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 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: 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
- 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

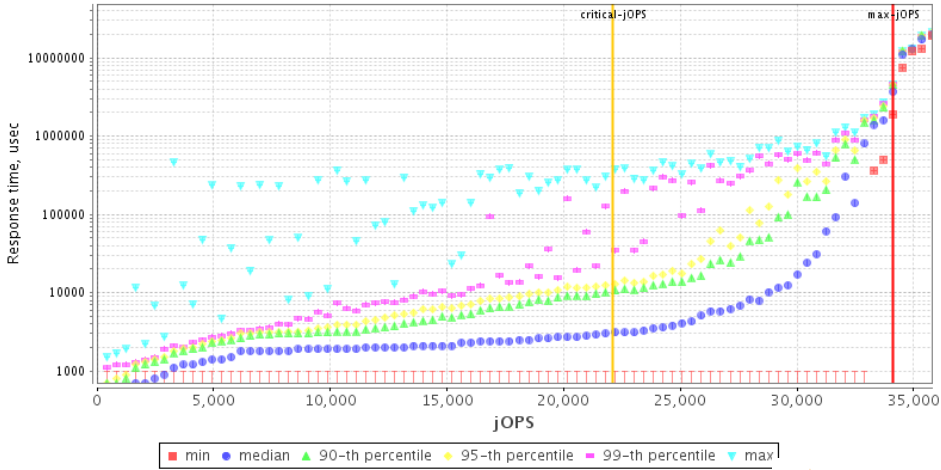
- Отличия в нагрузке:
  - λ обычно измеряют под пиковой нагрузкой
  - τ как правило измеряют под регулируемой нагрузкой
  - Нагрузка – это новая степень свободы!

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

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

  - Нужно мерить каждое событие, 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.

- 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!

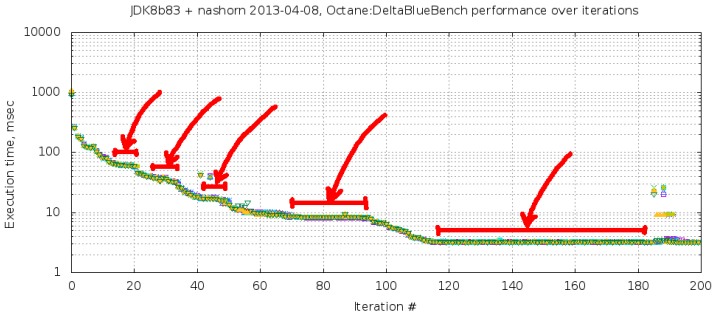
Как измеряем: Transients

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 измеряем: Иногда steady state нет!

```
public class FibonacciGen {
    BigInteger n1 = ONE; BigInteger n2 = ZERO;
    public BigInteger next() {
        BigInteger cur = n1.add(n2);
        n2 = n1; n1 = cur;
        return cur;
    }
}
```

- Каждый вызов `next()` всё медленней и медленней
- Имеет смысл измерять первые N операций – а там transients!

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, как растёт наша экспертиза
  - Всякий внешний бенч валидируется переписыванием под JMH

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, как растёт наша экспертиза
  - Всякий внешний бенч валидируется переписыванием под JMH
- Мы вынули столько неочевидных граблей из JMH, что не верим ни одному известному харнессу.

Практика: Цель

Цель всего этого мероприятия в том,
чтобы вас как следует напугать
и заставить бросить:

- (тупо) писать обёртки для бенчмарок
- (тупо) доверять обёрткам для бенчмаркок
- (тупо) доверять бенчмаркам

System: Задачка (A)

Давайте запустим простенький бенчмарк.
Система говорит, что у неё 4 CPU.

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

System: Задачка (A)

Давайте запустим простенький бенчмарк.
Система говорит, что у неё 4 CPU.

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

- Вопрос 1: Почему переход $2 \rightarrow 4$ потока такой хилый?

System: Задачка (A)

Давайте запустим простенький бенчмарк.
Система говорит, что у неё 4 CPU.

Threads	Ops/nsec	Scale
1	3.06 ± 0.10	
2	5.72 ± 0.10	1.87 ± 0.03
4	5.87 ± 0.02	1.91 ± 0.03

- Вопрос 1: Почему переход $2 \rightarrow 4$ потока такой хилый?
- Вопрос 2: Почему переход $1 \rightarrow 2$ потока всего 1.87x?

System: Power management

Запускаем простой бенчмарк,
+ и зажимаем частоту CPU в 2 GHz:

Threads	Ops/nsec	Scale
1	1.97 ± 0.02	
2	3.94 ± 0.05	2.00 ± 0.02
4	4.03 ± 0.04	2.04 ± 0.02

System: Power management

Многие подсистемы балансируют power-vs-performance

(Ex.: cpufreq, SpeedStep, Cool&Quiet, TurboBoost)

- **Засада**: ломает гомогенность времени
- **Костыль**: выключить PM, зажать частоту CPU
- **JMH**: работаем дольше, не паркуем потоки

System: OS Schedulers

Шедулеры OS балансируют affinity-vs-power

(Ex.: Solaris schedulers, Linux power-efficient taskqueues)

- **Засада**: ломает иллюзию симметричности CPU
- **Костыль**: зажать политики шедулинга
- **JMH**: работаем дольше, не паркуем потоки

System: Time Sharing

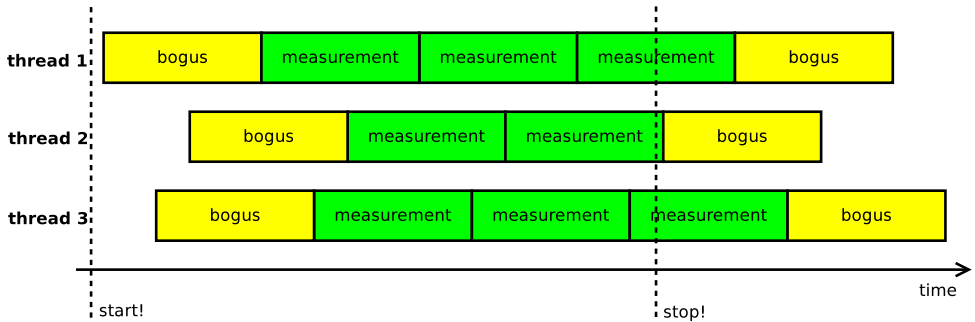
Системы разделяемого времени балансируют утилизацию.

(Ex.: тысячи их)

- **Засада**: старт/останов потоков не мгновенный, время работы потоков не детерминировано, загрузка во времени неоднородна
- **Костыль**: удостовериться, что все потоки начали исполнять работу
- **JMH**: «synchronize iterations»

System: Time Sharing, #2

JMH – synchronize iterations:



VM: Задачка (C)

```
@GenerateMicroBenchmark
public void baseline() {
}
```

0.5 ± 0.1 ns

```
@GenerateMicroBenchmark
public void measureWrong() {
    Math.log(x);
}
```

0.5 ± 0.1 ns

```
@GenerateMicroBenchmark
public double measureRight() {
    return Math.log(x);
}
```

34.0 ± 1.0 ns

VM: Dead-code elimination

Оптимизаторы очень хорошо удаляют код без наблюдаемых эффектов.

- **Засада**: может снести часть бенчмарка вдрабадан
- **Костыль**: наводим эффекты на результаты (суммируем, складываем, печатаем, и т.п.)
- **JMH**: поддержка в API

VM: Задачка (D)

```
@GenerateMicroBenchmark
public void baseline() {
}
```

0.5 ± 0.1 ns

```
@GenerateMicroBenchmark
public double measureWrong() {
    return Math.log(42);
}
```

1.0 ± 0.1 ns

```
private double x = 42;
@GenerateMicroBenchmark
public double measureRight() {
    return Math.log(x);
}
```

34.0 ± 1.0 ns

VM: Constant folding, etc.

Оптимизаторы довольно хороши в предвычислениях.

- **Засада**: может предоптимизировать часть бенчмарка
- **Костыль**: сделать входные данные непредсказуемыми
- **JMH**: поддержка в API

VM: CSE

JMH ломает спекуляцию о чтениях в разных вызовах @GMB метода

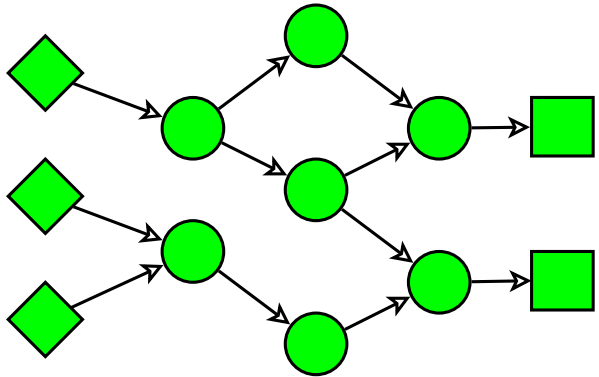
```
double x;

@GenerateMicroBenchmark
double doWork() {
  doStuff(x);
}
```

```
volatile boolean done;
void doMeasure() {
  while (!done) {
    doWork();
  }
}
```

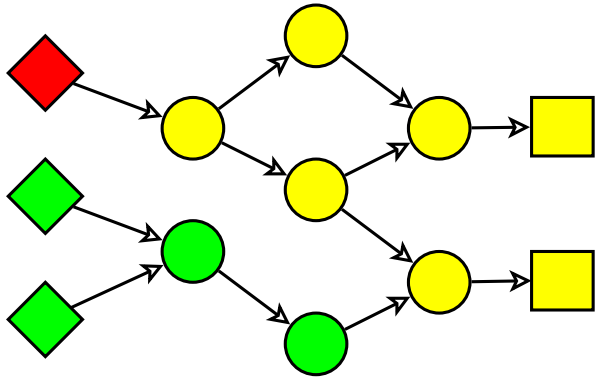
(Перевод: читаем всё из кучи $\Rightarrow$ мы спасены!)

VM: DCE, CSE... одно и то же!



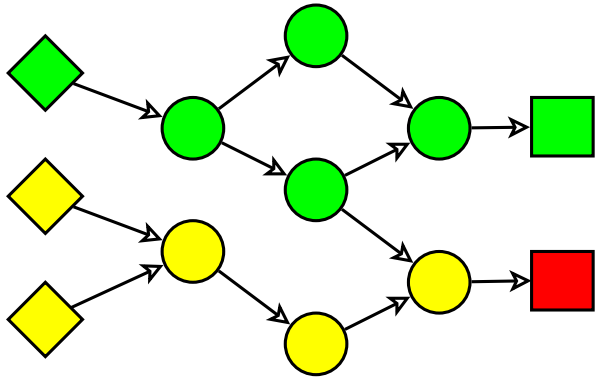
Потеря непредсказуемости что источника, что результата теряет и часть бенчмарка. Тихо и незаметно.

VM: DCE, CSE... одно и то же!



Потеря непредсказуемости что источника, что результата теряет и часть бенчмарка. Тихо и незаметно.

VM: DCE, CSE... одно и то же!



Потеря непредсказуемости что источника, что результата теряет и часть бенчмарка. Тихо и незаметно.

VM: Задачка (E)

```
// changing N, will performance differ?
static int N = 100;

@GenerateMicroBenchmark
public int test() { return doWork(N); }

int x = 1, y = 2;

private int doWork(int reps) {
  int s = 0;
  for (int i = 0; i < reps; i++)
    s += (x + y);
  return s;
}
```


VM: Задачка (E), #2

N	ns/call	ns/add
1	1.5 ± 0.1	1.5 ± 0.1
10	2.0 ± 0.1	0.1 ± 0.01
100	2.7 ± 0.2	0.05 ± 0.02
1000	68.8 ± 0.9	0.07 ± 0.01
10000	410.3 ± 2.1	0.04 ± 0.01
100000	3836.1 ± 40.6	0.04 ± 0.01

VM: Задачка (E), #2

N	ns/call	ns/add
1	1.5 ± 0.1	1.5 ± 0.1
10	2.0 ± 0.1	0.1 ± 0.01
100	2.7 ± 0.2	0.05 ± 0.02
1000	68.8 ± 0.9	0.07 ± 0.01
10000	410.3 ± 2.1	0.04 ± 0.01
100000	3836.1 ± 40.6	0.04 ± 0.01

Ну и какой строчке верить?

0.04 ns/add $\Rightarrow$ 25 adds/ns $\Rightarrow$ GTFO!

VM: Loop unrolling

Раскрутка циклов сильно расширяет диапазон оптимизаций.

- **Засада**: положим, одна итерация цикла занимает M нс. Тогда после раскрутки цикла она эффективно занимает αM ns, где $\alpha \in [0; +\infty)$
- **Костыль**: избегаем раскручиваемых циклов, минимизируем эффекты от раскрутки
- **JMH**: оказывается, что костыли для CSE/DCE так же ломают эффекты от раскрутки

VM: Задачка (F)

```
interface M {
  void inc();
}

abstract class AM implements M {
  int c;
  void inc() {
    c++;
  }
}

class M1 extends AM {}
class M2 extends AM {}
```


VM: Задачка (F), #2

```
M m1 = new M1();
M m2 = new M2();

@GenerateMicroBenchmark
public void testM1() { test(m1); }

@GenerateMicroBenchmark
public void testM2() { test(m2); }

void test(M m) {
    for (int i = 0; i < 100; i++)
        m.inc();
}
```


VM: Задачка (F), #3

test	ns/op
testM1	4.6 ± 0.1
testM2	36.0 ± 0.4

VM: Задачка (F), #3

test	ns/op
`testM1`	4.6 ± 0.1
`testM2`	36.0 ± 0.4
repeat `testM1`	35.8 ± 0.4

VM: Задачка (F), #3

test	ns/op
`testM1`	4.6 ± 0.1
`testM2`	36.0 ± 0.4
repeat `testM1`	35.8 ± 0.4
forked `testM1`	4.5 ± 0.1
forked `testM2`	4.5 ± 0.1

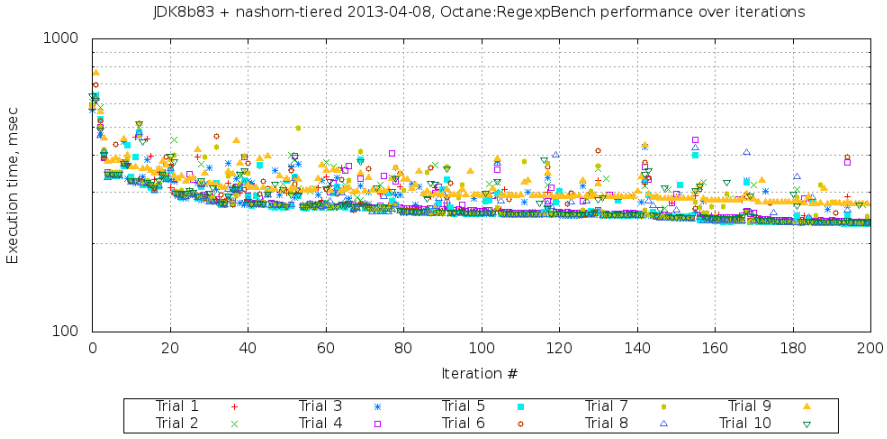
VM: Profile feedback

Динамические оптимизации
могут использовать информацию времени исполнения

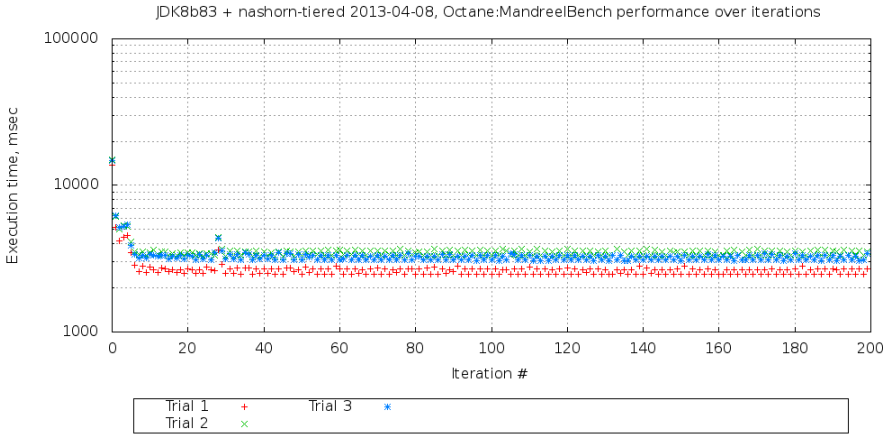
(Ex.: call profile, type profile, CHA info)

- **Засада**: Серьёзная разница между режимами «всё в одной VM» и «всё в отдельных VM»
- **Костыль**: Прогреваем все бенчмарки вместе, ИЛИ запускаем каждый бенчмарк в отдельной JVM
- **JMH**: поддержка bulk warmup; forking

VM: Задачка (G)



VM: Задачка (G), #2



VM: Run-to-run variance

Многие масштабируемые алгоритмы не детерминированы!

(Ex.: memory allocators, profiler counters, non-fair locks, concurrent data structures, ...)

- **Засада**: (потенциально) (огромная) разница между запусками в одной конфигурации
- **Костыль**: протоколирование решений на каждом уровне (e.g. запись решений компилятора), несколько запусков JVM
- **JMH**: несколько запусков JVM

CPU: Задачка (H)

```
@State
public class TreeMapBench {
  Map<String, String> map = new TreeMap<>();

  @Setup
  public void setup() { populate(map); }

  @GenerateMicroBenchmark
  public void test(BlackHole bh) {
    for(String key : map.keySet()) {
      String value = map.get(key);
      bh.consume(value);
    }
  }
}
```


CPU: Задачка (H), #2

```
@GenerateMicroBenchmark
public void test(BlackHole bh) {
  for(String key : map.keySet()) {
    String value = map.get(key);
    bh.consume(value);
  }
}
```

	Exclusive	Shared
Throughput, op/sec	615 ± 12	828 ± 21

CPU: Задачка (H), #2

```
@GenerateMicroBenchmark
public void test(BlackHole bh) {
  for(String key : map.keySet()) {
    String value = map.get(key);
    bh.consume(value);
  }
}
```

	Exclusive	**Shared**
Throughput, op/sec	615 ± 12	828 ± 21
Threads	4	4

CPU: Задачка (H), #2

```
@GenerateMicroBenchmark
public void test(BlackHole bh) {
  for(String key : map.keySet()) {
    String value = map.get(key);
    bh.consume(value);
  }
}
```

	Exclusive	**Shared**
Throughput, op/sec	615 ± 12	828 ± 21
Threads	4	4
Maps	4	1

CPU: Задачка (H), #2

```
@GenerateMicroBenchmark
public void test(BlackHole bh) {
  for(String key : map.keySet()) {
    String value = map.get(key);
    bh.consume(value);
  }
}
```

	Exclusive	**Shared**
Throughput, op/sec	615 ± 12	828 ± 21
Threads	4	4
Maps	4	1
Footprint, Kb	1024	256

CPU: Cache capacity

Основная память слишком далека.
Давайте кешировать всё прямо в проце!

- **Засада**: Производительность заметно отличается в зависимости от футпринта, путей обхода памяти, да и банальной удачи
- **Костыль**: Следим за футпринтом; пробуем разные размеры данных; пробуем разные shared/unshared случаи
- **JMH**: `@State` sharing

CPU: Задачка (J)[2]

Exhibit B.

```
int sum = 0;
for (int x : a) {
  if (x < 0) {
    sum -= x;
  } else {
    sum += x;
  }
}
return sum;
```

Exhibit P.

```
int sum = 0;
for (int x : a) {
  sum += Math.abs(x);
}
return sum;
```

Которая версия быстрее?

[2]Credits: Sergey Kuksenko (@kuksenk0)

CPU: Задачка (J)

E. Branched

```
L0:  mov   0xc(%ecx,%ebp,4),%ebx
     test  %ebx,%ebx
     jl    L1
     add   %ebx,%eax
     jmp   L2
L1:  sub   %ebx,%eax
L2:  inc   %ebp
     cmp   %edx,%ebp
     jl    L0
```

E. Predicated

```
L0:  mov   0xc(%ecx,%ebp,4),%ebx
     mov   %ebx,%esi
     neg   %esi
     test  %ebx,%ebx
     cmovl %esi,%ebx
     add   %ebx,%eax
     inc   %ebp
     cmp   %edx,%ebp
     jl    Loop
```

Которая версия быстрее?

CPU: Задачка (J)

Regular Pattern = (+, −)*

	NHM	Bldzr	C-A9	SNB
branch_regular	0.9	0.8	5.0	0.5
branch_shuffled	6.2	2.8	9.4	1.0
branch_sorted	0.9	1.0	5.0	0.6
predicated_regular	2.0	1.0	5.3	0.8
predicated_shuffled	2.0	1.0	9.3	0.8
predicated_sorted	2.0	1.0	5.7	0.8

time, nsec/op

CPU: Branch Prediction

Out-of-Order процессоры очень много спекулируют. Большую часть времени (99%+) делают это корректно!

- **Засада**: Провал в производительности, когда спекуляция проваливается
- **Костыль**: Реалистичные данные! Много, много разных наборов данных

Выводы

Выводы: Benchmarking is Serious Business

Огромное поле для ошибок.

- Написание тестов требует экспертизы
- Написание фреймворков требует ещё большей экспертизы
- Не верьте красивым репортам, верьте логичным результатам

Выводы: инструменты

1. Мозг
   - Плагин «данунеможетбыть» для перепроверок фактов
   - Плагин «щапридумаем» для построения гипотез и экспериментов
   - Плагин «чётоянепонял» для проверки консистентности гипотез
   - Плагин «ядурак» для лёгкого отвержения ложных гипотез
2. Руки
   - Прямые, для постановки аккуратных экспериментов
   - Сильные, для обработки тонн экспериментальных данных
3. Язык, уши, глаза и прочее I/O
   - Для обмена результатами и peer review
   - Для доступа к предыдущим экспериментам

Выводы: прочие инструменты

1. Фреймворки
   - JMH: `http://openjdk.java.net/projects/code-tools/jmh/`
2. Профилировщики
   - VisualVM, JRockit Mission Control, Oracle Studio Performance Analyzer
   - top, vmstat, mpstat, iostat, dtrace, strace
3. Дизассемблеры
   - `-XX:+PrintAssembly`
   - `https://wikis.sun.com/display/HotSpotInternals/PrintAssembly`

Backup

Backup: Time Sharing, Задачка (B)

```
public void measure() {
    long startTime = System.nanoTime();
    while(!isDone) {
        work();
    }
    println(System.nanoTime() - startTime);
}
```

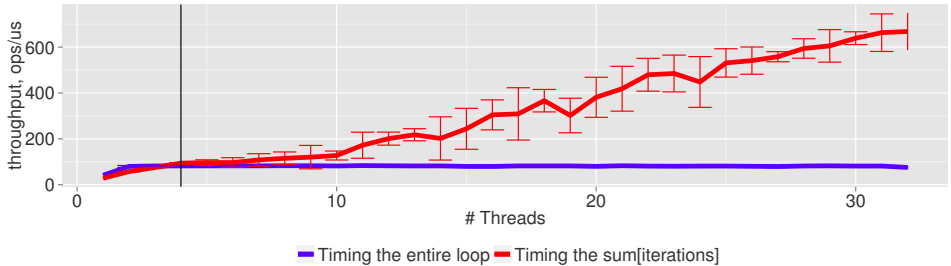

Backup: Time Sharing, Задачка (B)

«Is there a problem, officer?»

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


Backup: Time Sharing, Задачка (B)

Измеряем количество вызовов в секунду, да?



...а оно растёт за количество CPU – WTF?!

Backup: Time Sharing, Задачка (B)

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


Backup: Time Sharing

Разделение времени даёт иллюзию параллельной работы нескольких потоков

- **Засада**: эта иллюзия не всегда выполняется для производительности!
- **Костыль**: **никогда** не перегружать систему!
- **JMH**: большой красный флаг на въезде в страну

Backup: Inlining budgets

Подстановка (inlining) – это убер-оптимизация

- **Засада**: Всё заинлайнить нельзя ⇒ приходится думать о том, что инлайнится, а что нет
- **Костыль**: Методы мельче, циклы мельче, подглядываем в -XX:+PrintInlining, используем компиляторные хинты
- **JMH**: генерируем горячие циклы в мелкие методы, даём удобный `@CompileControl`

Backup: сгенерированный JMH код

JIT-компиляция и подстановки начинаются с этого метода:

```
public void testLong_loop
    (Loop loop, Result r, MyBenchmark bench) {
  long ops = 0;
  r.start = System.nanoTime();
  do {
    bench.testLong(); // @GenerateMicroBenchmark
    ops++;
  } while(!loop.isDone);
  r.end = System.nanoTime();
  r.ops = ops;
}
```


Backup: Задачка (I)

Насколько хорошо скалируется этот код?

```
@State(Scope.Benchmark) class Shared {
  final int[] c = new int[64];
}

@State(Scope.Thread) class Local {
  static final AtomicInteger COUNTER = ...;
  final int index = COUNTER.incrementAndGet();
}

@GenerateMicroBenchmark
void work(Shared s, Local l) {
  s.c[l.index]++;
}
```


Backup: Задачка (I), #2

Threads	Average ns/call	Hit
1	2.0 ± 0.1	
2	18.5 ± 2.4	9x
4	32.9 ± 6.2	16x
8	85.4 ± 13.4	42x
16	208.9 ± 52.1	104x
32	464.2 ± 46.1	232x

Почему?

Backup: Bulk method transfers

Система следит за когерентностью памяти по кусочкам. Характерный размер кусочка: 32, 64, 128 байта.

- **Засада**: модификация рядом лежащих данных разными потоками даётся с трудом (false sharing)
- **Костыль**: паддинг, subclass juggling, `@Contended`
- **JMH**: все внутренние структуры отпажжены, `@State`-объекты паддятся автоматически

