Seastar

高スループットなサーバアプリケーションの為の新しいフレームワーク

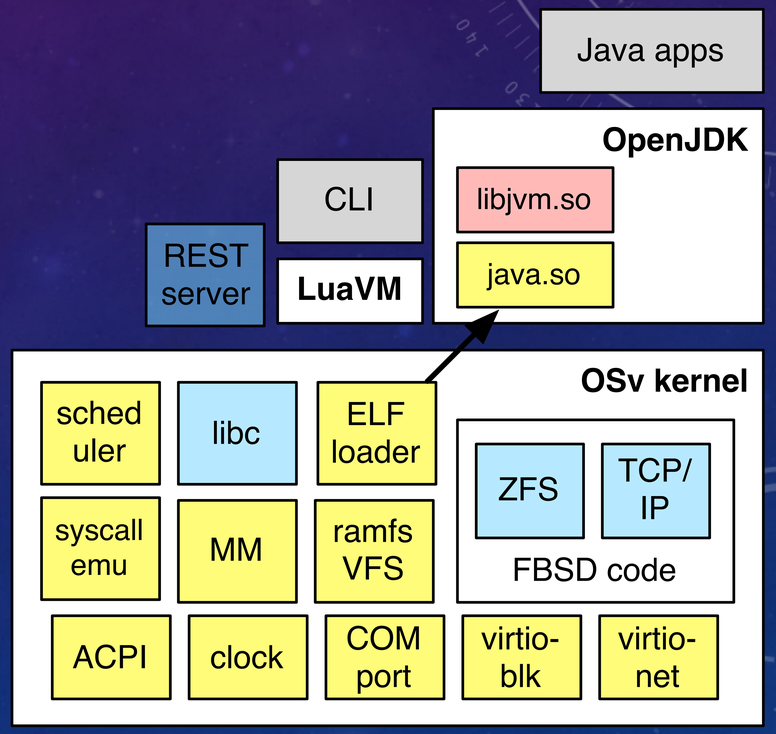
Takuya ASADA @ Cloudius Systems

Seastar?

- http://www.seastar-project.org/
- https://github.com/cloudius-systems/seastar
- @CloudiusSystems
- OSvを開発しているCloudius Systemsが最近リリースした新しいOSSプロダクト
- 高スループットなサーバアプリケーションの為の新しいフレームワーク
- Apache Lisence

OSv?

- http://osv.io
- http://github.com/cloudius-systems/osv/
- @CloudiusSystems
- JavaVMやmemcachedなど、単一プロセス＝単一アプリケーションだけを仮想マシン上で実行するための超軽量OS
- Linuxアプリケーションとの互換性を提供
- フルサイズのOSとは構造の異なる薄いレイヤ
- Linuxカーネルは不使用（コア部分はフルスクラッチ）
- 速い、軽い、管理しやすい　が目標



Cloudius Systems?

Avi Kivity | Dor Laor | Sholmi Livne | Takuya Asada | Glauber Costa | Pekka Enberg

Tomasz Grabiec | Nadav Har'El | Asias He | Amnon Heiman | Raphael S. Carvalho | Calle Wilund

Vladislav Zolotarov | Tzach Livyatan | Don Marti | Karin Nidam

目的

- NoSQL、memcachedなどのサーバアプリケーションを高速化したい
- OSvでの経験から、既存のソースコードを変更しないで行える高速化は数%～数割程度だろうということが分かっている
- 数倍高速化したい
- そのためにソースコードは書き直すことを前提にする

最近のハードウェア

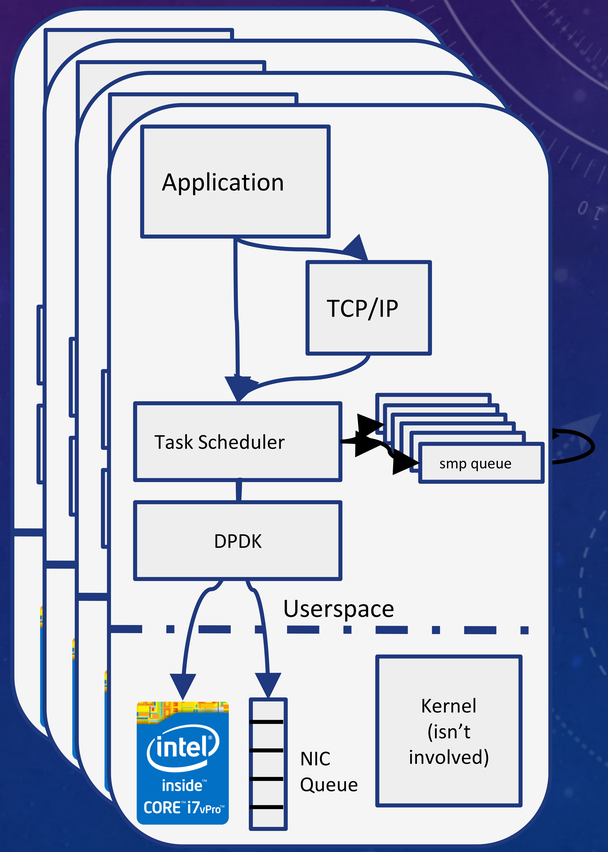
- 沢山のコア
- NUMA構成
- 1コアの性能はゆっくりとしか向上しない
- NIC、SSDの性能は劇的に向上
  - NIC: 1GbE → 10GbE, CPU: 1GHz → 3.2GHz, メモリ: CPUの1/10のペース

既存のソフトウェアスタック

- 同一の処理を行うスレッドを複数実行（複数のCPU上で並列に実行）
- 共有データにはロック機構を使う
- 問題点
  - ロックの使用は例え競合がなくてもコストが大きい
  - 競合が起きれば更にコストが高くなる
  - NUMA構成の場合、共有データが近いメモリ上になるとは限らない（よりコストが高くなる）
  - ある共有データに対する処理はどのCPU上でも実行されうるので、キャッシュ競合は起きやすい
  - そもそもカーネルのネットワークスタックをユーザランドのソケットAPIから呼び出す方式が遅い

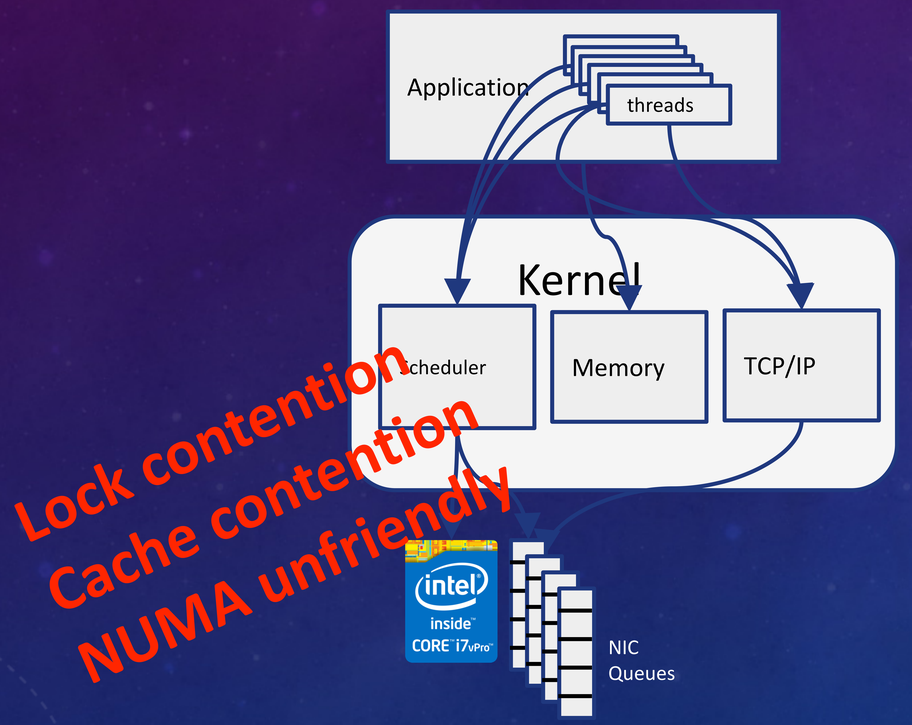
Seastarが提案する新しいプログラミングモデル

- シェアードナッシング方式
- シングルスレッドな非同期エンジンを各CPUで実行
  - 処理は小さなタスクとして記述、非同期エンジンがスケジュール＆実行
- データは共有しない
- CPU間の通信は全てメッセージパッシングで行う
- ネットワークIOにおいてはDPDKを用いてカーネルをバイパス
  フルスクラッチで実装されたTCP/IPエンジンを非同期エンジンで実行
- no thread, no context switch, no locking

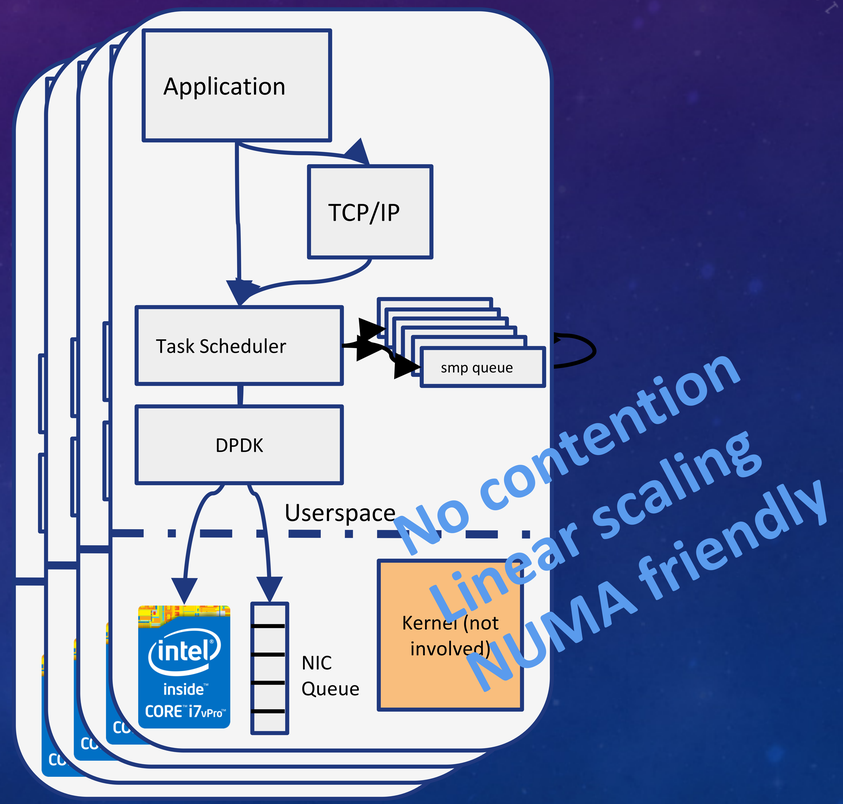


従来のスタックとSeastarの比較

Traditional stack



SeaStar's sharded stack



カーネルバイパスの必要性

- 従来のネットワークスタックだと…
- Zerocopyできない・やりずらい
- ソケット＆プロセス側とプロトコル処理側のコンテキストが別
  - CPUが別の事も→キャッシュ競合
  - レイテンシが増大
- プロトコルスタック内のロック競合
- システムコール、コンテキストスイッチのオーバヘッド

DPDK

- http://dpdk.org
- Intelが中心になって開発したカーネルをバイパスして高速に通信を行うためのフレームワーク
- 使用するCPU数は起動時に指定され、pinningされる
- メモリはhugetlbfsからアロケート、キャッシュラインまで意識したバッファ管理機構を持つ
- NICはuio・vfioと呼ばれるドライバを利用してユーザランドへ直接mmapし制御
- （基本的には）割り込み不使用、ポーリング
- ユーザランドにNICドライバが存在
- ネットワークスタックは持たない
- アプリケーションにはmbufの形でパケットが直接届く（スタックは自前で実装）
- HPETタイマードライバすらユーザランドに持っている（mmapして直接アクセス）

DPDK＋Seastarネットワークスタック

- DPDKがカーネルをバイパスしてNICから低オーバヘッド・SMPでパケットを送受信する機能を提供
- Sestarの非同期エンジン上にTCP/IPスタックを実装
- Zerocopy対応
- シェアードナッシングなのでキャッシュ競合、ロック競合しない
- 殆どユーザランドで処理するのでネットワークIOでシステムコールは呼ばれない
- プロトコルスタックとアプリは同スレッドで動くためコンテキストスイッチ回数も少ない
- ソケットAPI非互換、独自API

Seastarネットワークスタックの機能

- 対応プロトコル：TCPv4, UDPv4, IPv4, ARP, DHCP
- IPv6やマイナーなトランスポートプロトコルは非対応
- サーバアプリケーションにとって最低限な機能
- パケットフォワーディング的な処理を行う用途は今の所あまり考えられていない

Seastarのプログラミングモデル

- C++14
- Future/Promise/Continuationモデルに基づく非同期API
  - リアクティブプログラミングモデルのサブセット
  - C++標準にあるstd::future/std::promiseとは異なる独自の実装
    （C++標準の物や他言語のpromise/futureと似ている）
  - Future/Promiseベースのスケジューラ
  - Future/Promiseベースの独自API群（ネットワークIO、ファイルIO、タイマー、etc）

C++11/boostのfutures/promisesとの違い

- Seastarの実装に特化された独自実装
- ロックしない
- メモリアロケーションしない
- continuationsをサポート

Future

- Futureはまだ実行されていないかもしれない計算の結果を表す
  - ネットワークから受け取る予定のデータバッファ
  - n分後に時間が来る予定のタイマーの発火
  - （終わる予定の）ディスク書き込みの終了

Promise

- Futureの条件が満たされた結果を提供するオブジェクトまたは関数

簡単なfuture/promiseの例

```
future<int> get(); // 最終的にintが生成される事をpromiseする
future<> put(int) // intを入力する事をpromiseする

void f() {
    get().then([] (int value) { // .then()でget()が実行完了した時の処理をラムダ式で定義
        put(value + 1).then([] { // .then()でput(int)が完了した時の処理をラムダ式で定義
            std::cout << “value stored successfully¥n”; // put(value+1)が完了したらstdoutにメッセージ出力
        });
    });
}
```

Continuation

```
future<int> get(); // 最終的にintが生成される事をpromiseする
future<> put(int) // intを入力する事をpromiseする

void f() {
        get().then([] (int value) {// .then()でget()が実行完了した時の処理をラムダ式で定義
          return put(value + 1); // put()はfutureなのでputが実行完了するまでこのラムダ式は終了しない
        }).then([] {// .then()でget().then()が実行完了した時の処理をラムダ式で定義
          std::cout << "value stored successfully¥n";
        });
}
```

UDP Server

```
ipv4_addr listen_addr{port};
 chan = engine().net().make_udp_channel(listen_addr);

 keep_doing([this] {
    return chan.receive().then([this] (udp_datagram dgram) {
        return chan.send(dgram.get_src(), std::move(dgram.get_data())).then([this] {
            n_sent++;
        });
    });
 });
```

TCP Server

```
engine().listen(make_ipv4_address(addr), lo).accept().then([this, which] (connected_socket fd, socket_address addr) mutable {
        input_stream<char> read_buf(fd.input());
        output_stream<char> write_buf(fd.output());
        return read_buf.read_exactly(4).then([this] (temporary_buffer<char> buf) {
            auto cmd = std::string(buf.get(), buf.size());
            if (cmd == str_a) {
                return do_something_a();
            }else if(cmd == str_b) {
                return do_something_b();
            }
        });
});
```

Exception handling

```
void f() {
   receive().then([] (buffer buf) {
      return process(parse(std::move(buf));
}).rescue([] (auto get_ex) {
      try {
         get_ex();
      } (catch std::exception& e) {
         // your handler goes here
      }
   });
}
```

Zero copy RX

future<temporary_buffer> socket::read(size_t n);

- temporary_bufferはDPDKドライバが提供したページを直接指す
- ネットワークスタックはページテーブルを使ってバッファをsatter-gatherして連続した領域に見せられる
- 使い終わった後の回収処理は自動実行される

Zero copy TX

pair<future<size_t>, future<temporary_buffer>> socket::write(temporary_buffer);

- 最初のfutureはTCPウインドウがデータ送信可能な状態になったらreadyになる
- 次のfutureはバッファが解放可能になったらreadyになる
- どの順序でcompleteしても問題無い

複数CPUで実行

```
auto server = new distributed<udp_server>;
server->start().then([server = std::move(server), port] () mutable {
        server->invoke_on_all(&udp_server::start, port);
}).then([port] {
        std::cout << "Seastar UDP server listening on port " << port << " ...¥n";
});
```

CPU間通信

```
smp::submit_to(neighbor, [key] {
        return local_database[key];
}).then([key, neighbor] (sstring value) {
        print("The value of key %s on shard %d is %s¥n", key, neighbor, value);
});
```

ポインタ

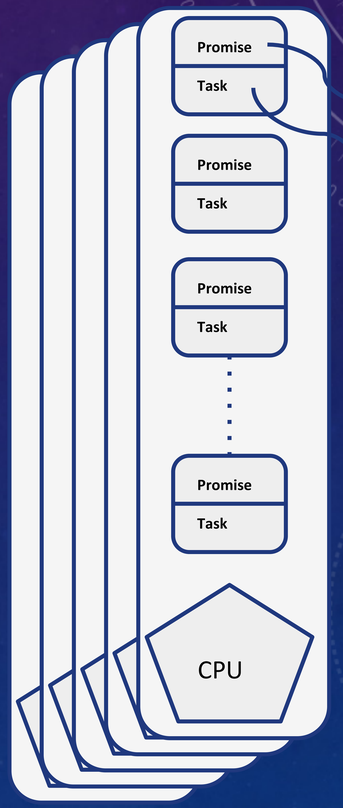
- ポインタはスマートポインタを含めて使用しない
- コピーがふさわしくない時はstd::move()を使う

Seastarのスケジューリング機構（SMP）

- app.run()から開始されたプログラムは常にcpu0で実行
  - pinningされている
- distributedクラスを用いることで任意のクラスインスタンスを全CPUで実行
  - それぞれのCPUにpinningされる
- クラスインスタンスはそれぞれのCPUで別々なので状態（変数）はCPU間で共有されない
  - smp::submit_to()を使うしかない

Seastarのスケジューリング機構（タスクの実行）

- future/promiseで書かれたプログラムは即時実行されない
- 実行される条件（.then()の手前の処理が実行されること）とラムダ式のポインタがスケジューラのランキューに登録される
- 非同期処理エンジンは条件が実行可能になったものから順に実行していく（条件以外の実行順序は保証されていない）
- シェアードナッシングかつpinningされているので、ランキューが空になっても別のCPUからタスクがマイグレーションされてくることはない



Promise is a pointer to eventually computed value

Task is a pointer to a lambda function

どこがタスクになるのか

```
engine().listen(make_ipv4_address(addr), io).accept().then([this, which] (connected_socket fd, socket_address addr) mutable {
        input_stream<char> read_buf(fd.input());
        output_stream<char> write_buf(fd.output());
        return read_buf.read_exactly(4).then([this] (temporary_buffer<char> buf) {
            auto cmd = std::string(buf.get(), buf.size());
            if (cmd == str_a) {
                return do_something_a();
            }else if(cmd == str_b) {
                return do_something_b();
            }
        });
});
```

丸で囲った部分（then()に渡されるfutureの部分）がそれぞれランキューに登録される

パケット着信処理のスケジューリング

- マルチキューNICのRSSを使用してフロー毎に着信キューを分散
- それぞれのCPUでネットワークスタックの受信処理を呼び出し

- RSSが使用出来ない環境ではソフトウェアハッシュによる分散方式にも対応可能

Seastarのメモリアロケータ

- malloc/free/realloc/memalign/new/deleteなどを上書き、独自関数を用意
- それぞれのCPUで非同期エンジン起動時に大きくmmapしてアロケータが呼ばれたらページリストから切り出し
- アロケーションと使用が絶対に同じCPUになるようにする、かつプログラムはシェアードナッシングなのでデータは基本的には共有されない

共有データのハンドル：TCPスタック

- TCPコネクションのステートは通常のOSのネットワークスタックでは共有データ（ロックが必要、全コアからアクセス）
- Seastarでは全く共有しない
- 同一コネクションのパケットがいつも同じCPUに届けば他のCPUからステート情報が見える必要が無い
- L2レイヤでフォワード先CPUを固定
- コネクション確立から切断まで同一CPUで処理

共有データのハンドル：memcachedのadd

- TCPコネクションの例と違ってパケットヘッダ数バイトを見てフォワード先を固定という訳にいかない
- distributed classを利用して、全コア上で別々にデータストア用クラスのインスタンスを保持
- このインスタンス上でaddメソッドを実行
- キャッシュ・ロック競合は起きないがデータが重複してもたれているのでメモリ使用量はncpus倍
- TCPスタックよりうまくいっていない、場合によってメモリ使用量が非効率になるかデータ共有が起きるかを選ぶ必要が出てくる

提供されるAPI

- Future/Promise/Continuation
- ネットワークIO
- ファイルIO
- タイマー
- HTTP
- JSON(swagger)
- RPC
- POSIX APIラッパー
- collectdクライアント

サンプルアプリ

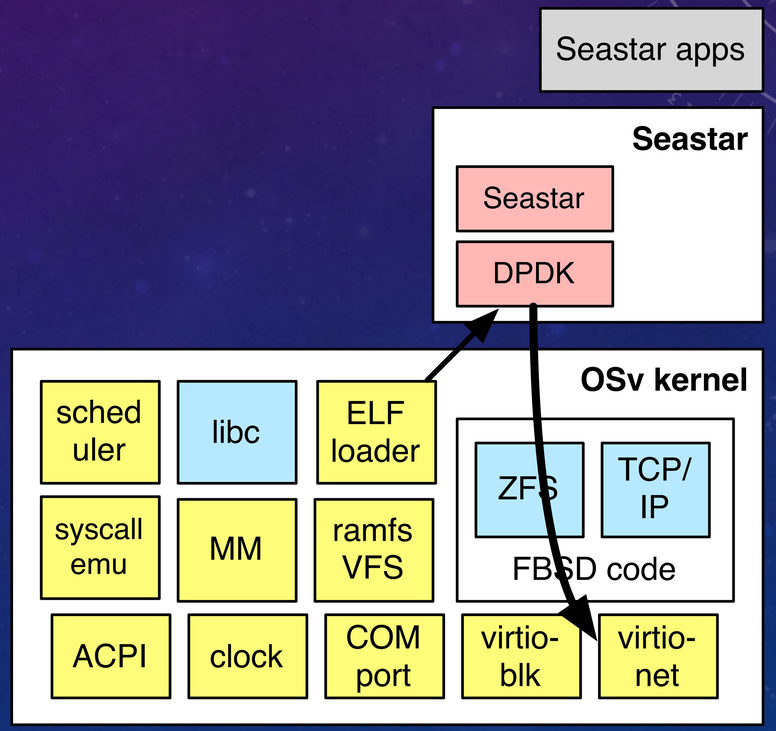
- HTTPD(swagger対応)
- memcached
- “seawreck” HTTP benchmark tool

選べる実行環境

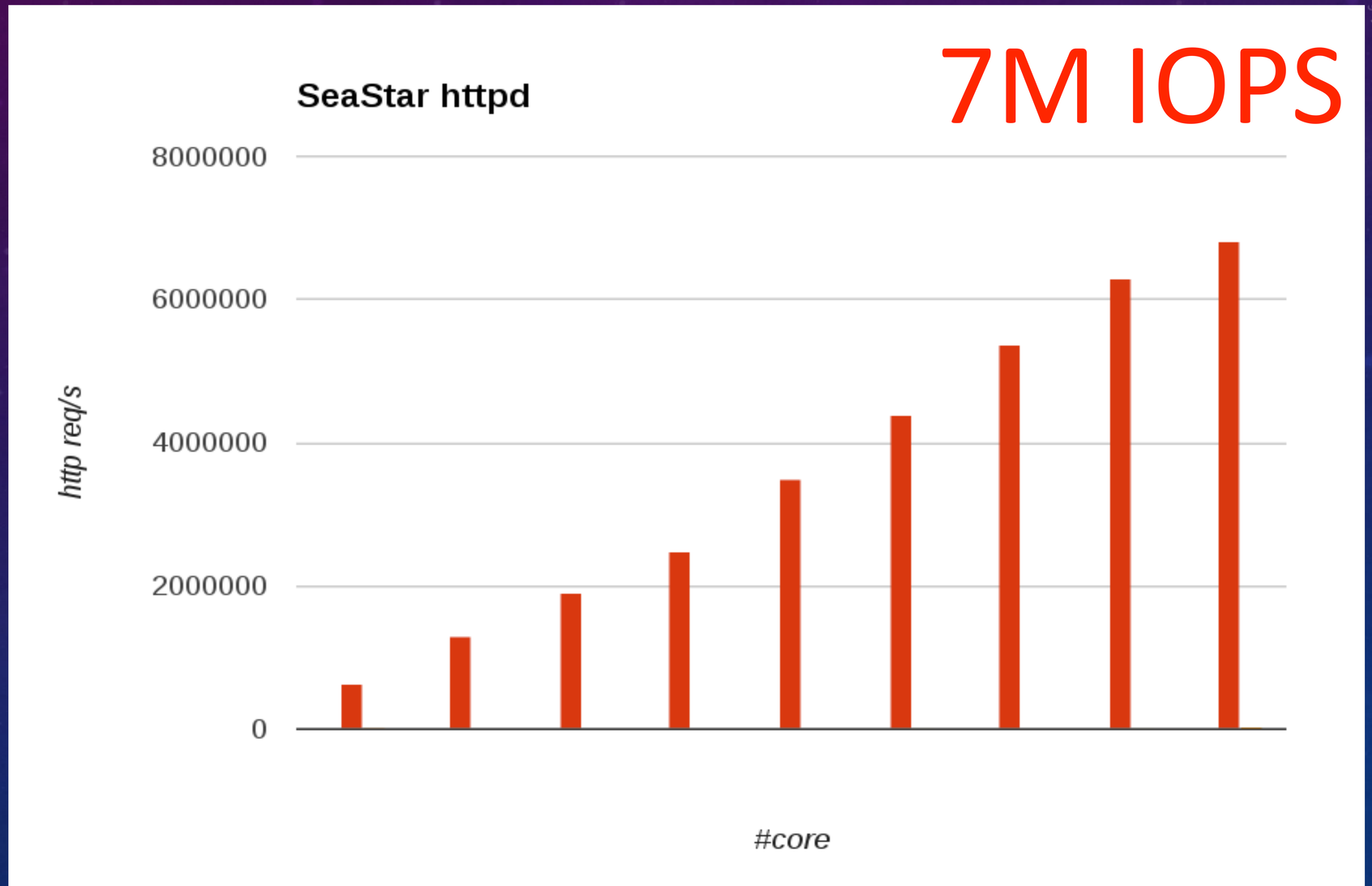
- OS
  - Linux on baremetal
  - Linux on VM
  - OSv on VM
- ネットワークバックエンド
  - DPDK + Seastarネットワークスタック
  - vhost-net + Seastarネットワークスタック
  - Xen + Seastarネットワークスタック
  - OSのネットワークスタック（ソケットAPI）
- ブロックバックエンド
  - OSのファイルシステム

Seastar on OSv

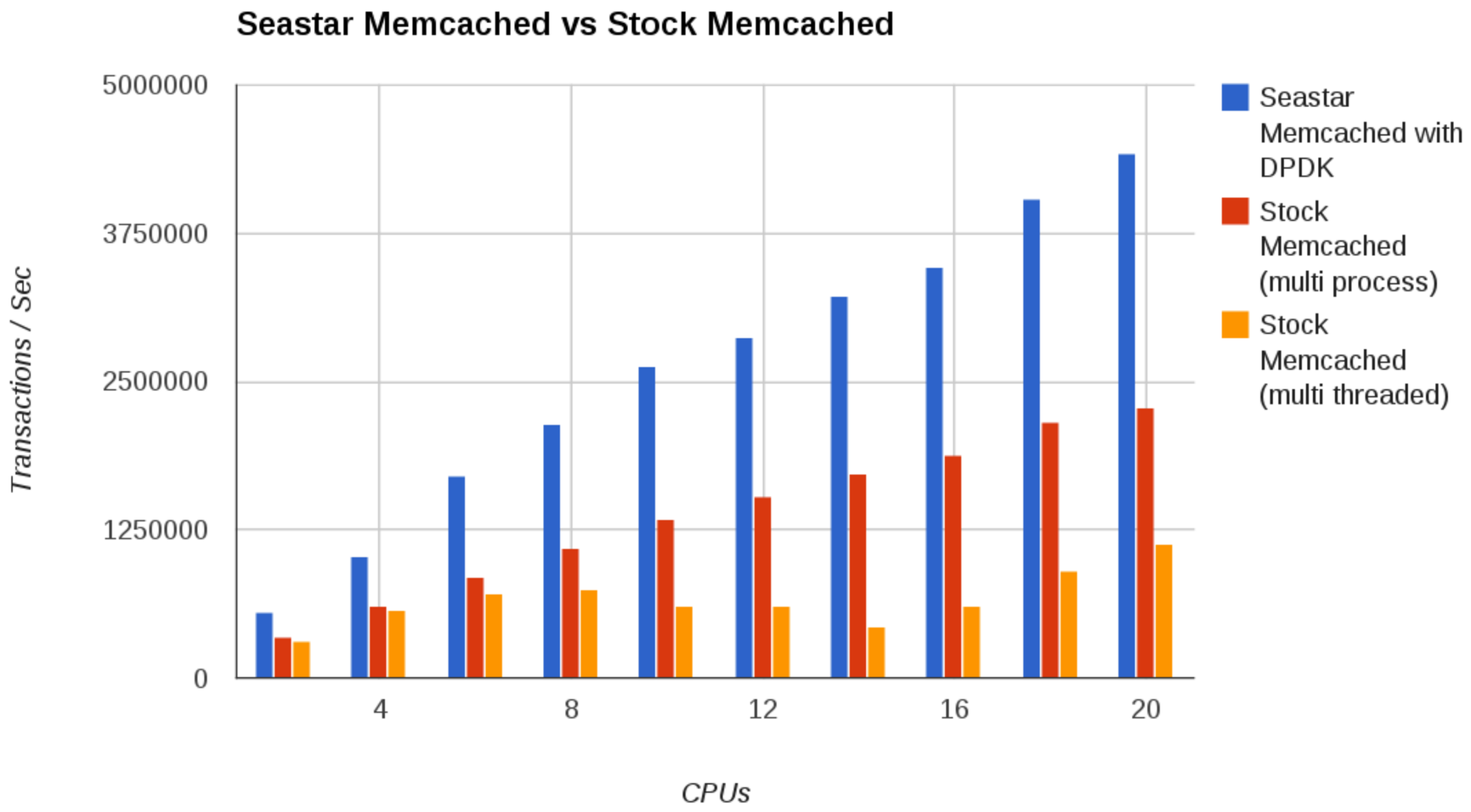
- 開発中
- ネットワークスタックを迂回、直接仮想NICへアクセス
- SR-IOV対応可
- ハードウェアを限定し、BIOS周りのコードにワークアラウンドパッチを当てる事によりベアメタル対応可（未実装）



パフォーマンス（httpd）



パフォーマンス（memcached）



計測結果

- 20コア超までリニアにスケール
- 250,000トランザクション/コア（memcached）
- 計測値がクライアントによって律速されており、計測方法の改善を行っている
- より細かいベンチマークやベンチマーク結果に基づくチューニングは後日実施予定

ベンチマーク環境

- 2x Xeon E5-2695v3, 2.3GHz
  35M cache, 14 cores
  (28 cores total, 56 HT)

- 8x 8 = 64 GB DDR4 Micron memory
- Intel Ethernet CNA XL710-QDA1（1OGbE/40GbE）

他のDPDK向けネットワークスタックとの比較

- 他のDPDK向けネットワークスタック
  - 従来のプログラミングモデルでのTCP/IPスタック実装
  - ソケットAPI（又はソケットAPIに近いもの）
  - 既存アプリケーションとの互換性は高いと考えられる
- Seastar
  - シェアードナッシングモデル
  - 独自API
  - 既存アプリケーションとの互換性は無い・又は低い
  - パフォーマンス優先
  - DPDK以外のネットワークバックエンドもサポート（コードを書き直す必要無し）
  - ファイルAPIなど包括的で最適化されたAPI（ネットワークスタックだけではなくて、必要なものは全て提供）

適用範囲

- HTTPサーバ
- NoSQL
- 分散ファイルシステム
- オブジェクトストア
- プロクシ
- キャッシュ（memcached, CDN）
- NFV
- etc

NoSQL周りのアプリケーションを移植中

他言語対応

- 他の言語にはBindingを介してサポートすることが議論されている
- 但し、スクリプト言語では言語インタプリタがボトルネックになる可能性がある
- 良い提案募集中（golangとか…？）

Thank you

http://www.seastar-project.org/

@CloudiusSystems