ORACLE

Java Benchmarking

как два таймстампа прочитать!

Алексей Шипилёв
aleksey.shipilev@oracle.com, @shipilev

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.

Введение

Введение: В качестве разогрева

«Сколько стоит создание одного String?»

```
long startTime = System.nanoTime();
for (int i = 0; i < 1000; i++) {
  String s = new String("");
}
long stopTime = System.nanoTime();
System.out.println("Time:" + (stopTime - startTime));
```


Теория

Теория: Зачем люди делают бенчмарки?

1. **Ради холивора**: Node.js – Но Java... – Node.js!
2. **Ради маркетинга**: проверить, что мы вкладываемся в установленные критерии
3. **Ради инжиниринга**: изолировать и зафиксировать перформансный феномен, чтобы была референсная точка для улучшения
4. **Ради науки**: понять, какой моделью описывается система, и на основании этой модели предсказать будущее поведение

Теория: Ради холивора

Прекрасный пример – Computer Language Benchmarks Game:[1]

- Многие реализации вообще не сравнимы: e.g. AOT vs. JIT
- Измеряют непонятно что: e.g. pidigits измеряет скорость интерфейса до GMP
- Куча дисклеймеров про то, что в реальной жизни всё может быть по-другому: и тогда этот проект нужен только ради лулзов
- Любят его потому, что CLBG даёт **числа**, которыми можно размахивать в холиворах

[1] http://benchmarksgame.alioth.debian.org/

Теория: Ради маркетинга

Прекрасный пример – SPEC benchmarks:

- Референсные наборы бенчмарок, одинаково хороших/плохих для большинства вендоров
- Позволяют иметь референсные точки, против которых можно выставлять критерии успешности продукта, писать в рекламе и т.п.
- Ну и что, что они не всегда репрезентативны – главное, что они «золотые»

Теория: Ради инжиниринга

«If you can't measure it, you can't optimize it»

- Нужны лабораторные условия, в которых зафиксировано конкретное состояние системы, чтобы можно было проверять внесённые изменения
- Эти бенчмарки обычно фокусируются на конкретных местах продукта, имеют бОльшую разрешающую способность, чем маркетинговые бенчи
- Размеры и охват этих бенчмарков зависит от укуренности инженеров

Теория: Ради науки

«Science Town PD: To Explain and Predict»

- Извлечь из результатов тестов правдоподобную модель производительности
- Из модели получить предсказания о будущем поведении, проверить эти предсказания, спокойно вздохнуть и деплойнуть в прод
- Самая трудоёмкая, и самая надёжная цель бенчмаркинга

Теория: Что интересно нам?

1. **Ради холивора**: мой язык лучше твоего языка
2. **Ради маркетинга**: проверить, что мы вкладываемся в конкретные критерии
3. **Ради инжиниринга**: изолировать и зафиксировать перформансный феномен, чтобы была референсная точка для улучшения
4. **Ради науки**: понять, какой моделью описывается система, и на основании этой модели предсказать будущее поведение

Теория: Бенчмарки – это эксперименты

Хорошие эксперименты отвечают на несколько групп вопросов:

1. **Что измеряем:** какие метрики используем? Что эти метрики нам говорят?
2. **Как измеряем:** какими способами мы измеряем метрики? какая точность у этих способов? какие подводные камни?
3. **Как проверяем:** как поверяем наши инструменты? как узнаём, что результатам можно доверять?

Что измеряем: Метрики

Две главные группы метрик:

Bandwidth (λ)

- Сравнительно легко измерить
- Легко вводится в steady state
- Скрывает большие задержки
- Упускает active-idle

Latency (τ)

- Трудно измерить корректно
- Средняя latency как правило бессмыслена
- Нужно считать квантили
- Семплы меньше 1мкс получить не практично

Что измеряем: путаем bandwidth и latency

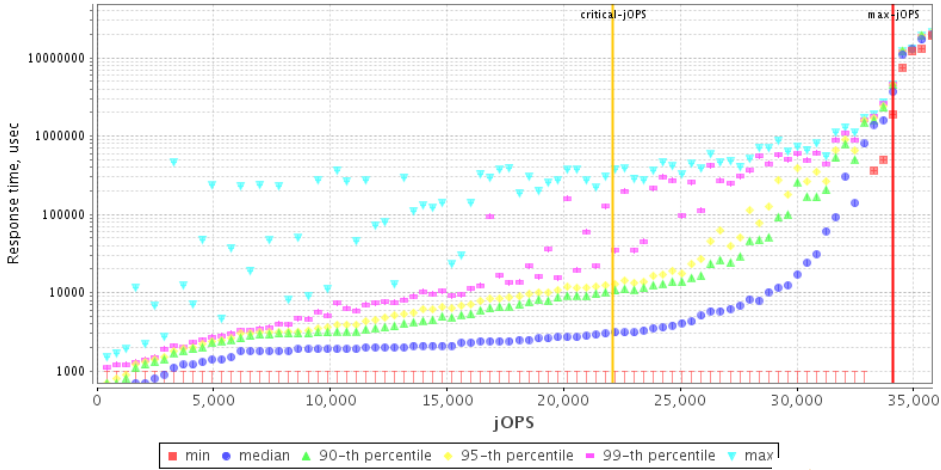
- Отличия в нагрузке:
  - λ обычно измеряют под пиковой нагрузкой
  - τ как правило измеряют под регулируемой нагрузкой
  - Нагрузка – это новая степень свободы!

- Отличия в подходах к измерению:
  - Средняя задержка – бессмысленная метрика, ибо

$$\tau_{avg} = \frac{1}{\lambda}$$

  - Нужно мерить отдельные события и считать квантили!
  - События короче 1 мкс? Мухаха.

Что измеряем: λ и τ – братья навек



Как измеряем: типы бенчмарок

Два главных подхода:

Time-based

- Исполняем постоянное время
- Делаем операции одну за одной
- В измерение умещаем порядочное количество операций

Fixed work

- Исполняем постоянное количество операций
- Чаще всего одну операцию
- Здорово экономит время!

Как измеряем: Transients

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

Как измеряем: Transients

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

- У нас полно переходных процессов.

Как измеряем: Transients

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

- У нас полно переходных процессов.
- Компиляция кода – не единственный переходный процесс!

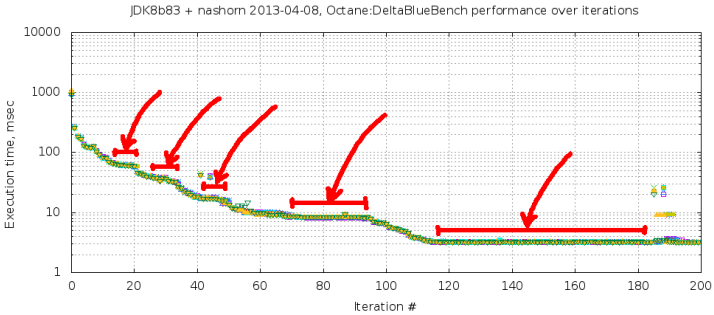
Как измеряем: Transients

Запомните

Мы имеем дело с динамическими системами.
«Прогрев» – это выжидание характерного времени переходного процесса.

- У нас полно переходных процессов.
- Компиляция кода – не единственный переходный процесс!
- «Мудрость»: «Следите за PrintCompilation». WTF?

Как измеряем: Steady state



- Система пришла в устойчивое состояние (steady state), когда все её переходные процессы устканились
- Любое изменение выбивает её из steady state

Как проверяем: Инженерный подход

Главный инженерный вопрос

Почему мой бенчмарк не может работать быстрее?

Ответ определяет качество эксперимента:

1. В какие ограничения упёрлись?
2. Работает та часть кода, которую мы «исследуем»?
3. Что сделать, чтобы исправить бенчмарк?

Как проверяем: Научный подход

Главный научный вопрос

Как бенчмарк реагирует на изменение внешних условий?

Отвечаем, насколько актуальная модель разнится с ментальной:

1. Проверка на дурака: имеют ли смысл эти результаты?
2. Негативный контроль: меняется ли результат от варирования переменной X_i, хотя не должен?
3. Позитивный контроль: не меняется ли результат от варирования переменной Y_i, хотя должен?

Практика

Практика: JMH

У нас тоже есть очень хороший харнесс:
`http://openjdk.java.net/projects/code-tools/jmh/`

- JMH is Serious Business:
  - Он учитывает тонну VM-ных эффектов
  - Мы его периодически допиливаем, когда меняется VM
  - Мы его периодически фиксим, когда растёт наша экспертиза
  - Всякий внешний бенч валидируется переписыванием под JMH

Практика: Старая песня

Весь этот жир в предыдущем докладе:

- System: Power management
- System: Time Sharing
- VM: Dead code elimination
- VM: Constant folding
- VM: Loop unrolling
- VM: Profiles and forking
- VM: Run-to-run variance
- VM: Inlining
- CPU: Caches
- CPU: Branches

Практика: JMH Samples

```
JMHSample_01_HelloWorld
JMHSample_02_BenchmarkModes
JMHSample_03_States
JMHSample_04_DefaultState
JMHSample_05_StateFixtures
JMHSample_06_FixtureLevel
JMHSample_07_FixtureLevelInvocation
JMHSample_08_DeadCode
JMHSample_09_Blackholes
JMHSample_10_ConstantFold
JMHSample_11_Loops
JMHSample_12_Forking
JMHSample_13_RunToRun
JMHSample_15_Asymmetric
JMHSample_16_CompilerControl
JMHSample_17_SyncIterations
JMHSample_18_Control
JMHSample_20_Annotations
JMHSample_21_ConsumeCPU
JMHSample_22_FalseSharing
JMHSample_23_AuxCounters
JMHSample_24_Inheritance
JMHSample_25_API_GA
JMHSample_26_BatchSize
JMHSample_27_Params
```


Про боттлнеки: Внеклассное чтение

В качестве примера про анализ боттлнеков,
тут должен был быть забавный пример про то,
как оптимизация `@tailrec` в `scalac` с последующей
оптимизацией в HotSpot приводит к пессимизации кода,
но поля этого слайда слишком узки...

`http://stackoverflow.com/questions/22581163/hidden-performance-cost-in-scala/22604185`

Про модели: модельная задачка

«Сколько стоит `volatile` write?»

- Казалось бы, простой вопрос: в нём всего четыре слова, и только два из них английские...
- Хочется взять и померить!

Про модели: Пишем...

```
public class VolatileWrite {
  int v; volatile int vv;

  @GenerateMicroBenchmark
  int baseline1()    { return 42; }

  @GenerateMicroBenchmark
  int incrPlain()    { return v++; }

  @GenerateMicroBenchmark
  int incrVolatile() { return vv++; }
}
```


Про модели: Мерим...

```
public class VolatileWrite {
  int v; volatile int vv;

  @GenerateMicroBenchmark
  int baseline1()    { return 42; }   //  2.0 +- 0.1 ns

  @GenerateMicroBenchmark
  int incrPlain()    { return v++; }  //  3.5 +- 0.1 ns

  @GenerateMicroBenchmark
  int incrVolatile() { return vv++; } // 15.1 +- 0.1 ns
}
```


Про модели: Фатальный недостаток

```
volatile int vv;

@GenerateMicroBenchmark
int incrVolatile() { return vv++; }
```

- Измеряем в условиях, когда бенчмарк занят только `volatile`-ами
- Если эта операция тяжёлая, то мы загоняем систему в её «пограничное» состояние, и оно почти наверняка не воспроизводится в проде
- Нам нужно знать «как стоимость `volatile` зависит от внешних условий»!

Про модели: Backoffs

```
@Param int tokens;

volatile int vv;

@GenerateMicroBenchmark
int incrVolatile() {
  BlackHole.consumeCPU(tokens); // burn time
  return vv++;
}
```

- «Сожжем» пару циклов перед операцией
- Варьируем `tokens` – варьируем микс операций

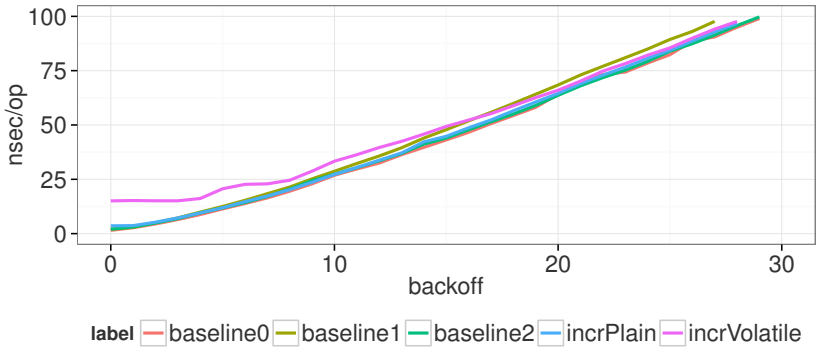
Про модели: Backoffs

```
@GenerateMicroBenchmark
void baseline0() { BH.consumeCPU(tokens); }

@GenerateMicroBenchmark
int  baseline1() { BH.consumeCPU(tokens); return 42; }

@GenerateMicroBenchmark
int  baseline2() { BH.consumeCPU(tokens); return v; }
```

- Чтоб два раза не вставать, заодно проверим разные baseline-ы: наперёд не ясно, какой из них надо использовать, чтобы контрастировать сам инкремент

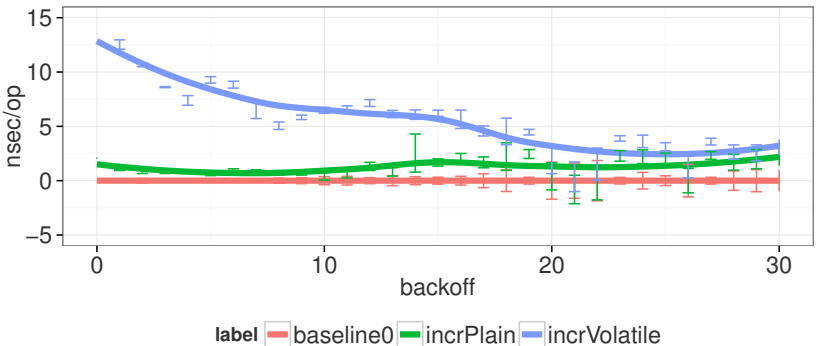
Про модели: Мерим...

Если вы рисуете такие графики, то вам уготовано место в аду:



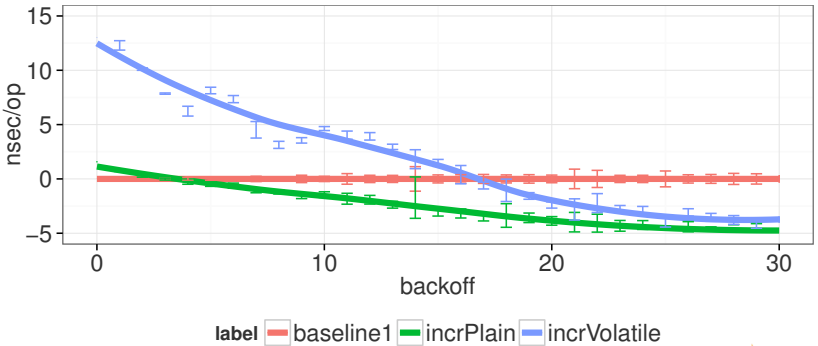
Про модели: вычитаем baseline0

- Абсолютная стоимость `volatile` сильно амортизируется!
- Можно ли вообще так вычитать baseline из теста?

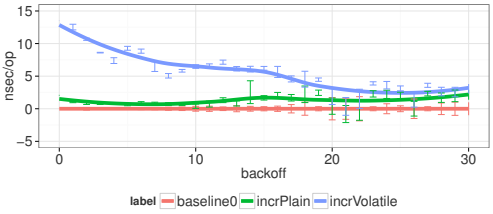


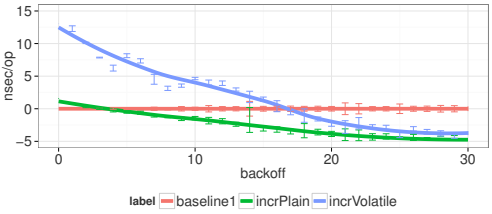
Про модели: вычитаем baseline1

- УПС! Добавили кода в тест, а он стал шустрее?!
- Ничего удивительного: performance is not composable



Про модели: WTF is different?

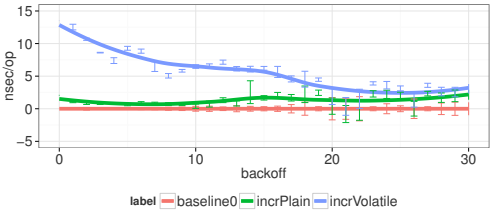


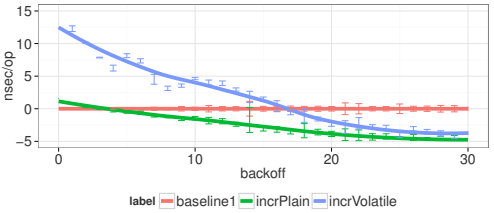



```
@GenerateMicroBenchmark
void baseline0() {
  BH.consumeCPU(tokens);
}
.
```

```
@GenerateMicroBenchmark
int baseline1() {
  BH.consumeCPU(tokens);
  return 42;
}
```


Про модели: WTF is different?

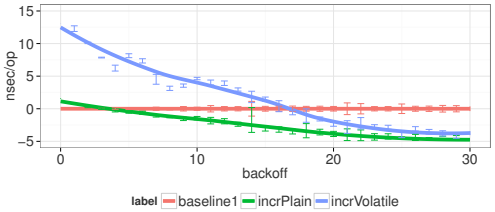





```
@GenerateMicroBenchmark
void baseline0() {
  BH.consumeCPU(tokens);
}
.
```

```
@GenerateMicroBenchmark
int baseline1() {
  BH.consumeCPU(tokens);
  return 42;
}
```

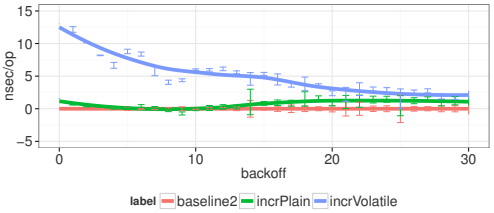

Про модели: WTF is different?

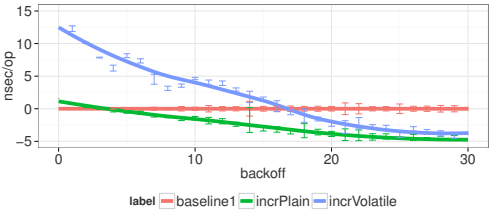



```
@GenerateMicroBenchmark
int baseline2() {
  BH.consumeCPU(tokens);
  return v;
}
```

```
@GenerateMicroBenchmark
int baseline1() {
  BH.consumeCPU(tokens);
  return 42;
}
```


Про модели: WTF is different?



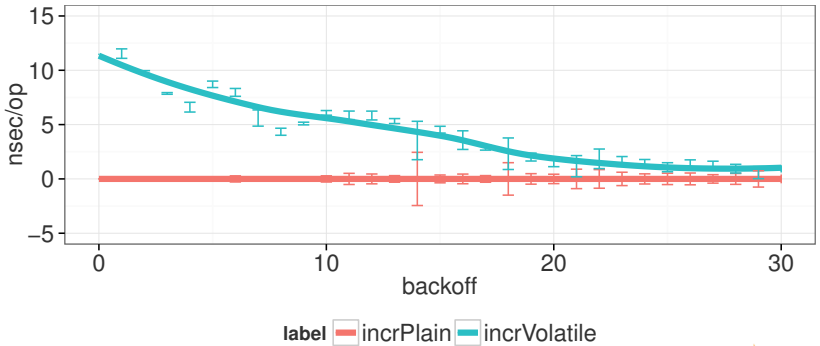



```
@GenerateMicroBenchmark
int baseline2() {
  BH.consumeCPU(tokens);
  return v;
}
```

```
@GenerateMicroBenchmark
int baseline1() {
  BH.consumeCPU(tokens);
  return 42;
}
```

Про модели: сухой остаток

- Разные baseline дают разные результаты...
- Ну и ладно, будем просто сравнивать plain и volatile:



Про модели: подытожим

Вот зачем нам нужны модели!

- Открывают поведение системы за пределами случайно выбранных «удачных» точек
- Позволяют предсказать поведение в разных условиях
- Помогают поймать проблемы в эксперименте (контроль)
- Комбинаторные эксперименты помогают по разному смешать операции и построить из них предположения об их самостоятельной производительности

Про модели: что-то ты, брат, заливаешь...

«Комбинаторные эксперименты помогают по разному смешать операции и построить из них предположения об их самостоятельной производительности»

Я сейчас возьму в руки секундомер/`nanoTime`
и всё померю по частям!

Про таймеры: ща измерим

А почему бы и нет?

```
// call continuously
public long measure() {
    long startTime = System.nanoTime();
    work();
    return System.nanoTime() - startTime;
}
```

Не кроется ли в этом какая-нибудь грабля?

Про таймеры: померим latency

Latency = время на вызов

```
@GenerateMicroBenchmark
public long latency_nanotime() {
    return System.nanoTime();
}
```


Про таймеры: померим granularity

Granularity = разрешающая способность

```
private long lastValue;

@GenerateMicroBenchmark
public long granularity_nanotime() {
    long cur;
    do {
        cur = System.nanoTime();
    } while (cur == lastValue);
    lastValue = cur;
    return cur;
}
```


Про таймеры: типичная картинка [Linux]

```
Java(TM) SE Runtime Environment, 1.7.0_45-b18
Java HotSpot(TM) 64-Bit Server VM, 24.45-b08
Linux, 3.13.8-1-ARCH, amd64

Running with 1 threads and [-client]:
   granularity_nanotime:      26.300 +- 0.205 ns
       latency_nanotime:      25.542 +- 0.024 ns

Running with 1 threads and [-server]:
   granularity_nanotime:      26.432 +- 0.191 ns
       latency_nanotime:      26.276 +- 0.538 ns
```

Про таймеры: типичная картинка [Solaris]

```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
SunOS, 5.11, amd64

Running with 1 threads and [-client]:
   granularity_nanotime:      29.322 +- 1.293 ns
       latency_nanotime:      29.910 +- 1.626 ns

Running with 1 threads and [-server]:
   granularity_nanotime:      28.990 +- 0.019 ns
       latency_nanotime:      30.862 +- 6.622 ns
```

Про таймеры: типичная картинка [Windows]

```
Java(TM) SE Runtime Environment, 1.7.0_51-b13
Java HotSpot(TM) 64-Bit Server VM, 24.51-b03
Windows 7, 6.1, amd64

Running with 1 threads and [-client]:
   granularity_nanotime:      371,419 +- 1,541 ns
       latency_nanotime:       14,415 +- 0,389 ns

Running with 1 threads and [-server]:
   granularity_nanotime:      371,237 +- 1,239 ns
       latency_nanotime:       14,326 +- 0,308 ns
```

Про таймеры: эпичная картинка [Windows]

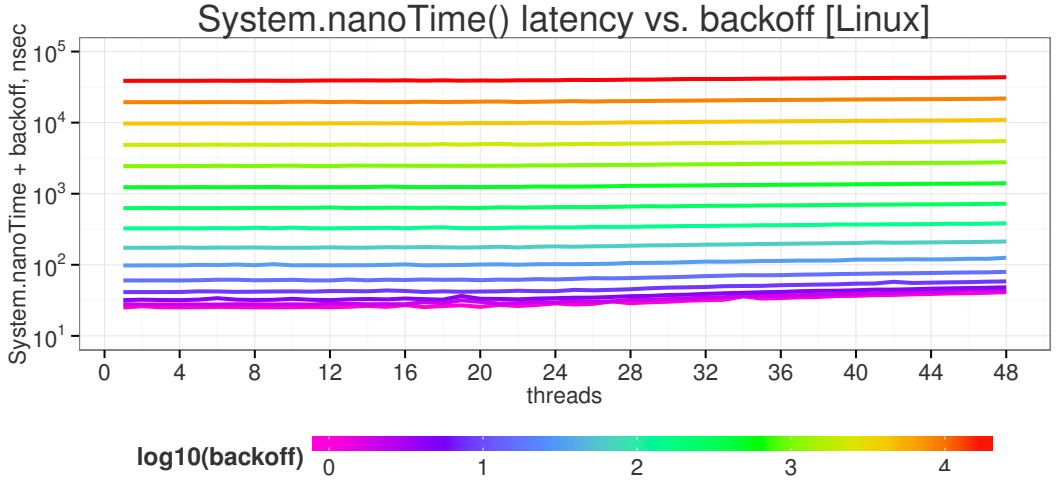
```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
Windows Server 2008, 6.0, amd64

Running with 32 threads and [-client]:
   granularity_nanotime:  15137.504 +-   97.132 ns
       latency_nanotime:  15190.080 +- 1760.500 ns

Running with 32 threads and [-server]:
   granularity_nanotime:  15118.159 +-  121.671 ns
       latency_nanotime:  15176.690 +- 1504.406 ns
```


Про таймеры: модельный эксперимент

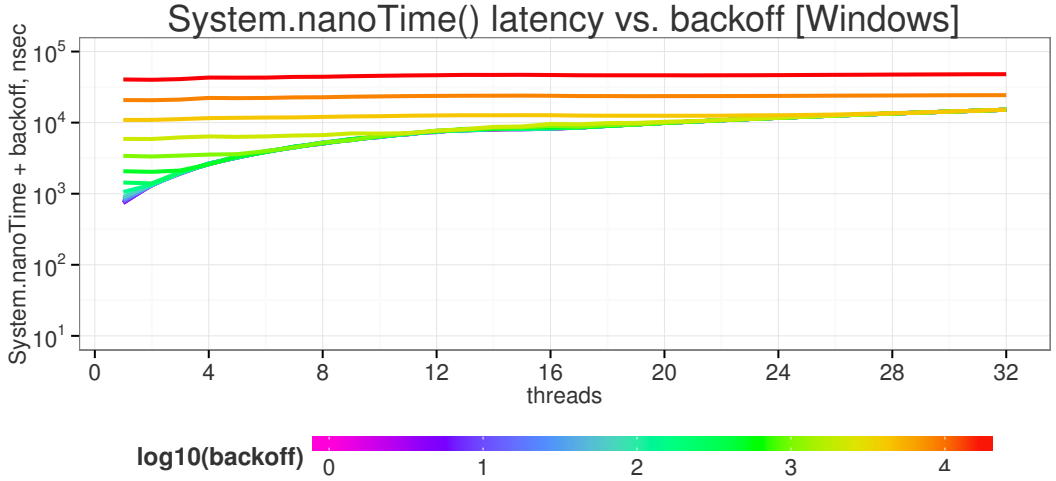
- Если `System.nanoTime()` дорогой и потенциально не скалирующийся, значит, он будет влиять на измерение
- Давайте найдём, насколько он влияет:

```
@Param
int backoff;

@GenerateMicroBenchmark
public long nanotime() {
    BlackHole.consumeCPU(backoff);
    return System.nanoTime();
}
```

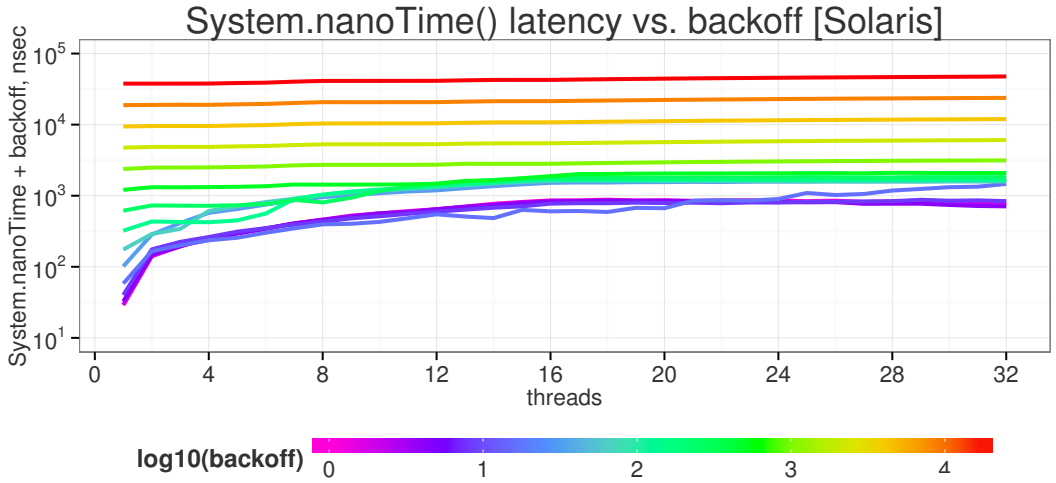

Про таймеры: тут пока всё в порядке



Про таймеры: воткнулись в сам nanoTime!



Про таймеры: платим за монотонность



Про таймеры: типичная картинка [Mac OS X]

```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
Mac OS X, 10.9.2, x86_64

Running with 1 threads and [-server]:
   granularity_nanotime:  1009.623 +-  2.140 ns
       latency_nanotime:   44.145 +-  1.449 ns

Running with 4 threads and [-server]:
   granularity_nanotime:  1044.703 +- 32.103 ns
       latency_nanotime:   56.111 +-  3.397 ns
```


Про таймеры: Подытожим

`System.nanoTime` – это новый `String.intern`!

- Юзер с `nanoTime` = обезьяна с гранатой
- `nanoTime` можно и нужно пользовать в отдельных случаях, но будьте готовы ко всем минусам этого предприятия
- Чаще всего прямое измерение нам не доступно, и приходится строить модели по косвенным уликам

Про таймеры: продолжаешь заливать?

Мои куски достаточно большие, чтобы не наступить на гранулярность и стоимость `nanoTime()`!

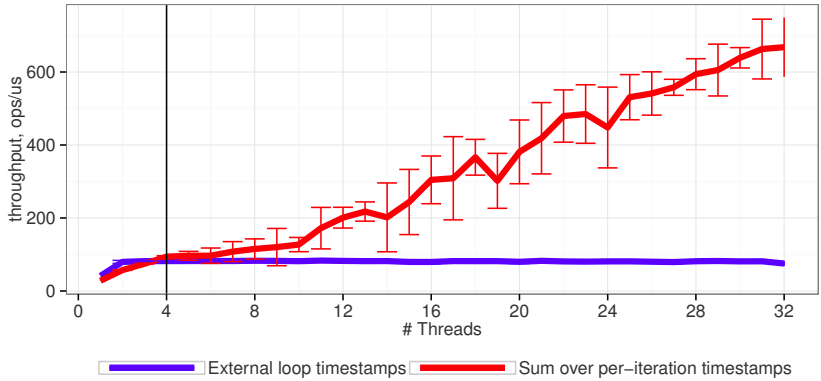
Про omission: жирный такой бенчмарк

```
public long measure() {
    long ops = 0;
    long startTime = System.nanoTime();
    while(!isDone) {
        setup(); // want to skip this
        work();
        ops++;
    }
    return ops/(System.nanoTime() - startTime);
}
```


Про omission: ща измерим отдельный кусок

```
public long measure() {
   long ops = 0;
   long realTime = 0;
   while(!isDone) {
      setup(); // skip this
      long time = System.nanoTime();
        work();
      realTime += (System.nanoTime() - time);
      ops++;
   }
   return ops/realTime;
}
```


Про omission: проверяем с пустым setup()...



Измеряем количество вызовов в секунду, да?

...а оно растёт за количество CPU?!

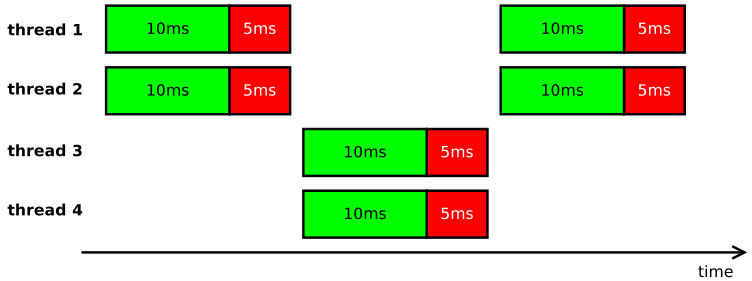
Про omission: подсказка

```
public long measure() {
   long ops = 0;
   long realTime = 0;
   while(!isDone) {
      setup(); // skip this
      long time = System.nanoTime();
        work();
      realTime += (System.nanoTime() - time);
      ops++;
      ...WHOOPS, WE DE-SCHEDULE HERE...
   }
   return (ops/realTime);
}
```


Про omission: базовый пример

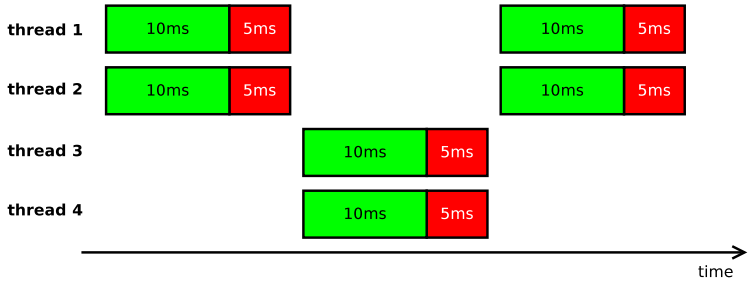
- Каждый поток измеряет только время одной операции, и оно в среднем 10 мс/оп $\Rightarrow$ каждый поток считает, что его индивидуальная $\lambda_i = 100$ оп/сек
- Два потока, и поэтому $\sum_{i=1}^{N} \lambda_i = 200$ оп/сек

Про omission: накидываем ещё потоков



- Каждый поток по-прежнему считает, что $\lambda_i = 100$ оп/сек!
- Четыре потока, и поэтому $\sum_{i=1}^{N} \lambda_i = 400$ оп/сек

Про omission: накидываем ещё потоков



- Каждый поток по-прежнему считает, что $\lambda_i = 100$ оп/сек!
- Четыре потока, и поэтому $\sum_{i=1}^{N} \lambda_i = 400$ оп/сек

Про omission: подытожим

Секундомеры могут мерить совсем не то,
что вы думаете или хотите думать

- ...особенно, если вы измеряете отдельные части
- ...особенно, если вы измеряете перегруженные системы
- ...особенно, если жонглируете λ и τ

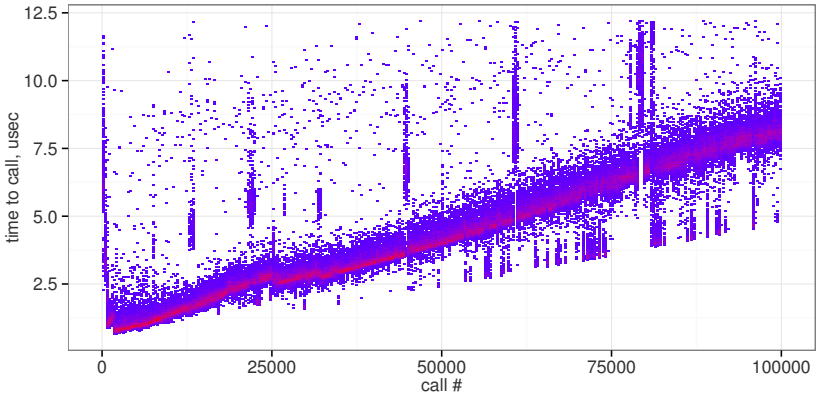
Про steady state: наш любимый fibonacci

Как вам такой бенчмарк?

```
public class FibonacciGen {
  BigInteger n1 = ONE; BigInteger n2 = ZERO;

  @GenerateMicroBenchmark
  public BigInteger next() {
    BigInteger cur = n1.add(n2);
    n2 = n1; n1 = cur;
    return cur;
  }
}
```

Про steady state: померим каждый вызов

Упс, у этого бенчмарка действительно нет steady state...



Про steady state: засада

Нет steady state – нельзя использовать time-based бенчмарки!

Чем дольше мерим, тем «медленнее» результат:

итерация, с	throughput, us/op
1	5.013 ± 0.006
2	7.087 ± 0.009
4	10.021 ± 0.017
8	14.159 ± 0.010

Про steady state: pick your poison

Мерим вызовы в единицу времени:

- мерим непонятно в каком состоянии
- как сравнивать две разные реализации?
  (и ещё повезёт, если модель линейная)

Мерим время на вызов:

- соберём грабли с latency/granularity таймеров
- соберём грабли с omission
- соберём transient-ы (прощайте, многопотоковые бенчи)

Про steady state: промежуточный вариант

Измеряем большими кусками (batching)!

```
@Setup(Level.Iteration)
public void setup() {
  n1 = BigInteger.ZERO; n2 = BigInteger.ONE;
}

@GenerateMicroBenchmark
@Measurement(batchSize = 5000)
public BigInteger next() {
  BigInteger cur = n1.add(n2);
  n2 = n1; n1 = cur;
  return cur;
}
```


Про steady state: засада

Batching хорош, ибо амортизирует стоимость таймеров, но:

- всё равно собирает transient'ы
- всё равно имеет проблемы с omission
- скрывает за собой локальные нелинейности

Про steady state: засада

Batching хорош, ибо амортизирует стоимость таймеров, но:

- всё равно собирает transient'ы
- всё равно имеет проблемы с omission
- скрывает за собой локальные нелинейности

Бенчмарки без steady state – страшный геморрой!

Выводы

Выводы: Benchmarking is Serious Business

Огромное поле для ошибок.

- Даже самые очевидные вещи могут быть ложными
- Даже лучший вариант может быть недостаточно хорошим
- Даже самые точные инструменты могут быть слишком грубы

