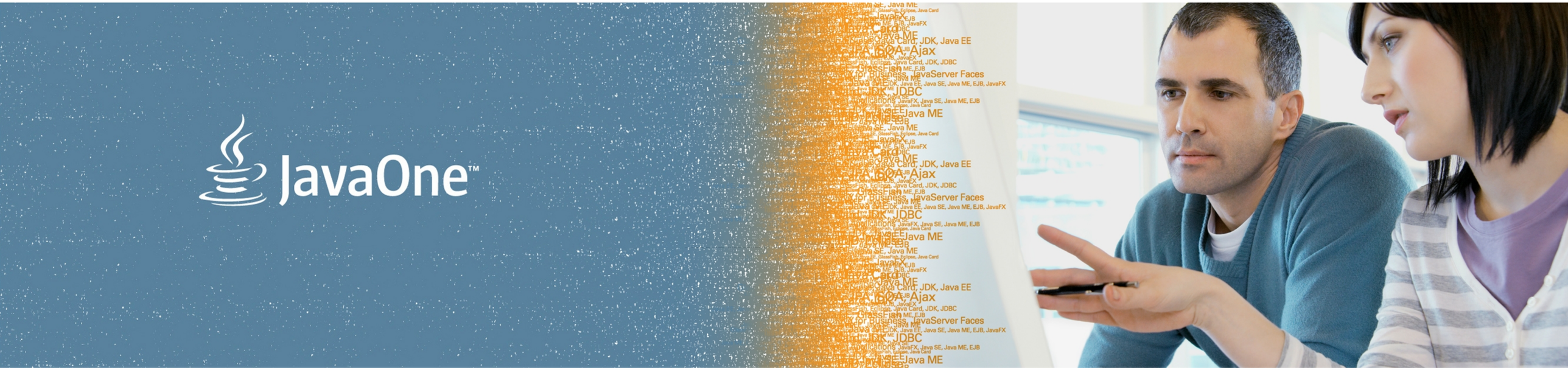

JavaOne™



ORACLE®

(The Art of) (Java) Benchmarking

Aleksey Shipilev
Java Platform Performance

Agenda

Введение

Общая теория

Java Benchmarking

Инструменты

Введение в введение

- Computer Science → Software Engineering
  - Строим приложения по функциональным требованиям
  - В большой степени абстрактно, в “идеальном мире”
    - Теоретически неограниченная свобода – искусство!
  - Рассуждения при помощи формальных методов
- Software Performance Engineering
  - “Real world strikes back!”
  - Исследуем взаимодействие софта с железом на типичных данных
    - Производительность уже нельзя предсказать
    - Производительность можно только измерить
  - Естественно-научные методы

Бенчмарки

- Что это такое?
  - Def.: “Программа, используемая для измерения производительности”
    - Жило-было приложение, добавили измерение времени – бац – уже бенчмарк
  - Каждый запуск бенчмарка – вычислительный эксперимент
- Типичные требования к бенчмаркам
  - Результат запуска: значение некоторой метрики
  - Надёжность (воспроизводимост)
  - Объективность (“test the right thing”)
  - Лёгкость запуска
  - Самовалидация

Классификация бенчмарок

- Реальные приложения
  - Запускаем руками, совершаем действия руками
  - Мерим секундомером, вольтметром, осциллографом
- Автоматические сценарии приложений
  - Зафиксировали какой-нибудь сценарий
  - Автоматически измерили время, мощность, трафик
- Синтетические (макро) бенчмарки
  - Написали приложение, похожее на типичное, эталонное
  - Автоматически измерили
- Микробенчмарки
  - Написали отдельную, маленькую часть
  - Выбросили всё остальное

ORACLE®

Откуда проблемы

- Эксперимент требует понимания объекта исследований
  - Чем тоньше область исследования, тем бóльше нужно знать
- Запуск бенчмарка НЕ последний шаг
  - Понять, что измеряем
  - Интерпретировать, что реально измерили
  - Обнаружить ошибку III рода
  - Заставить себя перепрогнать бенчмарк :)
- Быть хорошим бенчмаркером = делать меньше попыток:
  - Перевод: не делать откровенных глупостей → учиться на глупостях других
  - Обнаруживать ошибки с упреждением, и заранее их исправлять

You're Not Alone

- Transaction Processing Performance Council
  - Дизайн-документы на стандартные бенчмарки
  - Покрыто много угловых случаев для OLTP-бенчмарок
- Standard Performance Evaluation Corporation
  - Индустриальные стандарты бенчмарок
    - SPECcpu, SPECvirt_sc, SPECjbb, SPECjvm, SPECjEnterprise
    - Полные реализации: скачали → развернули → запустили → ??? → profit
  - Часто обновляются для пущей релевантности
- Performance teams
- Peer reviews
  - Как обычно в Интернетах, проверяйте источники

Agenda

Введение

Общая теория

Java Benchmarking

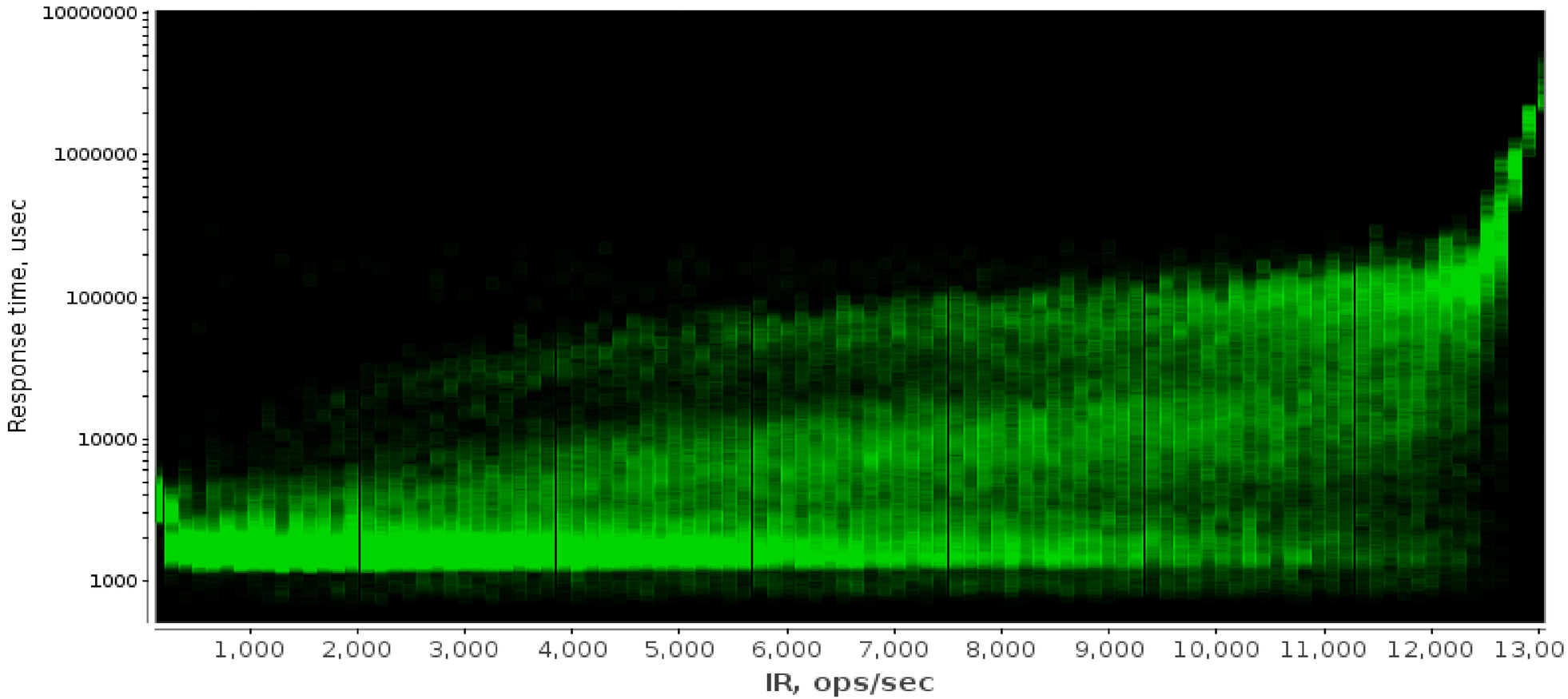
Инструменты

Метрики

- Производительность – композитная метрика
  - количество операций за время (λ, throughput, bandwidth)
  - время на одну операцию (τ, response time, latency)
  - “Bandwidth only tells parts of the story” © pingtest.net
- Оказывается, что проще улучшить λ:
  - DDR2 PC2-3200: 3200 Mb/sec, CL 4ns
  - DDR2 PC2-6400: 6400 Mb/sec, CL 5ns
- В большинстве систем, τ тесно связана с λ
  - “Девять женщин не могут родить ребёнка за один месяц”
    - Несколько женщин могут рожать детей каждый месяц, день, секунду :)
    - А что если на всех не хватает акушерок?

Метрики

- При прочих равных: $\tau \sim \exp(\lambda)$



Composability

- Предположим, есть функциональные блоки A и B
  - Есть ли разница при последовательном (...) и параллельном (||) исполнении?
- Общая функциональность:
  - *Functionality*(A ... B) = *Functionality*(A || B)
  - “Black Box”: поведение одинаково
- Общая производительность:
  - *Performance*(A ... B) **?** *Performance*(A || B)
  - Про это сказать толком ничего нельзя
    - A и B соревнуются за аппаратные ресурсы
    - Возможно, > (эффективный размер кеша меньше)
    - Возможно, < (два потока на HT машине)
    - Возможно, = (нет конфликтов)

Composability

- Важный урок: “чёрный ящик” не работает
  - В реальном мире процессы и потоки борются за ресурсы
  - Давайте рассчитывать на худший случай?
- Особенно важно в “manycore”-мире
  - Подавляющая часть кода исполняется неэксклюзивно
  - Однопоточные бенчмарки бесполезны для предсказания реальной производительности
- Проверять комбинации со всеми возможными программами?
  - Экспоненциальный взрыв пространства конфигураций
  - Хорошее приближение: несколько экземпляров бенчмарки в нескольких потоках

присказка ФизТеха

“Физики имеют дело не с дискретными величинами,
а с распределениями вероятностей”

С нами Бог играет в кости

- Эмпирические результаты случайны
  - Боремся репликацией опытов
    - Много запусков, статистические оценки
    - Запуски в разных условиях
- Контроль!
  - Положительный: проверяем влияние существенных факторов
  - Отрицательный: проверяем влияние несущественных факторов
- Проверка гипотез
  - Система A даёт 50 оп/сек, система B даёт 40 оп/сек, быстрее ли A?

"Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

С нами Бог играет в кости

- Эмпирические результаты случайны
  - Боремся репликацией опытов
    - Много запусков, статистические оценки
    - Запуски в разных условиях
- Контроль!
  - Положительный: проверяем влияние существенных факторов
  - Отрицательный: проверяем влияние несущественных факторов
- Проверка гипотез
  - Система A даёт 50 оп/сек, система B даёт 40 оп/сек, быстрее ли A?
    - Случай 1. A = **50** ± 12 ops/sec, B = **40** ± 24 ops/sec (*наверное*, нет)
    - Случай 2. A = **50** ± 3 ops/sec, B = **40** ± 2 ops/sec (*наверное*, да)

"Piled Higher and Deeper" by Jorge Cham
www.phdcomics.com

David Brent, “The Office”

“Those of you who think you know everything are annoying to those of us who do.”

“Кандидатский минимум”

- HW specifics
  - cpu/memory layout, cache sizes and associativity, power states, etc.
- OS specifics
  - threading model, thread scheduling and affinity, system calls performance, etc.
- Libraries specifics
  - algorithms, tips and tricks for better performance, etc.
- Compilers specifics
  - high-level and low-level optimizations, tips and tricks, etc.
- Algos specifics
  - algorithmic complexity, data access patterns, etc.
- Data specifics
  - representative sizes and values, operation mix, etc.

ORACLE®

И зачем?

- Главный Вопрос:

Как быстро работает мой бенчмарк?

И зачем?

- Главный Вопрос:

~~Как быстро работает мой бенчмарк?~~

Почему мой бенчмарк не может работать быстрее?

И зачем?

- Главный Вопрос:

~~Как быстро работает мой бенчмарк?~~

Почему мой бенчмарк не может работать быстрее?

- Ответ определяет качество эксперимента
  - В какие ограничения мы упёрлись?
  - Действительно ли работает ли та часть кода, которую мы “нагружаем”?
  - Что сделать, чтобы исправить бенчмарк?

Agenda

Введение

Общая теория

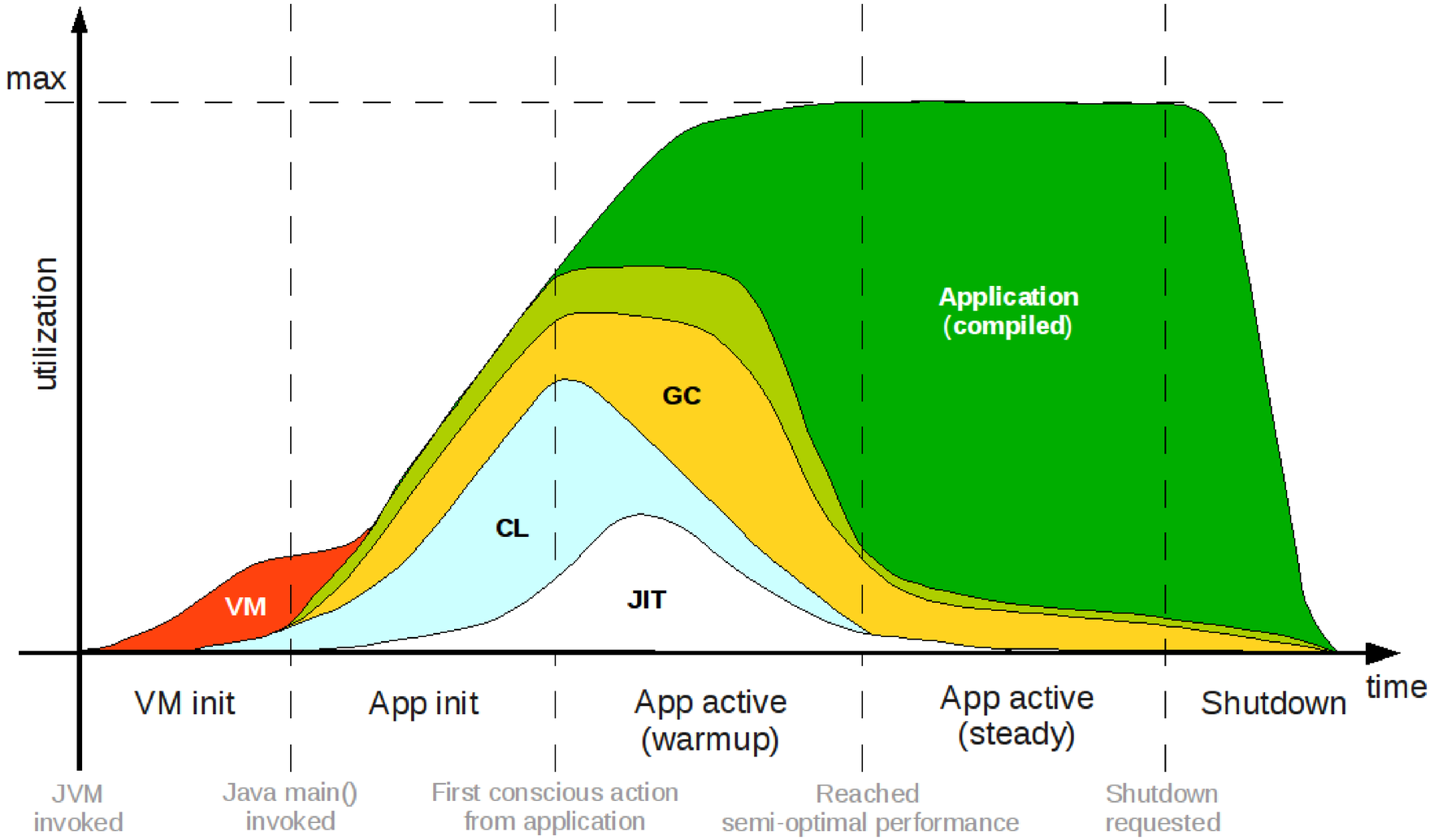
Java Benchmarking

Инструменты

Динамические среды поддают жару

- Помимо всего прочего, нужно ещё знать про:
  - Виртуальную машину
    - Загрузка классов, верификация байткода
  - JIT
    - Профилировка, планы компиляции, OSR
    - Агрессивные оптимизации
    - Штатные и нештатные режимы работы
  - GC
    - Алгоритмы, throughput vs. response time
    - Штатные и нештатные режимы работы

Типичный жизненный цикл JRE



Dr. Cliff Click

“Microbenchmarks are like a microscope.
Magnification is high, but what the heck are you looking at?”

Pitfall #0. Ошибки III рода

- Java-бенчмарк PiDigits из Computer Language Benchmarks Game
  - считает первые N цифр десятичного разложения π
    - требуется arbitrary precision для вычисления членов ряда
  - Java-бенчмарк?
    - Использует нативный GNU MP для операций над числами
      - До 90% времени проводится в нативных врапперах и GMP
    - Java используется, чтобы организовать вычисления и вывести результат

```
public void _run() {
    // ...

    acquire(sema[op1], 1);
    sema[op1].release();
    acquire(sema[op2], 1);
    sema[op2].release();

    if (instr == MUL) {
        GmpUtil.mpz_mul_si(...);
    } else if (instr == ADD) {
        GmpUtil.mpz_add(...);
    } else if (instr == DIV_Q_R) {

        GmpUtil.mpz_tdiv_qr(...);
            sema[op3].release();
    }
    sema[dest].release();
}
```

Pitfall #0. Ошибки III рода

Важные уроки:

Поймите, что вы хотите измерить.

Поймите, что вы *на самом деле* измерили.

Исправьте.

Повторите.

Pitfall #1. С места в карьер, нет warmup

- CLBG делает ставку на две метрики
  - execution time, время выполнения программы
    - т.е. почти “time java $...”
  - memory footprint
- Любой динамический рантайм *сразу* в проигрыше
  - В общее время *кроме самой программы* входит время на инициализацию, компиляцию, адаптивные настройки рантайма
    - Чем короче тест, тем больше относительные накладные расходы
  - Смотрим на время инициалиации...
    - Не используем сложные API, приматываем нативный код
    - Оптимизируем I/O исходных данных и результатов

Pitfall #1. С места в карьер, нет warmup

- Есть там “steady state” тесты:

```
public static void main(String[] args){
    pidigits m = new pidigits(Integer.parseInt(args[0]));
    for (int i=0; i<65; ++i) m.pidigits(false);
    m.pidigits(true);
}
```

- Сравним со “стандартным” тестом:

```
public static void main(String[] args){
    pidigits m = new pidigits(Integer.parseInt(args[0]));
    // for (int i=0; i<19; ++i) m.pidigits(false);
    m.pidigits(true);
}
```

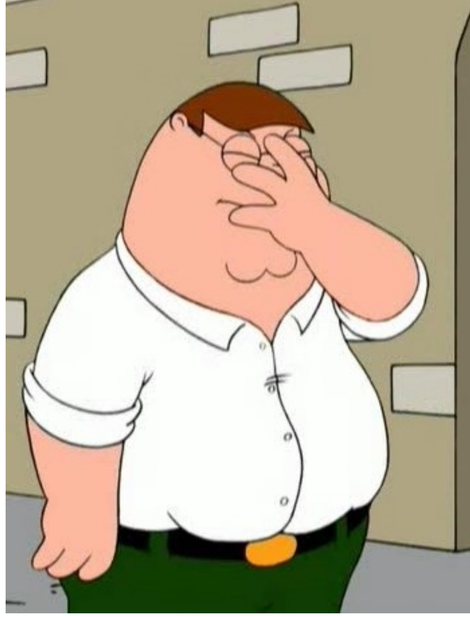
Pitfall #1. С места в карьер, нет warmup

- Есть там “steady state” тесты:

```
public static void main(String[] args){
    pidigits m = new pidigits(Integer.parseInt(args[0]));
    for (int i=0; i<65; ++i) m.pidigits(false);
    m.pidigits(true);
}
```

- Мейнтейнер CLBG даже написал мне пышущее ядом письмо:
  - “*... The JavaOne Moscow presenter's slides fail to show that combined time measurement* ***was correctly divided by 65 to give the average.***”

Pitfall #1. С места в карьер, нет warmup



- Есть там “steady state” тесты:

```
public static void main(String[] args){
    pidigits m = new pidigits(Integer.parseInt(args[0]));
    for (int i=0; i<65; ++i) m.pidigits(false);
    m.pidigits(true);
}
```

- Мейнтейнер CLBG даже написал мне пышущее ядом письмо:
  - “*... The JavaOne Moscow presenter's slides fail to show that combined time measurement* ***was correctly divided by 65 to give the average.***”
    - Ага, конечно. И тем не менее:
      - Если первая итерация будет дольше 20x раз, то среднее изменится >13%
      - Вообще говоря, усреднять метрики из *разных* режимов глупо

Pitfall #1. С места в карьер, нет warmup

Важные уроки:

Прогревайте код.

Прогрев – специальный этап в тесте.

Прогревайтесь на том же коде, и тех же данных.

Pitfall #2. Измерили посторонний эффект

- Найдено в тестах EclipseLink
  - Один из тестов измеряет скорость вызова “синхронизованных” методов
- Опубликованные данные
  - Отрицательный контроль провалился:
    - block: 8244087 usec
    - method: 13383707 usec
  - Вывод:
    - “Используйте synchronized(this)”
  - JVM-инженеры:
    - “WTF, эти два метода эквивалентны”

```
public class SynchTest {
    int i;

    @GenerateMicroBenchmark
    void testSynchInner() {
        synchronized (this) {
            i++;
        }
    }

    @GenerateMicroBenchmark
    synchronized void testSynchOuter() {
        i++;
    }
}
```

Pitfall #2. Измерили посторонний эффект

- Воспроизводим руками
  - Сделаем много вызовов каждого теста
    - Сначала 10 секунд synchInner
    - Потом 10 секунд synchOuter
    - 1 поток, монитор всегда свободен
  - Результаты (ops/sec):
    - synchInner:

      40988 ± 218
    - synchOuter:

      261602 ± 11511

```
public class SynchTest {
    int i;

    @GenerateMicroBenchmark
    void testSynchInner() {
        synchronized (this) {
            i++;
        }
    }

    @GenerateMicroBenchmark
    synchronized void testSynchOuter() {
        i++;
    }
}
```

Pitfall #2. Измерили посторонний эффект

- Решение
  - BiasedLocking включается не сразу
    - -XX:BiasedLockingStartupDelay=0
  - Результаты (ops/sec):
    - synchInner:
      287905 ± 12298
    - synchOuter:
      286962 ± 10114
- Старт JVM – переходный процесс
  - Измерения сразу после старта
    - Поменяли порядок тестов – противоположный результат

```
public class SynchTest {
    int i;

    @GenerateMicroBenchmark
    void testSynchInner() {
        synchronized (this) {
            i++;
        }
    }

    @GenerateMicroBenchmark
    synchronized void testSynchOuter() {
        i++;
    }
}
```

Pitfall #2. Измерили посторонний эффект

Важные уроки:

Любой результат должен быть объяснён.

Неожиданные результаты вскрывают неучтённую систематику.

Покажите ваши данные специалистам.

Pitfall #3. Странные режимы

- Не так давно в каком-то блоге:

“

Today I want to show how you could compare e.g. different algorithms.
You could simply do:

```
int COUNT = 1000000;
long firstMillis = System.currentTimeMillis();
for(int i = 0; i < COUNT; i++) {
  runAlgorithm();
}
System.out.println(System.currentTimeMillis()-firstMillis) / COUNT);
```

There are several problems with this approach, which results in very unpredictable results: [...]

You should turn off the JIT-compiler with specifying -Xint as a JVM option, otherwise your outcome could depend on the (unpredictable) JIT and not on your algorithm. You should start with enough memory, so specify e.g.-Xms64m -Xmx64m, because JVM memory allocation could get time consuming. Avoid that the garbage collector runs while your measurement: -Xnoclassgc

”

Pitfall #3. Странные режимы

- Не так давно в каком-то блоге:

> Today I want to show how you could compare e.g. different algorithms.
> You could simply do:
>
> ```
> int COUNT = 1000000;
> long firstMillis = System.currentTimeMillis();
> for(int i = 0; i < COUNT; i++) {
>   runAlgorithm();
> }
> System.out.println(System.currentTimeMillis()-firstMillis) / COUNT);
> ```
>
> There are several problems with this approach, which results in very unpredictable results: [...]
>
> **You should turn off the JIT-compiler with specifying -Xint as a JVM option, otherwise your outcome could depend on the (unpredictable) JIT and not on your algorithm.** You should start with enough memory, so specify e.g.-Xms64m -Xmx64m, because JVM memory allocation could get time consuming. Avoid that the garbage collector runs while your measurement: -Xnoclassgc

JavaOne

ORACLE

Pitfall #3. Странные режимы

- С каждой итерацией **bitset** всё рос
  - скорость всё падала и падала
  - списали всё на “непредсказуемость” JIT'а
- Выключили JIT
  - скорость остального кода резко упала
  - относительное замедление из-за растущего **bitset** на фоне общего замедления стало меньшим
  - “Ура, интерпретатор более предсказуем!”
- Написали в блог про “открытие”
  - В конце концов, заглянули в профиль и увидели там растущий BitSet

```
public class Test {
    private static BitSet bitset;

    private void doWork() {
        // do work
        bitset.set(n, r);
        // do more work
    }

    public void doIteration() {
        doWork();
    }
}
```

Pitfall #3. Странные режимы

Важные уроки:

Тестируйте в режимах, близких к боевым.

Отчётливо представляйте, *зачем* вы включаете другой режим.

Pitfall #4. Dead-code elimination

- Умные современные JIT-компиляторы...
  - Делают constant propagation and dead-code elimination автоматически
  - А совсем умные компиляторы (вроде C2 в HotSpot) могут найти “ненужные” блоки
    - Результат вычисления не используется → вычисление не делать
    - Что если этот “ненужный” блок как раз и измеряется?
- Довольно просто бороться
  - Обеспечить side-effect
    - Результат вывести через I/O
    - Сохранить результат в public-поле
    - Бросать exception по какому-нибудь нетривиальному невероятному результату
- Обычные симптомы
  - Ультра-быстрое “вычисление”

ORACLE®

Pitfall #4. Dead-code elimination

- Типичный пример
  - основан на тестах из Apache Commons Math
- Результат
  - StrictMath: 71 msecs
  - FastMath: 39 msecs
  - Math: 0 msecs
- Что случилось?
  - “x” не используется
  - компилятор заключил, что у Math.log(...) нет сайд-эффектов, и удалил весь цикл
  - Простая печать “x” сняла этот эффект:
    - Math: 21 msecs

```
@Test
public void testLog() {
  double x = 0;
  long time = System.nanoTime();
  for (int i = 0; i < RUNS; i++)
      x += StrictMath.log(Math.PI + i);
  long strictMath = System.nT() - time;

  x = 0;
  time = System.nanoTime();
  for (int i = 0; i < RUNS; i++)
      x += FastMath.log(Math.PI + i);
  long fastTime = System.nT() - time;

  x = 0;
  time = System.nanoTime();
  for (int i = 0; i < RUNS; i++)
      x += Math.log(Math.PI + i);
  long mathTime = System.nT() - time;
```

Pitfall #4. Dead-code elimination

Важные уроки:

Компьютеры ленивы.

Программы могут пропустить некоторые куски кода,
если результат никому не нужен.

Pitfall #5. (Не)сравнимые эксперименты

- Представим, что надо протестировать коллекцию
  - Скажем, это внутренне-синхронизованный Map
  - Протестировать доступ из одного и нескольких потоков
- Два теста
  - Одна коллекция на все потоки, #CPU = #Threads
    - Проверяем скалируемость на одновременном доступе
  - По одной коллекции на каждый поток, #CPU = #Threads
    - Проверяем скалируемость в эксклюзивном доступе
    - Можно было бы запустить только один поток?
- Тесты должны быть сравнимы
  - Одинаковое количество потоков отлично нагружает систему

Pitfall #5. (Не)сравнимые эксперименты

- Первое измерение:
  - 4 потока
  - Эксклюзивный доступ:

    615 ± 12 ops/sec
  - Общий доступ:

    828 ± 21 ops/sec
- Общий доступ быстрее?
  - Контринтуитивный результат

```
public class TreeMapTest {
    List<String> keys = new ArrayList<>();
    Map<String, String> map = Factory.<>newMap();

    @GenerateMicroBenchmark
    public void test() {
        for(String key : keys) {
            String value = map.get(key);
            if (!value.equals(key)) {
                throw new ISE("Violated");
            }
        }
    }
}
```

Pitfall #5. (Не)сравнимые эксперименты

- Привет, память
  - Средний размер коллекции ~250 Kb
  - L2 cache = 256 Kb
  - L3 cache = 3072 Kb
- На самом деле, тесты не сравнимы:
  - Эксклюзивный:
    - 4 threads x 250 Kb = 1000 Kb
  - Общий
    - 1 thread x 250 Kb = 250 Kb
- Разные условия запуска!

	Эксклюзивный	Общий
1 поток	314 ± 14	296 ± 11
2 потока	561 ± 21	554 ± 12
4 потока	615 ± 12	828 ± 21
8 потока	598 ± 15	815 ± 15
16 потока	595 ± 12	829 ± 16
32 потока	644 ± 43	915 ± 34

Pitfall #5. (Не)сравнимые эксперименты

Важные уроки:

Осторожнее со сравнениями!

(Оказывается, очень редко можно сравнивать данные между собой)

Pitfall #6. Мультиязыковые бенчмарки

- Чем больше объекты исследований, тем больше тонких отличий
  - Различия в алгоритмах, библиотеках, настройках и поддержке оборудования
  - Вывод: сравнивать большие платформы – мрак и ужас
- Сравнение $lang1 vs. $lang2
  - Лучший способ подлить масла в огонь священных войн
    - Всё вокруг неправильного вопроса: “Что быстрее: $lang1 или $lang2?”
  - Умные люди используют эту возможность для самообучения
    - Правильный вопрос: “**ПОЧЕМУ** $program1, работающая поверх $impl1 языка $lang1, быстрее/медленее $program2, работающей поверх $impl2 языка $lang2, на $hardware?”
    - Чаще всего требует глубокого понимания и $impl1, и $impl2

Pitfall #6. Мультиязыковые бенчмарки

Важные уроки:

Перестаньте тратить на них время.

Полезно лишь для понимания
“Почему производительность отличается?”

Pitfall #7. Накладные расходы на инфраструктуру

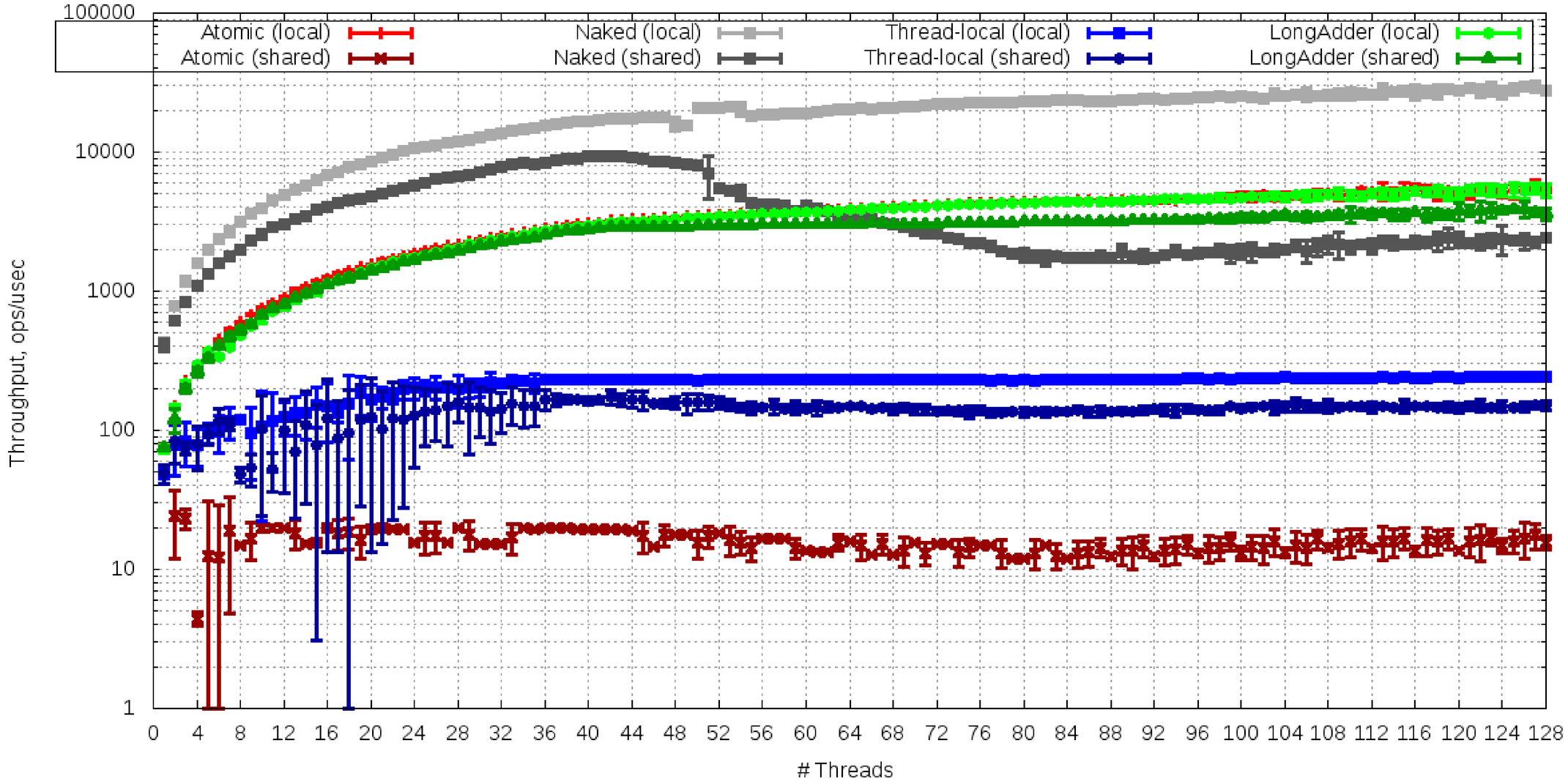
- Обслуживающий измерение код тоже исполняется
  - Для микробенчмарок размер инфраструктуры сравним с размером самого теста
  - Очень, очень, ОЧЕНЬ просто выстрелить себе в ногу
- Какие накладные расходы оказывает инфраструктура?
  - Считает операции
    - Мнимые блокировки, расходы на счётчики, и т.п.
  - Измеряет время
    - Ничего плохого, если не очень часто
    - Но некоторые бенчмарки ведут себя как Сумасшедший Шляпник: спрашивают время *миллионы* раз в секунду
  - Делает I/O
    - Непредсказуемо гадит в кеш, и т.п.

Pitfall #7. Накладные расходы на инфраструктуру

- Представим простой тест
  - Вызываем сложную функцию
  - Метрика: вызовы test() в секунду
  - Допустим, что мы НЕ знаем, в скольки потоках увеличивается счётчик
- Считаем количество вызовов:
  1. Простой счётчик
  2. Атомарный счётчик
  3. jsr166e LongAdder
  4. Тред-локальный счётчик

```
public class MyBrokenBenchmark {

    private long counter;

    private AtomicLong atomicCounter;

    private LongAdder longAdderCounter;

    private ThreadLocal<Long> tlCounter =
        new ThreadLocal<Long>();

    public void test() {
        // { increment counter here }
        // do work here
    }
}
```

Pitfall #7. Накладные расходы на инфраструктуру



Intel Xeon (Westmere-EP), 3.0 Ghz, 2x6x2 = 24 HW threads, SLES 11 x86_64, JDK 6u25

Pitfall #7. Накладные расходы на инфраструктуру

Важные уроки:

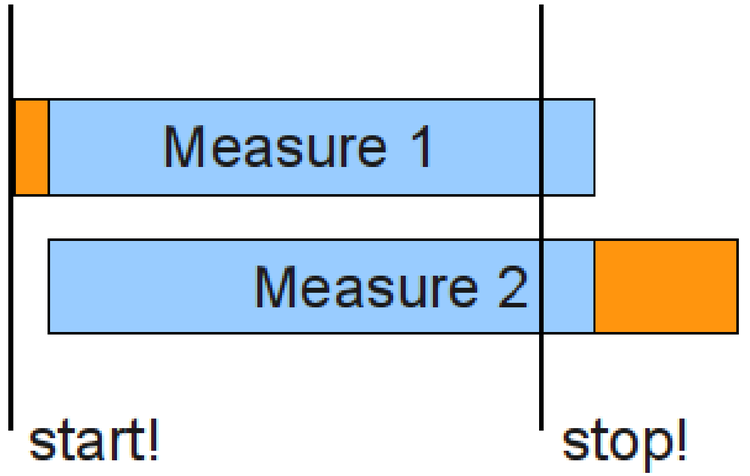
Ваша тестовая инфраструктура – ваш друг.

Иногда она же – ваш злейший враг.

Всегда подозревайте инфраструктуру.

Pitfall #8. Шедулинг

- Потоки не детерминированы
  - Время старта/останова потоков “дрожит”
  - Даже парковки на барьерах не гарантируют мелких отличий
- Представим, что N потоков делают одинаковое количество работы
  - Потоки завершатся в одно и то же время? Нет.
    - Другим потокам предоставится возможность работать в привилегированных условиях
      - “Размазывает” результаты
      - Даёт необоснованно завышенные показатели
    - Один из основных источников “дрожания”

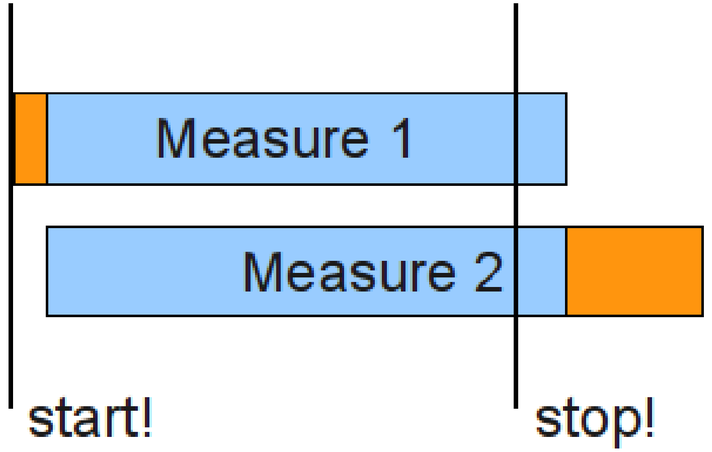


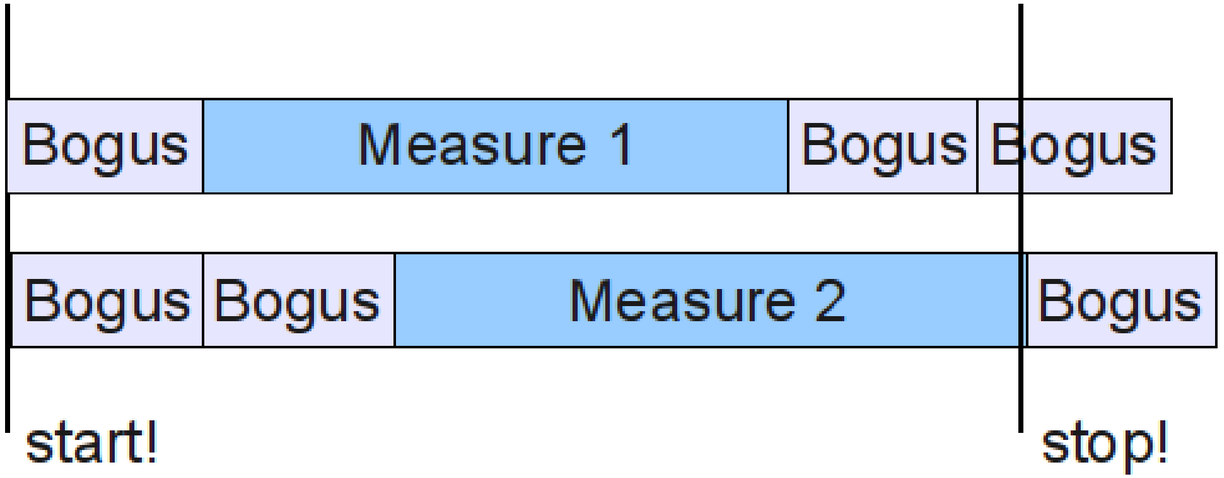
Pitfall #8. Шедулинг

- Возьмём обычный тест
  - 2x6x2 = 24 HW-потока
  - 24 Java-потока
- Оценим эффект
  - Без синхронизации потоков

    957 ± 32 ops/min
  - С синхронизацией потоков

    821 ± 5 ops/min
- Выводы
  - Куда более стабильный рез-т





Pitfall #8. Шедулинг

Важные уроки:

Большие шумы шепчут о проблемах с экспериментом.

Чаще всего признак того, что кто-то принимает критические решения у вас за спиной.

Pitfall #9. Планы компиляции

- Адаптивность рантайма
  - Играет на руку реальным приложениям
  - Играет против бенчмаркинга
- Задержка ре-адаптации
  - Оптимизироваться для одного случая
  - Запустить следующий тест
  - Работать в неоптимальных условиях
- Иногда консервативность рантайма зашкаливает
  - Вся история об “ошибках молодости” сохраняется

```
public class CompilePlanTest {
    Counter counter1 = new CounterImpl1();
    Counter counter2 = new CounterImpl2();

    @GenerateMicroBenchmark
    public void testM1() {
        test(counter1);
    }

    @GenerateMicroBenchmark
    public void testM2() {
        test(counter2);
    }

    public void test(Counter counter) {
        for(int c = 0; c < LIMIT; c++) {
            counter.inc();
        }
    }
}
```

Pitfall #9. Планы компиляции

- Результаты исполнения подряд:
  - testM1: 394 ± 7 ops/msec
  - testM2: 11 ± 1 ops/msec
- Результаты исполнения отдельно:
  - testM1: 396 ± 3 ops/msec
  - testM2: 381 ± 16 ops/msec
- Смешиваете тесты?
  - Будьте готовы, что профили тоже смешаются

```
public class CompilePlanTest {
    Counter counter1 = new CounterImpl1();
    Counter counter2 = new CounterImpl2();

    @GenerateMicroBenchmark
    public void testM1() {
        test(counter1);
    }

    @GenerateMicroBenchmark
    public void testM2() {
        test(counter2);
    }

    public void test(Counter counter) {
        for(int c = 0; c < LIMIT; c++) {
            counter.inc();
        }
    }
}
```

Pitfall #9. Планы компиляции

Важные уроки:

Прогревайте всё сразу, если интересуют средние метрики.

Прогревайте по отдельности, если интересуют пиковые метрики.

Стартуйте свежие JVM между тестами.

Agenda

Введение

Общая теория

Java Benchmarking

Инструменты

JavaOne™

ORACLE®

Иструменты

- Мозг
  - Плагин “данунеможетбыть” для проверок и перепроверок фактов
  - Плагин “щапридумаем” для построения гипотез и способов их проверки
  - Плагин “чётоянепонял” для проверки консистентности гипотез
  - Плагин “ядурак” для лёгкого отвержения ложных гипотез
- Руки
  - Прямого профиля, для постановки аккуратных экспериментов
  - Сильного профиля, для обработки тонн экспериментальных данных
- Язык, уши, глаза и прочее I/O
  - Для обмена результатами и peer review
  - Для доступа к предыдущим экспериментам

Прочие инструменты

- Профили приложений
  - VisualVM, JRockit Mission Control, Sun Studio Performance Analyzer
- Профили системы
  - top, vmstat, mpstat, iostat, dtrace, strace
- Профили JRE
  - -XX:+PrintCompilation, -verbose:gc, -verbose:class
- Дизассемблеры
  - http://wikis.sun.com/display/HotSpotInternals/PrintAssembly
- Системные счётчики
  - Sun Studio Performance Analyzer, oprofile


JavaOne™

The preceding 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.