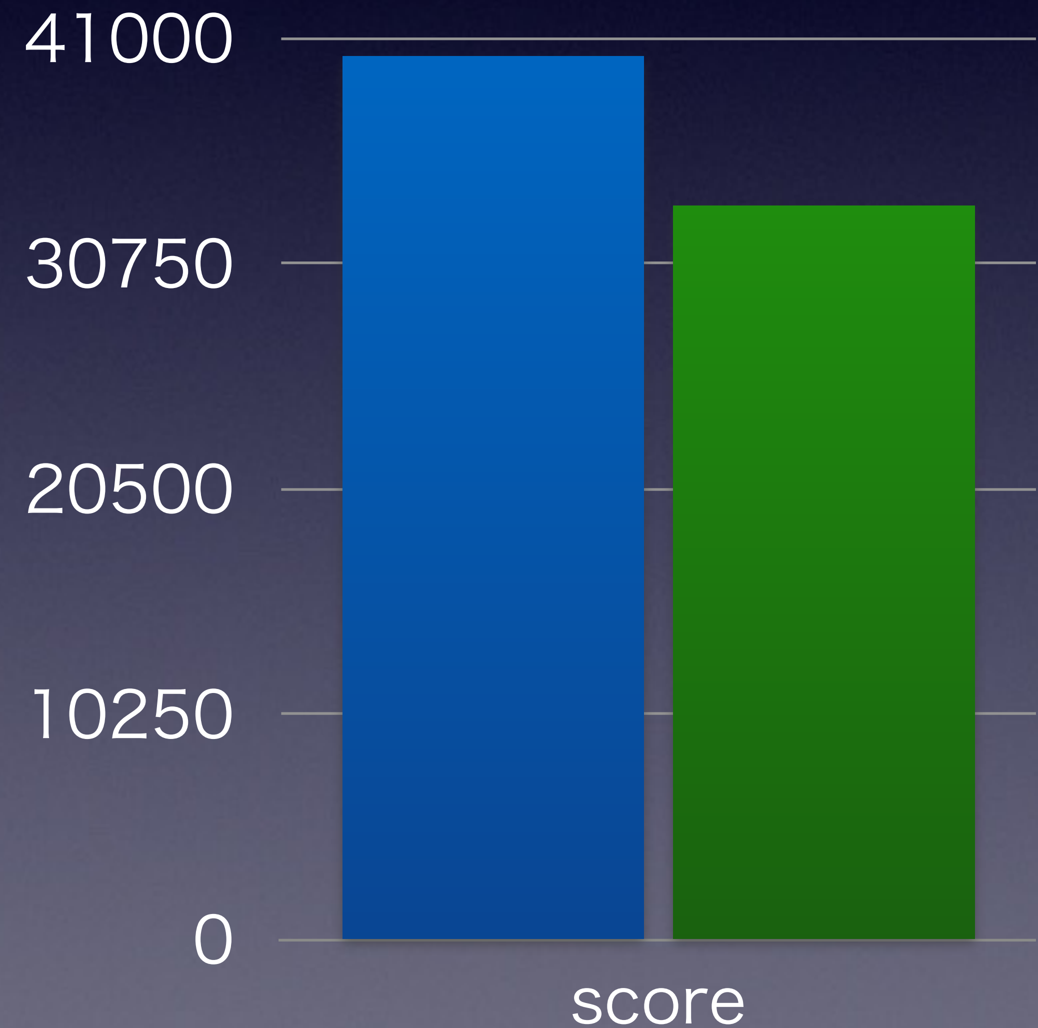
- mruby = 65MB
- Ruby = 75MB
- OpenJDK = 110MB

ブート時間

- 1秒（DHCP、ZFS初期化込み）

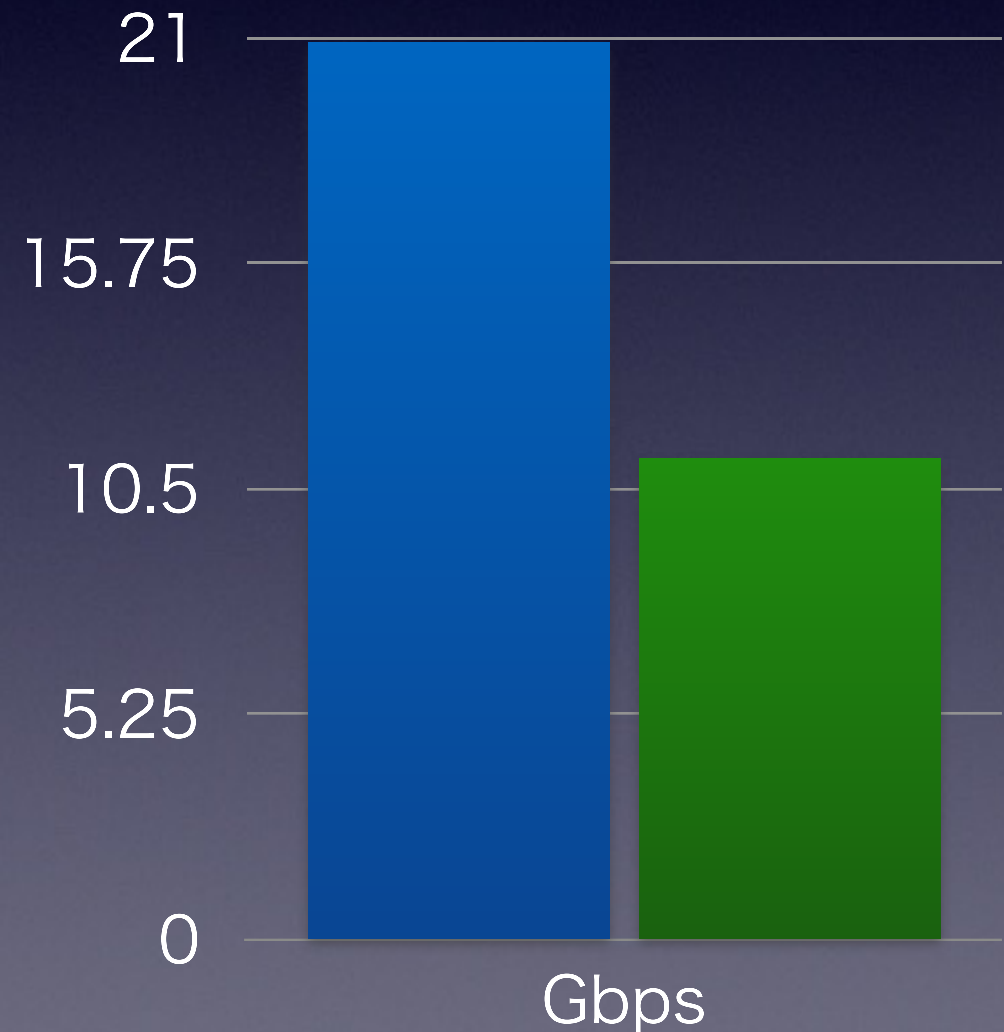
SPECjbb2005(Java)

- 120% faster than Linux guest



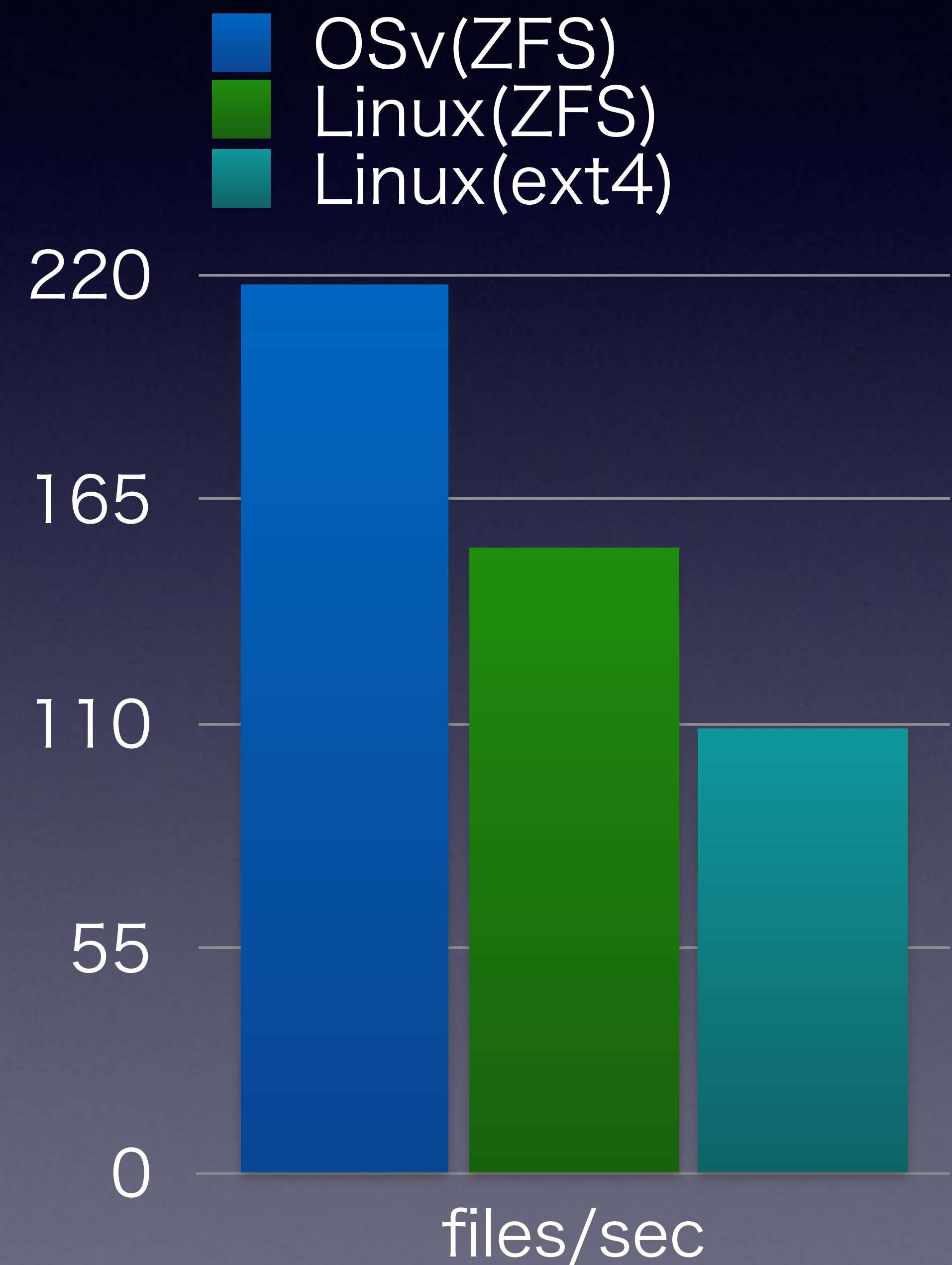
iperf(network)

- 186% faster than Linux guest



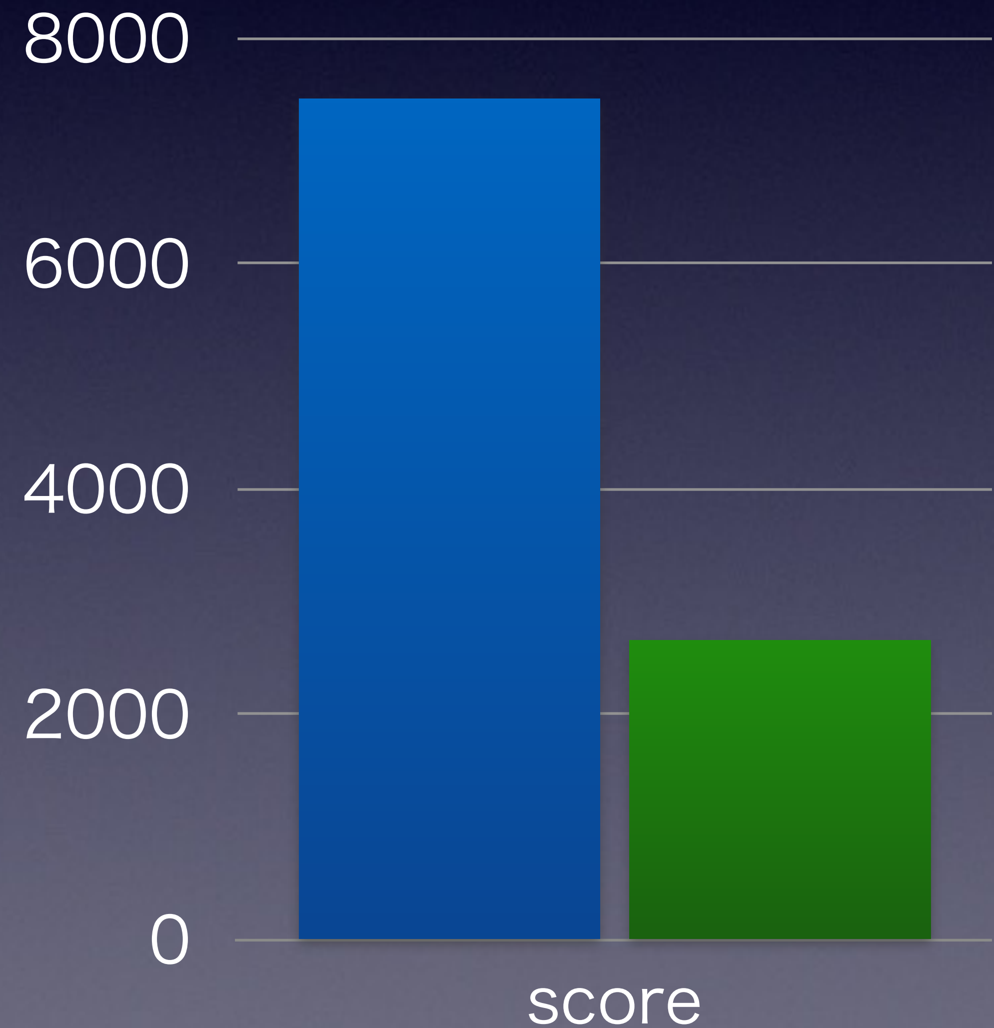
fsmark

- 142% faster than Linux guest



memcached

- ※ネットワークスタックを迂回する独自版memcachedでの比較
- 280% faster than Linux guest



デモ

OSvの実装

OSvの設計（１）

- OSvは複数のメモリ空間を持たない
  メモリ空間は全プロセス＆カーネルで共通
- OSvはカーネルとユーザプロセス間で権限のモード切替を行わない
  従って，カーネルの機能はlibc経由の関数コールで実現
- メモリの保護や権限の制限はハイパーバイザや言語ランタイムに任せる
  ネイティブコードがメモリ保護エラーでOSvカーネルのメモリ領域を破壊するのはユーザ責任
  - イメージとしては言語ランタイムをベアメタル環境に移植している状態に近い

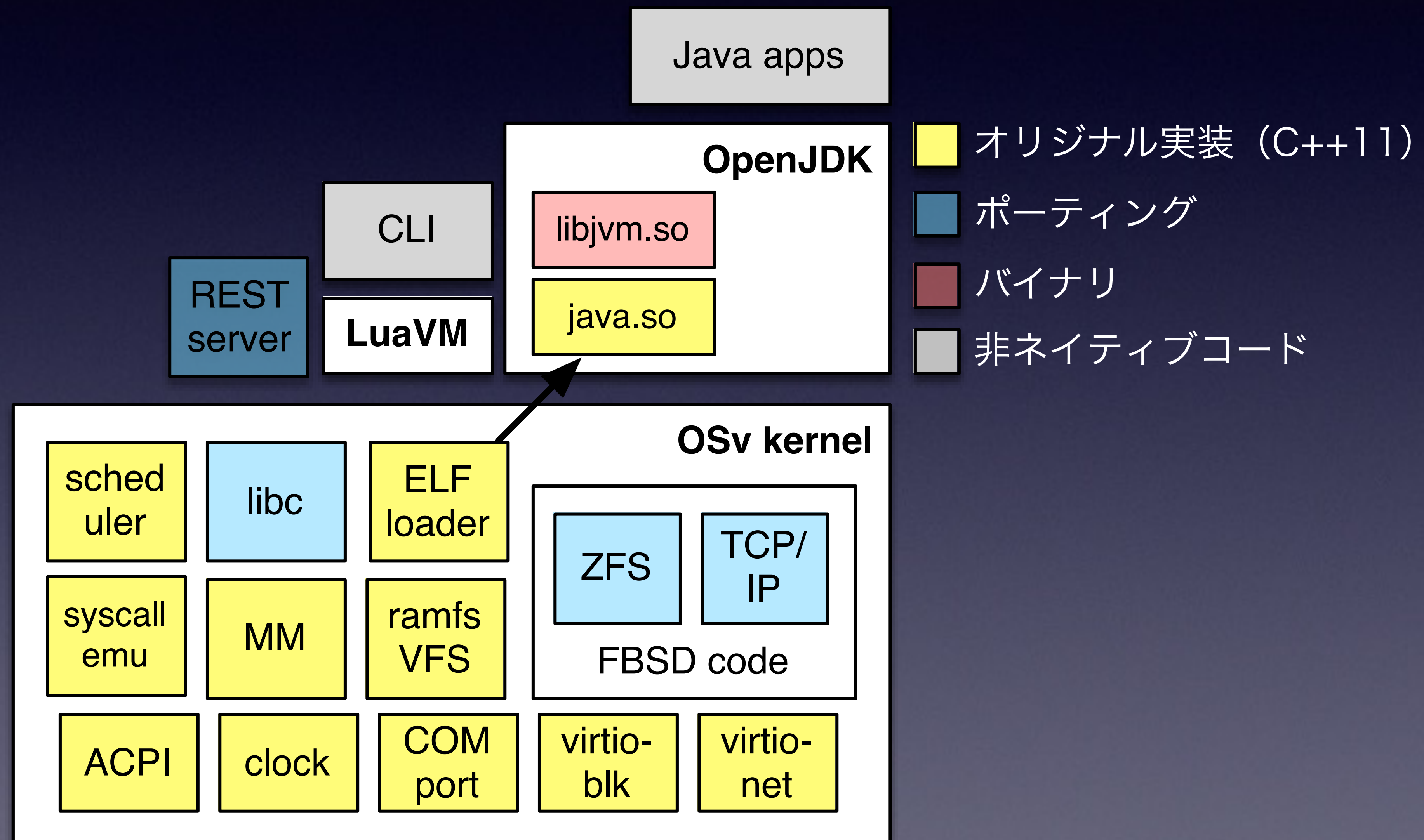
OSvの設計（２）

- これにより、TLBミスやモード切替のコストを削減しパフォーマンスを上げることが出来る
- （JVMによる制限が無ければ）ユーザプロセスからデバイスが丸見え
  - むしろ「アプリからvirtioを直接扱うことによって従来より性能を上げられる」と主張

OSvの構成要素

- C++11でスクラッチから書かれたカーネルの主な部分
  - メモリマネージャ、スケジューラ、ELFローダ、ドライバ、VFS、ramfs、ACPI、システムコール、libc関数（一部）
- FreeBSDからの移植
  - ZFS、TCP/IPスタック（v4のみ）
- musl-libc
- アプリケーション
  - lua VM & CLI、REST server、OpenJDK、Ruby…

OSvの構造



デバイスドライバ

- 仮想マシン専用なので準仮想化デバイス+最低限のデバイスをサポート
  - virtio-blk, virtio-net, virtio-scsi, virtio-rng
  - vmware-pvscsi, vmxnet3
  - xen pv driver
  - SATA
  - HPET、PV clock(KVM, Xen)
  - ACPI
  - Some legacy devices(IDE,VGA, COM, PS2)

Linuxとの互換性

- Fedora向けのOpenJDKバイナリが動作するレベル
- glibcの全関数が提供されている訳ではない
- musl-libcから必要に応じてAPIが移植されてきている
- 今まで必要無かったAPI、musl-libcに存在しないAPIが欠けている事がある

アプリケーション

- OSvが提供するlibcの範囲で動き、-fPIC（-shared）でビルドされている必要がある（最近PIE Executablesをサポート）
- アプリの実行＝現在のメモリ空間への<app>.soのロードとmain関数の実行
- fork() / exec()はサポートされない
  →内部でコマンド実行するプログラムとの互換性がない
- マルチスレッドはサポートされる

複数アプリケーションの実行

- プロセスは１つだが、スレッドを新しく作って<app>.soをロード＆実行する事で複数アプリを実行する事は可能
- 但し、空間が共有されるため同じプログラムを２インスタンス起動する事が出来ない

ネットワークIOの
高速化

net channel（１）

- Linuxで提案されていたネットワークスタックの改善案
  - Socket APIはプロセスのCPUで動き、パケット受信処理はソフト割り込みがスケジュールされたCPUで動くため、ロック競合とキャッシュ競合が起きる
  - ドライバでパケットヘッダをチェックする最低限のコードを走らせて、宛先ソケットを特定
即ソケットキューにパケットをキューイング
プロセスがソケットを見に来るまで待つ

net channel（2）

- OSvではこれを実装（TCPのみ）
- TCPコネクション確立時にpacket classifierにルールを登録
- ドライバでpacket classifierを呼び出し
  マッチしたらnet_channelがこれを受け取り
  マッチしないなら通常パスへ
- マッチしたパケットはnet_channelのキューに登録してソケットでブロックしているプロセスを起床させる
- プロセスコンテキストでnet_channelにキューされたパケットがネットワークスタックで処理されプロセスに渡される

net channel bench results

	orig-1vcpu	vjnc-1vcpu	orig-4vcpu	vjnc-4vcpu
TCP MAERTS	36571.80	38833.18	30570.87	37904.51
TCP STREAM	30078.82	46939.61	24550.12	32396.08
TCP REQUEST/RESPONSE	68786.90	70834.32	62702.22	66967.52

zerocopy TX/RX

- Socket APIを通してネットワークIOを行うと必ずコピーが発生してしまう
- 独自APIを追加することでゼロコピーを実現
- OSvはプロセスとメモリ空間が共有されているのでゼロコピーの実現が一般的なOSほど難しくない

zerocopy API

- ssize_t zcopy_tx(int sockfd, struct zmsghdr *zm);
- void zcopy_txclose(struct zmsghdr *zm);
- ssize_t zcopy_rx(int sockfd, struct zmsghdr *zm);
- int zcopy_rxgc(struct zmsghdr *zm);

zerocopy bench result

- HostOS <-> KVM上のOSvで計測
  - Normal RX: 57.8 Gbps
  - Normal TX: 57.6 Gbps
  - Zerocopy RX: 72.4Gbps (≒125 %)
  - Zerocopy TX: 72.1Gbps (≒125 %)

PF filter

- FreeBSDネットワークスタックにはパケットフィルタ用のフックが存在
- これをユーザプログラムから直接呼び出し

```
int pfil_add_hook(int (*func)(void *, struct
mbuf **, struct ifnet *, int, struct inpcb *),
void *, int, struct pfil_head *);
```

- これを利用したオリジナルのmemcached実装
  https://github.com/vladzcloudius/osv-memcached
- オリジナル比の性能：122%

net channel from app

- ※テスト実装無し
- net channel APIをユーザプログラムから直接呼び出し
- ほぼnetmap/DPDKなどと同じことが実現可能
- これに限らず、OSvの独自APIを呼び出す前提ならばより高速化が可能

virtio-app

- ※テスト実装無し
- ドライバをバイパスして直接virtio-netにアクセスすることも可能
- 実装コストはnet_channelやPF filterより更に上がるが、限界まで性能をだすことが出来ると考えられる

メモリ管理

メモリ管理

- 汎用OSと同様の仮想メモリシステムを実装
- mmapによるアプリケーションからのメモリ要求をサポート（CassandraなどのアプリはJVMをバイパスしてJNIでmmapを行っている）
- 大きなマッピング要求ではhuge pageを使用
- スワップやpage evictionはサポートされない

malloc()

- malloc()はカーネルとユーザアプリで共通のメモリプールからメモリが割り当てられる
- リクエストサイズ別に複数のプールを持つ（slab allocatorと同じ）
- プールごとに最低１ページアロケート、足りなくなったらページを足す

mmap()

- mmap()でanonymous mapping要求を受けたらリクエストサイズ分のページを割り付け
- file mappingの場合はZFSへmmap要求
FreeBSD版ZFSのコードがbuffer cacheを管理

Shrinker

- OSの残りメモリ量が少なくなった時に、アプリケーション側から登録したコールバックを実行、アプリケーションからメモリを解放
- 足りなくなったらOSへ返す事によりメモリ不足を回避
- JVMやbuffer cacheで使用
- 他のアプリからも使用可（OSv版memcachedなどで実装されている）

JVM ballooning

- OS起動時にほぼ全てのメモリをJVMヒープに割り当て
- OS側のメモリが足りなくなってきたらShrinkerを使って通知、Java側でByteArrayを作成し（GC対策）、この領域をOS側へ返還
- OSが使わないメモリ領域は引き続きJava側が使用出来る
- JVM自体のコードは無変更

buffer cache shrinking

- JVMと同じくShrinkerの通知を受けてbuffer cacheをリリース、メモリをOSに返還

まとめ

- OSvは単一のアプリケーションをハイパーバイザ・IaaSでLinuxOSなしに実行するための新しい仕組み
- より簡単に管理でき、より速い性能を得られます
- Github上で開発されており、誰でも参加出来ます
- 日本語Wiki：https://github.com/syuu1228/osv-ja-wiki/wiki

OSv

designed for the cloud

アプリの移植

Hello Worldの例

hello.cc

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!" << std::endl;
}
```

Hello Worldの例：Makefile

```
CXXFLAGS  = -g -Wall -std=c++11 -fPIC $(INCLUDES)

TARGET = hello

OBJ_FILES = hello.o

all: $(TARGET).so

%.o: %.cc

	$(CXX) $(CXXFLAGS) -c -o $@ $<

$(TARGET).so: $(OBJ_FILES)

	$(CXX) $(CXXFLAGS) -shared -o $(TARGET).so
```

Hello Worldの例：Capstanfile

```
base: cloudius/osv-base

cmdline: /tools/hello.so

build: make

files:

  /tools/hello.so: hello.so
```

Hello World on Javaの移植

Hello.java

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Hello World on Javaの移植
Makefile

```
module: Hello.class

%.class: %.java

    javac $^

clean:

    rm -rf *.class
```

Hello World on Javaの例：Capstanfile

```
base: cloudius/osv-openjdk

cmdline: /java.so Hello

build: make

files:

  /Hello.class: Hello.class
```

mrubyの移植

build_config.rbを以下のように変更

```
MRuby::Build.new do |conf|
  # load specific toolchain settings
  toolchain :gcc

  # C compiler settings
  conf.cc do |cc|
      cc.flags << "-O0 -fPIC -Wall"
  end
```

memcachedの移植

- SASLを無効化
- -fpieでビルド

MySQLの移植

- rootでの実行を拒否してmysqlユーザに切り替える処理をコメントアウト
- ファイルのパーミッションをチェックしてworld writableな場合実行を拒否する処理をコメントアウト
- -fPIC -sharedでビルド

CRubyの移植

- libc関数を25個追加（うちスタブ1/3）
- obstackライブラリのスタティックリンク
- ./configureによるフラグ設定の調整
- OpenSSLのビルド・スタティックリンク

Ruby on Rails on CRubyの移植

- Bundlerが依存パッケージの解決のためRails実行時にgemコマンドを実行するのでOSvでは動かない
- Bundlerによるライブラリロードを全削除
- 手動でパッケージ群をrequire
- 全てのgemをデプロイ用ディレクトリにダウンロードしてきてOSvのファイルシステムへコピー