

ORACLE®

# JDK 8 JVM Improvements

David Buck  
Java SE Sustaining Engineering  
日本オラクル株式会社  
#jdt2014\_C2



MAKE THE  
FUTURE JAVA

# Java Day Tokyo 2014

#javadaytokyo

ORACLE®

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

# Hello Everybody!

- バック・デイビッド
- Java SE の Sustaining Engineering
- 元 JRockit のエンジニア
- HotSpot と JRockit 両方の JVM を担当
- Blog: <https://blogs.oracle.com/buck/>

# Agenda

- PermGen の廃止
- Tiered Compilation
- その他の改善

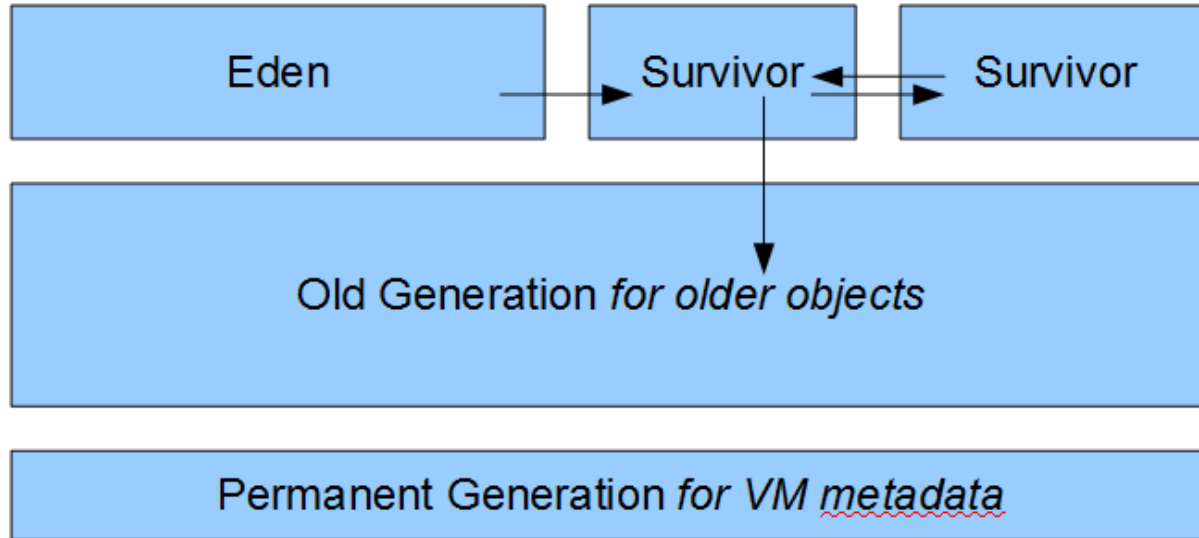
# PermGen の廃止

JEP 122: Remove the Permanent Generation

# PermGen とは

- クラスのメタデータなどを格納する場所
- 例えば：
  - バイトコード
  - intern された文字列
  - static フィールドの値

# Java Memory Layout with PermGen



# PermGen の良くないところ

- サイジングが困難
  - 考えられるポイント
    - ロードするクラスの数
    - ロードするクラスの大きさ
    - クラスのオーバーヘッド
  - 結局、試行錯誤
  - デフォルトが小さい : 64MB-85MB
- パフォーマンスへの悪影響

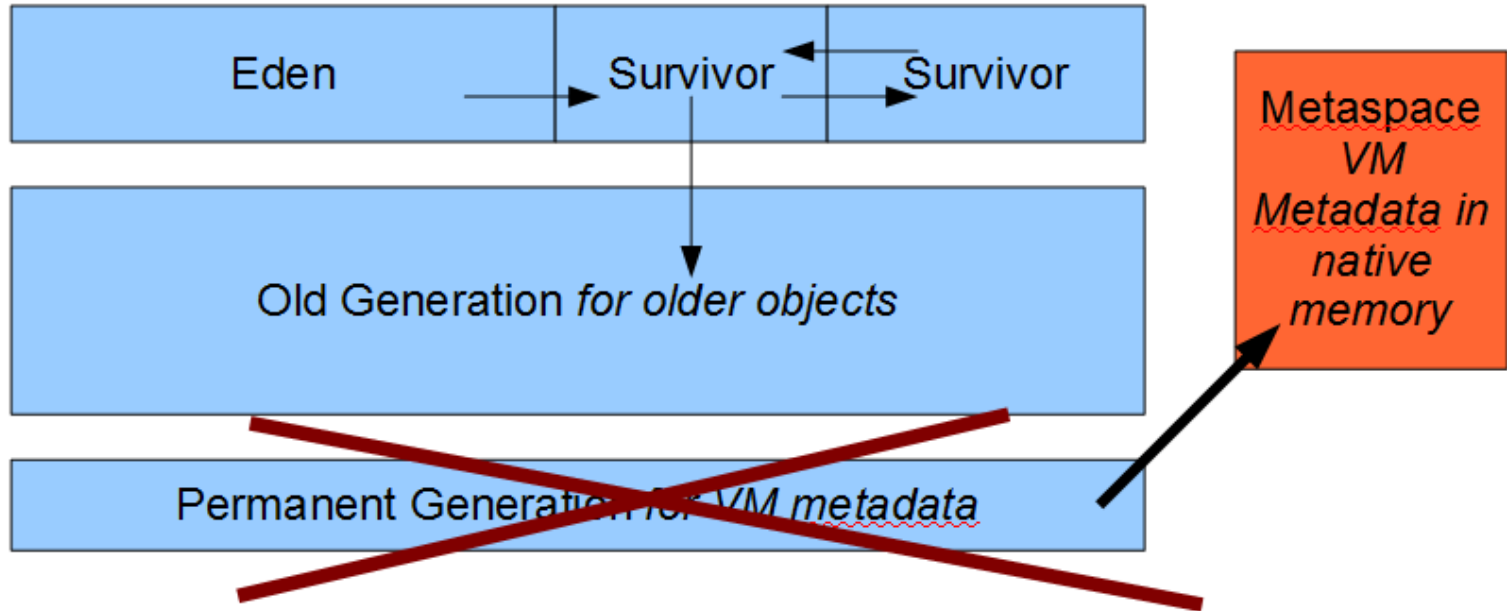


# PermGen の良いところ

# ソリューション

- PermGen を廃止
- Metaspace メタスペースを導入

# Where is Metadata now?



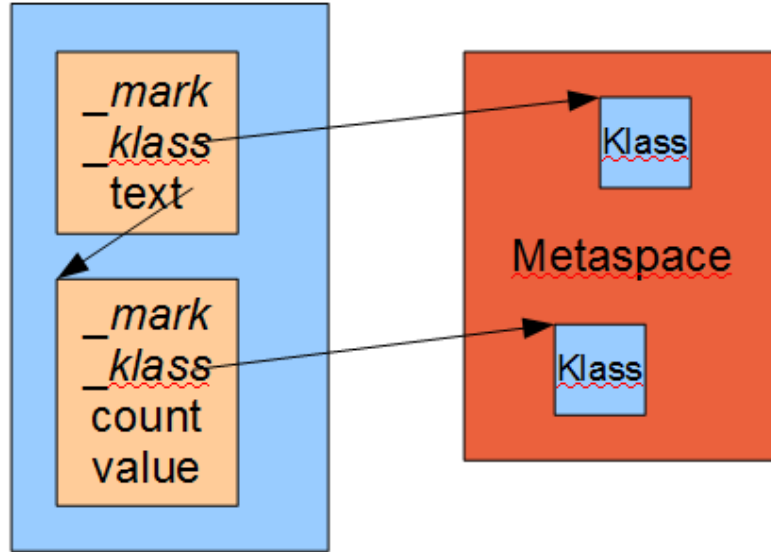
# Metaspace vs. PermGen

- (デフォルトでは) サイズの制限がない  
ユーザが意識する必要がない
- GC システムが管理する必要がない  
GC のパフォーマンスが良くなる
- アンロードを ClassLoader の単位で行う  
断片化が発生しにくい

# Java Object Memory Layout

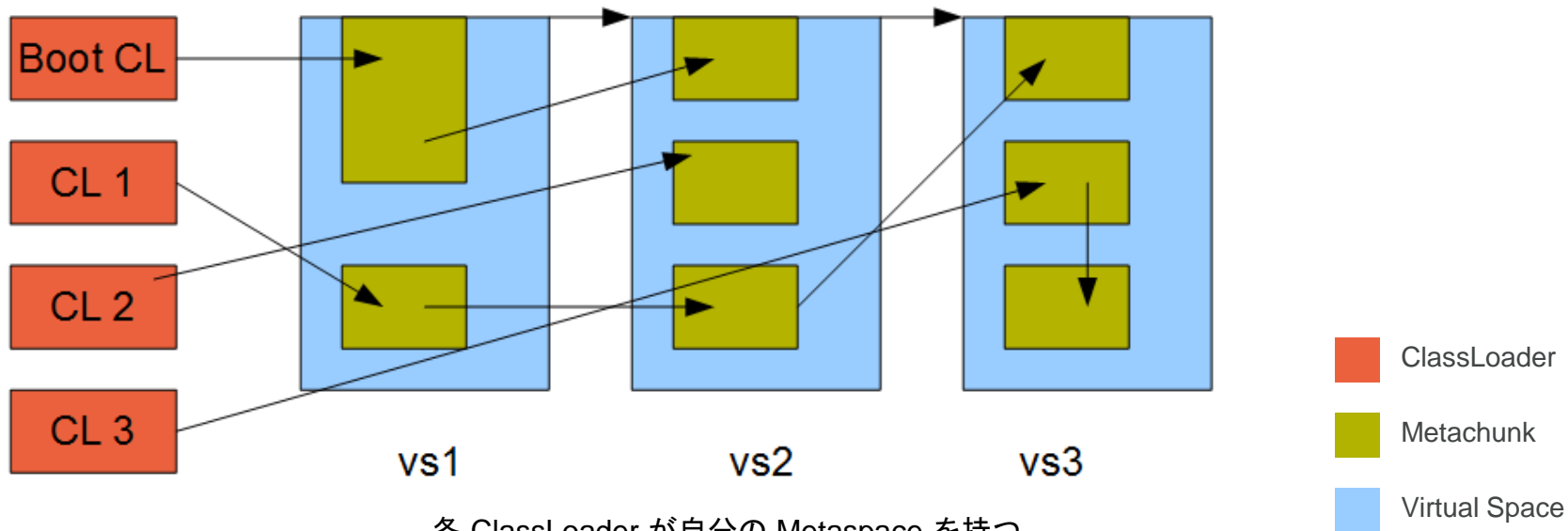
```
class Message {  
    // JVM adds _mark and  
    // _klass pointer  
    String text;  
    void add(String s) { ... }  
    String get() { ... }  
}
```

Java Heap



# Metaspace Allocation

## Metachunks in virtual spaces (vs1, vs2, vs3...)



各 ClassLoader が自分の Metaspace を持つ  
Metaspace が複数の Metachunk を持つ  
Virtualspace が複数の Metachunk を含む

# High Water Mark (高水位標)

- Full GC が発生しないと Metaspace のコレクションが行われない
- Full GC の頻度が低いシステムのメモリ使用量を制御する必要がある
- Metaspace のサイズが HWM を超えると Full GC が実行される
- 調整する必要がある場合
  - Full GC の頻度が高すぎる時
  - メモリの使用量が大きすぎる時

# Compressed Oops (圧縮参照) の概念

- 64-bit のマシンでも、オブジェクトのアドレスを 32-bit に格納する
  - Java ヒープの使用量を節約
  - ヒープのベースアドレスからのオフセットを利用
  - さらにアドレスの LSB を省略

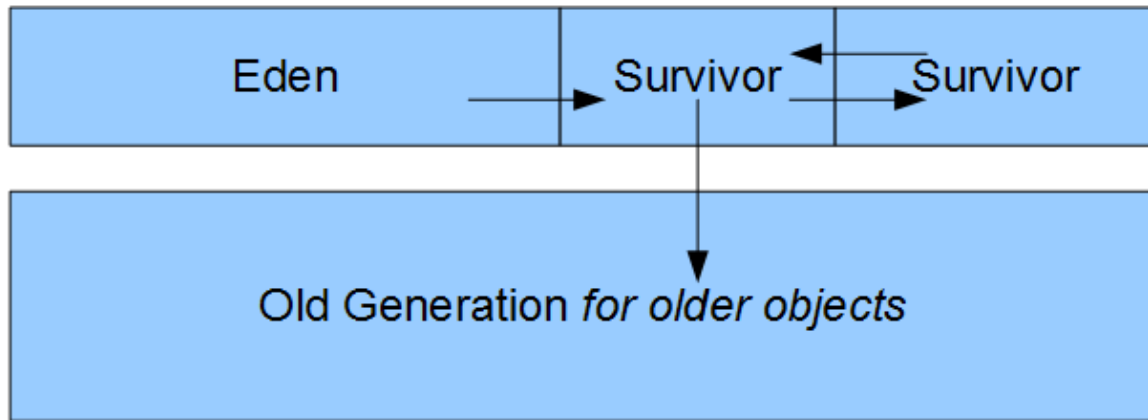


# Klass ポインターも圧縮しましょう

- Compressed Oops と同じように Java Heap を節約
- Compressed Class Pointer Space (CCPS) というメモリエリアを用意
- パフォーマンスに最も影響を及ぼすデータだけを格納する
  - InstanceKlass、ArrayKlass
  - vTable
- それ以外は MetaSpace
  - メソッド、バイトコード
  - ConstantPool など

# Compressed Class Pointer Space

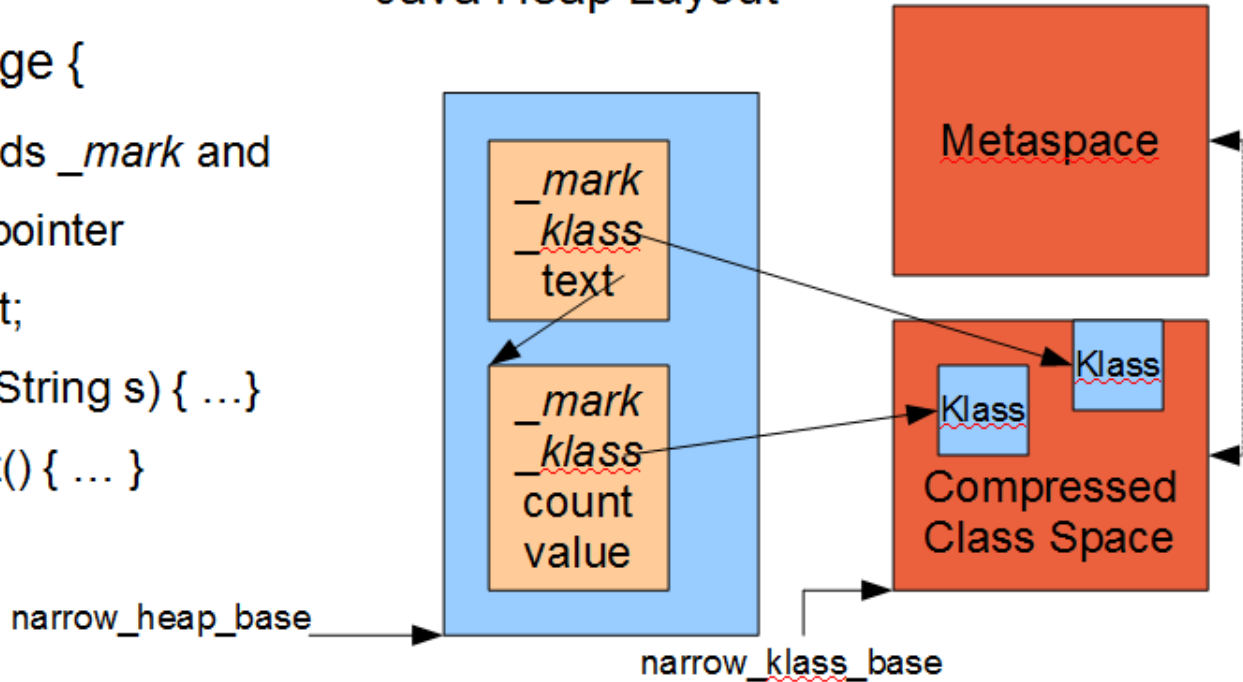
For 64 bit platforms, to compress JVM *klass* pointers in objects, introduce a compressed class pointer space



# Java Object Memory Layout with Compressed Pointers

```
class Message {  
    // JVM adds _mark and  
    // _klass pointer  
    String text;  
    void add(String s) { ... }  
    String get() { ... }  
}
```

Java Heap Layout



# 新しいコマンドラインのオプション

- -XX:MaxMetaspaceSize
- デフォルト = unlimited
- MetaSpace の最大サイズ (バイト数) を設定する

# 新しいコマンドラインオプション

- -XX:MetaspaceSize
- デフォルト = 21MB
- 起動時に Full GC の頻度を減らすために大きくする

# 新しいコマンドラインオプション

- -XX:MinMetaspaceFreeRatio
  - デフォルト = 40
- -XX:MaxMetaspaceFreeRatio
  - デフォルト = 70
- High Water Mark の拡張と縮小を制御する

# 新しいコマンドラインオプション

- `-XX:+UseCompressedClassPointers`
  - 64-bit では、デフォルトで有効
- `-XX:CompressedClassSpaceSize`
  - デフォルト = 1G
  - 変更が出来ないため、デフォルト値が大きい
  - 起動時にはメモリを reserve するだけ
  - 必要に応じて commit していく

# MBean の変更

- 新しい MemoryManagerMXBean: MetaspaceManager
- 新しいメモリプール MXBean: Metaspace と CompressedClassSpace
- 両方とも MemoryType.NON\_HEAP
- Metaspace の Usage は両方 (MS と CCS) の合計
- PermGen メモリプールがなくなりました



# ツールも対応する

- `jmap -permstat` → `jmap -clstats`
- `jstat -gc`
- `jcmd <pid> GC.class_stats`
  - 対象の JVM のコマンドラインで `-XX:+UnlockDiagnosticVMOptions` を追加することが必要

# Tiered Compilation

(階層型コンパイル)

# ちょっと待って、 これは JDK8 の新機能ではないでしょう！

- 古い実装は HotSpot Express で 6u25 までバックポートされました
- JDK8 ではようやくデフォルトで有効！
- JDK8 の新しい実装は従来のバージョンよりかなり充実

# 背景: HotSpot の二つの JIT コンパイラ

- C1 (-client)
  - コンパイル処理が速い
  - 生成されるコードが（比較的）速くない
- C2 (-server)
  - コンパイル処理に時間がかかる
  - 生成されるコードが速い

# つまり

- 起動を速くしたい場合: C1
- 起動後のパフォーマンスが必要な場合: C2

# Tiered Compilation の概要

- Tiered Compilation は両方のコンパイラを平行で利用
- 速い起動
- 起動後のパフォーマンスもいい

# Tiered Compilation の概要

## 従来の流れ

### C1 (client)

- ステップ 1 : インタプリタ  
実行しながら Hot なメソッドを検出する
- ステップ 2 : ネイティブ  
Hot と判断したメソッドを  
JIT コンパイルし、実行する

# Tiered Compilation の概要

## 従来の流れ

C2 (server)

- ステップ 1 : インタプリタ  
実行しながらプロファイリングする
- ステップ 2 : ネイティブ  
Hot と判断したメソッドを  
1 で取得したプロファイリングデータを使って  
JIT コンパイルし、実行する



# Tiered Compilation の概要

## 重要なポイント！


- C2 がプロファイリングデータを必要とするので、データを取得するためにインタプリタのフェーズが長い

# Tiered Compilation の概要

- インタプリタと C2 の間に C1 を入れて C1 でプロファイリングのデータを取得

# コンパイルのレベル

- level 0 - インタプリタ
- level 1 - C1 フル最適化 (プロファイリングなし)
- level 2 - C1 呼び出し (invocation) とループ (back-edge) のプロファイリング
- level 3 - C1 フルプロファイリング (level 2 + MDO)
  - Level 2 より約 30%遅い
- level 4 - C2

- 
- branch
  - call receiver type
  - typecheck

# コンパイルレベルの遷移

## 一番理想的なシナリオ

- level 0 -> level 3 -> level 4
  - 0 : インタプリタで実行され、Hot メソッドとして検出される
  - 3 : フルプロファイリングのC1でコンパイルされる
  - 4 : C2が3のデータを使って、再コンパイルする

# コンパイルレベルの遷移

## コンパイラのキューイング

- それぞれのコンパイラにキューが存在
  - C1 キュー
  - C2 キュー
- コンパイルスレッドを待っているタスクがキューイングされる
- C1 キューの長さによってコンパイルの閾値 (CompileThreshold) が自動的に調整される
- C2 のキューの長さによってメソッドが 0 から 2 にコンパイル

# コンパイルレベルの遷移

- 0 -> 3 -> 4 (一番理想的).
- 0 -> 2 -> 3 -> 4 (C2 のキューが長すぎ).
- 0 -> (3->2) -> 4 (キューで行き先が変更される).
- 0 -> 3 -> 1 or 0 -> 2 -> 1 (trivial、C2 がコンパイル出来ないメソッド).
- 0 -> 4 (C1でコンパイルが出来ない, インタプリタでフルプロファイリング).
- (1,2,3,4) → 0 ( 脱最適化 (deoptimization) )

# コマンドラインのオプション

トラブルシューティングで役に立つ

- -XX:+PrintCompilation

63 1 3 java.lang.String::equals (81 bytes)

63 2 n 0 java.lang.System::arraycopy (native) (static)

64 3 3 java.lang.Math::min (11 bytes)

64 4 3 java.lang.String::charAt (29 bytes)

66 6 3 java.lang.String::indexOf (70 bytes)

時間(ミリ秒) ID コンパイルレベル メソッド名 サイズ

# コマンドラインのオプション

## トラブルシューティングで役に立つ

### ▪ -XX:+PrintTieredEvents

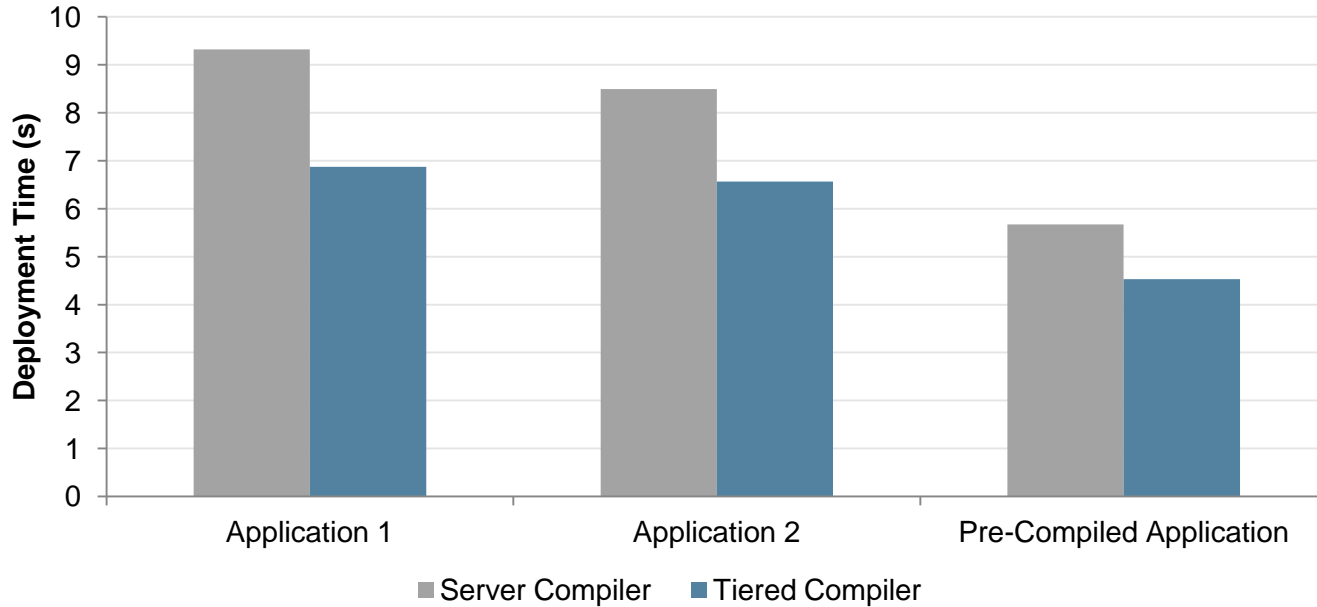
0.169833: [call level=0 [java.lang.Object.<init>()]V] @-1 queues=0,0 rate=n/a k=1.00,1.00 total=128,0 mdo=0(0),0(0) max levels=0,0 compilable=c1,c1-osr,c2,c2-osr status=idle]

- イベント: call, loop, compile, remove-from-queue, update-in-queue, reprofile
- レベル: 0,2,3 (イベントが発生したレベル)
- キューの状態 (C1 キュー, C2 キュー)
- 呼び出しのレート
- 呼び出しのカウンター: 合計, mdo
- コンパイルの可能性
- 状態 (idle、in-queue、など)



# JEE Application Deployment

## Tiered Compilation



# その他の改善

# JMX で診断コマンドを実行

- ローカルで jcmd コマンドだけではなく
- ネットワーク経由で診断コマンドの実行も可能

# フォールスシェアリングの回避

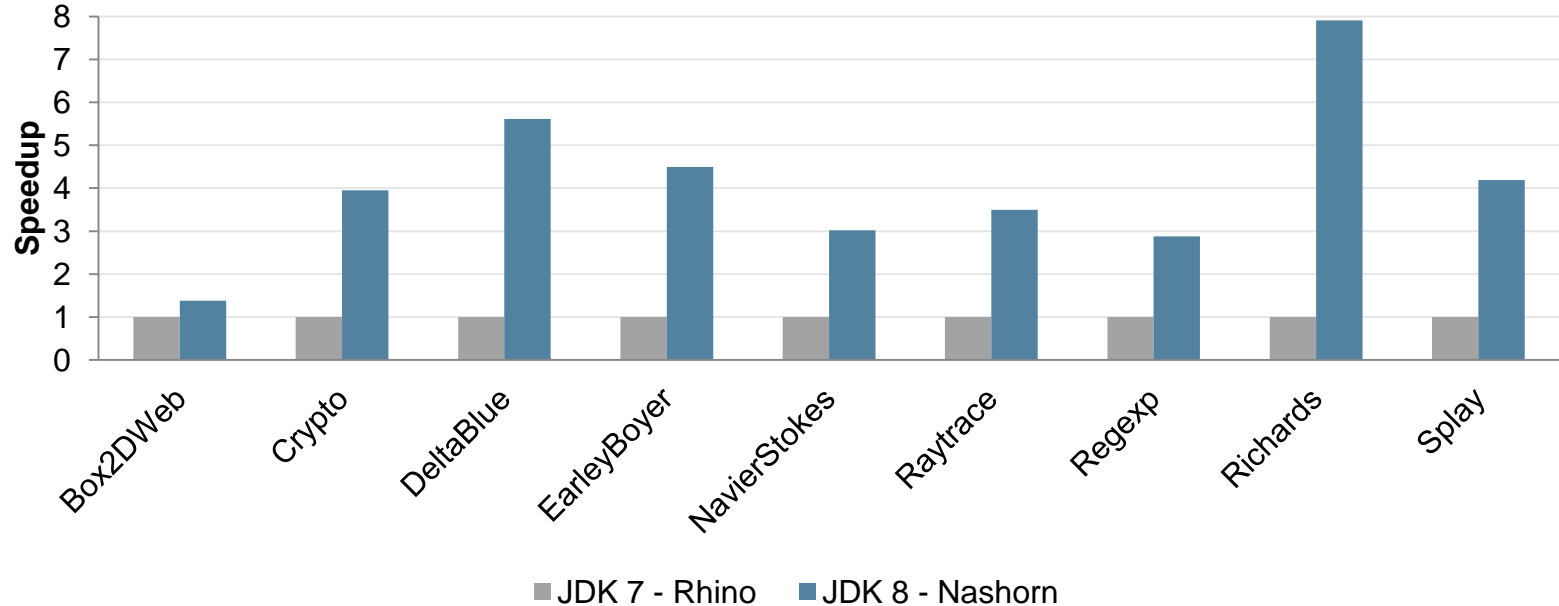
- JEP 142: Reduce Cache Contention on Specified Fields
- キャッシュラインのフォールスシェアリングの回避
- Fork / Join で競合されるフィールドが唯一のキャッシュラインに格納されるように、オブジェクトのレイアウト（パッド）を自動的に行う

# JSR-292 のパフォーマンス改善

- Invoke Dynamic の実装が一新され、かなり速くなった
- Lambda と Nashorn のパフォーマンスに大きい貢献

# JavaScript Engine – Nashorn

## JSR 292 - InvokeDynamic



# JVM 側で直接のサポートの追加

- `java.util.concurrent` のパフォーマンス改善
- AES 暗号化のパフォーマンス改善
  - X86/AMD64 の AES 専用命令を利用
- など
  
- JEP 171: Fence Intrinsic
- JEP 164: Leverage CPU Instructions for AES Cryptography
- +α

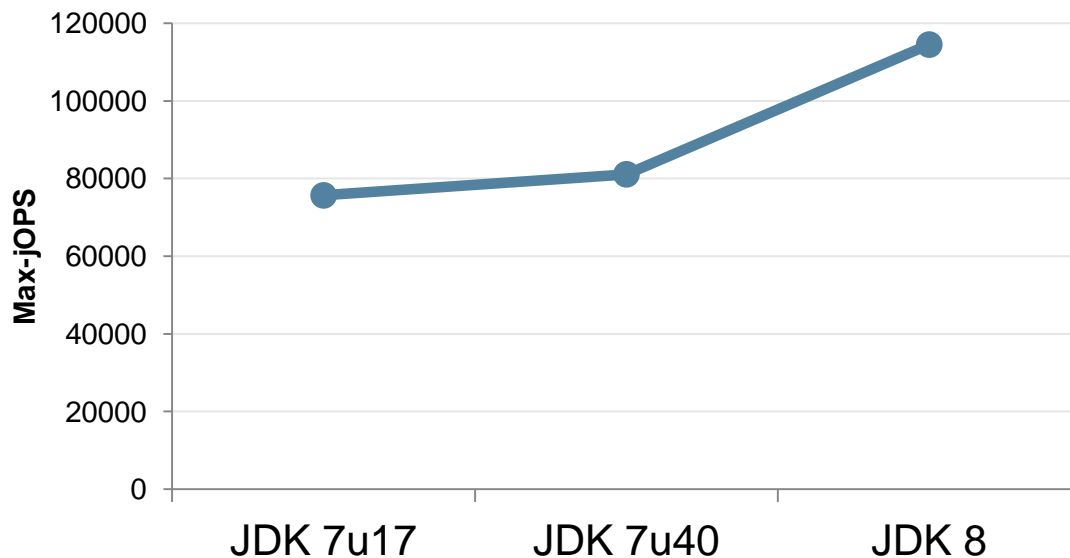
# もう使う意味がない GC の組み合わせが非推奨に

- JEP 173: Retire Some Rarely-Used GC Combinations
- JDK 9 でサポートされなくなる予定
- `-XX:-UseParNewGC -XX:+UseConcMarkSweepGC` (DefNew + CMS)
- `-XX:+UseParNewGC` (ParNew + SerialOld)
- `-Xincgc` (ParNew + iCMS)
- `-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC` (ParNew + iCMS)
- `-XX:+CMSIncrementalMode -XX:+UseConcMarkSweepGC -XX:-UseParNewGC` (DefNew + iCMS)



# 結果として

## SPECjbb2013 Improvement



Oracle JDK 7u17 – Oracle SPARC T5-2 – 75658 SPECjbb2013-MultiJVM Max-jOPS, 23334 SPECjbb2013-MultiJVM Critical-jOPS

Oracle JDK 7u40 – Oracle SPARC T5-2 – 81084 SPECjbb2013-MultiJVM Max-jOPS, 39129 SPECjbb2013-MultiJVM Critical-jOPS

Oracle JDK 8 – Oracle SPARC T5-2 – 114492 SPECjbb2013-MultiJVM Max-jOPS, 43963 SPECjbb2013-MultiJVM Critical-jOPS

Source: [www.spec.org](http://www.spec.org) as of March 10<sup>th</sup> 2014

SPEC and the benchmark name SPECjbb are registered trademarks of Standard Performance Evaluation Corporation (SPEC)

# Thank You!

