ORACLE

Java Benchmarking

как два таймстампа прочитать!

Aleksey Shipilëv
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.

Intro

Intro: Для разогрева

«Сколько стоит один String?»

```
long time1 = System.nanoTime();
for (int i = 0; i < 1000; i++) {
  String s = new String("");
}
long time2 = System.nanoTime();
System.out.println("Time:" + (time2 - time1));
```


Теория

Теория: Зачем люди делают бенчмарки?

1. **Ради холивора**: Node.js – Ho 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: 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?

Теория: 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

Научный подход

Научный подход: TL;DR;

В этой части мы рассмотрим некоторые методологические грабли в разработке бенчмарков: проблемы с моделями, инфраструктурой и проч.

Полная история и конспект здесь:
http://shipilev.net/blog/2014/nanotrusting-nanotime/

Модели: модельная проблема

«Jessie, it's time to cook some benchmarks...»

«Сколько стоит volatile-запись?»

Кажется, что это очень простой вопрос...
Сейчас возьмём и измерим!

Java

Модели: Остоооо...

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

  @Benchmark
  int baseline1()    { return 42; }

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

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


Модели: ...рооожненько!

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

  @Benchmark
  int baseline1()     { return 42; }   // 2.0 ns

  @Benchmark
  int incrPlain()     { return v++; }  // 3.5 ns

  @Benchmark
  int incrVolatile() { return vv++; } // 15.1 ns
}
```


Модели: Фатальный Недостаток

```
volatile int vv;

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

- Измеряем производительность в очень неприятном случае, когда система утонула в volatile-записях: так почти никогда не бывает в продакшене
- На самом деле, мы хотим узнать,
  «Сколько стоит volatile-запись в реалистичных условиях?»

Модели: Backoffs

```
@Param int tokens;

volatile int vv;

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

- «Сожжём» немножко циклов, перед тем как делать тяжёлую операцию
- Меняем `tokens` ⇒ меняем «скважность»

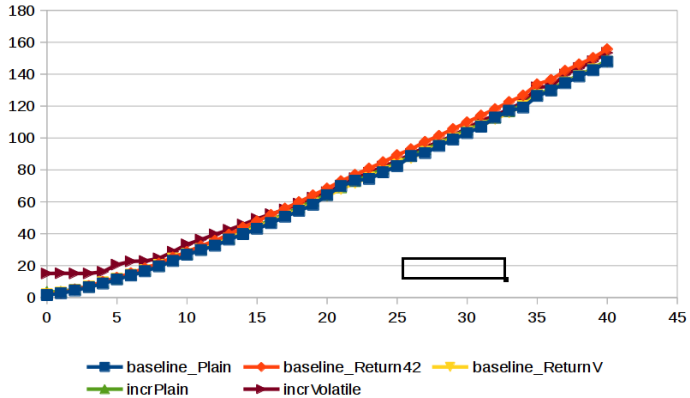
Модели: backoffs

- Ну и заодно попробуем ещё и разные baseline-ы:

```
@Benchmark
void baseline_Plain()
   { BH.consumeCPU(tokens); }

@Benchmark
int baseline_Return42()
   { BH.consumeCPU(tokens); return 42; }

@Benchmark
int baseline_ReturnPlain()
   { BH.consumeCPU(tokens); return v; }
```

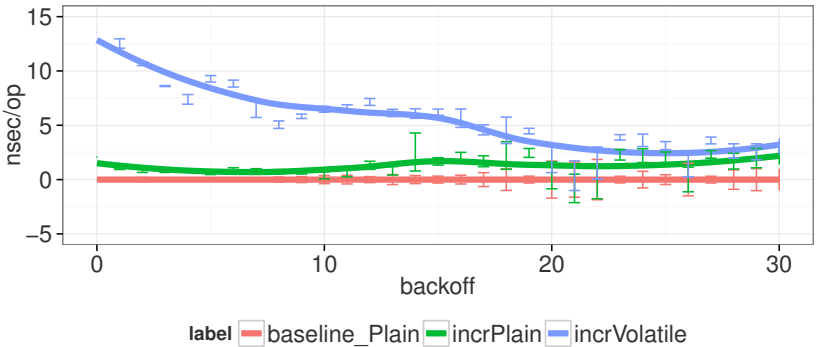

Модели: мерим...

«Bender B. Rodriguez жалеет о рисованных в экселе графиках»



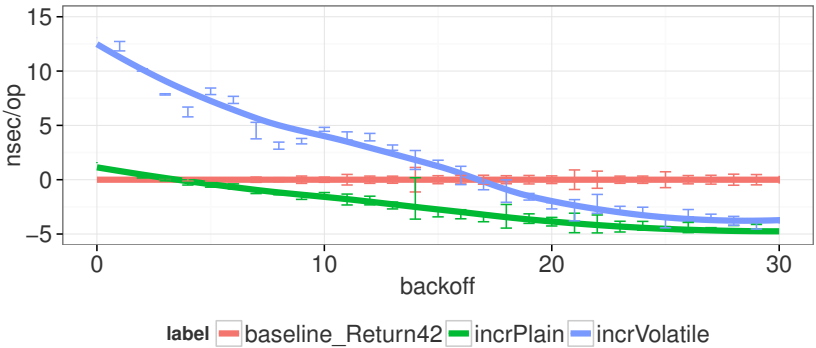
Модели: вычитаем `baselinePlain`

- Абсолютная стоимость `volatile`-записи сильно амортизирована!
- Можно ли так вычитать?

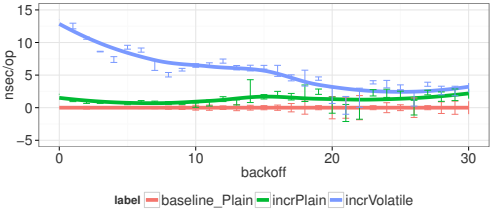


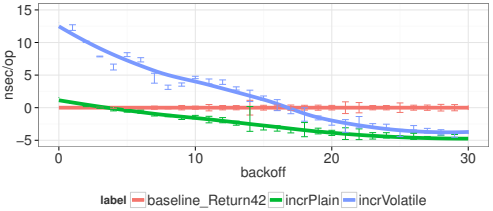
Модели: вычитаем `baseline_Return42`

- Добавили кода, а стало *быстрее*?
- Ничего удивительного: *производительность не композируется*



Модели: WTF is different?

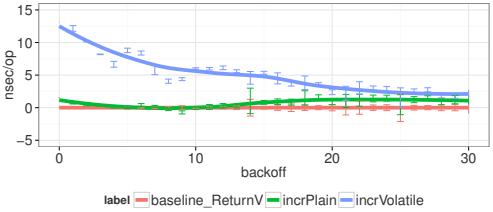


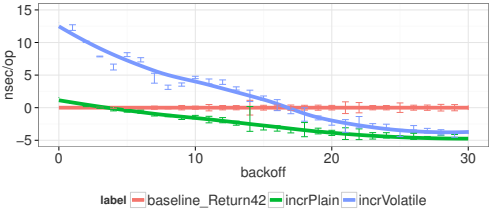



```
@Benchmark
void base_Plain() {
  BH.consumeCPU(tkns);
}
.
```

```
@Benchmark
int base_Ret42() {
  BH.consumeCPU(tkns);
  return 42;
}
```


Модели: WTF is different?



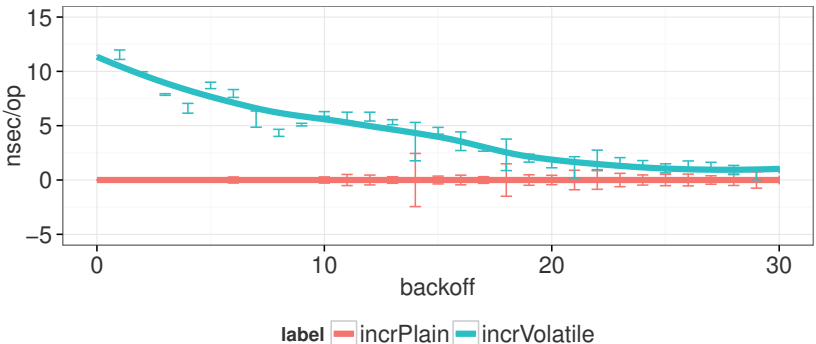



```
@Benchmark
int base_RetV() {
  BH.consumeCPU(tkns);
  return v;
}
```

```
@Benchmark
int base_Ret42() {
  BH.consumeCPU(tkns);
  return 42;
}
```


Модели: подытожим

- Разные baseline дают разные результаты: они **тесты** сами по себе!
- Поэтому можно просто сравнить `plain` и `volatile`:



Java

Модели: выводы

Вот для чего 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

Java

Модели: шутишь?

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

Я возьму в руки секундомер и измерю всё по частям!

Java

Таймеры: проверяем инфраструктуру

Почему бы и нет?

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


Таймеры: измеряем латентность

Латентность = время на вызов System.nanoTime

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


Таймеры: измеряем гранулярность

Гранулярность = разрешающая способность
(минимальная ненулевая разница между измерениями)

```
private long lastValue;

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


Таймеры: типичный результат [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
```


Таймеры: типичный результат [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
```


Таймеры: типичный результат [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
```


Таймеры: эпичный результат [Windows]

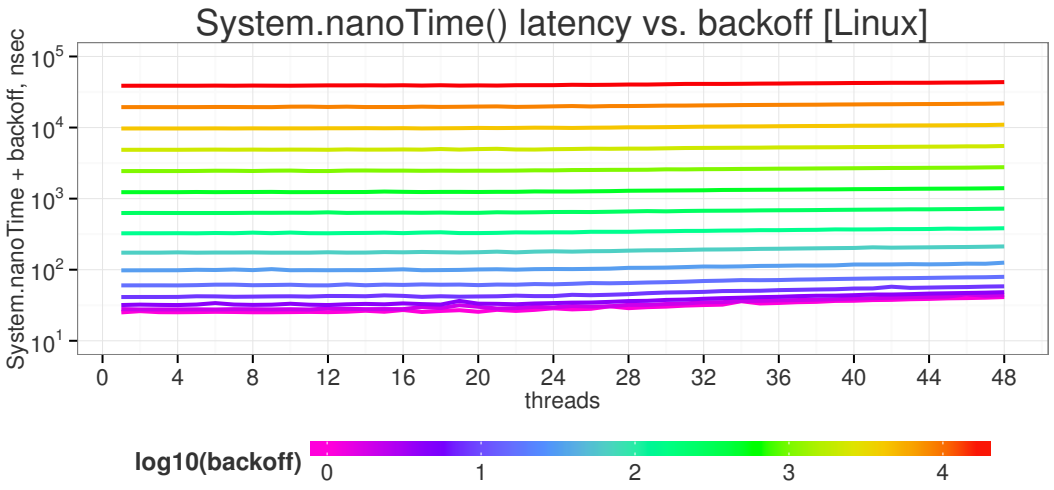
```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
Windows Server 2008, 6.0, amd64

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
```


Таймеры: модельнй эксперимент

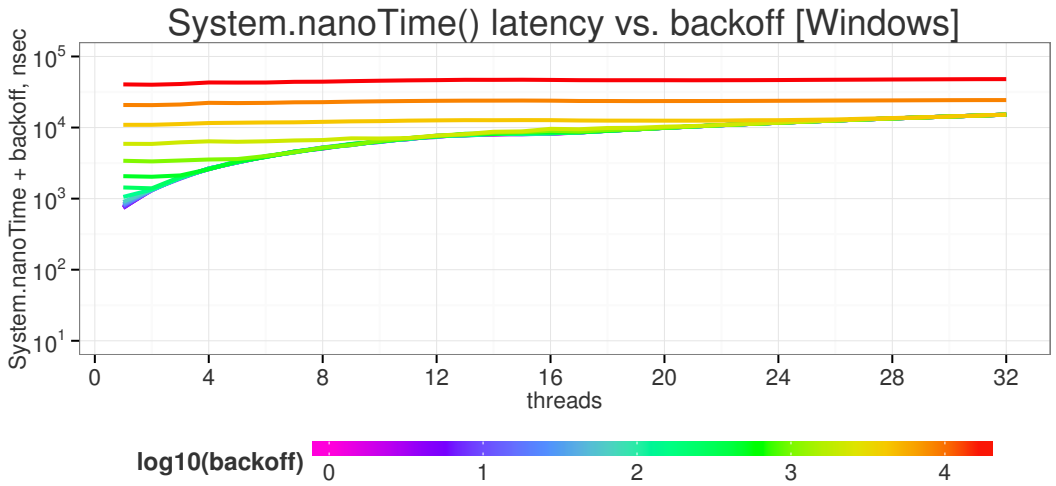
- Но если `System.nanoTime()` тяжёлый и потенциально не скалирующийся, наш тест ставит систему на колени?
- Надо смотреть, когда всё становится совсем плохо:

```
@Param
int backoff;

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

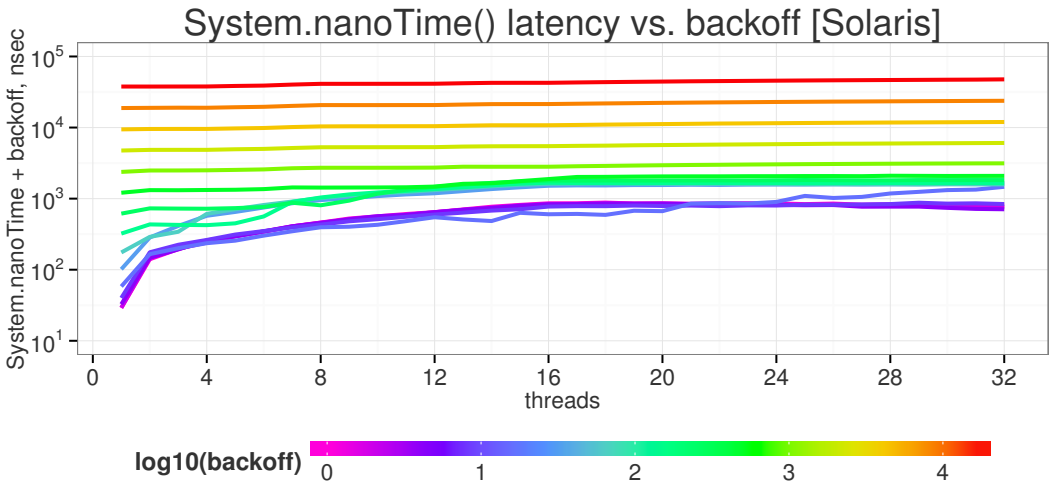

Таймеры: выглядит нормально [Linux]



Таймеры: что за... [Windows]



Таймеры: платим за монотонность [Solaris]



Таймеры: типичный результат [Mac OS X]

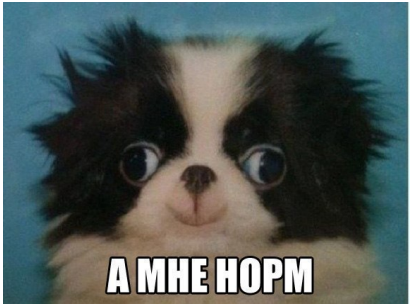
```
Java(TM) SE Runtime Environment, 1.8.0-b132
Java HotSpot(TM) 64-Bit Server VM, 25.0-b70
Mac OS X, 10.9.2, x86_64

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
```


Таймеры: подытожим

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

- Пользователь с `nanoTime` – как обезьяна с гранатой
- `nanoTime` можно и нужно использовать в некоторых случаях, когда вы отчётливо понимаете последствия
- Во многих случаях прямое измерение невозможно, и нам приходится строить модели на косвенных данных

Таймеры: продолжаешь шутить?



Наши куски кода достаточно
большие и плотные, чтобы не наступить
на гранулярность и латентность
`System.nanoTime()`!

Java

Omission: тяжёлый бенчмарк такой бенчмарк

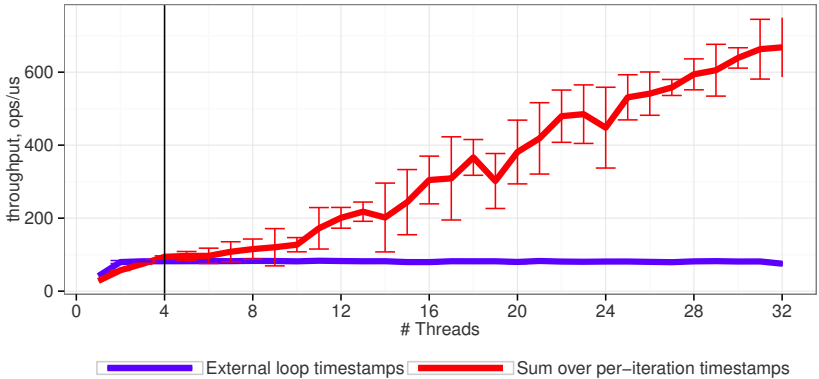
```
public long measure() {
  long ops = 0;
  long startTime = System.nanoTime();
  while(!isDone) {
    setup(); // want to skip this
    work();
    ops++;
  }
  return ops / (System.nanoTime() - startTime);
}
```


Omission: измеряем в отдельном блоке

```
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;
}
```


Omission: проверяем пустой setup()...

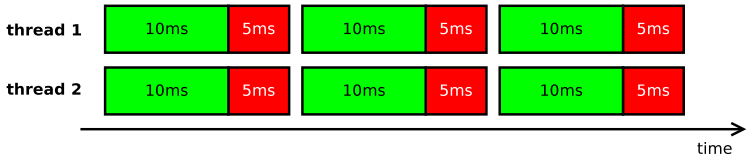
Измеряем throughput... и он растёт, даже когда процы кончились?



Omission: Hint

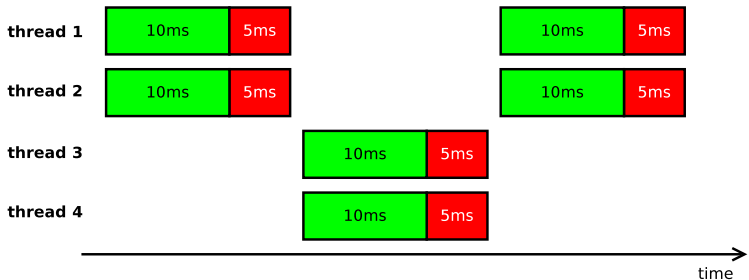
```
public long measure() {
  long ops = 0;
  long realTime = 0;
  while(!isDone) {
    setup(); // skip this
    long time = System.nanoTime();
      work();
    realTime += (System.nanoTime() - time);
    ops++;
    ...WHOOPS, WE DE-SCHEDULE HERE...
  }
  return ops / realTime;
}
```


Omission: на пальцах



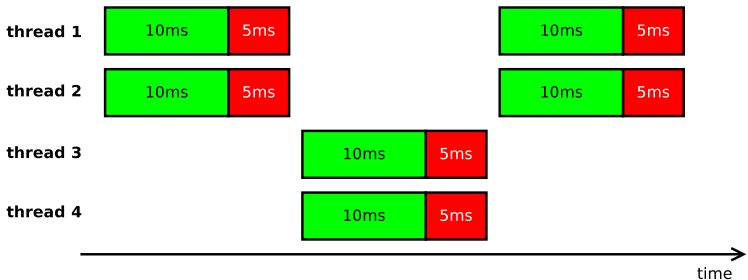
- Измеряем время на операцию, в среднем 10 мс/оп $\Rightarrow$ каждый i-тый поток считает, что его собственный throughput $\lambda_i = 100$ оп/с
- У нас два потока, и поэтому $\sum_{i=1}^{N} \lambda_i = 200$ оп/с

Omission: ещё немножко потоков



- Каждый поток всё ещё считает, что $\lambda_i = 100$ оп/с!
- Но теперь у нас четыре треда $\Rightarrow \sum_{i=1}^{N} \lambda_i = 400$ оп/с

Omission: ещё немножко потоков



- Каждый поток всё ещё считает, что $\lambda_i = 100$ оп/с!
- Но теперь у нас четыре треда $\Rightarrow \sum_{i=1}^{N} \lambda_i = 400$ оп/с

Omission: подытожим

"Phillip J. Fry is experiencing the major safepoint event"

Секундомеры могут измерять совсем не то, что вы думаете

- Проблемы со всеми метриками, содержащими время
- Очень просто получить проблемы на перегруженных системах
- Очень просто получить, если измерители завязаны на рабочую нагрузку

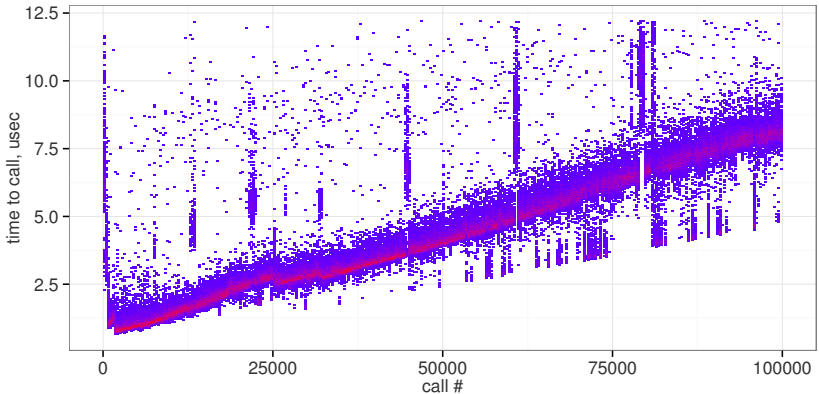
S.S.: (TGIF) Thank God It's Fibonacci

Какие проблемы, начальник?

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

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


S.S.: мерим каждый вызов...

А, ну да, у бенчмарка нет steady state:



Java

S.S.: проблема

Нет steady state – **нельзя** использовать time-based бенчмарки!
Чем дольше мы мерим, тем «хуже» кажется результат:

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

S.S.: Pick Your Poison

Time-based бенчмарки:

- Измеряем в хрен знает каком режиме
- Как сравнить две разные реализации?
  (А если ещё и модель нелинейная, то сразу кранты...)

Work-based бенчмарки:

- Проблемы с латентностью/гранулярностью таймеров
- Проблемы с omission
- Проблемы с переходными процессами

S.S.: подытожим

«The only winning move
is not to play at all»

Приходится выбирать между двумя очень плохими вариантами: проще вообще не выбирать.

Бенчмарки без steady state –
– это полная О.П.Ж.А.!

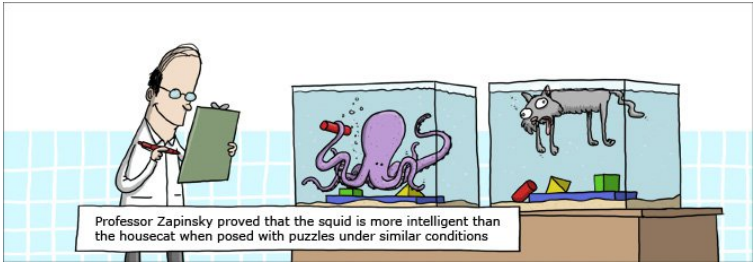
S.S.: паллиатив

Измеряем батчами:

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

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


Инженерный подход

Инженерный подход: бенчмарки



Очень хочется сравнимых результатов

- Каждая мелкая неподконтрольная деталь – это свободная переменная
- Библиотеки – это огромные комплексы мелких деталей
- Языковые среды – это **галактики** мелких деталей

Java

Инженерный подход: TL;DR;

В этой секции мы рассмотрим забавную историю сравнения Java vs. Scala. Произошла она на далёком-далёком StackOverflow, где роятся люди, что уверены в том, что хвостовая рекурсия – это лучшее изобретение человечества – в доме, что построил Том.

Полная история и конспект здесь:

`http://shipilev.net/blog/2014/java-scala-divided-we-fail/`

Инженерный подход: Scala, @tailrec

```
@tailrec private def
isDivisible(v: Int, d: Int, l: Int): Boolean = {
  if (d > l) true
  else (v % d == 0) && isDivisible(v, d + 1, l)
}

@Benchmark
def test(): Int = {
  var v = 10
  while(!isDivisible(v, 2, l))
    v += 2
  v
}
```


Инженерный подход: Java, отсутствие tailrec

```
private boolean isDivisible(int v, int d, int l) {
  if (d > l) return true;
  else
    return (v % d == 0) && isDivisible(v, d+1, l);
}

@Benchmark
public int test() {
  int v = 10;
  while(!isDivisible(v, 2, l))
    v += 2;
  return val;
}
```


Инженерный подход: мерим...

Benchmark	lim	Score	Score error	Units
ScalaBench	1	0.002	0.000	us/op
ScalaBench	5	0.494	0.005	us/op
ScalaBench	10	24.228	0.268	us/op
ScalaBench	15	3457.733	33.070	us/op
ScalaBench	20	2505634.259	15366.665	us/op
JavaBench	1	0.002	0.000	us/op
JavaBench	5	0.252	0.001	us/op
JavaBench	10	12.782	0.325	us/op
JavaBench	15	1615.890	7.647	us/op
JavaBench	20	1053187.731	20502.217	us/op

Инженерный подход: профилируем Java

```
Result: 12.719 +-(99.9%) 0.284 us/op [Average]

....[Thread state distributions].....................
 91.3%         RUNNABLE
  8.7%         WAITING

....[Thread state: RUNNABLE].........................
 58.0%  63.5% n.s.JavaBench.isDivisible
 32.9%  36.1% n.s.JavaBench.test

....[Thread state: WAITING]..........................
  8.7% 100.0% <irrelevant>
```


Инженерный подход: профилируем Scala

```
Result: 24.076 +-(99.9%) 0.728 us/op [Average]

....[Thread state distributions].....................
 91.4%        RUNNABLE
  8.6%        WAITING

....[Thread state: RUNNABLE].........................
 90.6%  99.1% n.s.ScalaBench.test
  0.9%   0.9% n.s.generated.ScalaBench_test.test_avgt_jmhLoop

....[Thread state: WAITING]..........................
  8.6% 100.0% <irrelevant>
```


Инженерный подход: грубые профайлеры

Грубые профайлеры (уровня отдельных методов) практически бесполезны в разборе nano- и micro-бенчмарков.

Кроме того, они почти всегда семплят на safepoint'ах, что отдельный фейл.

Инженерный подход: JMH perfasm

```
java -jar benchmarks.jar ... -prof perfasm
```

На удивление, достаточно просто подружить три вещи:

1. Linux `perf` даёт нам легковесное HWC сэмплирование
2. JVM даёт нам отображение нативных адресов на нативные методы
3. `-XX:+PrintAssembly` даёт отображение адресов на Java-код

На деле, у нас дофига хороших профайлеров, но их не всегда удобно использовать.

Инженерный подход: самое горячее место в Scala

Настоящее x86-деление, одна штука:

```
 clocks    insns    code
-------------------------------------------------------
; n.s.g.ScalaBench_test::test_avgt_jmhLoop
...
  0.27%    0.17%    cltd
  2.24%   17.26%    idiv   %ecx
 77.99%   66.44%    test   %edx,%edx
...
```

Как вообще можно выиграть в два раза?

Инженерный подход: самое горячее место в Java

```
 clocks    insns   code
-------------------------------------------------------
; n.s.JavaBench::isDivisible
...
  1.68%    2.76%   cltd
  0.06%    0.16%   idiv   %ecx
 27.59%   36.37%   test   %edx,%edx
 ...
  0.04%            cltd
                   idiv   %r10d
 12.24%    1.54%   test   %edx,%edx
 ...
  0.01%            callq  <recursive-call>
```


Инженерный подход: 2-ое горячее место в Java

```
 clocks    insns    code
-------------------------------------------------------
; n.s.g.JavaBench_test::test_avgt_jmhLoop
 ...
  1.34%    0.21%    imul   $0x55555556,%rdx,%rdx
  1.25%    0.20%    sar    $0x20,%rdx
  1.15%    2.36%    mov    %edx,%esi
  0.95%    1.51%    sub    %r10d,%esi             ; irem
 ...
```

[2] http://www.hackersdelight.org/divcMore.pdf

Инженерный подход: 2-ое горячее место в Java

```
 clocks    insns    code
-------------------------------------------------------
; n.s.g.JavaBench_test::test_avgt_jmhLoop
 ...
  1.34%    0.21%    imul   $0x55555556,%rdx,%rdx
  1.25%    0.20%    sar    $0x20,%rdx
  1.15%    2.36%    mov    %edx,%esi
  0.95%    1.51%    sub    %r10d,%esi            ; irem
 ...
```

Широко известный трюк замены взятия остатка на умножение и сдвиг![2]

[2] http://www.hackersdelight.org/divcMore.pdf

Инженерный подход: на пальцах

```
// inlines twice, specializes for d={2,3}
private boolean isDivisble(int v, int d, int l) {
  ...
  return (v % d == 0) && isDivisble(v, d+1, l);
}

@Benchmark
public int test() {
  int v = 10;
  while(!isDivisble(v, 2, l))
    v += 2;
  return val;
}
```


Инженерный подход: делаем «d» непредсказуемым

Benchmark	lim	Score	Score error	Units
ScalaBench	1	0.002	0.000	us/op
ScalaBench	5	0.489	0.002	us/op
ScalaBench	10	23.777	0.116	us/op
ScalaBench	15	3379.870	5.737	us/op
ScalaBench	20	2468845.944	2413.573	us/op
JavaBench	1	0.003	0.000	us/op
JavaBench	5	0.465	0.001	us/op
JavaBench	10	22.989	0.095	us/op
JavaBench	15	3453.116	16.390	us/op
JavaBench	20	2518726.451	4374.482	us/op

Инженерный подход: подытожим

«Days since the last benchmarking accident: 0»
(@gvsmirnov)

Бенчмарки без анализа сильно расстраивают Шипилёва.

Хоть запоказывайтесь графиков: Language A vs. Language B, Nashorn vs. Rhino, Graal vs. C2, и т.п, но всё, что я вижу в этих графиках, это

БАЙЕСОВЫЙ ШУМ

Fin

Fin: вывод

«If you don't analyze the benchmarks, you've gonna waste a good time»

Поверхностные заключения почти всегда исходят из существующих предубеждений.

Бенчмарки нужны для того, чтобы исследовать реальность, а не потворствовать предубеждениям.