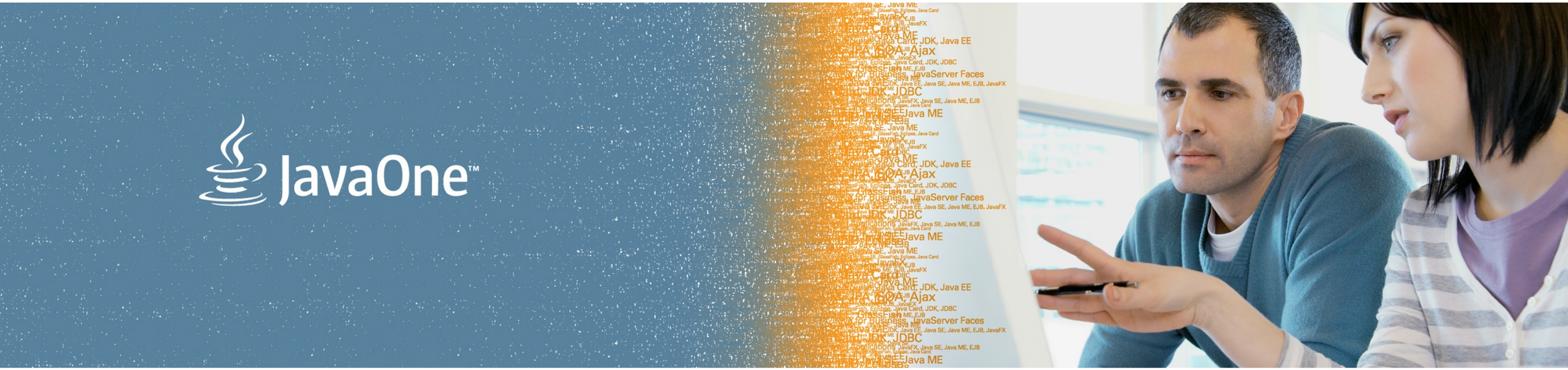

JavaOne™



ORACLE®

(The Art of) (Java) Benchmarking

Aleksey Shipilev
Java Platform Performance

Введение

JavaOne™

ORACLE®

Кто-то из мудрых

“Есть легенда, что в тот момент, когда мы построим полную картину происходящего, мир тут же исчезнет и на его месте появится что-то абсолютно непознаваемое.

Кое-кто считает, что это уже произошло.”

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

Про научный метод

- Два базовых течения
  - Теоретический
    - Построение системы знаний, проверка предсказательной силы
    - “Насколько выводы из теории соотносятся с экспериментом?”
  - Эмпирический
    - Исследование связей между феноменами
    - “Как теоретически объяснить практически происходящее?”
- Основной урок: истина vs. предубеждение
  - Научный метод как способ приблизиться к истине
  - Истина может быть установлена только наблюдением или опытом

Бенчмарки

- Что это такое?
  - Def.: “Программа, используемая для измерения производительности”
    - Жило-было приложение, добавили измерение времени – бац – уже бенчмарк
  - Каждый запуск бенчмарка – вычислительный эксперимент
    - Чистая эмпирика
    - Рвёт мозг математикам, физики хихикают
- Типичные требования к бенчмаркам
  - Результат запуска: значение некоторой метрики
  - Должен быть объективным (“test the right thing”)
  - Должен быть надёжным (воспроизводимым)

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 у́же область исследования, тем бóльше нужно знать
  - Написанный как попало “дикий” бенчмарк бесполезен
    - Никто не гарантирует, что “tests the right thing”
  - Наивные бенчмаркеры объект исследований не понимают
    - Да-да, это можешь быть и ты, *%username%*!
- Выводы:
  - Большинство “диких” бенчмарок изначально бесполезны
    - Рано кипятиться, примеры чуть позже
  - Хотите писать релевантные бенчмарки? Это сложно, но не “rocket science”
    - Добивайтесь понимания того, что вы делаете
    - Пробуйте, пробуйте, и ещё раз пробуйте

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

Теория

JavaOne™

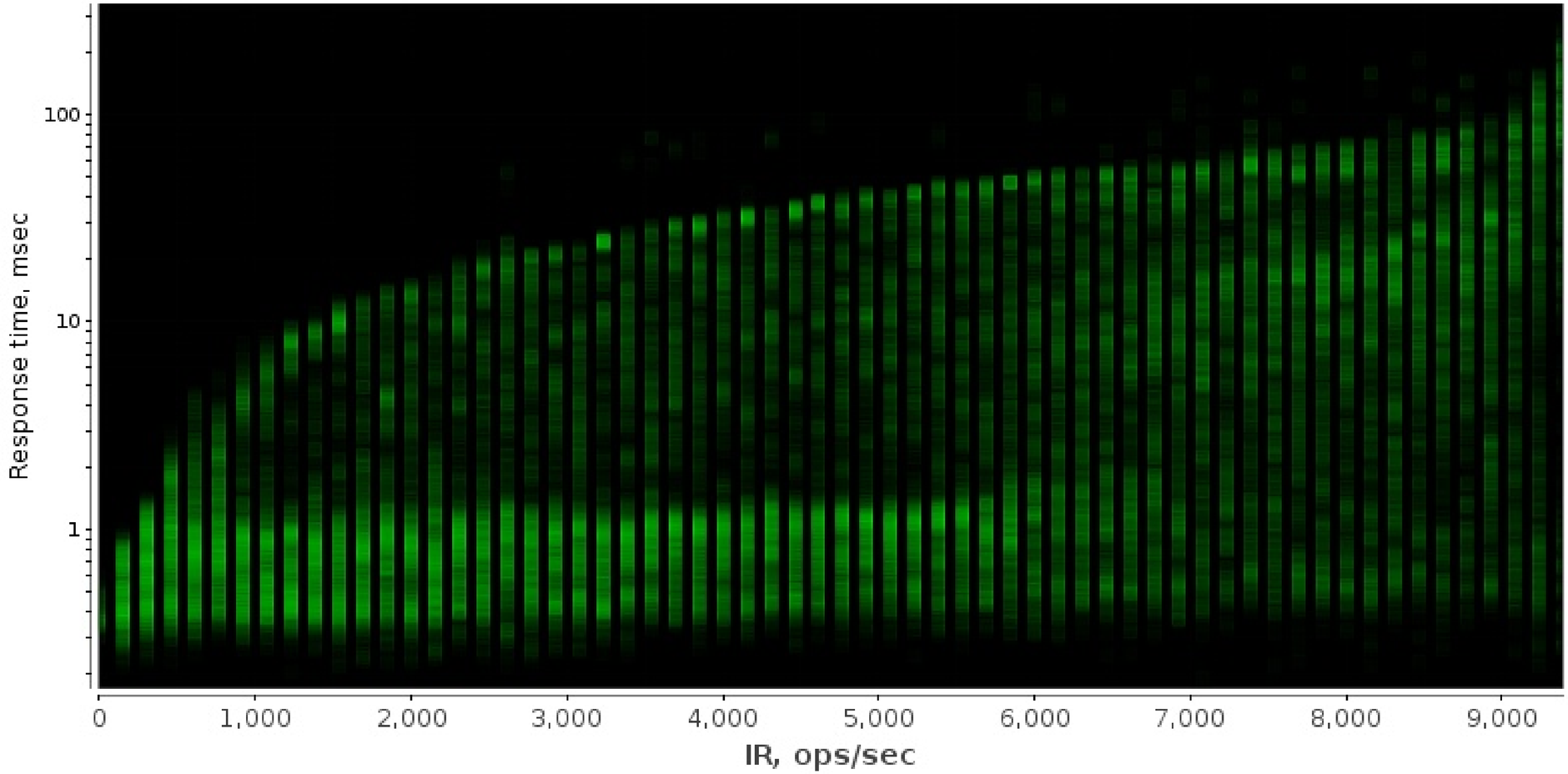
ORACLE®

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
- Чаще улучшают λ:
  - DDR2 PC2-3200: 3200 Mb/sec, CL 4ns
  - DDR2 PC2-6400: 6400 Mb/sec, CL 5ns
  - DDR3 PC3-6400: 6400 Mb/sec, CL 5ns
  - DDR3 PC3-17000: 17000 Mb/sec, CL 15ns
- При прочих равных: $\tau \sim \exp(\lambda)$

Метрики

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



JavaOne™

ORACLE®

Composability

- Предположим, есть функциональные блоки A и B
  - Будем исполнять их в двух сценариях: последовательно и параллельно
- Общая функциональность:
  - F(A) ⊥ F(B) = F(A) || F(B)
  - “Black Box”: F(A) не зависит от F(B)
- Общая производительность:
  - P(A) ⊥ P(B) **?** P(A) || P(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 подвержены случайностям
  - Боремся репликацией опытов
    - Много запусков, статистические оценки
    - Запуски в разных условиях
- Контроль!
  - Положительный
    - Проверяем влияние существенных факторов
    - Зависит ли результат от проверяемых параметров?
  - Отрицательный
    - Проверяем влияние несущественных факторов
    - Зависит ли результат от несущественных параметров?

При чём тут мы?

JavaOne™

ORACLE®

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.

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

Сам бенчмарк

- Throughput-based vs. time-based
  - Что и как измеряете?
- Инфраструктура
  - Накладные расходы на поддержку
  - (поговорим на примерах)
- Жизненный цикл
  - initialization, warmup, measurement
  - (тоже поговорим)
- Ожидаемые результаты
  - В т.ч. ожидаемая реакция на изменения параметров

При чём тут Java?

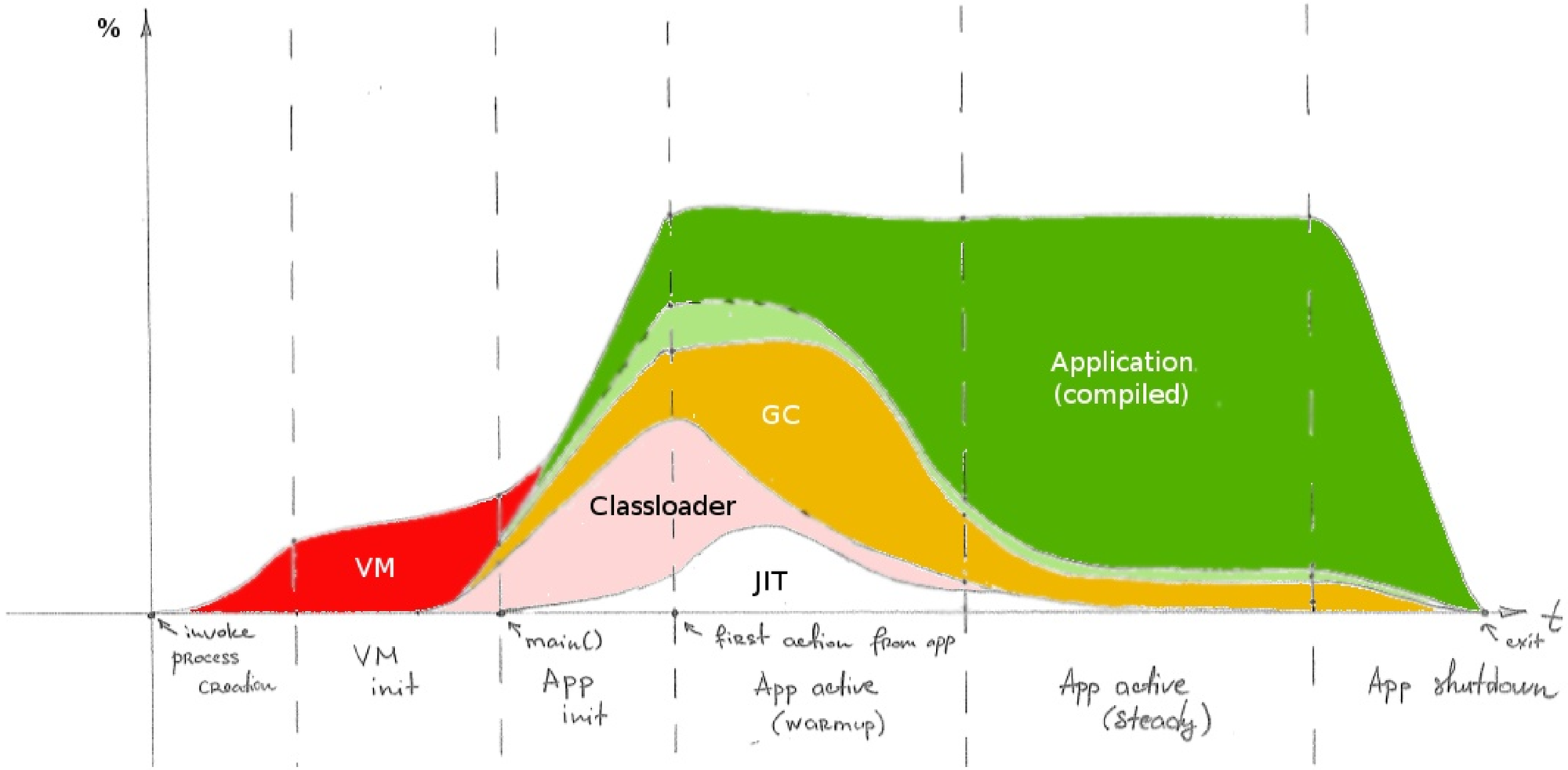
JavaOne™

ORACLE®

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. Объект исследований

- 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 #1. С места в карьер, нет warmup

- CLBG делает ставку на две метрики
  - execution time, время выполнения программы
    - натурально, “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
  - CLBG поощряет написание бенчмарков с минимальными расходами на инициализацию
    - Например, примотать нативный GMP!
    - Заставляет фокусироваться на I/O исходных данных и результатов

Pitfall #1. С места в карьер, нет warmup

- В CVS-истории CLBG видны попытки эту проблему решить

```
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);
}
```

- “Прогрелось”. Но работает минимум в 65x медленнее
  - Поэтому супер-оптимизация:

Pitfall #1. С места в карьер, нет warmup

- В CVS-истории CLBG видны попытки эту проблему решить

```
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);
}
```

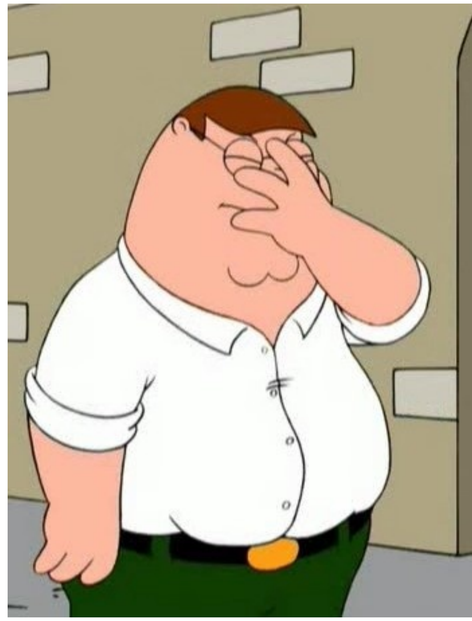
- “Прогрелось”. Но работает минимум в 65x медленнее
  - Поэтому супер-оптимизация:

```
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

- В CVS-истории CLBG видны попытки эту проблему решить

```
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);
}
```



- “Прогрелось”. Но работает минимум в 65x медленнее
  - Поэтому супер-оптимизация:

```
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

- Отечественный производитель:
  - Пишет оптимизированную библиотеку для линейной алгебры
    - Написал микробенчмарк для умножения матриц
    - Имхо, поторопился, надо было интегрироваться в существующие тесты
  - Java: HotSpot(TM) Client VM (build 17.0-b17, mixed mode, sharing)
    - Потерял кучу агрессивных оптимизаций, сделанных в -server
    - Зато написал их в своём коде, и героически “разогнал” библиотеку
  - Померил время на исполнение по пяти запускам, взял лучшее время

```
t1 = System.currentTimeMillis();
routine();
t2 = System.currentTimeMillis();
t3 = t2 – t1;
```

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:

      40988.04 ± 218.95
    - synchOuter:

      261602.51 ± 11511.81

```
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:
      287905.52 ± 12298.57
    - synchOuter:
      286962.10 ± 10114.94
- 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 #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

”

Pitfall #3. Странные режимы

- С каждой итерацией **bitset** всё рос
  - скорость всё падала и падала
  - списали всё на “непредсказуемость” JIT'a
- 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 #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 #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<>();
    SortedMap<String, String> map = new STreeMap<>();

    @Setup
    public void setup() {
        // populate keys and map
    }

    @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 #6. Совсем не сравнимые эксперименты

- Чем больше объекты исследований, тем больше тонких отличий
  - Различия в алгоритмах, библиотеках, настройках и поддержке оборудования
  - Вывод: сравнивать большие платформы – мрак и ужас
- Типичный пример: сравнение .Net vs. Java на самописных деревьях
  - Набор данных опирается на ГПСЧ, специфичные для платформ
  - Измеряем структурно *разные* деревья
- Особые дисциплины Специальной олимпиады
  - “Мой C++ быстрее Java”
  - “Моя Java быстрее C++”
  - “Мой Haskell вас всех … лучше”
  - “А у Erlang синхронизация дешевле”

Pitfall #7. Накладные расходы на инфраструктуру

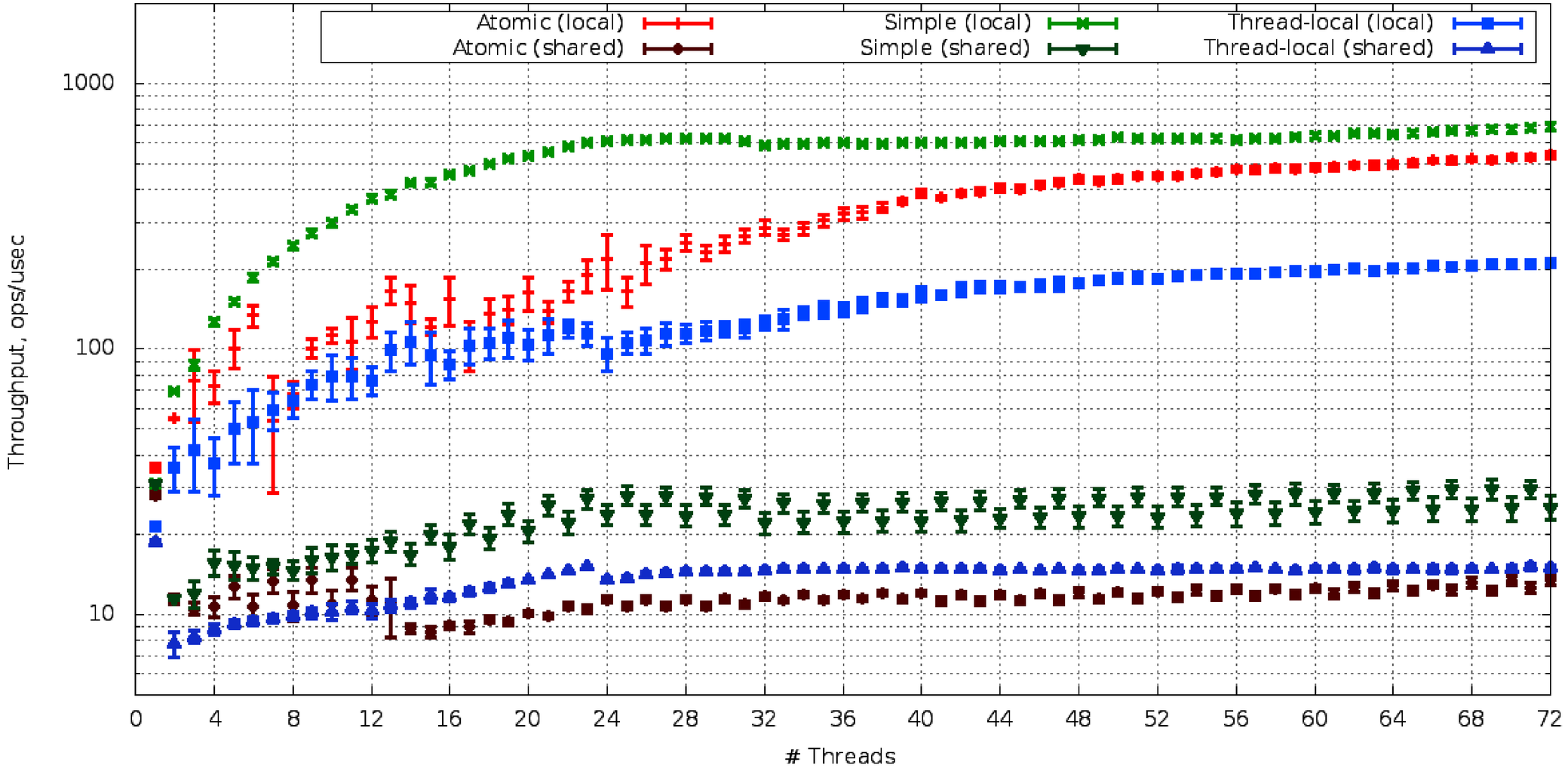
- Обслуживающий измерение код тоже исполняется
  - Для микробенчмарок размер инфраструктуры сравним с размером самого теста
  - Очень, очень, ОЧЕНЬ просто выстрелить себе в ногу
- Какие накладные расходы оказывает инфраструктура?
  - Считает операции
    - Мнимые блокировки, расходы на счётчики, и т.п.
  - Измеряет время
    - В хорошем случае trap, в плохом случае syscall
  - Делает I/O
    - Непредсказуемо гадит в кеш, и т.п.

Pitfall #7. Накладные расходы на инфраструктуру

- Представим простой тест
  - Вызываем сложную функцию
  - Метрика: количество вызовов test() за единицу времени
- Нужно подсчитать количество вызовов
  - Простой счётчик
  - Атомарный счётчик
  - Тред-локальный счётчик

```
public class SinTest {

    private AtomicInteger atomicCounter;

    private int counter;

    private ThreadLocal<Integer> tlCounter =
        new ThreadLocal<Integer>() {
            @Override
            public Integer initialValue() {
                return 0;
            }
        };

    public double test(double a) {
        // { increment counter here }
        return Math.sin(a);
    }
}
```

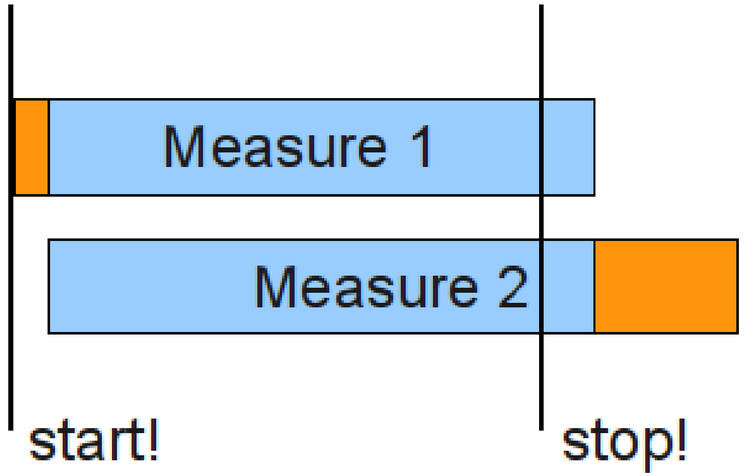
Pitfall #7. Накладные расходы на инфраструктуру



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

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”

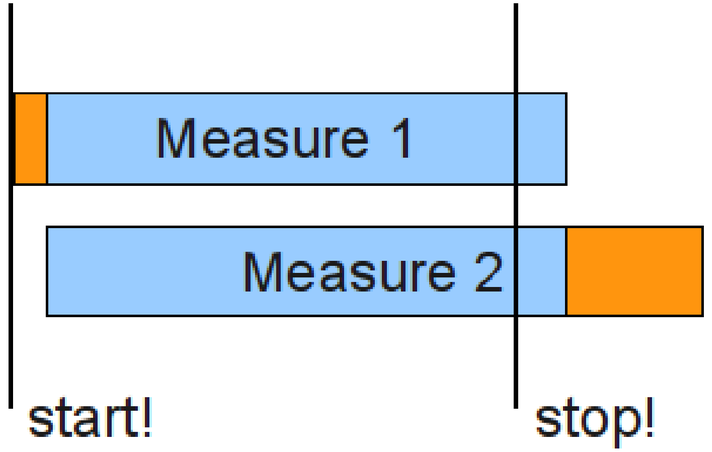


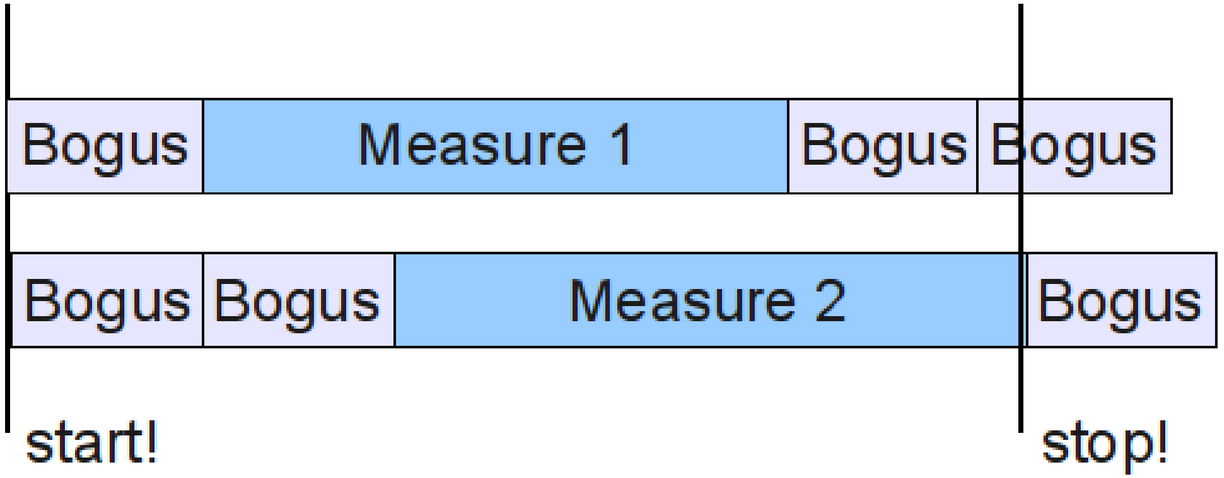
Pitfall #8. Шедулинг

- 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 #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();
        }
    }
}
```

Что делать?

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.