PyPy による OpenStack* Swift* パフォーマンスの最適化

この記事は、インテル® デベロッパー・ゾーンに公開されている「Optimizing OpenStack* Swift* Performance with PyPy*」の日本語参考訳です。

はじめに

Python* は、オープンソースの汎用プログラミング言語です。Python* ベースのアプリケーションは、クラウド・コンピューティングや同様のアプリケーションのデータセンターで使用されています。Python* コア言語 (インタープリター) を最適化することで、Python* で実装されたほぼすべてのアプリケーションのパフォーマンスを向上できます。例えば、主要なオープンソース・オブジェクト・ストレージ・ソリューションである OpenStack* Swift* は、ほとんど Python* で記述されています。この記事では、インタープリターを切り替えるだけで OpenStack* Swift* のパフォーマンスを向上できることを示します。ベンチマークを使用した測定では、スループットが最大 2.2 倍向上し、レイテンシーが最大 78% 向上しました。

この記事では、JIT Python* インタープリターである PyPy JIT を使用して、最適な OpenStack* Swift* パフォーマンスを達成するための技術的な情報を共有します。JIT ソリューションをベースにアプリケーションのパフォーマンスを最適化する最も一般的な手法 (BKM) を紹介します。

用語

CPython: Python* は一般にプログラミング言語として知られていますが、厳密には言語仕様です。これは、Python* で記述された (または Python* 仕様に準拠する) アプリケーションのソースコードは、異なるランタイム実装で解釈できるためです。CPython と呼ばれるリファレンスまたは標準 Python* インタープリターは、C プログラミング言語で実装されています。CPython はオープンソースの実装であり、多くの開発者サポート・コミュニティーがあります。実験では、ベースライン・インタープリターとして CPython を使用します。

Python* バージョン: CPython には 2.7 と 3 の 2 つの主要ブランチがあり、これらは通常 Python* 2 と Python* 3 と呼ばれています。この記事では、CPython 2.7 および CPython 2.7 と互換性のある PyPy JIT に注目します。ここでは特に明記しない限り、CPython は CPython 2.7 を、PyPy JIT は PyPy2 JIT を指します。Python 3 と PyPy 3 はここでは取り上げません。

Python* インタープリター: ここでインタープリターという用語は、ランタイム、コア言語、コンパイラーの 3 つを指します。インタープリターは、Python* アプリケーション・ソースコードを理解して、ユーザーのために実行するコード (バイナリー) です。

モジュール、ライブラリー、拡張: Python* インタープリターには、標準ライブラリーと呼ばれる一般的な関数が含まれることがあります。これは、インタープリターに同梱されています。これらの関数は、インタープリターのバイナリー (ビルトイン)、純粋な Python* スクリプトで記述されたライブラリー (*.py ファイル)、または C プログラミング言語で記述されたライブラリー (Linux* では *.so ファイル、Windows* では *.dll ファイル) に実装されています。Python* 開発者は、カスタマイズ C 拡張と呼ばれる追加の C モジュールを作成することもできます。多くの場合、一般的な Python* アプリケーション開発者にとって、モジュール、ライブラリー、拡張は同義です。

Python* アプリケーション: アプリケーションは、C モジュールの有無に関係なく、純粋な Python* スクリプト (ASCII テキストファイル) で記述できます。純粋な Python* コードはハードウェアに依存せず、ソフトウェア製品として配布できます。一方、C モジュールは通常、製品として出荷する前に特定のハードウェア向けのバイナリーにコンパイルされます。

Python* 実行: Python* の動作を簡単に説明するため、図 1 にアプリケーション、C 拡張、Python* ランタイムの関係を示します。アプリケーションは解釈されるか、ランタイムの下で実行される必要があります。インタープリター (またはランライム) は、入出力 (I/O) やネットワークなどのシステムサービスを取得するため、オペレーティング・システムのシステム・ライブラリーを呼び出すことができます。

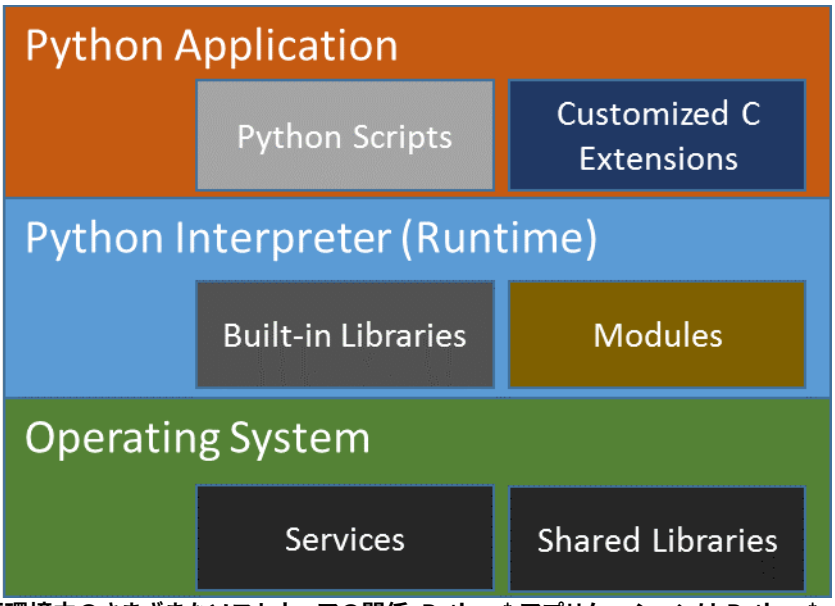

図 1. Python* 実行環境内のさまざまなソフトウェアの関係。Python* アプリケーションは Python* インタープリターに依存し、Python* インタープリターはオペレーティング・システムとサービスに依存します。

動的変換: Python* は、動的なスクリプト言語です。「動的」という用語を理解する簡単な方法は、変数が実行時にあるコード行では文字列を保持し、別のコード行では整数を保持できることを想像してみてください。これは、コンパイル時にデータ型を宣言し、実行時にそれを変更できない静的コンパイル言語と対照的です。動的変換は、パフォーマンス向上のため、実行時 (動的) にバイトコードがマシンコードにコンパイルされることと言い換えることができます。

JIT コンパイル: コンパイル済みコードのスピードと解釈の柔軟性を組み合わせた動的変換の 1 つです。JIT 手法は、HHVM、Node.js*、Lua* などのパフォーマンスを設計目標とする動的スクリプティング言語では一般的です。JIT コード領域は、実行時に実行マシン向けに動的に生成される最適化された命令が格納されるメモリー位置です。JIT コードは、実行時に何度も生成され破棄されます。

PyPy: PyPy は、実行速度を高速化する Python* の代替実装です。JIT 機能に対応していない CPython とは異なり、PyPy は JIT コンパイル機能を提供します。また、CPython はすべての Python* 実装の中で最大のユーザー数を持ち、非常に大きな開発者コミュニティーがあります。一方、PyPy は小規模な開発者コミュニティーによって維持されています。ここでは、PyPy と PyPy JIT は同義語とします。

OpenStack* と Swift*: OpenStack* は、クラウド・コンピューティング向けのオープンソースのソフトウェア・プラットフォームであり、ほとんどの場合 IaaS (Infrastructure as a Service) として配備されます。ソフトウェア・プラットフォームは、データセンター全体の処理、ストレージ、ネットワーク・リソースの多様なマルチベンダー・ハードウェア・プールを制御する、相互に関連するコンポーネント (またはサービス) で構成されています。Swift* は、スケーラブルで冗長なストレージシステムであり、OpenStack* サービスの一部です。オブジェクトとファイルは、1 つ以上のデータセンターに分散されたサーバーの複数のディスクドライブに書き込まれ、OpenStack* ソフトウェアが、クラスター全体のデータの複製と整合性を保証します。ストレージクラスターは、新しいノードを追加するだけで水平方向にスケールできます。この実験では、Swift* 2.11 を使用しました。Swift* については、以降のセクションで詳しく説明します。

ここでノードは、単一の物理マシンまたはベアメタルを指します。プロキシーサーバーは、通常、プロキシーノード上で実行し、クライアントから受信する要求をリスニングするデーモンプロセスを指します。通常、プロキシーノードは複数のプロキシー・サーバー・プロセスを実行し、ストレージノードは複数のストレージ・サーバー・プロセスを実行します。便宜上、ここではサーバーとノードは同義語とします。

Python* パフォーマンス

Python* は、主に 2 つの理由から人気があるスクリプト言語です。Python* はコードを簡単に素早く作成でき、豊富なオープンソース・ライブラリーがあります。

確かに、Python* は多くの利点を提供しますが、動的スクリプト言語であるため低速です。ハードウェアに依存しない CPython (標準 Python* インタープリター) は、純粋なインタープリター・モードで動作します。インタープリター・モードでは、実行時に (動的に) ターゲット・ハードウェア上で実行を開始する前に、アプリケーション・ソースコードを解析する必要があります。一方、静的にコンパイルされる言語で記述されたアプリケーションは、製品を配布する前のコンパイル時にターゲット・ハードウェア (このケースでは CPU) 向けのマシンコードに変換されます。そのため非常に高速に実行できます。ほとんどの開発者は、Python* が遅いことを認識しており、それを受け入れています。しかし、一部のエンタープライズ・アプリケーション (OpenStack* など) では、より高いパフォーマンスが求められます。

Python* パフォーマンスを向上する一般的なアプローチ

Python* コミュニティーによって採用されたソリューションの 1 つは、C モジュールを記述して、CPU 負荷の高い操作をハイパフォーマンスな C コードにオフロードするというものです。Cythonizing と呼ばれる同様のアプローチは、既存の Python* コードを C コードに変換します。このタイプのアプローチはコードを複雑にし、開発、配備、保守のコストが増加します。

科学分野では、アプリケーション内にデコレーターを使用してコードブロックを記述するのが一般的です。デコレーターは、Python* アプリケーション・コード・ブロックを最適化モードまたは JIT モードで実行するようにインタープリターに指示できます。

別の方法として、Golang (別名 Go*) などの異なるプログラミング言語でアプリケーションを記述し直します。この場合も、開発、実装、保守のコストは考慮すべき大きな要因です。

OpenStack* Swift* で CPython の代わりに PyPy JIT を使用する

我々は異なるアプローチを選択しました。インタープリターとして Python* JIT エンジン (PyPy JIT) を使用しました。このアプローチは、アプリケーション・ソースコードの変更や追加のハードウェアを必要としません。OpenStack* コードベースは何百万行もあることから、これが競合ソリューションに匹敵する優れたパフォーマンスとスケーラビリティーを維持しつつ、総所有コスト (TCO) を抑える最良の方法の 1 つであると考えました。

オープンソース・コミュニティーの充実

我々の目標の 1 つは、オープンソース・コミュニティーを充実させることでした。OpenStack* Swift* を使用して改善することで、PyPy の機能と可能性を広く知ってもらいたいと考えました。そのため、ここで紹介する情報は、ソフトウェア・アーキテクトと開発者の両方に役立つでしょう。読者は、PyPy のパフォーマンスの利点を理解し、PyPy の導入に関連した課題、ソリューション、最も一般的な手法 (BKM) について理解を深めることができるでしょう。

OpenStack* Swift*

概要

OpenStack* Swift* は、非常に大きな非構造化データを永続的に保存して、高速にアクセスするため高可用性を保持するように設計されています。これを実現するため、Swift* はクライアントに HTTP アプリケーショ

ン・プログラミング・インターフェイス (API) を提供しています。API 要求は標準の HTTP 動詞を使用し、API 応答は標準の HTTP 応答コードを使用します。

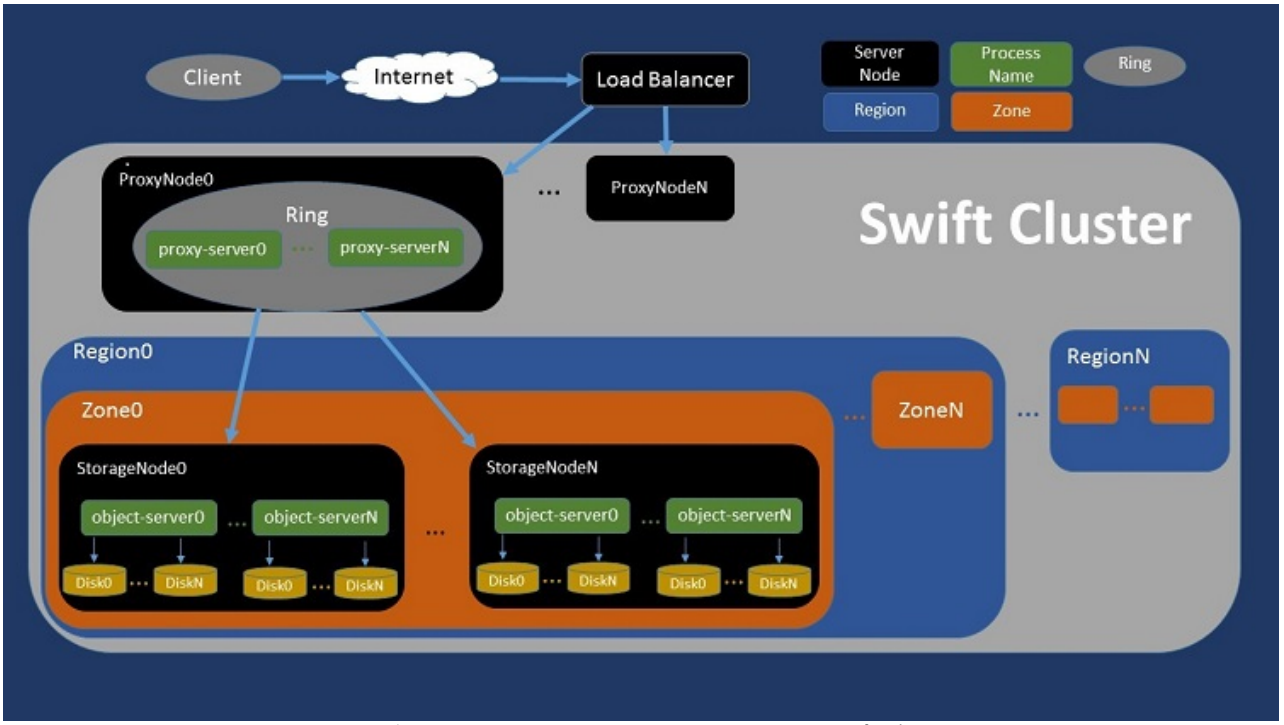

図 2. 汎用 OpenStack* Swift* アーキテクチャーの概略図。

高レベルでは、オブジェクト (写真など) を要求する顧客は、ロードバランサーを経由して Swift* クラスターに到達します。Swift* クラスターは複数のプロキシー・サーバー・ノードを含むことができ、ストレージノードは複数の地域 (例えば、アメリカのデータセンターとヨーロッパのデータセンター) に分割できます。各地域には多数のゾーン (例えば、データセンター内のフルラック) があり、各ゾーンには通常複数のストレージノードが含まれます。ユーザーからの要求は、サービス・プロバイダーの施設にあるディスクの 1 つからオブジェクト・ファイルをフェッチすることで (あるいは、ディスクへファイルを書き込むことで) 最終的に実現されます。

プロキシーサーバーとストレージサーバー

Swift* 内部には、主に 2 つの論理部分があります。

プロキシーサーバー: ほとんどの API を実装します。クライアントとストレージノード間のすべての通信の調整と仲介も行います。また、クラスター内のデータの場所を特定します。さらに、ストレージサーバーが最初の要求をどのように処理したかに応じて、クライアントに送信する適切な応答を選択します。基本的に、プロキシーサーバーは、クライアントからのネットワーク接続を受け入れ、ストレージサーバーへのネットワーク接続を作成して、2 つの間でデータを移動します。

ストレージサーバー: 永続的なメディアへのデータの保存、要求があった場合のデータ提供、複製、正当性のためのデータ監査を行います。

プロキシーサーバーのみに注目

範囲を限定するため、ここでは過去に重大な CPU 依存のパフォーマンス・ボトルネックが観察されたプロキシーサーバーのパフォーマンス向上のみに注目します。ストレージサーバーは一切変更しません。これは、ストレージサーバーで CPU パフォーマンスの問題が発生しないことを意味するものではありません。この件に関しては、個別の調査が必要です。

オブジェクトの読み書き

Swift* への新しいオブジェクトの書き込み:

- クライアントが、新しいオブジェクトの完全な名前を表す URI (Uniform Resource Identifier) へ PUT 要求を送ります。要求のボディーには、クライアントがシステムに保存したいデータが含まれています。
- プロキシーサーバー (プロキシー) が要求を受け付けます。
- そして、決定論的にデータの正しい格納場所を選択し、バックエンドのストレージサーバーとの接続を開きます。
- 次に、クライアント要求のボディーからバイトを読み取り、そのデータをストレージサーバーへ送ります。
- 冗長性のために、新しいオブジェクトは通常複数のストレージサーバーへ書き込まれます。

オブジェクトの読み取り:

- クライアントが、新しいオブジェクトの完全な名前を表す URI へ GET 要求を送ります。
- プロキシーサーバーが要求を受け付けます。
- そして、決定論的にクラスター内のデータの場所を選択し、必要なバックエンドのストレージサーバーとの接続を開きます。

ベンチマーク、システム構成、パフォーマンス・メトリック

システム・ヘルス・チェックとチューニングのマイクロベンチマークを選択

OpenStack* Swift* のパフォーマンスを評価する前に、システムレベルのヘルスチェックを行い、一貫したパフォーマンスが得られるようにシステムを構成する必要があります。この実験に取り掛かった当初、Python* 開発者コミュニティーで推奨されていた Python* ベンチマークは、Grand Unified Python* Benchmark (GUPB) スイートのみでした。最初のシステム・ヘルス・チェックとパフォーマンス比較には、これを使用しました。このベンチマーク・スイートには、50 を超えるマイクロベンチマーク (それぞれ、正規表現、レイトレース・アルゴリズム、JSON パーサーなどの特定のタスクを実行するシングルスレッドの Python* アプリケーション) が含まれていました。以前は Python* Mercurial リポジトリー (英語) でホストされていましたが、この記事の執筆時点ではプロジェクトが終了しています。現在は、同様の別のベンチマーク・スイートが推奨されており、GitHub* (英語) でホストされています。

実行時のパフォーマンスのばらつきを抑えるシステム構成の BKM

GUPB を実行していくつかの実験を行った後、実行時のパフォーマンスのばらつきを最小限に抑えるため、次の最も一般的な手法 (BKM) を確立しました。

1. すべての CPU コアを同じ固定周波数で実行するように設定します。そのためには、システムの起動時に BIOS で P ステートを無効にします。Ubuntu* オペレーティング・システムでは、sudo ユーザーで c "/sys/devices/system/cpu/cpu*/cpufreq/scaling_max_freq" と "/sys/devices/system/cpu/cpu*/cpufreq/scaling_min_freq" のパラメーター値を設定することで CPU 周波数を設定できます。アプリケーション・パフォーマンスは、実行時に CPU が 1GHz で動作した場合と 2GHz で動作した場合では大きく異なる可能性があります。動的周波数調整は、熱や冷却を含むさまざまな要因に応じて、ハードウェアまたはオペレーティング・システムによって行われます。
2. 次のコマンドを実行して、デフォルトで有効に設定される Linux* セキュリティー機能の ASLR (アドレス空間配置のランダム化) を無効にします。
   ```
   echo 0/proc/sys/kernel/randomize_va_space
   ```

次の 2 つのグラフは、ベースラインとして CPU 周波数を固定した後、ASLR を無効にする前と後に GUPB スイートの CALL_METHOD マイクロベンチマークを実行した結果の違いを示しています。このマイクロベンチ

マークは、Python* 関数呼び出しのオーバーヘッドを評価します。この例では、パフォーマンス・メトリックは実行時間 (秒) であり、値が小さいほどハイパフォーマンスです。データの散乱状況を示すため、最初にすべてのデータポイントから中央値を計算して、次に各データポイントと中央値の差分をグラフに描画しました。

ASLR を無効にする前:

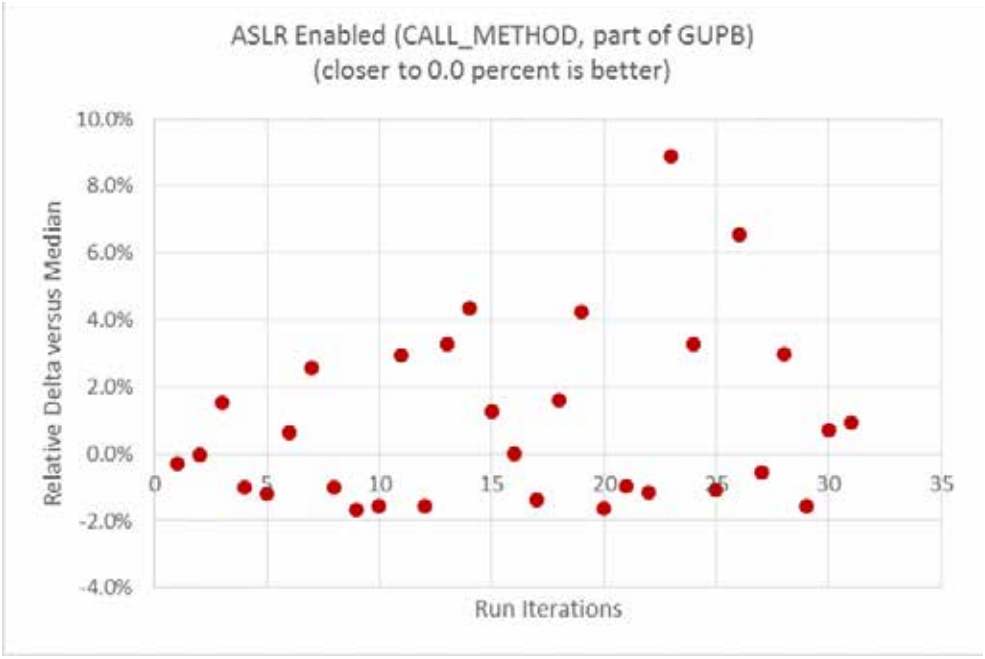

図 3. マイクロベンチマーク CALL_METHOD で測定した ASLR を無効にする前の実行時のパフォーマンスでは大きなばらつきが見られる。

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

ASLR を無効にした後:

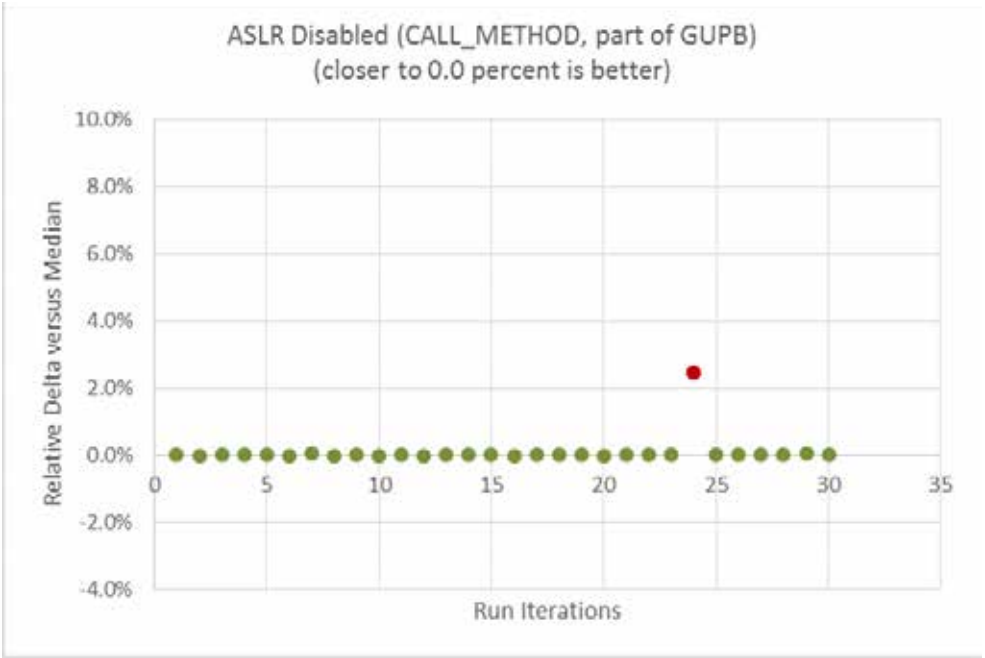

図 4. マイクロベンチマーク CALL_METHOD で測定した ASLR を無効にした後の実行時のパフォーマンスではばらつきが大幅に軽減されている。

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

ASLR が有効な場合 (図 3)、実行時の差分 (ばらつき) は最大 11% でした。繰り返し 30 回実行することでばらつきの散乱状況が分かります。ASLR が無効な場合 (図 4)、同じマイクロベンチマークとハードウェアで実行した実行時の差分 (ばらつき) はほぼゼロになり、外れ値は 1 つだけになります。

OpenStack* Swift* のパフォーマンスを測定するベンチマークを選択

HTTP ロードを発行し、OpenStack* Swift* クラスターにストレスをかけて、うまくスケーリングするには、ベンチマーク・ツールが必要です。各要求のパフォーマンスを追跡し、ベンチマーク全体の実行のヒストグラムを作成できるツールを見つける必要があります。この条件に一致したツールは、ssbench と呼ばれるオープンソースのツールだけでした。このツールは Python* で記述されており、元々 SwiftStack* によって開発されました。

ssbench は、クライアントからサーバーへの要求の送信をシミュレーションするため、プロキシーサーバーへのペイロードを生成します。実行中に実行ステータスがコンソールに表示されるため非常に便利です。また、最後にサマリーが提供され、パフォーマンス・メトリックとして 1 秒間に処理された要求数 (スループット) と、秒単位で測定された各要求を完了するのに必要なラウンドトリップ時間 (レイテンシー) が報告されます。

入力パラメーターを指定して ssbench を実行

以下は、100% READ シナリオ ([0, 100, 0, 0]) に設定された ssbench 向けのサンプル CRUD (作成、読み取り、更新、削除) 入力ファイルです。このファイルは、ssbench パッケージの scenario フォルダーにある既存のテンプレートを使用して作成しました。

この実験では、ストレージノードの I/O レートを最大にするため、オブジェクト・サイズを 4k (4,096 バイト) にしました。これは、ディスク上のファイルシステムのブロックサイズと同じです。特に明記しない限り、この実験では 4k がデフォルトのオブジェクト・サイズです。4k チャンクの読み取りまたは書き込みは、ブロック境界でアライメントされており、ディスクからバイトを読み書きする最も効率的で高速な方法です。デバイスのブロックサイズを確認する 1 つの方法は、次のコマンドを実行することです。

```
sudo blockdev --getbsz /dev/sda1 4096
```

次のサンプルコードでは、読み取り専用の場合に tiny と small の両方のオブジェクトで 4k サイズを使用しました。

```
{
 "name": "Small test scenario",
 "sizes": [{
  "name": "tiny",
  "size_min": 4096,
  "size_max": 4096
 }, {
  "name": "small",
  "size_min": 4096,
  "size_max": 4096
 }],
 "initial_files": {
  "tiny": 50,
  "small": 50
 },
 "operation_count": 500,
 "crud_profile": [0, 100, 0, 0],
 "user_count": 4,
 "container_base": "ssbench",
 "container_count": 100,
 "container_concurrency": 10
}
```

一部のパラメーターは、コマンドライン入力でオーバーライドできます。

以下は、コマンドラインの例です。

```
ssbench-master run-scenario -f ./very_small.scenario –A
http://controller:8080/auth/v1.0 -U system:root -K testpass --pctile 90 --workers
4 -r 600 -u 256 -s ./ssbench-results/very_small.scenario.out
```

このテストケースでは、v1.0 認証方式を使用しました。Keystone* サービスで別のパフォーマンス・ボトルネックが見つかり、個別の調査が必要なため、ここでは v2.0 や Keystone* は使用しませんでした。

上記のコマンドラインは、256 の同時使用ユーザーを指定しています。入力ファイルで設定されている user_count 値は、この値でオーバーライドされます。特に明記しない限り、この実験ではこれがデフォルト値です。コマンドラインは、結果を ASCII テキスト形式で “very_small.scenario.out” に出力します。この実験では、クライアント・マシンで利用可能な 4 つの CPU コアに合わせて、ワーカー数は常に 4 とします (--workers 4)。一方で、サーバーへの負荷やストレスレベルを調整するため、同時使用ユーザーは変更します。"-r 600" パラメーターは、各 ssbench を 600 秒間継続して実行するように指定しています。

Memcached のチューニングとスケーリング問題の解決

Memcached は、OpenStack* Swift* ソフトウェア・スタックの重要なコンポーネントです。Linux* サービスとして実行し、キー/値ペアをキャッシュとして提供します。アプリケーションは C で記述されており、非常に効率的です。しかし、正しく使用しないと、Memcached は次のスループットのグラフに示すようなパフォーマンスの問題を引き起こします。

最初に、我々は Memcached のパフォーマンスの問題とその症状を調査しました。次に、パフォーマンスを最適化するため、Memcached の実行方法を調整しました。

図 5 は、ssbench 実行時のスループットのヒストグラムです。それぞれの点は、実行中の 1 秒間の単一のスループットを表します。

Memcached の問題を解決する前は、赤色の点がヒストグラムのより広い範囲に散乱していました。

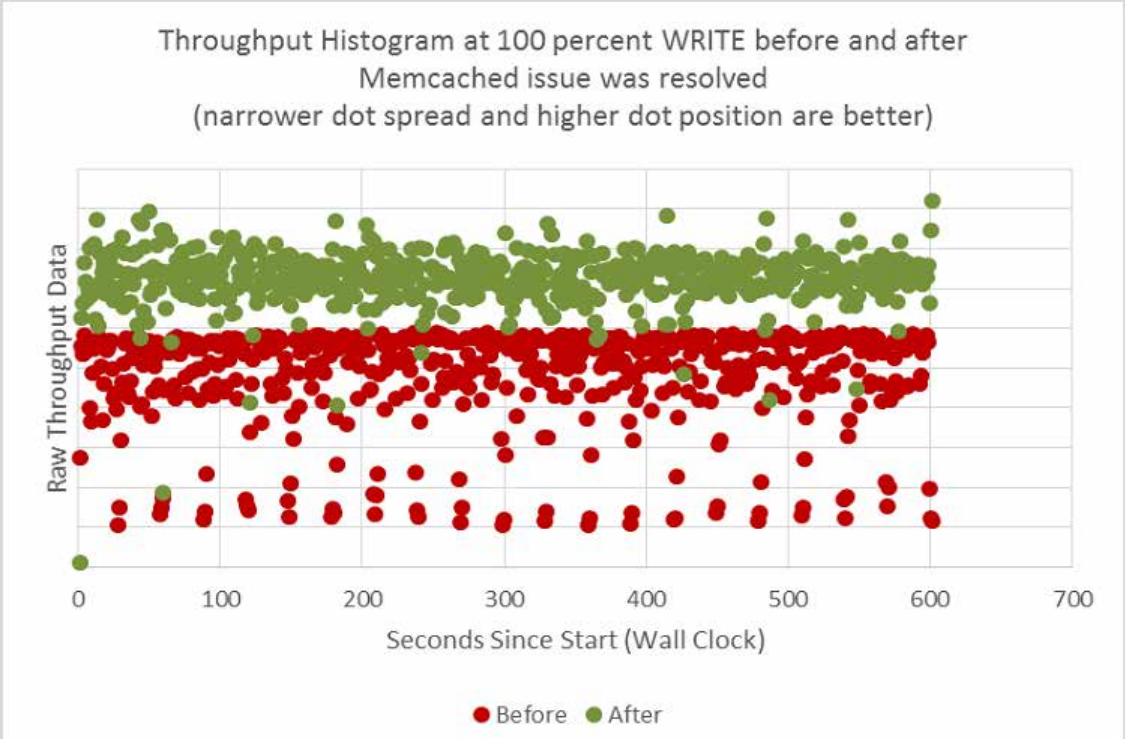

図 5. Memcached をチューニングする前と後のスループット・データ・ヒストグラムの比較。
赤色の点は「チューニング前」の結果を示し、緑色の点は「チューニング後」の結果を示します。

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

ssbench クライアントのターミナル/コンソールには、Swift* プロキシーサーバーから多数のエラーが返されました。プロキシーサーバーのログファイルを調査したところ、大量のタイムアウトが発生していたことが分かりました。原因として、ストレージノードの I/O ボトルネック、ネットワーク・パケットの断続的な損失、プロキシーまたはストレージ CPU の飽和など、複数の要因が考えられます。実際の環境では、タイムアウトの原因はさらに複雑となる可能性があります。

しかし、我々が目にしたエラーの原因は、一般的なものではありませんでした。アプリケーション・コードのインストルメンテーションとネットワーク・トラフィックのスニッフィングを行うことで、原因を Memcached に絞り込むことができました。具体的には、Memcached の応答が想定よりも遅かったことでした。

これは驚きでした。最初に我々は、Memcached ベンチマークを実行して、実際に応答が想定される時間範囲内であることを確認しました。次に、ワークロードを分散しました。2 ～ 5 の個別のノードに Memcached をインストールして実行しました。この Memcached の負荷分散は、/etc/swift/proxy-server.conf ファイルの [filter:cache] で設定しました。次に例を示します。

```
memcache_servers = 192.168.0.101:11211,192.168.0.102:11211,192.168.0.103:11211
```

(memcache サーバーが 3 台の個別のマシンから 3 つの異なる IP アドレスで実行されるように指定)

ワークロードの分散はある程度の効果がありましたが、Memcached サービスをホストするため 5 つのノードを追加した後も問題が解決されませんでした。また、同時使用ユーザー数 (-u オプション) を ssbench から引き上げてストレスレベルを上げたところ、プロキシーサーバーから返される応答エラーの数が増えました。詳細な調査の結果、プロキシー設定ファイルで 1 つのパラメーター (proxy-server.conf) がコメントアウトされていることが判明しました。

```
#memcache_max_connections = 2
```

つまり、Memcached サーバーはデフォルト値である最大同時接続数 2 で実行されるように設定されていたのです。この実験では、この値を 256 に変更しました。また、同じプロキシーサーバー上で実行する Memcached サーバーを 1 つのみにしました。これらの変更により、すべてのタイムアウト・エラーが解決されました。

さらに、データの散乱が大幅に減少しただけでなく、最大スループットが向上しました (図 5 の緑色の点)。Memcached をチューニングしただけで、平均で 1.75 倍のスピードアップを達成し、スループットが 75% 向上しました。

ssbench の実行で一貫したスループットを生成するためのシステム・チューニングと設定が完了し、適切なベースラインを作成できたため、PyPy を使用する前と後のパフォーマンス・データを収集して結果を比較する準備が整いました。

PyPy による OpenStack* Swift* パフォーマンスの向上

100% READ と 100% WRITE の 2 つの実験で ssbench を 600 秒間ずつ実行しました。ssbench は実行全体のスループットを追跡できるため、ベンチマーク実行の最初から最後までの CPython と PyPy のスループットの差を確認することが可能です。図 6 では、緑色の点は PyPy の 100% READ のスループットへの影響を示し、青色の点は 100% WRITE のスループットへの影響を示します。0.0% よりも上にある点は、PyPy によりパフォーマンスが向上したことを示しています。

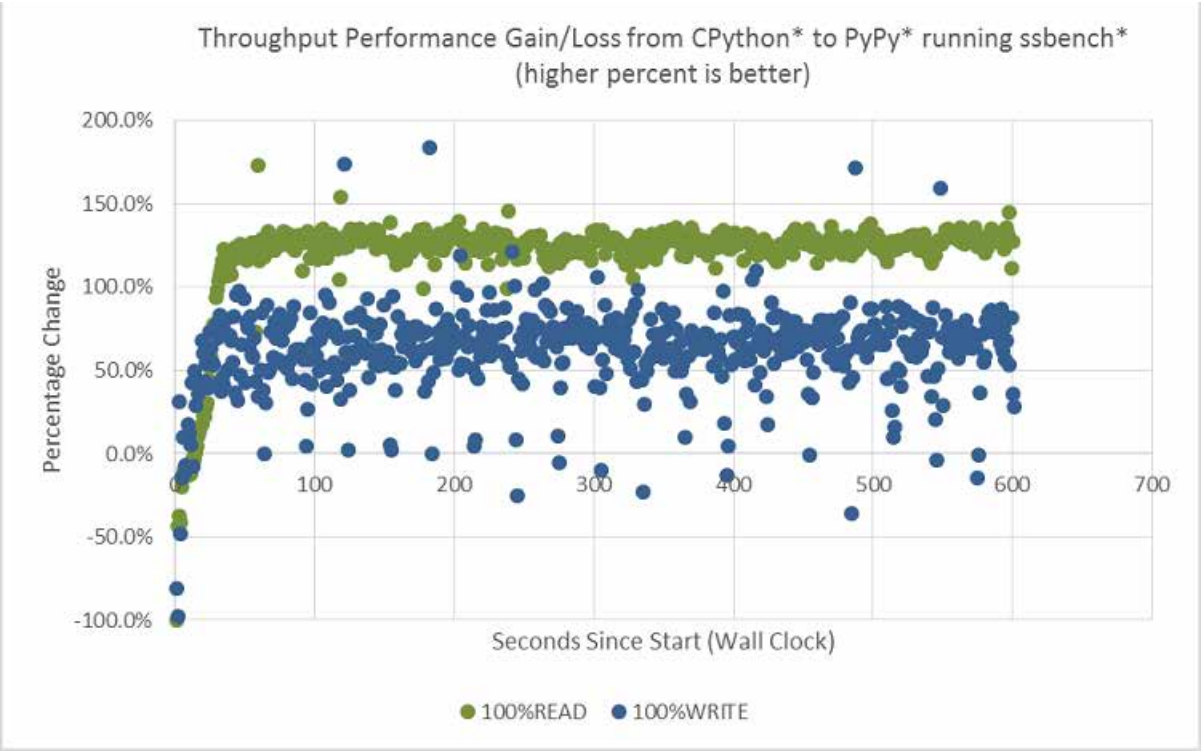

図 6. OpenStack* Swift* で ssbench* を 100% READ と 100% WRITE で実行した場合の PyPy と CPython の差分を示すヒストグラム。

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

図 6 から、READ (緑色)/WRITE (青色) に関係なく、PyPy のほうが CPython よりも全体的にスループットが高いことが分かります (多くの点が 0.0% よりも上にあるため)。しかし、最初の 10 秒間は、PyPy が CPython を下回っています。次の 20 秒ほどで PyPy は上昇を続けて徐々に CPython を上回り、横ばいになります。また、グラフから、WRITE よりも READ のほうが PyPy によるパフォーマンス・ゲインが大きいことが分かります (READ を表す緑色の点の多くが WRITE を表す青色の点の上にあります)。

PyPy ウォームアップ

最初の 30 秒間の PyPy の動作は、ウォームアップと呼ばれます。この間 PyPy インタープリターは、(CPython と同様に) インタープリター・モードで実行しながら、命令トレースの収集、JIT 命令の生成、生成された命令の最適化を行います。これは、PyPy の初期オーバーヘッドであり、避けることはできません。

100% READ において PyPy は CPython よりも最大 2.2 倍スピードアップ

ウォームアップ期間の後、PyPy と CPython の差分は横ばい (ほぼ一定のスループット率) になります。前述のとおり、サーバープロセスは長時間実行されるため、平均値や 30 秒を超える長い実行時間の傾向を比較することは妥当です。このことを考慮して、100% READ では、CPython と比較して PyPy はスループットを約 120% 向上し、2.2 倍のスピードアップを達成しています。

100% WRITE において PyPy は CPython よりも最大 1.6 倍スピードアップ

図 6 の 100% WRITE の比較では、PyPy はウォームアップ後に平均で CPython よりもスループットが約 60% 向上し、1.6 倍のスピードアップを示しています。100% READ の 2.2 倍のスピードアップと比べるとわずかですが、これは WRITE のほうが READ よりも I/O の制約を受けるためです。READ では、さまざまなデータ・キャッシュ・メカニズムにより、OpenStack* Swift* ストレージノードのディスク上で実際の I/O アクティビティーが軽減され、CPU 依存になります。100% WRITE の実験では、ディスク使用率が 60% に達し (オープンソース・ツールの iostat で収集されたデータに基づく)、I/O への依存が高まっています。同じ理由から、いく

つかの青色の点は 0.0% を下回っており、PyPy によるパフォーマンスの低下を示しています。このような場合、高速なプロキシーサーバーが、バックエンドのストレージサーバーの処理能力を超える要求を送信するとディスクに負荷がかかり、すべてが遅くなります (利点よりも弊害のほうが上回ります)。

オブジェクト・サイズと同時使用ユーザー数のパフォーマンスへの影響

前述の実験では、オブジェクト・サイズは 4KB、同時使用ユーザー数は 256 に固定していました。次の実験では、さまざまなオブジェクト・サイズ (I/O に影響) と同時使用ユーザー数 (負荷に影響) を使用して PyPy によるパフォーマンスへの影響を調査します。図 7 に結果を示します。グラフから、オブジェクト・サイズが 1KB の 100% READ では、約 2 倍 (100%) のパフォーマンス・ゲインを達成していることが分かります。しかし、オブジェクト・サイズが 10MB の READ では、パフォーマンス・ゲインは 10 ～ 20% と非常に小さくなります。WRITE のパフォーマンス・ゲインは、オブジェクト・サイズが 1KB の場合は 8 ～ 20%、10MB の場合は 40% を超えます。この実験では、同時使用ユーザー数を 200 から 300、400 へと増やすことで、パフォーマンス・ゲインがわずかに向上しました。

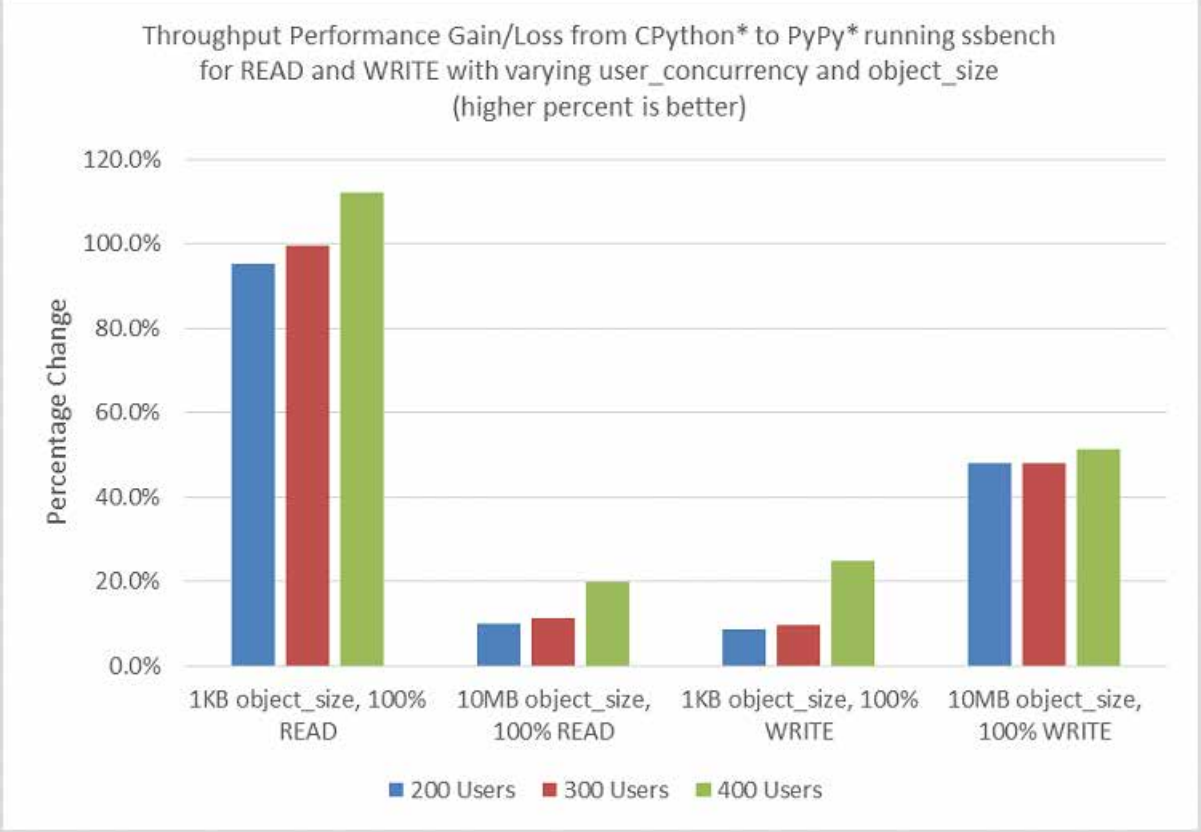

図 7. さまざまなオブジェクト・サイズと同時使用ユーザー数で 100% READ と 100% WRITE を実行した場合の CPython と PyPy の ssbench スループットの比較。

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

READ の応答レイテンシーを最大 78% 向上

RPS を測定するスループットは、パフォーマンスを評価する 1 つの方法です。別の方法として、要求の処理にかかった時間を秒単位で測定できます。図 8 から、100% READ の応答レイテンシーは、オブジェクト・サイズが 4k で同時使用ユーザー数が 256 の場合は 57% 向上し、オブジェクト・サイズが 4k で同時使用ユーザー数が 2048 の場合は 78% 向上することが分かります。スループットの影響は、サービス・プロバイダーのバックエンド・サーバー側に現れるのに対して、レイテンシーの影響は、顧客が直接感じるものです。そのため、レイテンシーは、ピーク・トラフィック時の顧客への影響をシミュレーションしたり、予測するのに役立ちます。この結

果では、同時使用ユーザー数が 2048 の高負荷状態のほうが (同時使用ユーザー数が 256 のベースラインと比較して) PyPy によるパフォーマンスの利点が大きくなっています。

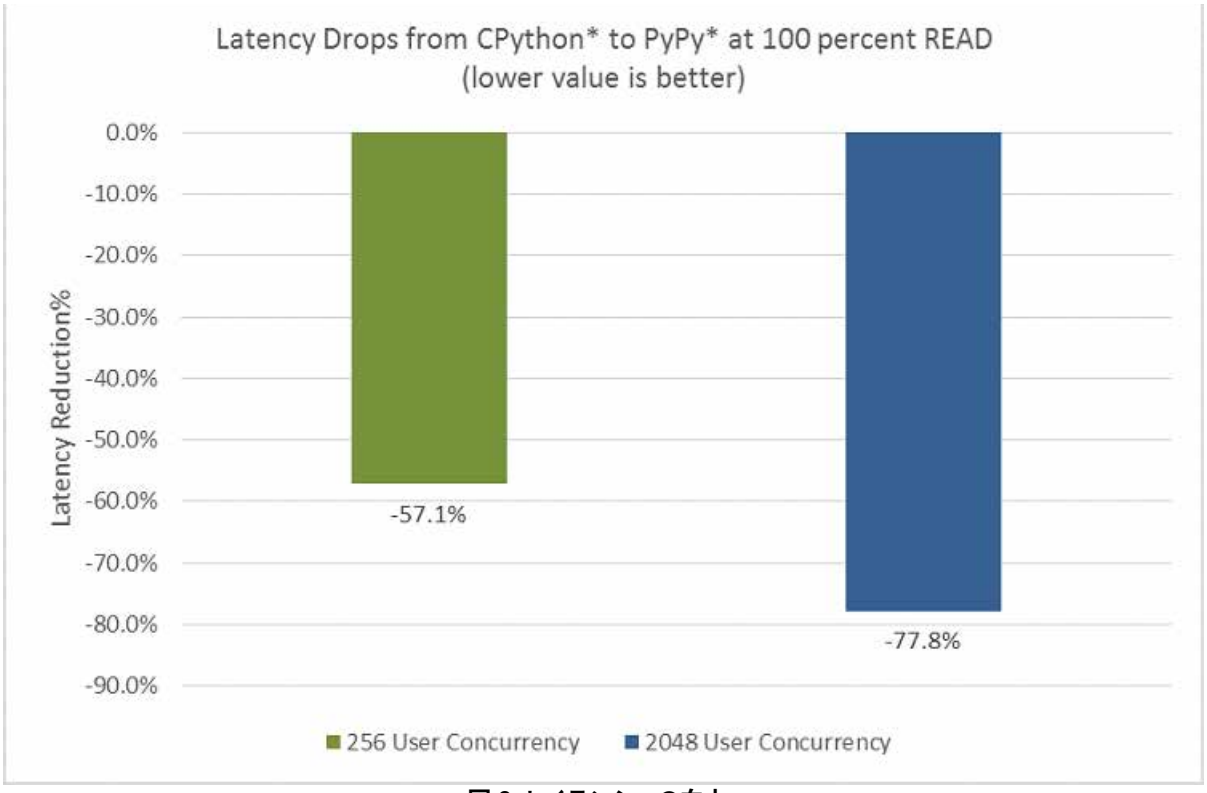

図 8. レイテンシーの向上。

ベンチマーク結果は、「Spectre」および「Meltdown」と呼ばれる脆弱性への対処を目的とした最新のソフトウェア・パッチおよびファームウェア・アップデートの適用前に取得されたものです。パッチやアップデートを適用したデバイスやシステムでは同様の結果が得られないことがあります。

追加情報

PyPy によりパフォーマンスを向上する方法と理由

ここでは、PyPy によりパフォーマンスを向上できることを示しました。PyPy は、マシンコードや JIT 命令を動的に最適化して、ループ内で繰り返し行われるデータ型のチェックなどの一部の冗長な呼び出しを軽減または排除します (変数のデータ型が一定の場合)。つまり、PyPy は 同じ作業量に必要な CPU サイクルを減らします。これは、CPU パス長の短縮と呼ばれます。より詳細な考察には、専用のホワイトペーパーが必要でしょう。

PyPy は CPU 依存の場合にのみ役立つ

PyPy の価値を発揮するには、アプリケーションは CPU 依存であり、メモリー、ネットワーク、I/O などのレイテンシーの長いほかのハードウェアの制約による影響が比較的小さくなくてはなりません。実験では、適切なペイロードを作成するため、高速なネットワーク・スイッチとメモリー DIMM を使用するとともに、ssbench の同時使用ユーザー数を調整しました。最初の実験では、テストに使用したインテル® Xeon® プロセッサー・ベースのプロキシーノードの CPU 使用率が 10% と非常に低く、これは ssbench の実行に使用する同時使用ユーザー数を変更しても変わりませんでした。詳細な調査の結果、ストレージがボトルネックであることが分かりました。この実験用のシステム構成や使用シナリオでは、プロキシーノード上で Python* インタープリターを変更しても大きなパフォーマンス・ゲインは得られないことが分かりました。プロキシーノード上では CPU 依存にならないためです。しかし、前述のとおり、この実験ではプロキシーノードにのみ注目したいと考えていました。そこで、プロキシーノードがパフォーマンス・ボトルネックとなるように、利用可能な CPU コア数を 10 に減らしました。その結果、プロキシーノード上のすべての CPU コアの使用率が 80% 以上になりました。しかし、

前述のとおり、I/O によってまだパフォーマンスが部分的に (特に 100% WRITE において) 制限されていました。

Linux* オペレーティング・システムでは、Top というツールを使用して CPU 使用率を動的に監視できます。アプリケーションが CPU 依存でない場合、プログラミング言語自体のパフォーマンスは重要ではなくなり、手動でアセンブリー・コードを調整しても効果がありません。クライアント・マシンで ssbench のパフォーマンス・データを収集する間、プロキシーサーバーとストレージサーバーの CPU、メモリー、ネットワーク、I/O を含むシステムのパフォーマンス・データも収集しました。高負荷時の CPU 動作を監視するため、Linux* オペレーティング・システムの Perf ツールを使用して CPU プロファイルも収集しました。Perf データから、Python* インタープリターが CPU サイクルの 80% を占めており、CPython のメインループ関数である PyEval_EvalFrameEx は CPU サイクルの 24 % を占めていることが判明しました。Top の結果は CPU 依存であることを示しており、PyPy を使用する前の Perf データは CPython がパフォーマンス・ボトルネックであることを示していました。

この記事で紹介した実験結果は、PyPy によってパフォーマンスの利点が得られる可能性があることを示しました。複数の要因が存在し相互に影響し合う実際の環境では、Python* ランタイムの最適化を検討する前に総合的なシステム・パフォーマンス解析を行うことを推奨します。

PyPy とモジュールの互換性

OpenStack* Swift* の設計において重要なことの 1 つは、サードパーティー拡張をサポートしていることです。これらのミドルウェア・モジュールは、通常プロキシーにロードされ、要求または応答のいずれかで動作します。Swift* の多くの機能はミドルウェアとして実装されています。実際に、エコシステムにはこの方法で実装されているさまざまな機能があります。例えば、S3 API の互換性などです。Swift* のプロキシーサーバーのパフォーマンスを向上する方法を検討する場合、要件の 1 つとして、これらのサードパーティー・モジュール (C 拡張を含む) とのシームレスな統合を確実にする必要があります。実験では、Eventlet モジュールのメモリーリークの問題など、この記事の執筆時点で分かっている PyPy と OpenStack* Swift* の統合に関する問題をすべて解決しました。Eventlet の問題は、パッチが作成され、Eventlet 0.19 以降に組込まれました。

OpenStack* Swift* アプリケーションの最適化

OpenStack* Swift* のソースコードとランタイムの特性を解析中に、いくつかの追加の問題が見つかりました。まず、プロキシーサーバー上の最もホットな Python* アプリケーション・コードが、ストレージサーバーと通信中にネットワーク・ソケットの作成と破棄を繰り返して CPU サイクルの大半を占めていました。これはアーキテクチャーの問題であり、実装言語には関係ありません。

ストレージサーバーと通信する別の方法があります。プロセスが起動して実行を開始したらすぐに、プロキシーサーバーから個々のストレージサーバーへ接続プールを作成することを推奨します。この変更により、パフォーマンスが改善される可能性が高いでしょう。また、プロキシーサーバーのソフトウェア・レベルで優れたデータキャッシュを実装すると、ストレージサーバーへのアクセスを最小化または排除できます。これにより、READ などの使用シナリオをより効率的に実行できるようになります。

まとめ

この記事では、アプリケーション (Swift*) ソースコードを変更したり、ハードウェアをアップグレードすることなく、PyPy により OpenStack* Swift* のスループットを最大 2.2 倍スピードアップし、応答レイテンシーを最大 78% 向上できることを示しました。これは、総所有コスト (TCO) を増やすことなく既存のシステム・パフォーマンスを向上できる良い方法であり、OpenStack* Swift* アーキテクトと開発者にお勧めのアプローチです。ここでは、ASLR と Memcached を適切に設定することでシステムを最適な状態にするチューニングの BKM も共有しました。これらの技術情報は、パフォーマンスが設計要件の 1 つであるほかの大規模な Python* アプリケーションにも適用できるでしょう。

システム構成

	クライアントノード	プロキシーノード	ストレージノード
ノード数	1	1	15
プロセッサー	インテル® Core™ i7-4770 プロセッサー	インテル® Xeon® プロセッサー E5-2699 v4	Intel Atom® プロセッサー C2750
ノードごとの CPU コア数	4	10	8
CPU 周波数	3.40GHz	1.80GHz	2.40GHz
CPU ハイパースレッディング	オフ	オフ	オフ
メモリー	8GB	32GB	8GB
オペレーティングシステム	Ubuntu* 14.04 LTS	Ubuntu* 14.04 LTS	Ubuntu* 14.04 LTS
OpenStack* Swift* バージョン	N/A	2.11	2.11
ssbench バージョン	0.2.23	N/A	N/A
Python* バージョン	2.7.10	2.7.10	2.7.10

コンパイラーの最適化に関する詳細は、最適化に関する注意事項を参照してください。