ORACLE®

Java Platform Performance BoF

Aleksey Shipilev, Sergey Kuksenko
Java Platform Performance, Oracle

Java Platform Performance BoF @ JEE Conf

о чём у нас

- **Отвечаем на вопросы**
  - В основном, предварительно собранные в разных местах Рунета
  - Есть возможность задать вопрос прямо здесь
    - Пишем на бумажке, передаём вперёд ;)
- **Здесь и сейчас:**
  - Performance Engineering
  - Benchmarking
  - JIT
  - Concurrency and synchronization
  - Startup
- **Не здесь или не сейчас, а на других сессиях:**
  - “Диагностика проблем и настройка GC”

Performance Engineering

Performance Engineering

абстрактно и отлично об отличиях в абстракциях

- **Computer Science → Software Engineering**
  - Строим приложения по функциональным требованиям
  - В большой степени абстрактно, в “идеальном мире”
    - Теоретически неограниченная свобода – искусство!
    - Можно строить воздушные зАмки
  - Рассуждения при помощи формальных методов

- **Software *Performance* Engineering**
  - “Real world strikes back!”
  - Исследуем взаимодействие софта с железом на типичных данных
    - Производительность уже нельзя оценить
    - Производительность можно только *измерить*
  - Естественно-научные методы
    - Основываемся на эмпирических данных

Performance Engineering

первый шаг

- **Классические ошибки первого шага**
  - “я *вижу*, что метод foo() реализован неэффективно”
  - “по профилю *видно*, что метод bar() – самый горячий и занимает 5%”
  - “*по-моему,* у нас тормозит БД, и необходимо перейти с DB_X на DB_Y”

- **Правильный первый шаг:**
  - **Необходимо выбрать метрику**
    - ops/sec, transactions/sec
    - время исполнения
    - время отклика
  - **Убедиться в корректности метрики**
    - релевантна (учитывает реальный сценарий работы приложения)
    - повторяема

ЦЕЛЬ → улучшение метрики!

Performance Engineering

анализ узких мест (tips)

- **Низкая утилизация CPU**
  - Высокая дисковая, сетевая активность
  - Конфликт блокировок
  - Конфликт ресурсов ОС
  - Слабая параллелизация приложения
- **Высокая утилизация ядра ОС**
  - Частые блокировки
  - Частое обращение к ОС
- **Высокая утилизация CPU**
  - Неоптимальная архитектура приложения
  - Неправильное использование API
  - Неоптимизированные горячие методы
  - Неоптимальные настройки GC

Performance Engineering

инструменты для анализа системы

	Solaris	Linux	Windows	Что смотрим
Сеть	netstat, dtrace	netstat	perfmon	количество соединений, объем трафика
Диск	iostat, dtrace	iostat	perfmon	количество обращений к диску, задержка
Память	vmstat, prstat, dtrace	vmstat, top	perfmon	подкачка страниц, размер памяти
Процессы	ps, vmstat, mpstat, prstat, dtrace	ps, vmstat, top	perfmon	количество нитей, состояние нитей, переключения контекста
Ядро ОС	mpstat, lockstat, plockstat, dtrace, intrstat, vmstat	vmstat	perfmon	kernel time, блокировки, системные вызовы, прерывания ...

Performance Engineering

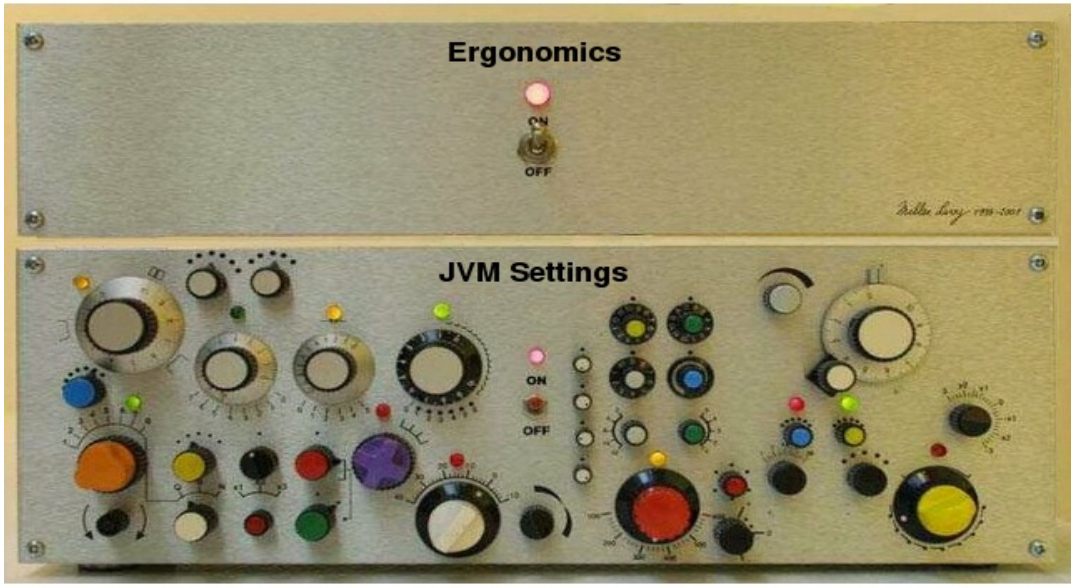
tools, tools, tools again, more tools

- VisualVM
  - http://visualvm.dev.java.net
- JRockit Mission Control
  - http://www.oracle.com/technetwork/middleware/jrockit/mission-control/index.html
- Sun Studio Analyzer
  - http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html
- NetBeans Profiler
  - http://www.netbeans.org
- DTrace
  - http://www.oracle.com/technetwork/systems/dtrace/dtrace/index.html
- Ещё могут быть полезны:
  - JProbe
  - OptimizeIt
  - YourKit

JVM tuning

настройка параметров JVM

- **Что настраивать?**
  - http://blogs.sun.com/watt/resource/jvm-options-list.html
  - http://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html



JVM tuning

настройка параметров JVM

- **JVM сама подбирает оптимальные параметры своей работы**
  - Server vs. Client
  - Large pages (Solaris)
  - CompressedOops (64-bit VM)
- **Так что же настраивать?**
  - GC/Heap tuning
  - -XX:+UseNUMA (Solaris, Linux)
  - -XX+:UseLargePages (Linux, Windows)
    - http://www.oracle.com/technetwork/java/javase/tech/largememory-jsp-137182.html
- **Не забыть**
  - Использовать последнюю версию JDK

Benchmarking

Benchmarking

на чём измерять производительность?

	“микро” бенчмарки	синтетические бенчмарки	авто-сценарии реальных приложений	реальные приложения
релевантность	плохая	средняя	хорошая	отличная
расходы на разработку и поддержку	низкие	средние	средние	высокие
расходы на запуск	низкие	низкие	средние	высокие
сложность интерпретации результатов	очень сложно	средняя	средняя	сложно

Benchmarking

наша стратегия

- **Сделать реальные приложения быстрее**
  - Но, слишком сложно поддерживать и запускать
  - Поэтому...
- **Сфокусироваться на синтетических бенчмарках и авто-сценариях**
  - SPECjbb*, SPECjvm*, SPECjAppServer*, SPECjEnterprise*
  - Glassfish, Weblogic, NetBeans, и т.п.
  - Большие in-house нагрузки
- **Пользоваться микробенчмарками с опаской**
  - Иногда от них никуда не деться
  - Очень легко написать, очень легко запустить
  - Очень легко “выстрелить себе в ногу”
  - ...что чуть ли не каждую неделю демонстрируется в профильных блогах

Benchmarking

итак, вы решили написать свой бенчмарк

- **Опасайтесь типичных ошибок**
- **Дизайн эксперимента**
  - Что хотим измерить?
  - Каким способом будем измерять?
- **Реализация эксперимента**
  - Отсутствие прогрева, “мёртвый код”, etc.
  - Многопоточность, утилизация
  - Контроль переменных: результат воспроизводится?
  - Контроль переменных: результат вообще зависит от входов?
- **Интерпретация результатов**
  - Что в итоге измерили?
    - *Почему мой бенчмарк не может работать быстрее?*
  - Значим ли результат?
    - 1000 ops/sec против 1050 ops/sec – прирост или нет?

JIT

JIT

факты

- **… есть**
- **… работает**
- **… работает хорошо**
- **… знает о железе всё:**
  - Количество и тип CPU
  - Поддерживаемые инструкции (SSEx, AVX, VIS)
  - Топологию памяти (в т.ч. размеры кэшей и их характеристики)
- **… знает о приложении много всего:**
  - Иерархию загруженных классов
  - Актуальную статистику создания объектов
  - Горячий код
  - Какие ветвления исполнялись
  - Какие значения использовались
  - Многое другое
- **… не боится использовать эти знания для компиляции**

JIT

оптимизации

- compiler tactics
  - delayed compilation
  - tiered compilation
  - on-stack replacement
  - delayed reoptimization
  - program dependence graph representation
  - static single assignment representation
- proof-based techniques
  - exact type inference
  - memory value inference
  - memory value tracking
  - constant folding
  - reassociation
  - operator strength reduction
  - null check elimination
  - type test strength reduction
  - type test elimination
  - algebraic simplification
  - common subexpression elimination
  - integer range typing
- flow-sensitive rewrites
  - conditional constant propagation
  - dominating test detection
  - flow-carried type narrowing
  - dead code elimination
- language-specific techniques
  - class hierarchy analysis
  - devirtualization
  - symbolic constant propagation
  - autobox elimination
  - escape analysis
  - lock elision
  - lock fusion
  - de-reflection
- speculative (profile-based) techniques
  - optimistic nullness assertions
  - optimistic type assertions
  - optimistic type strengthening
  - optimistic array length strengthening
  - untaken branch pruning
  - optimistic N-morphic inlining
  - branch frequency prediction
  - call frequency prediction
- memory and placement transformation
  - expression hoisting
  - expression sinking
  - redundant store elimination
  - adjacent store fusion
  - card-mark elimination
  - merge-point splitting
- loop transformations
  - loop unrolling
  - loop peeling
  - safepoint elimination
  - iteration range splitting
  - range check elimination
  - loop vectorization
- global code shaping
  - inlining (graph integration)
  - global code motion
  - heat-based code layout
  - switch balancing
  - throw inlining
- control flow graph transformation
  - local code scheduling
  - local code bundling
  - delay slot filling
  - graph-coloring register allocation
  - linear scan register allocation
  - live range splitting
  - copy coalescing
  - constant splitting
  - copy removal
  - address mode matching
  - instruction peepholing
  - DFA-based code generator

JIT

performance urban legends

final дает лучшую производительность

копируйте поля в локальные переменные!

Reflection – дорого

volatile запрещает JIT оптимизировать доступ к полю

вызов виртуального метода - дорого

избегайте get/set методов внутри самого класса

вручную вылизанный метод лучше аналога из classlib

immutable классы – плохо

native методы дорогие, System.arraycopy() нативный – значит ...

вручную выполненый inline – хорошо

Java медленна потому, что нельзя вручную выключить проверку выхода индекса за границы массива

создание объектов дорого – используйте Object pooling

JIT

как писать код

- **Используйте стандартные библиотеки**
  - Зачем писать собственный стандартный контейнер?
- **Используйте высокоуровневый API:**
  - java.util.*
  - java.util.concurrent.*
  - NIO, NIO.2
  - вообще библиотеки
- **Код должен правильным и понятным**
  - Сначала правильно
  - Потом алгоритмически “быстро”
  - Код не должен быть JIT-oriented
- **Правильно используйте возможности языка**
  - EPIC FAIL: штатная передача управления exception'ами
  - FAIL: Возврат “исключения” через return <код_ошибки>
  - FAIL: int вместо Enum или boolean
  - ...

JIT

для любопытных

Как получить ассемблерный код метода?

- Обычным дебаггером ;)
- JVMTI
- -XX:+PrintAssembly
  - http://wikis.sun.com/display/HotSpotInternals/PrintAssembly

Concurrency

Concurrency

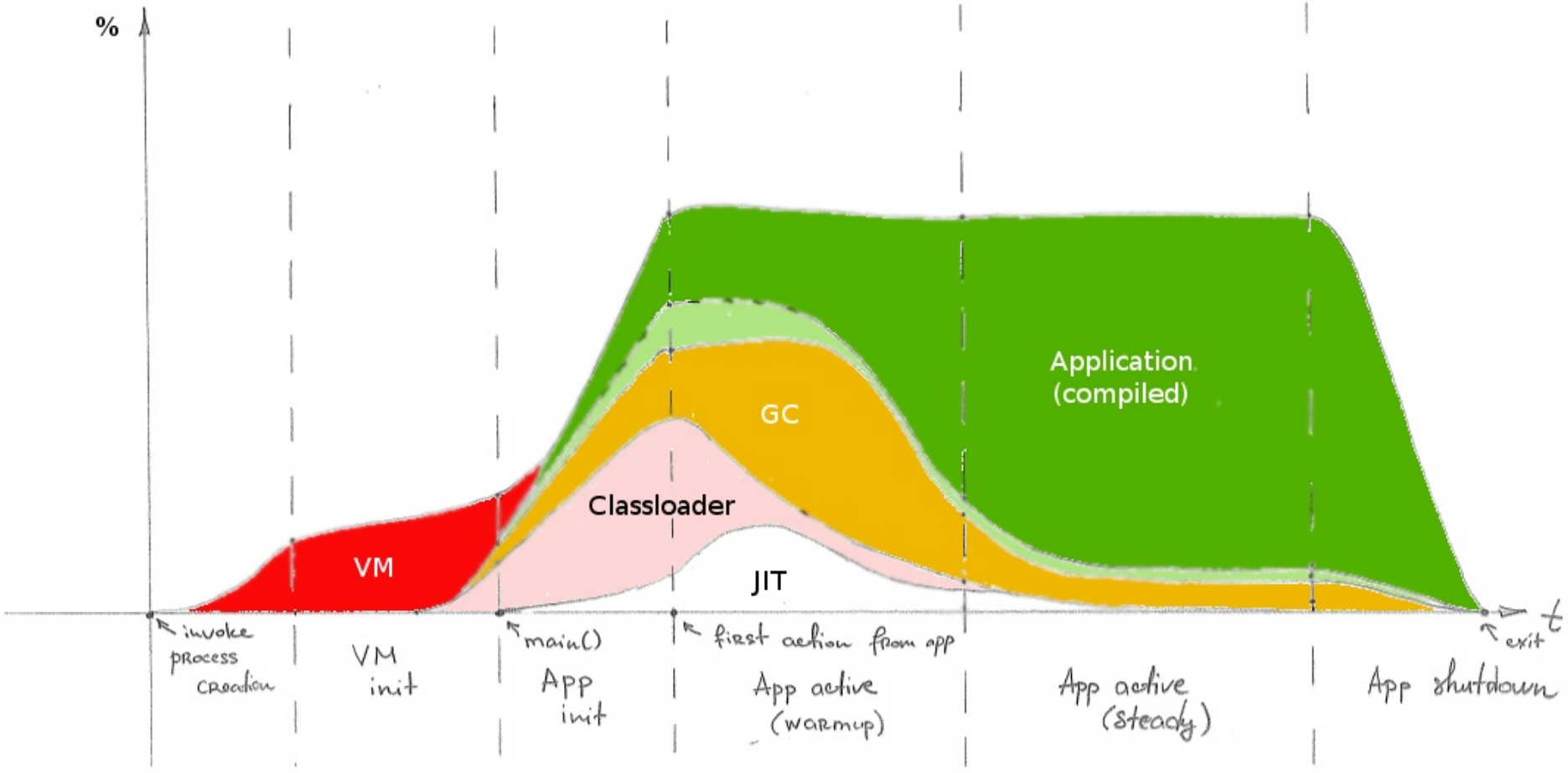
с чем его есть

- **Только мы научились программировать, новая беда**
  - Новое входное данное в программе: *время*
  - Подавляющее большинство concurrency-багов – гейзен-баги!
  - TDD “отдыхает”
- **Читаем хорошие книги**
  - “Учиться, учиться и ещё раз учиться” (с)
  - “Java Concurrency in Practice”
  - “The Art Of Multiprocessor Programming”
  - “JSR 133 Cookbook”
  - “A Little Book of Semaphores”
- **Пользуемся высокоуровневыми примитивами**
  - java.util.concurrent.*
  - … или другими, построенными на их основе
  - … при условии, что знаем, как их можно композировать без потери корректности
  - … при условии, что вообще знаем, как показать корректность
  - … при условии, что понимаем, сколько стоит тот или иной подход
  - … при условии, что обладаем 42-ухкратным запасом терпения и усидчивости

Startup

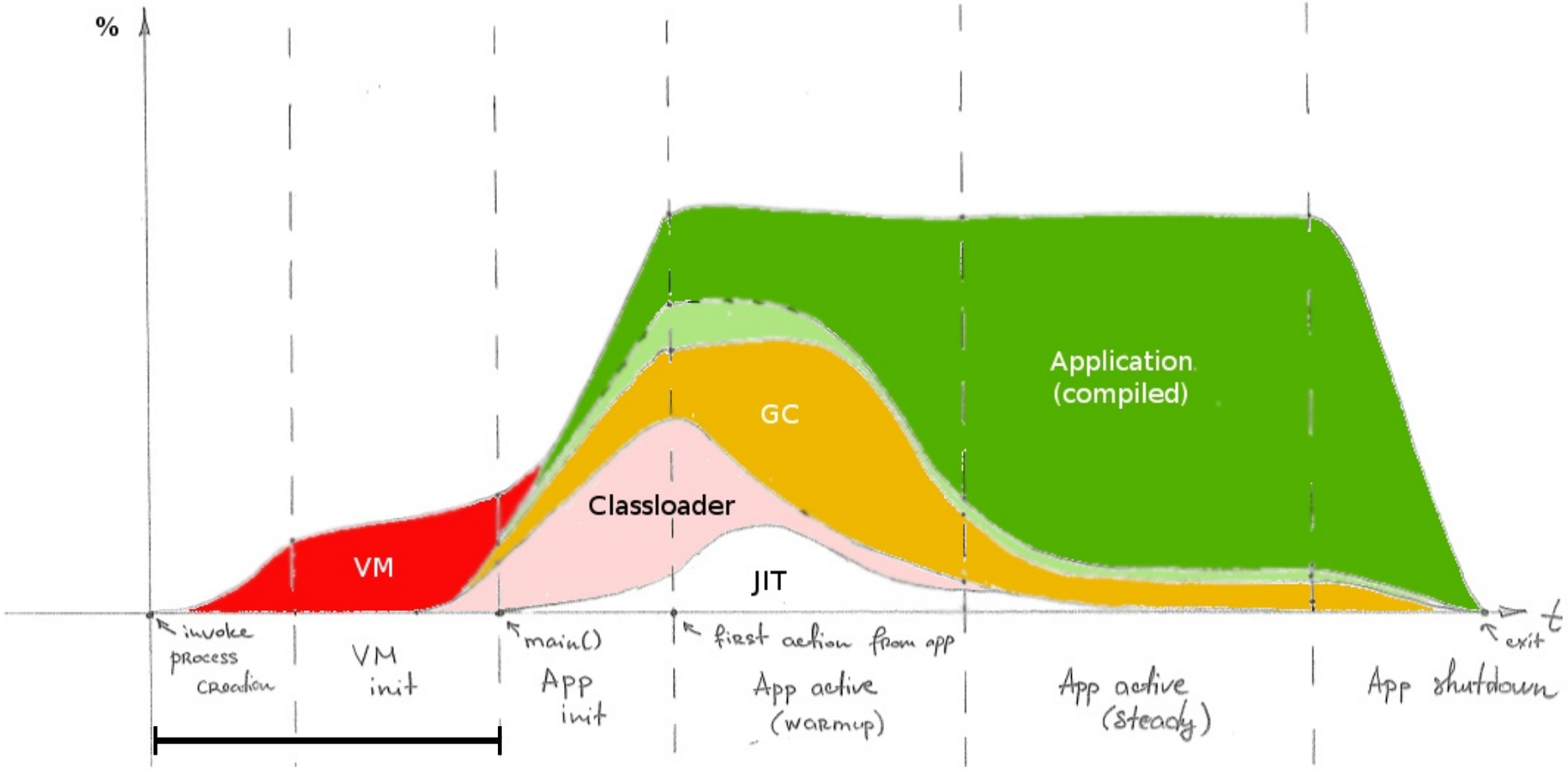
Startup

как измерять?



Startup

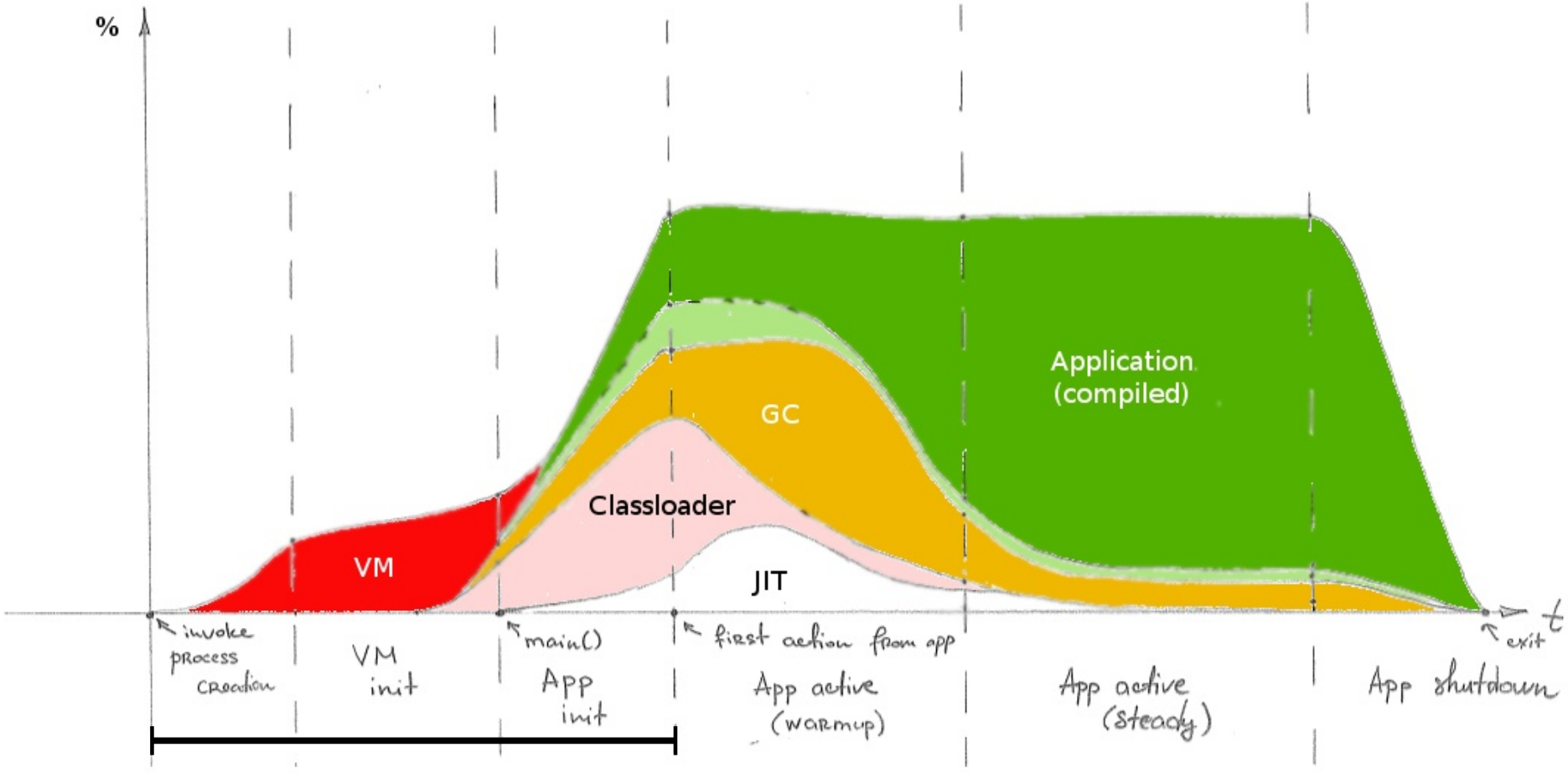
как измерять?



До входа в main()?

Startup

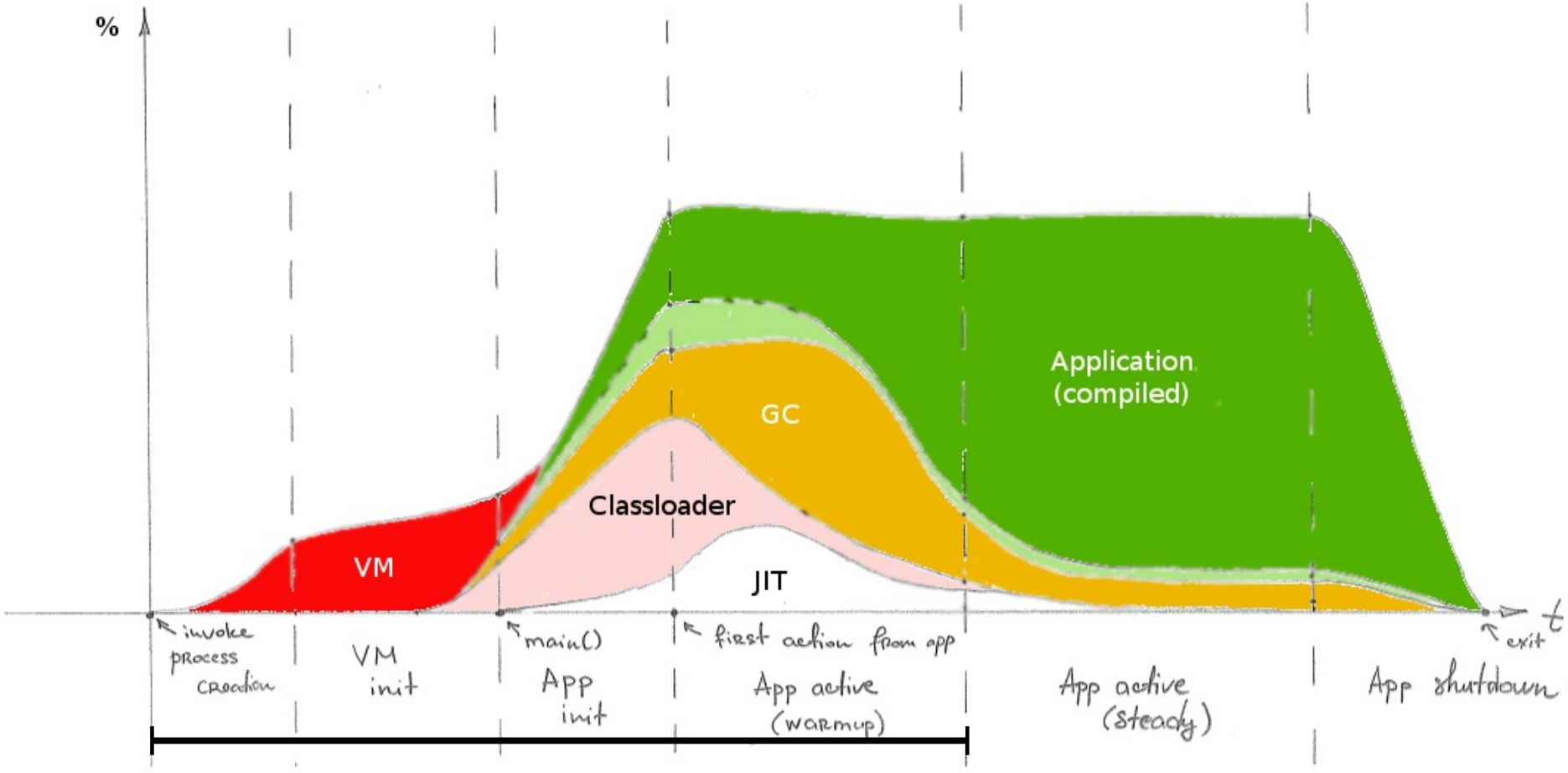
как измерять?



До первого ответа от приложения?

Startup

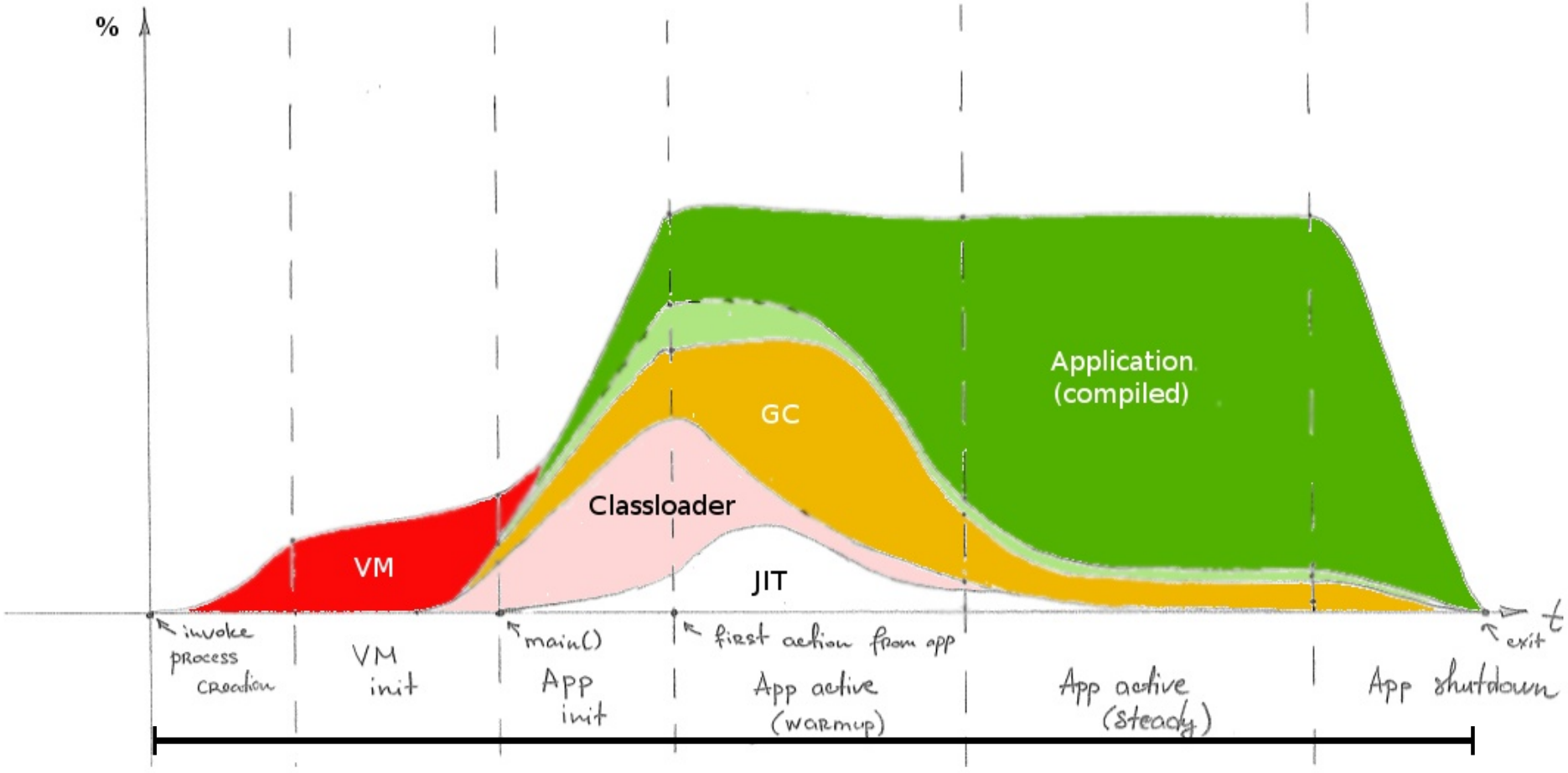
как измерять?



До “стабильного стостояния”?

Startup

как измерять?



Время на пуск-остановку?

Startup

Eclipse

- **Типичная конфигурация:**
  - 2x2 Intel i5 2.6 Ghz, Ubuntu 10.10 i686, JDK 6u25
  - Eclipse JDT (Galileo)
- **Типичные метрики:**
  - 6.000 загруженных классов
  - 1.000 методов скомпилировано
  - 512 Mb зарезервированного пространства в куче
  - 25 Mb кучи использовано после стартапа
- **Известные “проблемы”:**
  - Загрузка и верификация классов
  - JIT-компиляция

Startup

Eclipse

- **Метрика: секунд на запуск-завершение**
  - Файловые кеши прогреты, практически нулевой дисковый I/O

	default	CDS	CDS + no-verify
абсолютное время	5.83 [5.74; 5.92]	4.81 [4.73; 4.84]	4.61 [4.56; 4.74]
загрузка классов	5.01 [4.91; 5.11]	3.39 [3.12; 3.66]	3.01 [2.94; 3.08]
компиляция	0.51 [0.43; 0.59]	0.51 [0.44; 0.58]	0.51 [0.42; 0.60]

Startup

длинные приложения

- **Важно только для коротких приложений**
  - Чем дольше работает приложение, тем меньше удельные затраты на загрузку классов и компиляцию
- **Пример: 8 часа работает IntelliJ IDEA 10.x:**
  - 26.600 классов загружено
  - 5315 методов скомпилировано
- **Загрузка классов:**
  - Всего потрачено 202 с., ~0.7% общего времени
  - 10 мсек на класс
- **Компиляция:**
  - Всего потрачено 112 с., ~0.03% общего времени
  - 20 мсек на метод

What's Next

- **Послушать ещё?**
  - ~~JavaOne Moscow (апрель 2011)~~
- **Проблема со слайдами?**
  - Найдите нас на конференции
  - Напишите нам письмо
- **Проблема с JDK?**
  - http://openjdk.java.net
  - Задайте вопрос в OpenJDK
  - ...а лучше сделайте патч и дайте его в OpenJDK на review

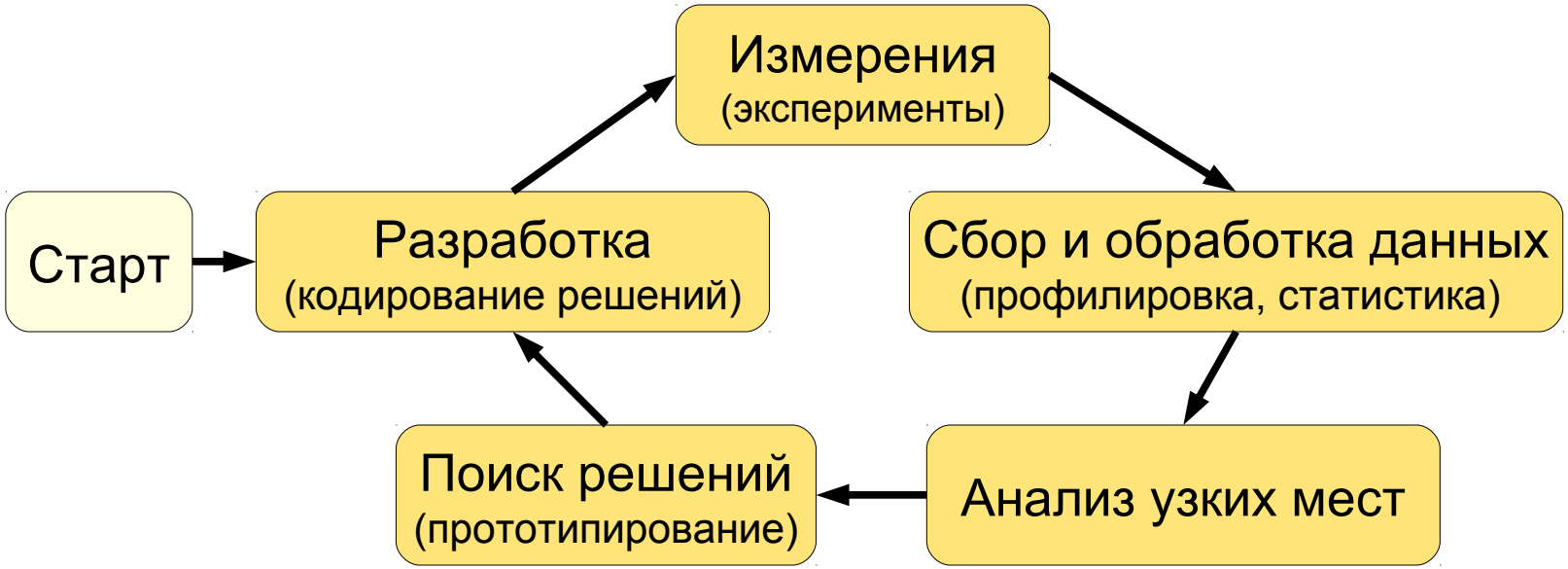

НЕСМОТРЯ НА ДЕТАЛЬНЫЙ АНАЛИЗ ТЕКУЩЕЙ СИТУАЦИИ, Я ТАК И НЕ СМОГ СОСТАВИТЬ ЧЁТКОЕ ПРЕДСТАВЛЕНИЕ ОБ ОБСУЖДАЕМОЙ ПРОБЛЕМЕ В СИЛУ ВОЗНИКШЕГО КОНГИТИВНОГО ДИССОНАНСА.

Q/A

Backup

Performance Engineering

итеративный подход



Важно:

- Одно изменение за цикл!
- Документировать все изменения

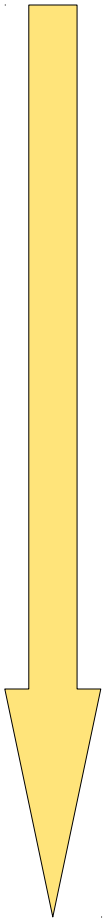
Performance Engineering

анализ узких мест

- **Что ограничивает скорость работы приложения?**
  - CPU
  - Ядро ОС
  - I/O (Сеть, Диск)

Performance Engineering

”нисходящий” метод поиска узких мест



- **Уровень системы**
  - Сеть
  - Диск
  - Database
  - Операционная система
  - Процессор/память
- **Уровень приложения**
  - Блокировки, синхронизация
  - Execution Threads
  - API
  - Алгоритмические проблемы
- **Уровень JVM**
  - Выбор JVM
  - Heap tuning
  - JVM tuning

New Features, Compatibility, Support, and beyond

Java 7, Java 8

что ожидать в области производительности

- **Java 7**
  - invokedynamic
  - NIO.2
  - Concurrency and Collection updates (Fork/Join)
  - XRender pipeline for Java2D (client)

- **Java 8 (или позже)**
  - Модульность
  - λ-выражения (замыкания)
  - Collection updates (filter, map, reduce)

Обратная совместимость

- **Мешает ли улучшать производительность JVM?**
  - **Да**, поэтому иногда расширяемся:
    - invokedynamic
    - Модульность
- **Стоит ли расширяться по первому требованию?**
  - **Нет**, развитие JVM/JIT, реализация новых методов оптимизации позволяет получить бОльший выигрыш
    - Вложенные классы
    - Reflection

Лучшая ОС для Java

угадайте, какая?

Solaris

- **“Engineered Together”**
- **Высокопроизводительный TCP/IP стек**
  - low-latency
  - up to 50% faster
- **DTrace**
  - мониторинг
- **NUMA**
  - MPO, Memory Placement Optimization
- **Large Pages**
  - Автоматическая аллокация
  - Разные размеры

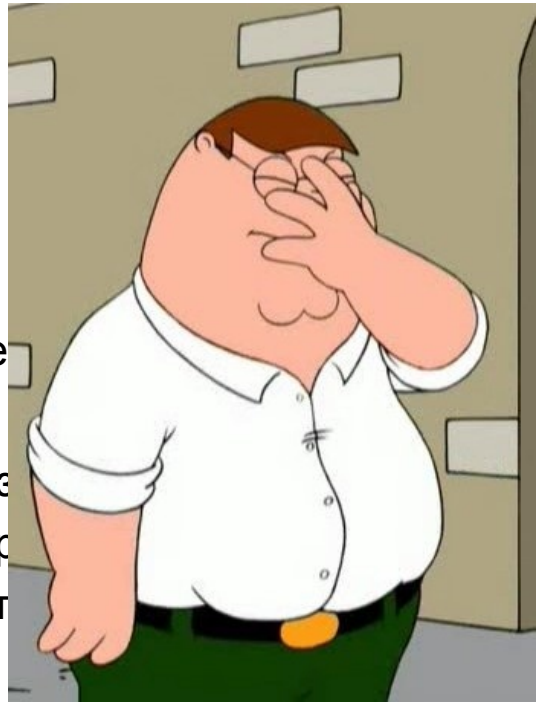
Benchmarking

ближе к делу, пример

- **Computer Language Benchmark Game**
  - http://shootout.alioth.debian.org/
  - Самый известный полигон для расстрела невинных ног
- **Тест “PiDigits”**
  - Вычислить первые N цифр в десятичном разложении π
  - Вычисляется приближение π специальным рядом
    - Требуется arbitrary precision для представления рациональных членов ряда
  - Потенциально параллелизуема
  - Две версии Java-теста:
    - Текущая использует GNU MP, внутренне параллельна, 240 строк кода, Java + нативные врапперы + GMP
    - Предыдущая использует BigInteger, не параллельна, 130 строк кода, plain Java

Benchmarking

ближе к делу, пример



- **Computer Language Benchmark Game**
  - http://shootout.alioth.debian.org/
  - Самый известный полигон для расстрела не
- **Тест “PiDigits”**
  - Вычислить первые N цифр в десятичном раз
  - Вычисляется приближение π специальным р
    - Требуется arbitrary precision для предст членов ряда
  - Потенциально параллелизуема
  - Две версии Java-теста:
    - Текущая использует GNU MP, внутренне параллельна, 240 строк кода, Java + нативные врапперы + GMP
    - Предыдущая использует BigInteger, не параллельна, 130 строк кода, plain Java

Benchmarking

ближе к делу, пример

- **Метрика: время исполнения**
  - `time java -server pidigits 1000`

- **“Прогреем”:**

```
pidigits.javasteady:
   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);
   }
```

Benchmarking

ближе к делу, пример

- **Метрика: время исполнен**
  - `time java -server pidi`
- **“Прогреем”:**

```
pidigits.javasteady:
   public static void mai
      pidigits m = new pi
      for (int i=0; i<65;
      m.pidigits(true);
   }
```

```
pidigits.java:
   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);
   }
```

Benchmarking

в век многоядерных систем

- **Обычная платформа**
  - 2x2 Intel i5 2.6 Ghz, Ubuntu 10.10 i686, JDK 6u25
- **Три варианта эксперимента:**
  - 1 экземпляр бенчмарка
    - по “методологии CLBG” (+ корректный прогрев)
    - GMP будет использовать 1x4=**4** потока
    - BI будет использовать 1x1=**1** поток
  - 4 экземпляра бенчмарка
    - чтобы утилизировать все ядра = снормировать на систему
    - GMP будет использовать 4x4=**16** потоков
    - BI будет использовать 4x1=**4** потока
  - 16 потоков
    - ради одинаковой нагрузки на OS = снормировать на систему+OS
    - GMP будет использовать 4x4=**16** потоков
    - BI будет работать в 16x1=**16** потоков
- **Какой из них наиболее релевантен?**

Benchmarking

а вот и данные

- **Метрика: операции в секунду**
  - 20 итераций
  - [a; b] – доверительный интервал на 95%

	1 экземпляр	4 экземпляра	16 потоков
“GMP”	**8.84** [8.69; 8.99]	**13.28** [12.86; 13.71]	**13.28** [12.86; 13.71]
“BigInteger”	**6.21** [6.19; 6.24]	**13.46** [13.35; 13.56]	**14.34** [14.21; 14.46]

Concurrency

элементная база

- **OS Threading**
  - мьютексы
    - mutex_lock()/mutex_unlock()
  - conditional waits
    - cond_wait()/cond_signal()
    - WaitForSingleObject
- **Compare-and-Swap (CAS)**
  - CAS(x1, x2, x3) = { if (x1 == x2) { x1 = x3 }; }
  - атомарная операция, поддерживаемая в “железе”: из нескольких одновременных CAS'ов успешно завершается только один
  - Миф: локальный CAS блокирует шину, и ст***о***ит больше на многопроцессорных системах
  - Факт: *глобальный* CAS требует трафика на шине

http://blogs.sun.com/dave/entry/biased_locking_in_hotspot

Concurrency

atomics

- **java.util.concurrent.Atomic***
  - обеспечивают атомарные операции над примитивами и указателями
  - альтернатива: synchronized {} или Lock'и
- **Трюк в использовании CAS'a:**
  - Изменение состояния атомика делается при помощи одного CAS'a
  - Чтение состояния не требует CAS'a

```
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

```
mov     %ecx,%edx
mov     0x8(%ecx),%eax
lea     0x8(%ecx),%edi
mov     %eax,%ecx
inc     %ecx
lock cmpxchg %ecx,(%edi)
mov     $0x0,%ebx
jne     [ok]
mov     $0x1,%ebx
test    %ebx,%ebx
je      [ok]
```

Concurrency

volatile

- **Volatile определяет порядок чтения-записей в поле**
  - Точная семантика определена в Java Memory Model
    - ***НЕ*** обеспечивает атомарности
  - Реализуется расстановкой барьеров
    - Какие из них вставятся в код, зависит от Hardware Memory Model
    - Эффект барьера зависит от HMM

```
PUSHL   EBP
SUB     ESP,8
MOV     EBX,[ECX + #12]
MEMBAR-acquire
MEMBAR-release
INC     EBX
MOV     [ECX + #12],EBX
MEMBAR-volatile
LOCK ADDL [ESP + #0], 0
ADD     ESP,8
POPL    EBP
TEST    PollPage,EAX
RET
```

```
push    %ebp
sub     $0x8,%esp
mov     0xc(%ecx),%ebx
inc     %ebx
mov     %ebx,0xc(%ecx)
lock addl $0x0,(%esp)
add     $0x8,%esp
pop     %ebp
test    %eax,0xb779c000
ret
```

http://g.oswego.edu/dl/jmm/cookbook.html

Concurrency

intrinsic synchronization

- **synchronized(object) { }**
- **4 состояния:**
  - ***Init***
    - Начальное “неопределённое” состояние
  - ***Biased***
    - Захватывается одним “владеющим” потоком, нет конфликтов
    - Захват владельцем: проверка на threadID
    - Захват не-владельцем: переход либо в ***Biased***, либо в ***Thin***
  - ***Thin***
    - Захватывается несколькими потоками, но конфликтов нет
    - Захват: CAS
    - Конфликтный захват: переход в ***Fat***
  - ***Fat***
    - Захватывается несколькими потоками, конфликт на блокировке
    - Вызов примитива синхронизации из ОС

Concurrency

java.util.concurrent.Lock

- **Построены на базе j.u.c.AbstractQueueSynchronizer**
  - Использует атомики
    - CAS
  - Использует Unsafe.park()/unpark()
    - интринзики для cond_wait()/cond_signal()/WaitForSingleObject()
- **ReentrantLock**
  - **По семантике эквивалентен synchronized {}**
    - Рекурсивный захват: проверка на threadID
    - Ставит потоки во внутреннюю очередь и делает park()
  - ***Non-Fair (default)***
    - Не гарантирует отсутствие starvation
    - Barging FIFO (CAS)
    - Лучшая производительность
  - ***Fair***
    - Гарантирует отсутствие starvation
    - FIFO
    - Честность в обмен на производительность

http://gee.cs.oswego.edu/dl/papers/aqs.pdf

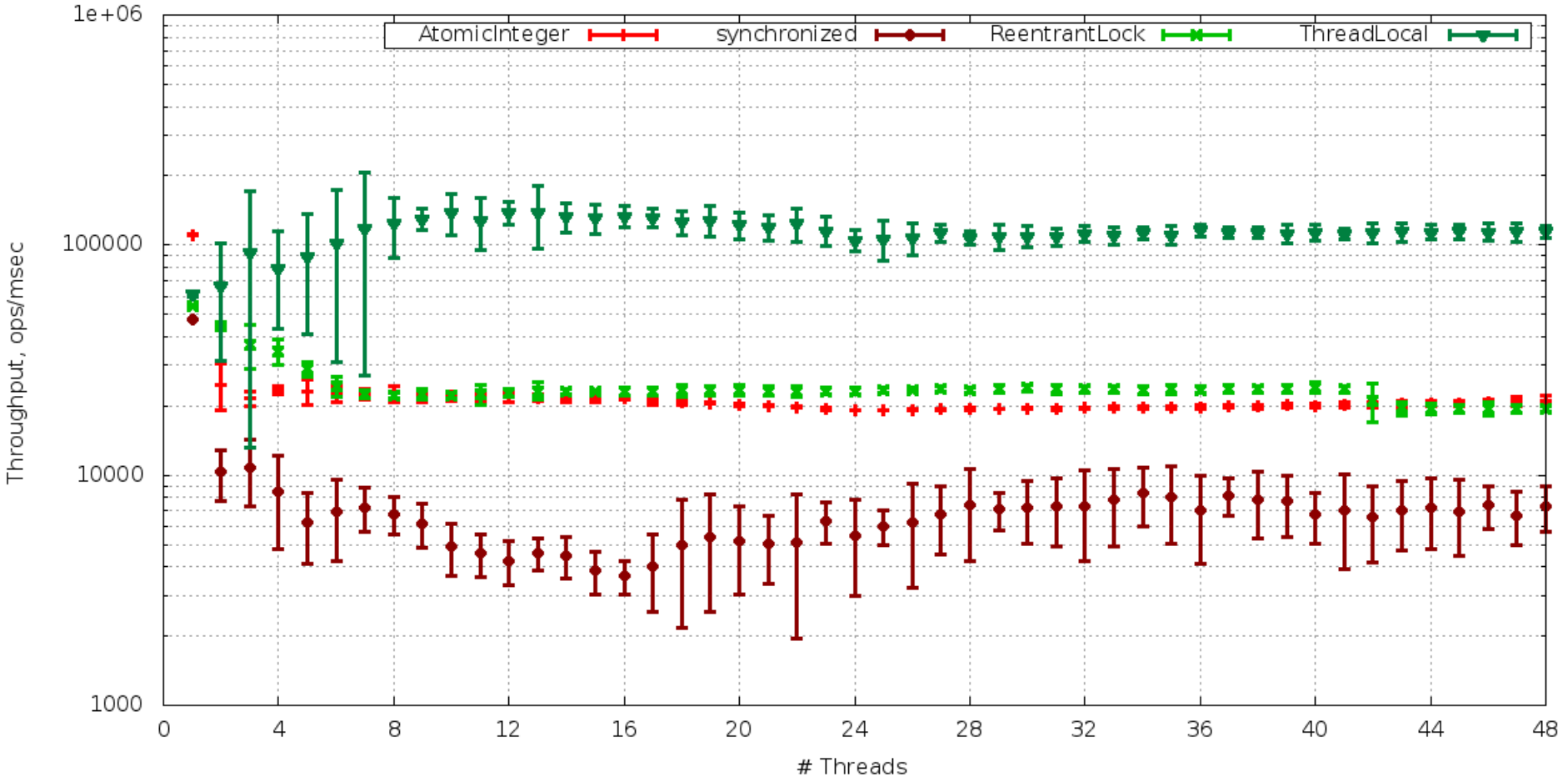
Concurrency

атомарный счётчик

```
private AtomicInteger atomic = new AtomicInteger();
private ReentrantLock lock = new ReentrantLock();
private final Object intrinsicLock = new Object();
private int primCounter = 0;

@GenerateMicroBenchmark
public void testAtomicInteger() {
    atomic.incrementAndGet();
}

@GenerateMicroBenchmark
public void testReentrantLock() {
    lock.lock();
    primCounter++;
    lock.unlock();
}

@GenerateMicroBenchmark
public void testIntrinsicLock() {
    synchronized (intrinsicLock) {
        primCounter++;
    }
}
```

Concurrency

атомарный счётчик



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

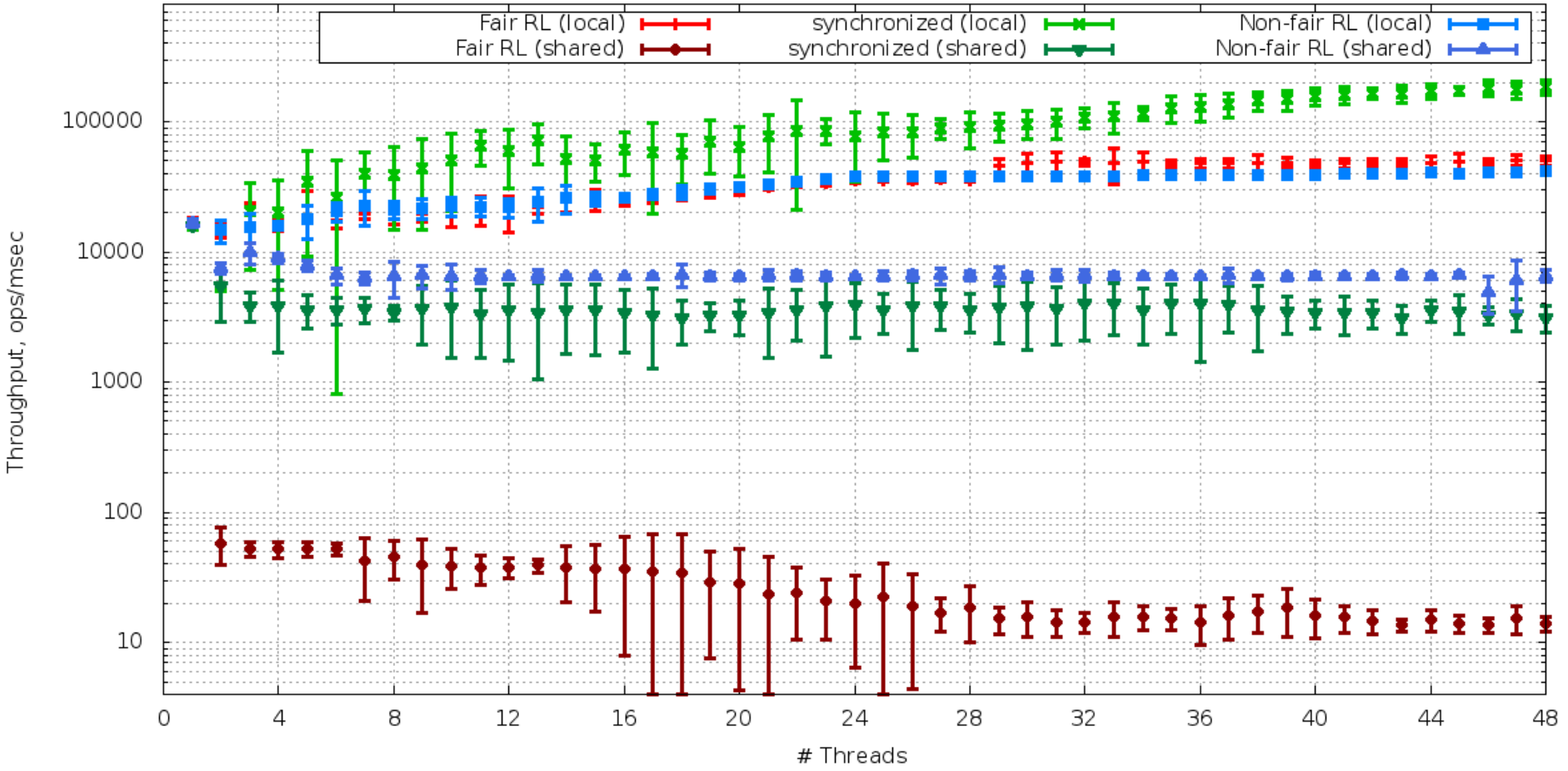
Concurrency

ReentrantLock vs. synchronized

- **Семантика одинакова**
  - Требования к видимости памяти
  - Рекурсивный
- **Плюсы j.u.c.RL**
  - Очередь потоков держится на стороне JVM
    - опционально, FIFO-политика при захвате-освобождении
    - позволяет быть “честным” на любой платформе
  - Barging FIFO policy
    - lock() может быть сразу удовлетворён, даже если в очереди есть потоки
    - сильно улучшает производительность при конфликте блокировок
  - Допускается несколько Condition
- **Минусы j.u.c.RL**
  - Нет scope'ов, требуется ручной unlock() через *finally*
    - Должно быть проще с ARM в JDK7

Concurrency

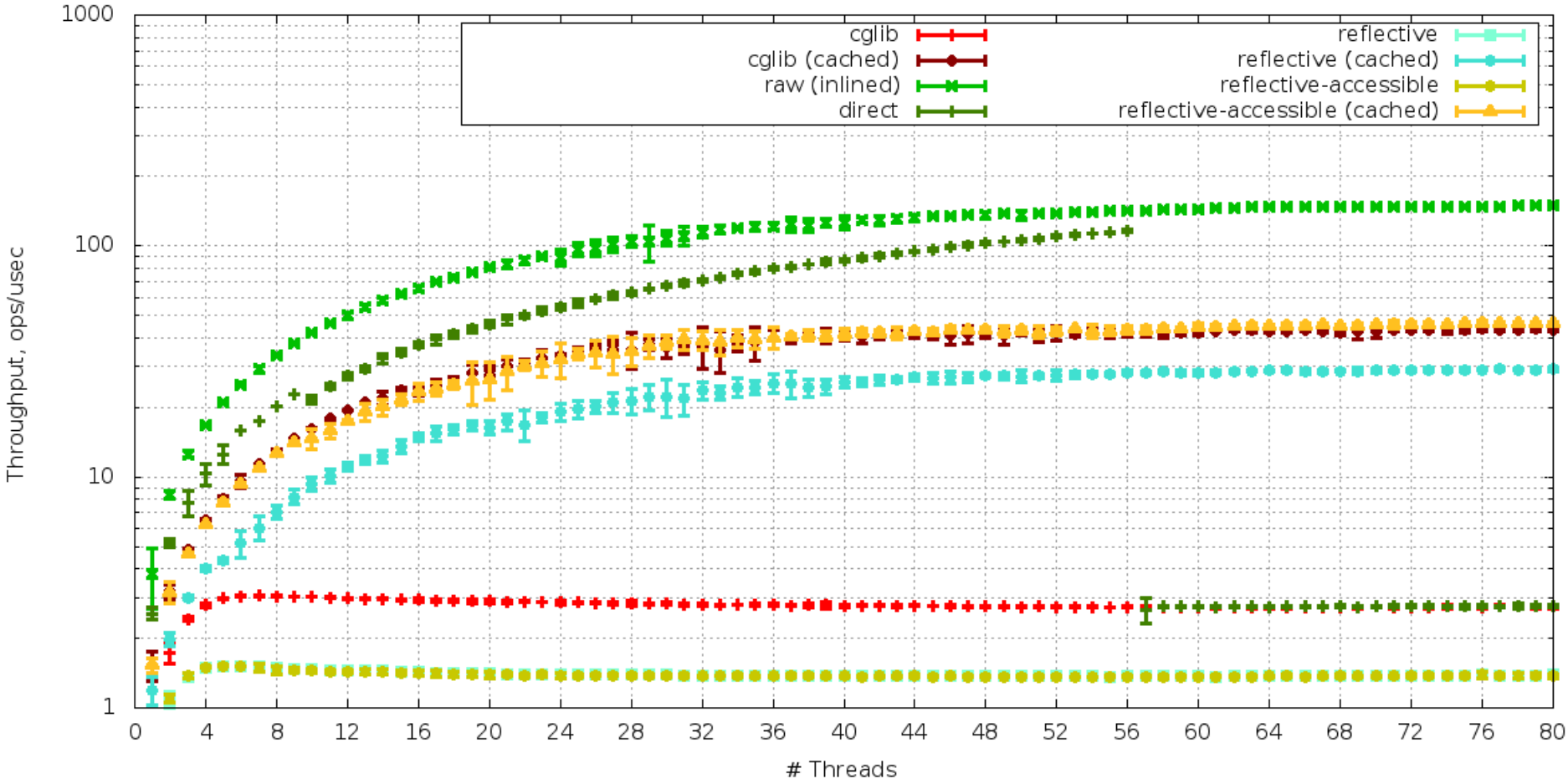
synchronized vs. ReentrantLock



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

Reflection

direct vs. reflection vs. cglib



Intel Xeon (Nehalem) 3.0 Ghz, 8x4x2 = 64 HW threads, Solaris 10, JDK 7b141


ORACLE®