Профилируем с точностью до инструкций и микросекунд

Sergey Melnikov

Райффайзен БАНК

Обо мне

- Совсем **не Enterprise** Java-разработчик, только **Java SE**
- В прошлом Compiler **Performance** Engineer @ **Intel** Compiler Lab
- Контрибьютил в Android (AOSP), FreeBSD, GCC, ocperf, jmh, …

Оглавление

1. Введение

Не думай о миллисекундах свысока

1 секунда	1,000 миллисекунд	1,000,000 микросекунд	1,000,000,000 наносекунд

Не думай о миллисекундах свысока

1 секунда	1,000 миллисекунд	1,000,000 микросекунд	1,000,000,000 наносекунд
4,000,000,000 тактов	4,000,000 тактов	4,000 тактов	4 такта

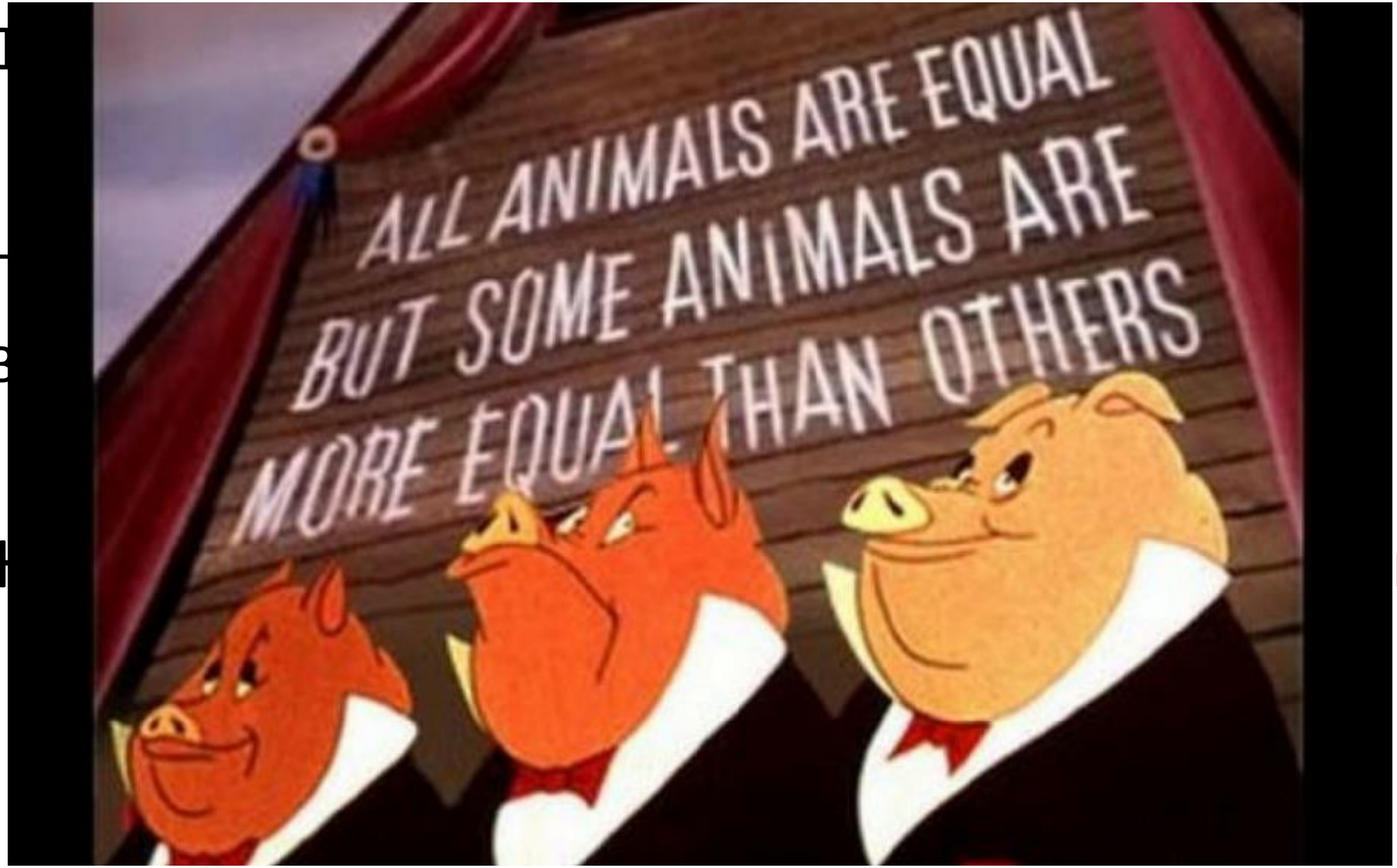
Предметная область

- Low-latency торговое приложение
- Все приложение критично к быстродействию

Но некоторые участки кода “более критичны” к быстродействию!

Пред

- L
- B


ALL ANIMALS ARE EQUAL
BUT SOME ANIMALS ARE
MORE EQUAL THAN OTHERS

K l” к

Самый критичный к скорости код

Выполняется сотни микросекунд и напрямую влияет на финансовый результат

100мкс могут стоить миллион рублей на каждой сделке

Самый критичный к скорости код

Всегда хочется быстрее!

Пример: Московская Биржа

Отправка нескольких ордеров через low-latency engine для работы с сетью

Нужно отправить быстро и **все** ордера **одновременно**

2. Выбираем средство анализа

Как же это профилировать?

- Участок кода сложно выделить в микробенчмарк
- Код затрагивает сеть
- Хочется получить профиль максимально похожий на реальный

Как будем профилировать

- Инструментирующие профилировщики?
- Сэмплирующие профилировщики?

Инструментирующие профилировщики

- Значительные накладные расходы
- Могут существенно влиять на исполняемый код

А вообще, на сколько это то же самое, что и на PRODe?

Потенциально некоторые оптимизации работают не так, как без инструментации

Инструментирующие профилировщики

Вывод: смотрим что-то более подходящее

Сэмплирующие прифилировщики

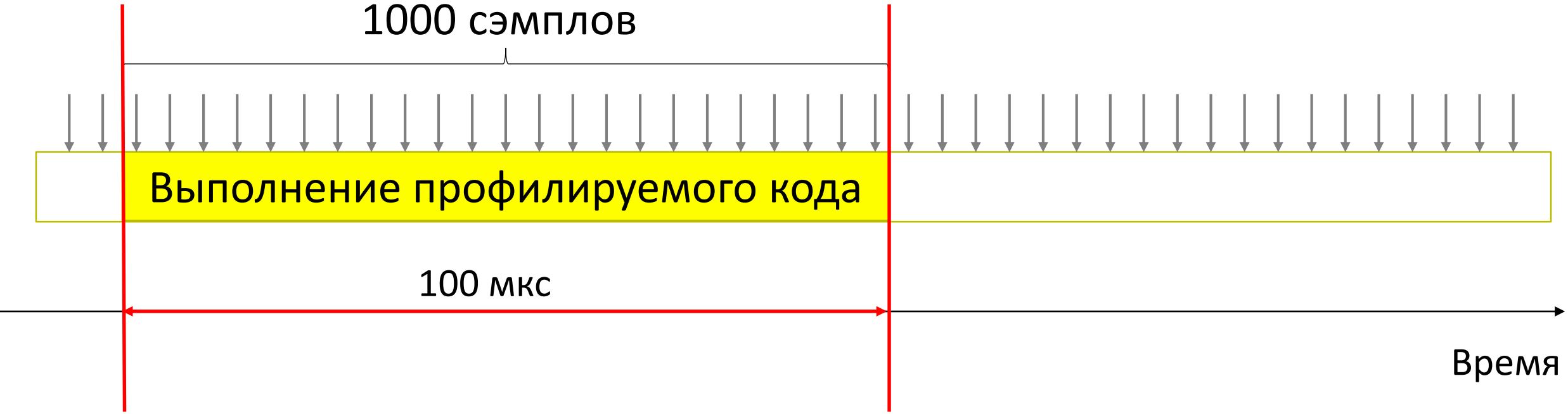
- Накладные расходы от умеренных до минимальных
- Не влияют напрямую на исполняемый код
- Анализ результатов требует некоторой экспертизы

С какой частотой сэмплируем?

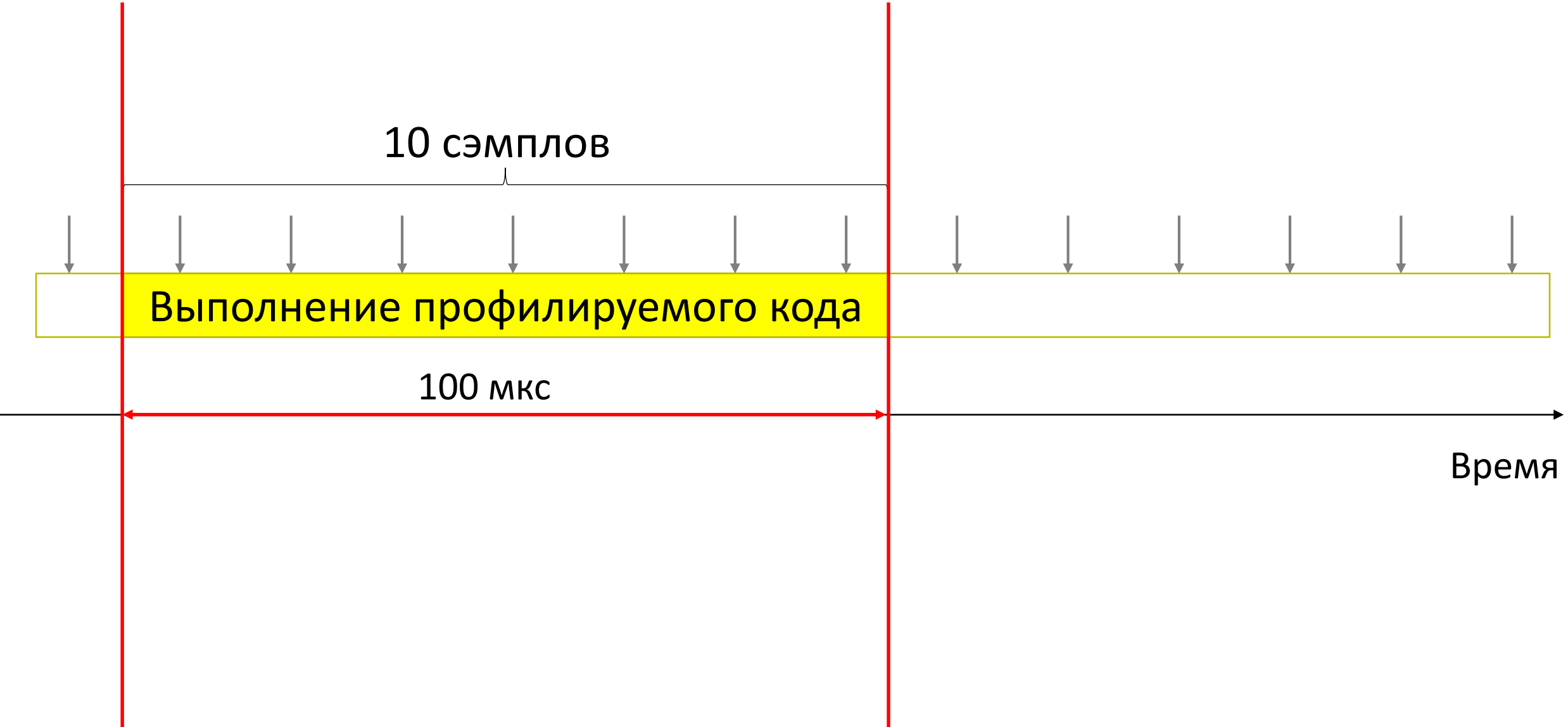
Допустим, мы хотим собрать 1000 сэмплов за 100мкс

Для этого нужна частота сэмплирования **10,000,000** сэмплов/с

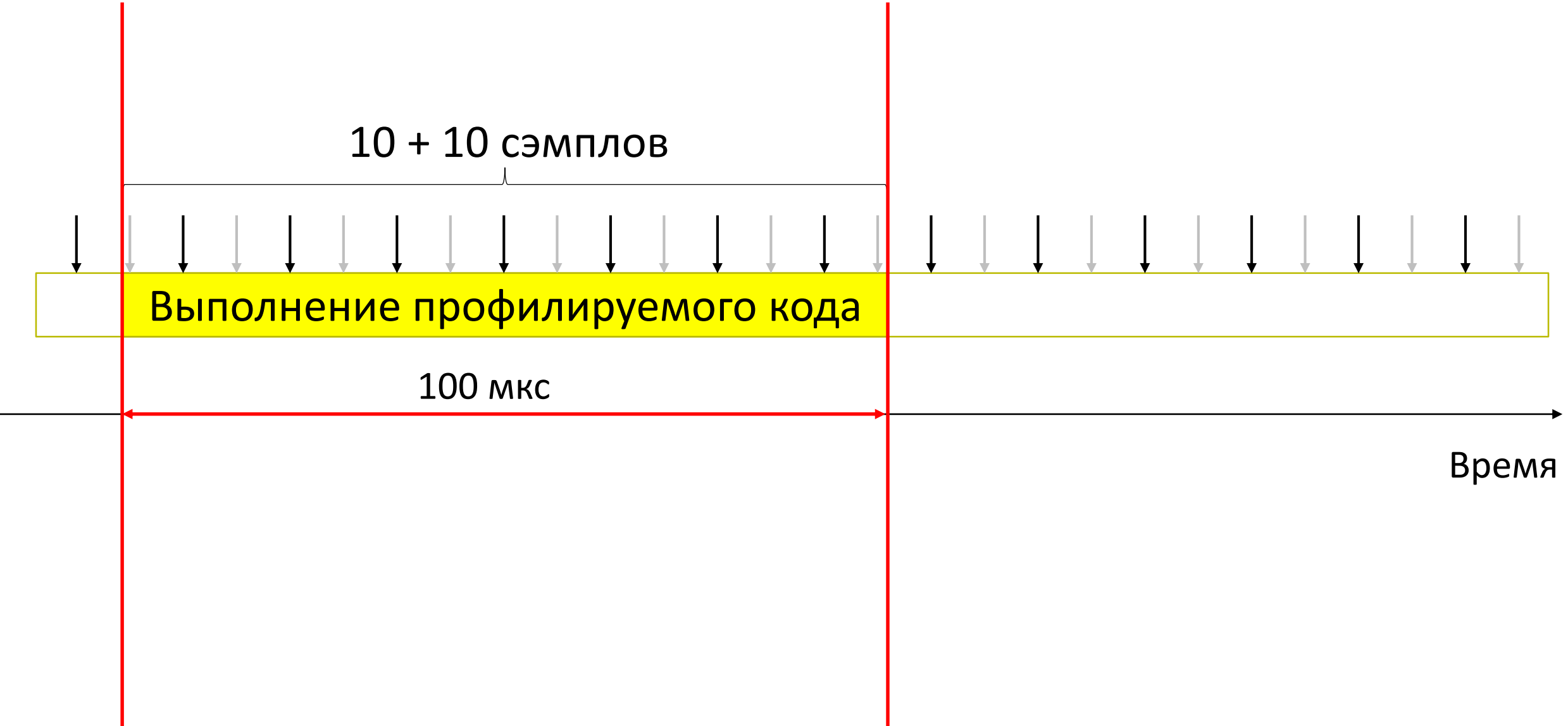
Сэмплирование на частоте 10,000,000 сэмплов в секунду



А если частота сэмплирования меньше?

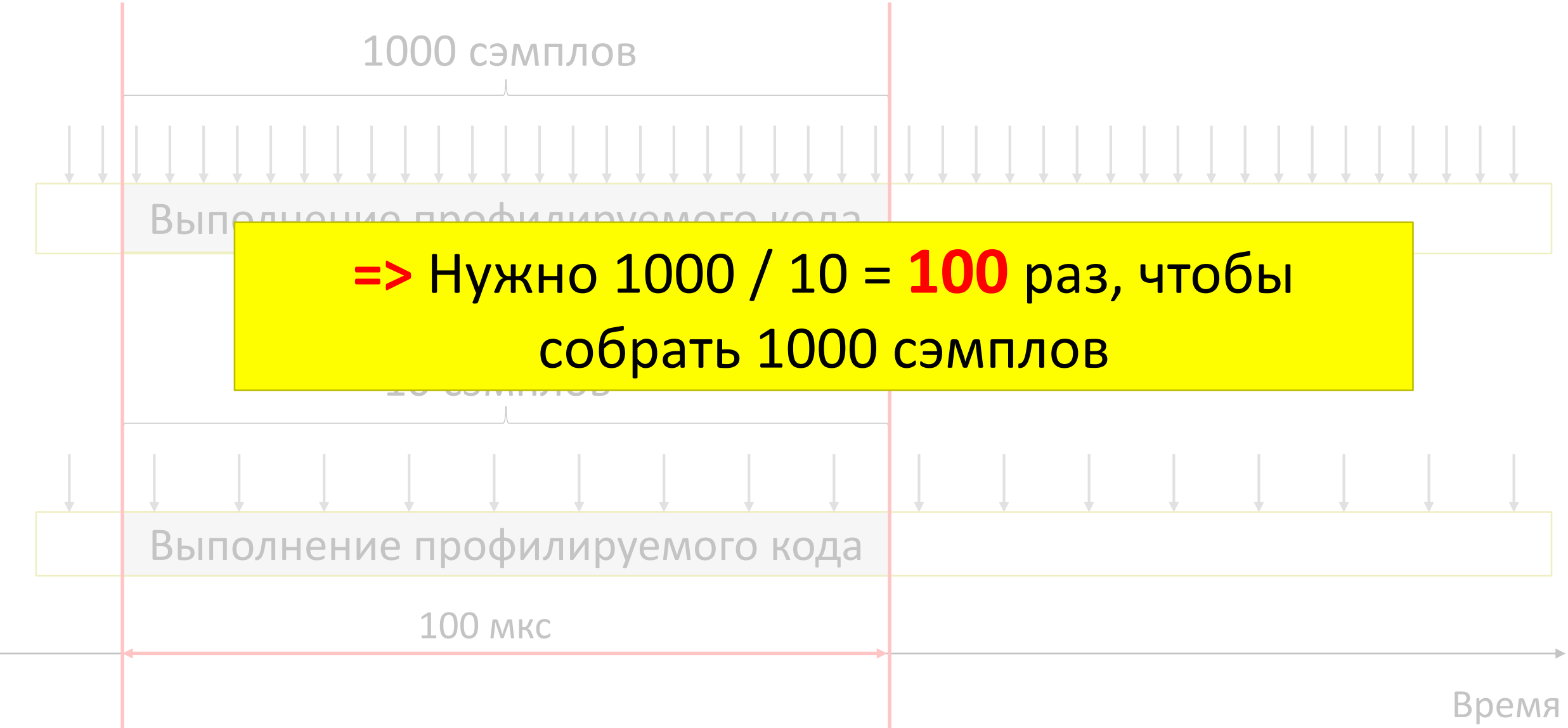


А если частота сэмплирования меньше?



А если частота сэмплирования меньше?

Тогда время профилирования пропорционально увеличивается



Async-profiler

Используется AsyncGetCallTrace, т.е.

- Полный Java stack
- Инструкции байт-кода

Штатная частота сэмплирования 1000 сэмплов/сек

или 0.1 сэмпл в 100мкс

=> Время профилирования 1000 / 0.1 = 10,000 сек ~= 2.8 часа

Async-profiler – попробуем разогнать

Из README.md:

“`-i N - sets the profiling interval, in nanoseconds`”

То, что надо - будем пробовать!

Получили 1 сэмпл в 380мкс

Т.е. эффективная частота ~2600 сэмплов/сек

Или 100 * 1 / 380 ~= 0.26 сэмпла в 100 мкс

Время профилирования

1000 / 0.26 ~= 3800 сек или **1 час**

Здесь и далее замеры производились на Intel HSW i7-4790 32Gb RAM

Частоты (существенно) больше 1000Гц

Use the ~~force~~ ***perf***, Luke!

Отказываемся от AsyncGetCallTrace

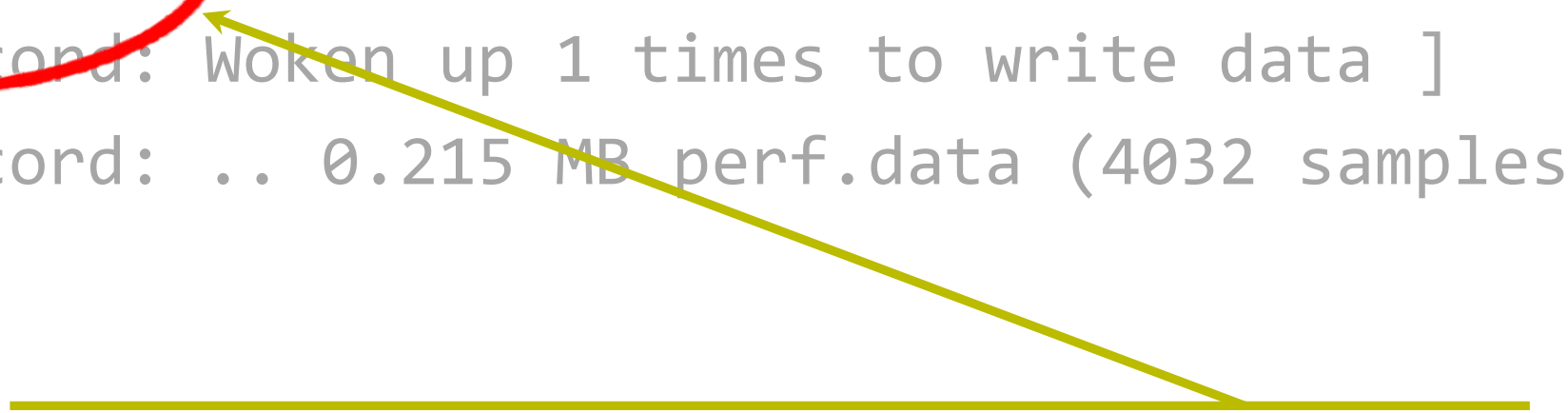
3. Учим perf собирать подробный java-профиль

Запускаем perf

```
$ perf record –F 1000 -p PID -g -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: .. 0.215 MB perf.data (4032 samples) ]
```

Запускаем perf

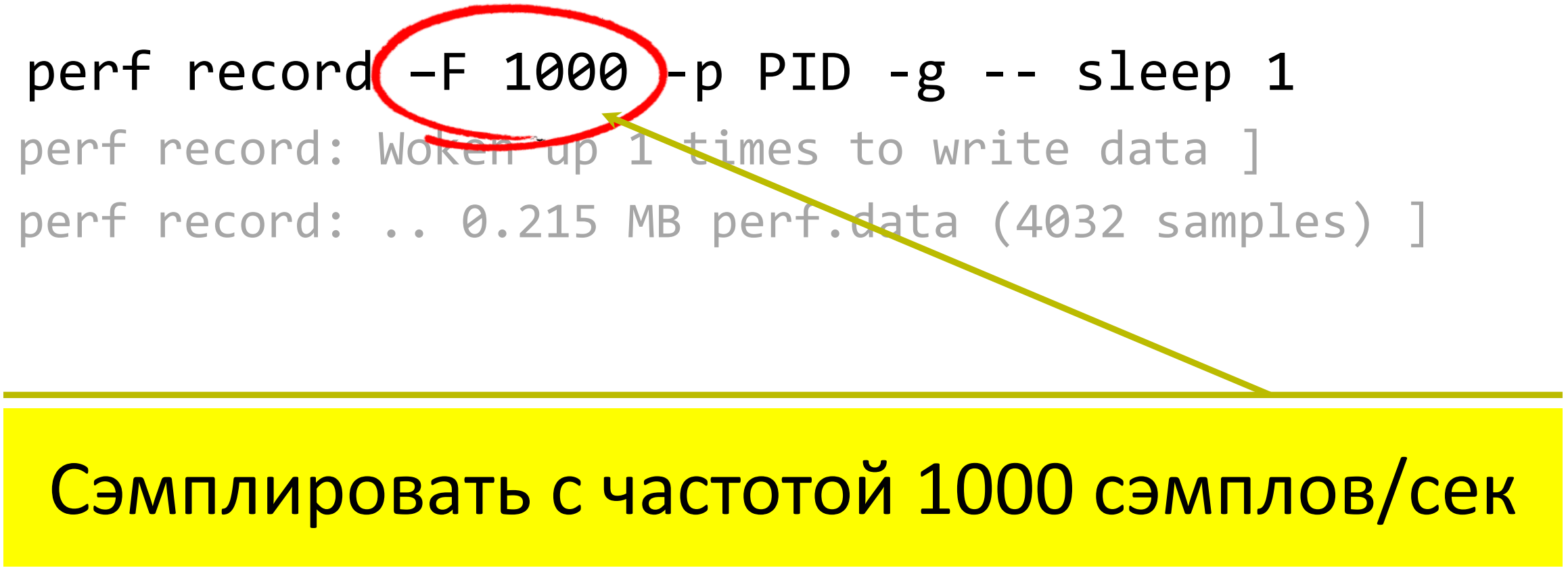
```
$ perf record -F 1000 -p PID -g -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: .. 0.215 MB perf.data (4032 samples) ]
```



Начать запись профиля perf’ом

Запускаем perf

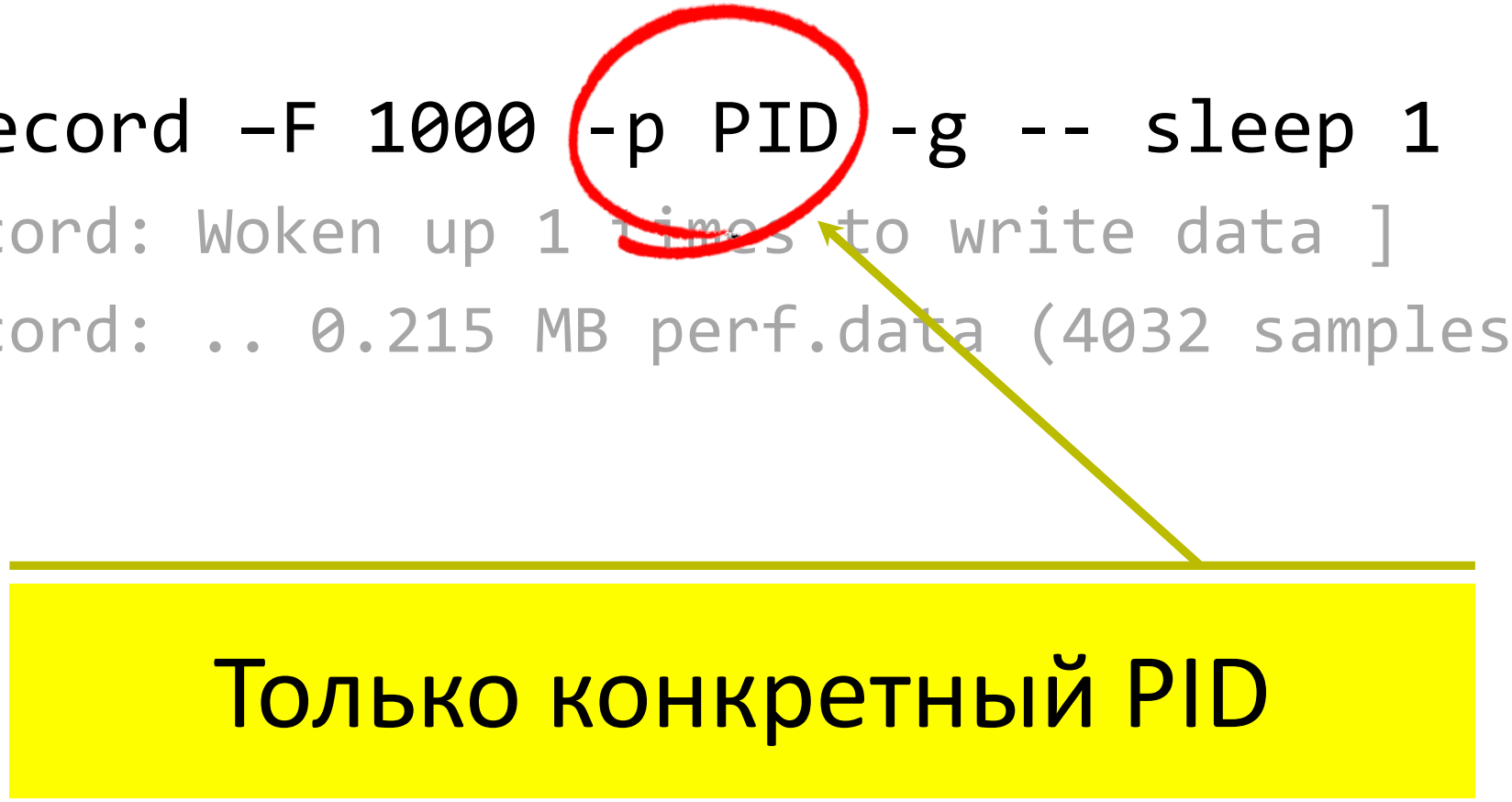
```
$ perf record -F 1000 -p PID -g -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: .. 0.215 MB perf.data (4032 samples) ]
```



Сэмплировать с частотой 1000 сэмплов/сек

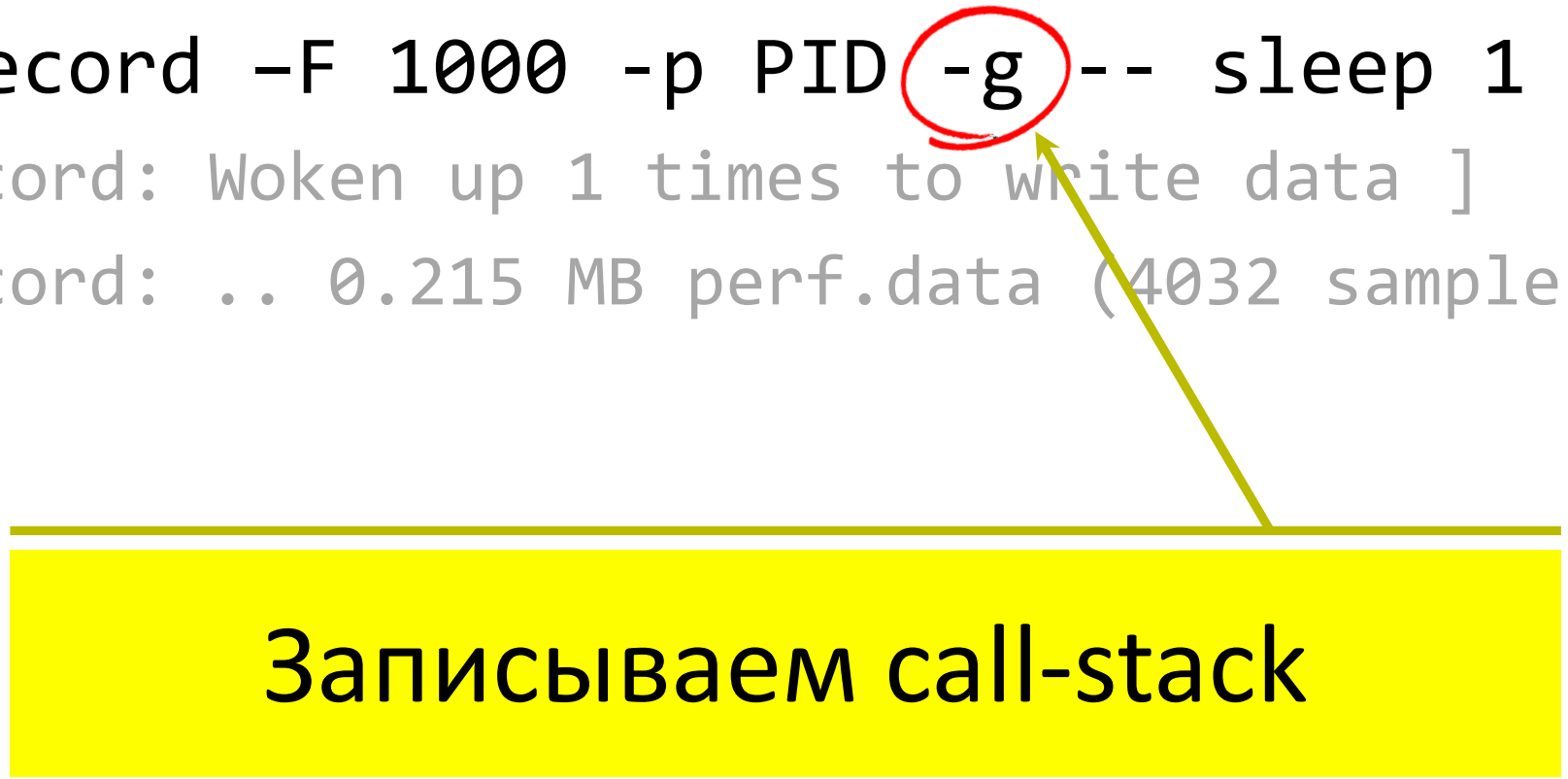
Запускаем perf

```
$ perf record –F 1000 -p PID -g -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: .. 0.215 MB perf.data (4032 samples) ]
```



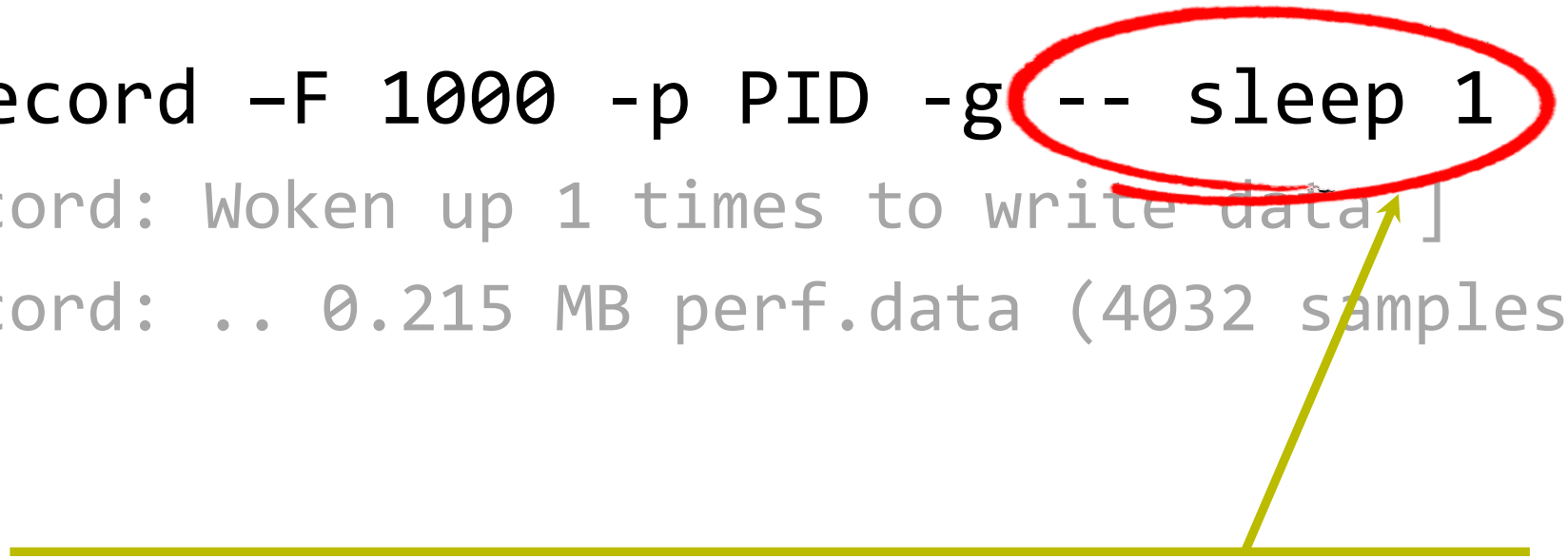
Запускаем perf

```
$ perf record –F 1000 -p PID -g -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: .. 0.215 MB perf.data (4032 samples) ]
```



Запускаем perf

```
$ perf record –F 1000 -p PID -g -- sleep 1
[ perf record: Woken up 1 times to write data ]
[ perf record: .. 0.215 MB perf.data (4032 samples) ]
```



Запись длится 1 секунду

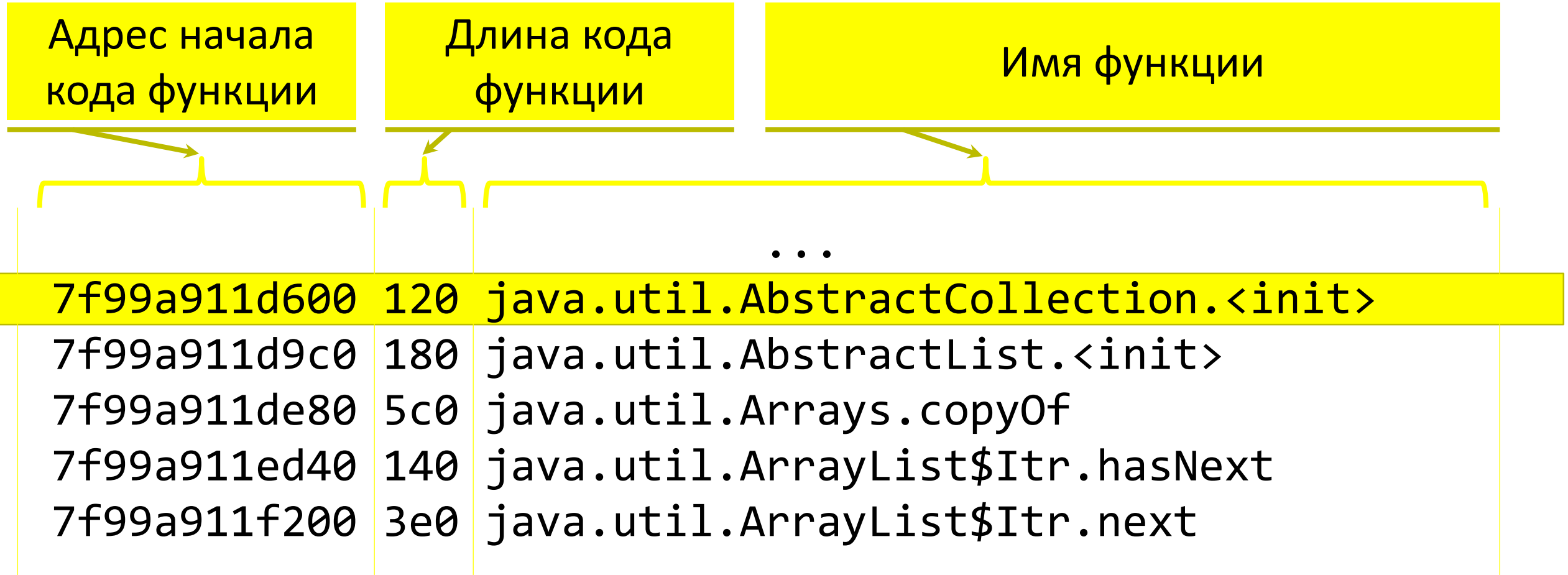
Смотрим профиль

```
$ perf script

java 8079 2008793.746571:    3745505 cycles:uppp:
            7fa1e88b53f8 [unknown] (/tmp/perf-11038.map)
java 8079 2008793.747565:    3728336 cycles:uppp:
            7fa1e88b5372 [unknown] (/tmp/perf-11038.map)
java 8079 2008793.748613:    3731147 cycles:uppp:
            7fa1e88b53ef [unknown] (/tmp/perf-11038.map)
```

И где мои функции?

Как perf узнает названия функций из java?

Perf читает файл /tmp/perf-*PID*.map со списком адресов и названий методов



Скрипт create-java-perf-map.sh из проекта perf-map-agent

```
$ bin/create-java-perf-map.sh 8079

$ ls  /tmp/perf-8079.map
/tmp/perf-8079.map

$ tail -n 2 /tmp/perf-8079.map
7fa1e88b1e20 380 Loop3.doRecursiveCall
7fa1e88b5360 180 Loop3.doRecursiveCall
```

Смотрим профиль. Вторая попытка

```
$ perf script

java  8080 1895245.867498:    cycles:uppp:
     7fb2dd10f527 Loop3.doRecursiveCall (/tmp/perf-8079.map)

java  8080 1895245.868176:    2127960 cycles:uppp:
     7fb2dd10f57f Loop3.doRecursiveCall (/tmp/perf-8079.map)

java  8080 1895245.868737:    1959990 cycles:uppp:
     7fb2dd10f627 Loop3.doRecursiveCall (/tmp/perf-8079.map)
```

Смотрим профиль. Вторая попытка

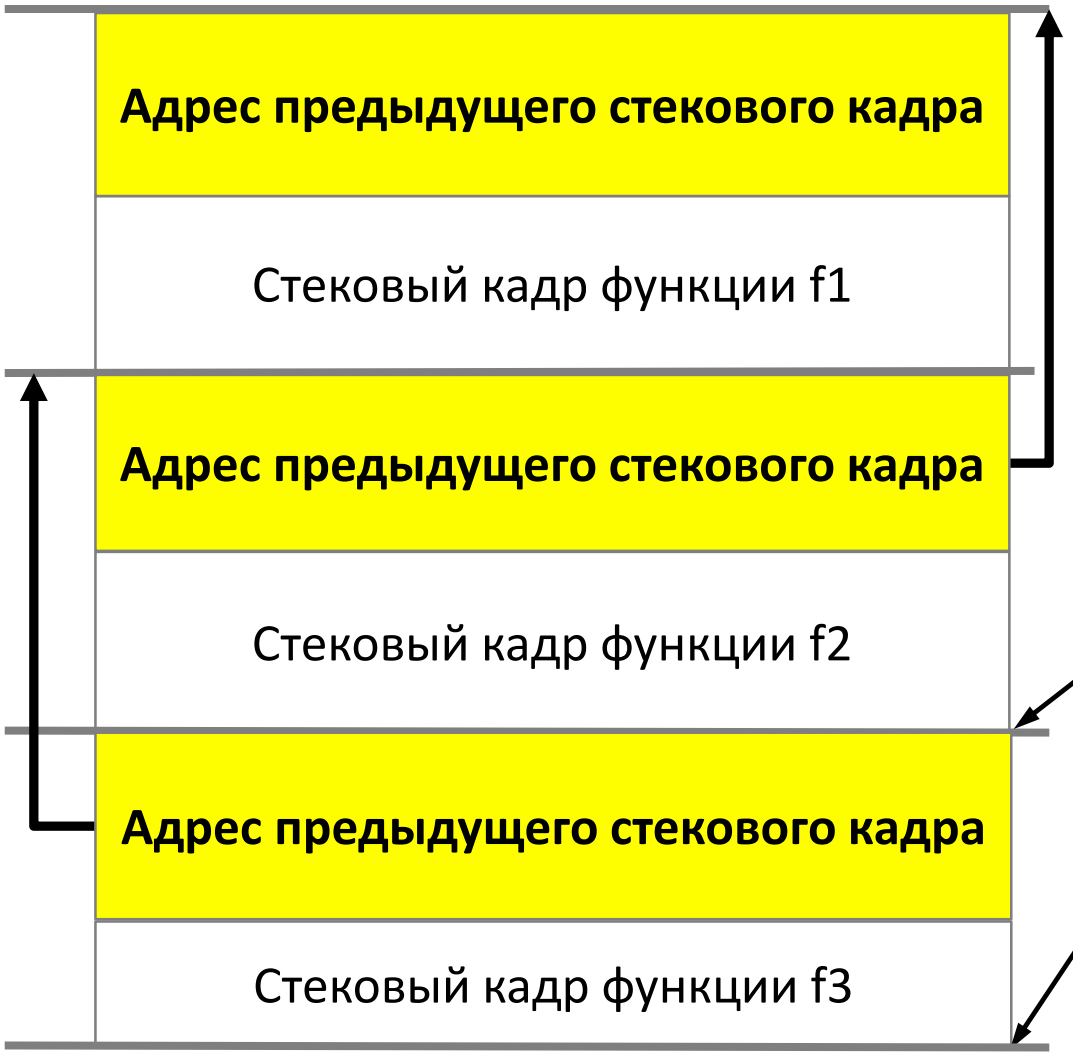
```
$ perf script

java  80
     7fb                                              9.map)

java  80
     7fb2dd10f57f Loop3.doRecursiveCall (/tmp/perf-8079.map)

java  8080 1895245.868737:    1959990 cycles:uppp:
     7fb2dd10f627 Loop3.doRecursiveCall (/tmp/perf-8079.map)
```

А где мои call-stack'и?

А как можно получить call-stack?

```
void f1() {
    f2();
}
void f2() {
    f3();
}
void f3() {
    ...
}
```

А как можно получить call-stack?



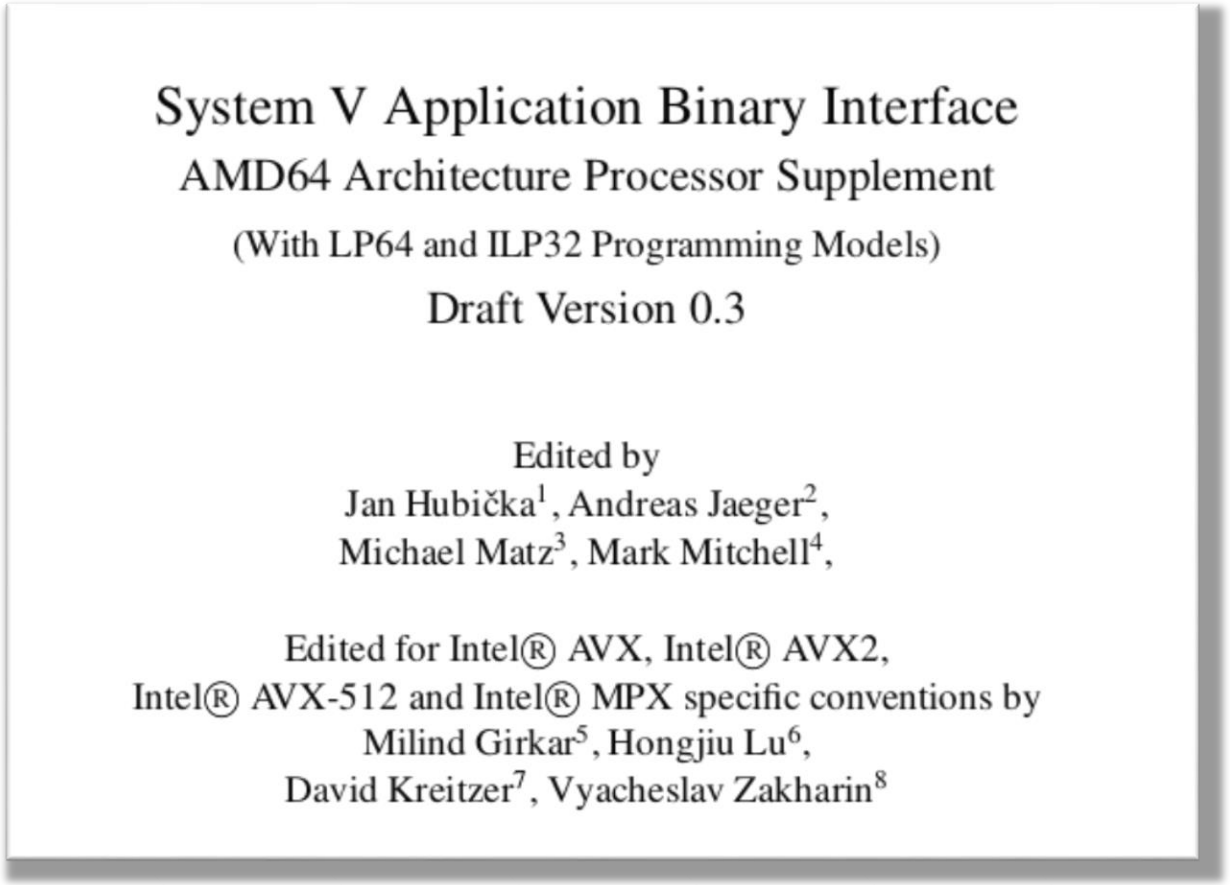
Значение в регистре %rbp позволяет **perf**'у получить стек вызовов

Регистр %rbp

Регистр %rsp

```
void f1() {
    f2();
}
void f2() {
    f3();
}
void f3() {
    ...
}
```

Подробнее, как работает вызов функций в Linux, можно прочитать в *System V Application Binary Interface*



System V Application Binary Interface

AMD64 Architecture Processor Supplement

(With LP64 and ILP32 Programming Models)

Draft Version 0.3

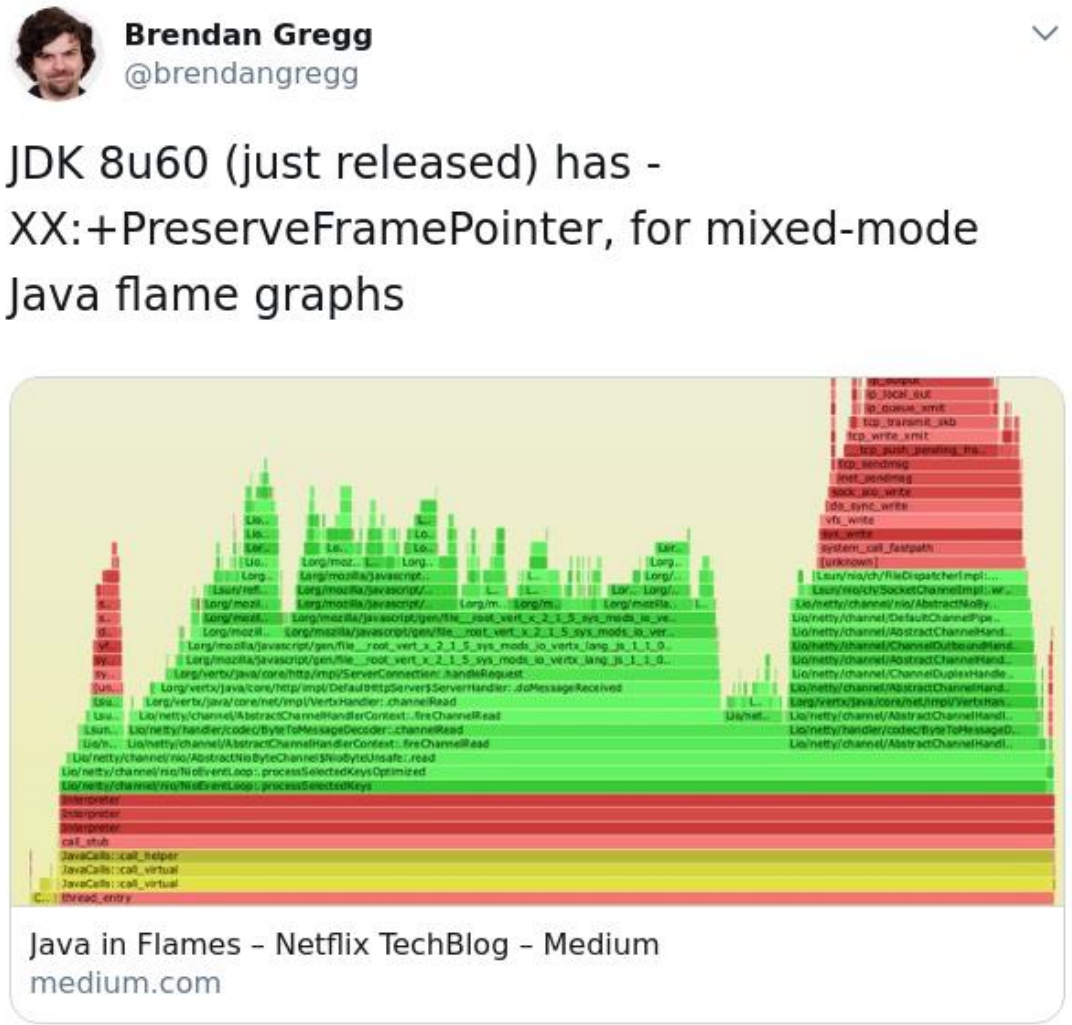
Edited by

Jan Hubička[1], Andreas Jaeger[2],
Michael Matz[3], Mark Mitchell[4],

Edited for Intel® AVX, Intel® AVX2,
Intel® AVX-512 and Intel® MPX specific conventions by
Milind Girkar[5], Hongjiu Lu[6],
David Kreitzer[7], Vyacheslav Zakharin[8]

https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf

Воспользуемся флагом -XX:+PreserveFramePointer



-XX:+PreserveFramePointer заставляет JVM хранить в регистре %rbp указатель на текущий стековый кадр

=> Можно получить адрес предыдущего стекового кадра

Запускаем perf еще раз

```
$ java -XX:+PreserveFramePointer -cp . ...

$ bin/create-java-perf-map.sh 18657

$ perf record ...

$ perf script ...
```

Смотрим профиль. Третья попытка

```
$ perf script

java 18657 1901247.601878:     979583 cycles:uppp:
            7fbfd1101edc Loop3.doRecursiveCall (...)
            7fbfd1101edc Loop3.doRecursiveCall (...)
            7fbfd1101edc Loop3.doRecursiveCall (...)
            7fbfd1101edc Loop3.doRecursiveCall (...)
            7f285d007b10 Interpreter (...)
            7f285d0004e7 call_stub (...)
                  67d0db [unknown] (... libjvm.so)
...
                    708c start_thread (... libpthread-2.26.so)
```

Проект perf-map-agent

Скрипт perf-java-record-stack из perf-map-agent:

- Генерирует файл с названиями java-функций для perf’а
- Запускает perf и профилирует java-приложение

Частота сэмплирования задается переменной окружения PERF_RECORD_FREQ

```
$ PERF_RECORD_FREQ=100000 ./bin/perf-java-record-stack PID
```

Запускаем

```
$ PERF_RECORD_FREQ=100000 ./bin/perf-java-record-stack PID
...
Maximum frequency rate (30000) reached.
Please use -F freq option with lower value or consider
tweaking /proc/sys/kernel/perf_event_max_sample_rate.
...
```

Выкручиваем нужный sysctl

```
$ echo '1000000' |
sudo tee /proc/sys/kernel/perf_event_max_sample_rate
```

или

```
$ sudo sysctl kernel.perf_event_max_sample_rate=1000000
```

Неужели получилось?

```
$ PERF_RECORD_FREQ=200000 ./bin/perf-java-record-stack PID
Recording events for 15 seconds (adapt by setting
PERF_RECORD_SECONDS)
[ perf record: Woken up 417 times to write data ]
[ perf record: Captured ... (1.051.541 samples) ]
```

1051541 / 15 ~= **70100** сэмплов/сек

А тем временем в dmesg…

```
[84430.412898] perf: interrupt took too long (1783 > 200),
lowering kernel.perf_event_max_sample_rate to 89700

...

[84431.618452] perf: interrupt took too long (2229 > 2228),
lowering kernel.perf_event_max_sample_rate to 71700
```

А еще быстрее? Способ есть!

1. `$ sudo sysctl kernel.perf_cpu_time_max_percent=70`
2. `$ sudo sysctl kernel.perf_event_max_sample_rate=200000`

Мы у цели!

```
$ PERF_RECORD_FREQ=200000 ./bin/perf-java-record-stack PID
Recording events for 15 seconds ...
...
[ perf record: Captured ... (2.961.252 samples) ]
```

2,961,252 / 15 ~ 197416 сэмплов/сек

Накладные расходы…

vmlinux = ядро linux

```
$ perf script
...
java ... native_write_msr (/.../vmlinux)
java ... Loop2.main (/tmp/perf-29621.map)
java ... native_write_msr (/.../vmlinux)
...
```

Что же делать?

Может, посмотрим на **аппаратные** события?

4. Используем PMU/PEBS в perf

Performance Monitoring Unit (PMU) Precise Event Based Sampling (PEBS)

Каждые X событий (**период**) мы записываем, где это произошло

Пример: сигнализируем каждую **20** (период) инструкцию процессора

А если итерация цикла – 5 инструкций?

А PMU сигнализирует каждую 20 инструкцию

```
mov aaa, bbbb
xor     %rdx, %rdx
L_START:
mov     $0x0(%rbx, %rdx),%r14
add     %r14, %r13
inc     %rdx
cmp     %rdx,100000000
jne     L_START
```

PMU

Инструкция: 18

А если итерация цикла – 5 инструкций?

А PMU сигнализирует каждую 20 инструкцию

PMU

Инструкция: 19

```
mov aaa, bbbb
xor    %rdx, %rdx
L_START:
mov    $0x0(%rbx, %rdx),%r14
add    %r14, %r13
inc    %rdx
cmp    %rdx,100000000
jne    L_START
```

А если итерация цикла – 5 инструкций?

А PMU сигнализирует каждую 20 инструкцию

```
mov aaa, bbbb
xor    %rdx, %rdx
L_START:
mov    $0x0(%rbx, %rdx),%r14
add    %r14, %r13
inc    %rdx
cmp    %rdx,100000000
jne    L_START
```

PMU

Инструкция: 20

А если итерация цикла – 5 инструкций?

А PMU сигнализирует каждую 20 инструкцию

```
mov aaa, bbbb
xor    %rdx, %rdx
L_START:
mov    $0x0(%rbx, %rdx),%r14
add    %r14, %r13
inc    %rdx
cmp    %rdx,100000000
jne    L_START
```

PMU

Инструкция: 1

А если итерация цикла – 5 инструкций?

А PMU сигнализирует каждую 20 инструкцию

```
mov aaa, bbbb
xor    %rdx, %rdx
L_START:
mov    $0x0(%rbx, %rdx),%r14
add    %r14, %r13
inc    %rdx
cmp    %rdx,100000000
jne    L_START
```

PMU

Инструкция: 19

А если итерация цикла – 5 инструкций?

А PMU сигнализирует каждую 20 инструкцию

```
mov aaa, bbbb
xor    %rdx, %rdx
L_START:
mov    $0x0(%rbx, %rdx),%r14
add    %r14, %r13
inc    %rdx
cmp    %rdx,100000000
jne    L_START
```

PMU

Инструкция: 20

А если итерация цикла – 5 инструкций?

```
xor     %rdx, %rdx
L_START:
mov     $0x0(%rbx, %rdx),%r14
add      %r14, %r13
inc     %rdx
cmp     %rdx,100000000
jne     L_START
```

PEBS будет нотифицировать каждую 20 / 5 = **4-ю итерацию** об одной и той же горячей инструкции

5 инструкций

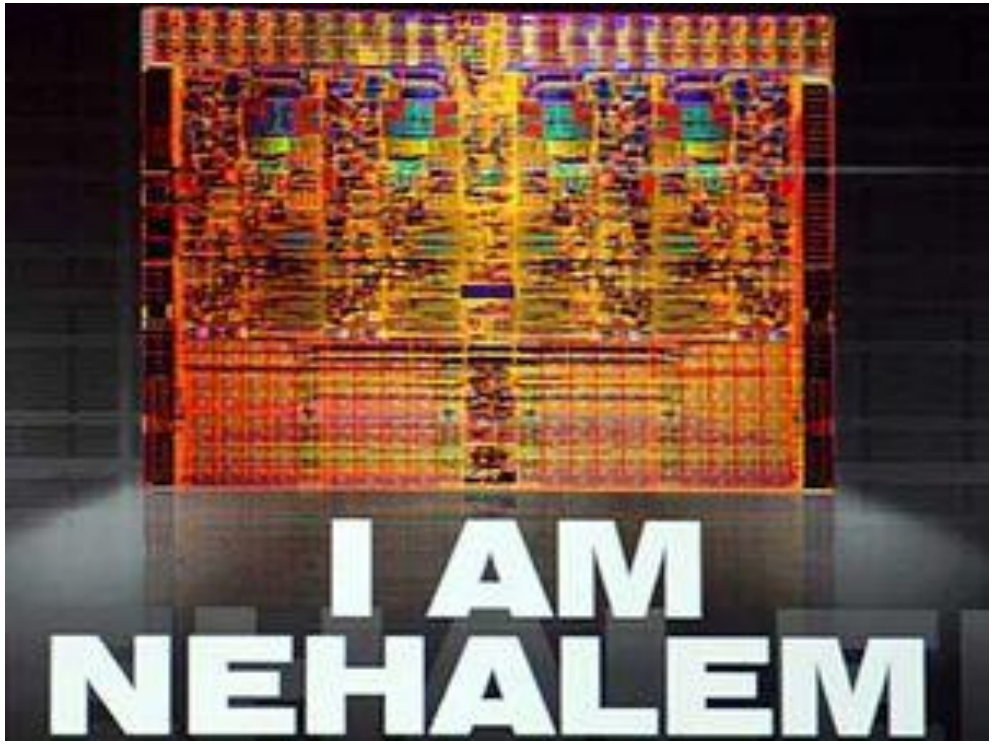
О каком количестве событий нужно сигнализировать?

Необходимо минимизировать вероятность наличия общего делителя между периодом и длиной цикла

=> Лучший выбор для периода – **простое** число
Например 23

Поддержка железа

Точно поддерживается с ***Nehalem*** (2009)



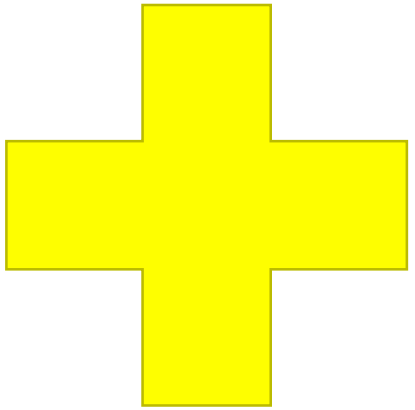
$ perf list

```
...
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]
...
```

Модифицируем скрипт perf-java-record-stack

```
...
sudo perf record -F $PERF_RECORD_FREQ ...
...
```

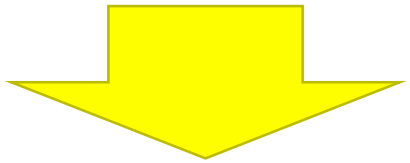
```
...
  -F, --freq=
      Profile at this frequency
...
  -e, --event=
      Select the PMU event
```



```
-c, --count=
    Event period
      to sample.
```

Модифицируем скрипт perf-java-record-stack

```
...
sudo perf record -F $PERF_RECORD_FREQ ...
...
```



```
...
sudo perf record -e cycles –c 100XX ...
...
```

Quiz: Какой период выбрать?

1. 10,000
2. 10,003
3. 10,007

Quiz: Какой период выбрать?

1. 10,000 <- 10,000 = 1000 x 10
2. 10,003 <- 10,003 = 1429 x 7
3. **10,007** <- 1230-е простое число

Получилось лучше!

```
$ ./bin/perf-java-record-stack-2 PID
Recording events for 15 seconds ...
...
[ perf record: Captured ... (4,597,167 samples) ]
```

4,597,167 / 15 ~= 306500 сэмплов/сек

Какую частоту мы получили?

300 000 сэмплов в секунду ~= 3мкс/сэмпл

или **33** сэмпла за 100 мкс

Сборка всего профиля 1000 / 33 ~= **30 секунд** – можно жить!

А если нужный код выполняется реже?

Если профилируемый код выполняется раз в 5 секунд?

Тогда профилирование займет 30 * 5 = **150** сек

А если нужный код выполняется реже?

Необходимы 45,000,000 сэмплов. Из которых *нужна лишь 1000 сэмплов*

99.998% собранных данных – **мусор**

А это замедляет работу всех последующих тулов и добавляет накладные расходы

Данных слишком много…

И что же делать?

А может железо умеет еще что-то полезное?

Умеет!

5. Intel Processor Trace – что это такое и как профилировать Java

Intel Processor Trace

Записывает результат выполнения каждой инструкции ветвления

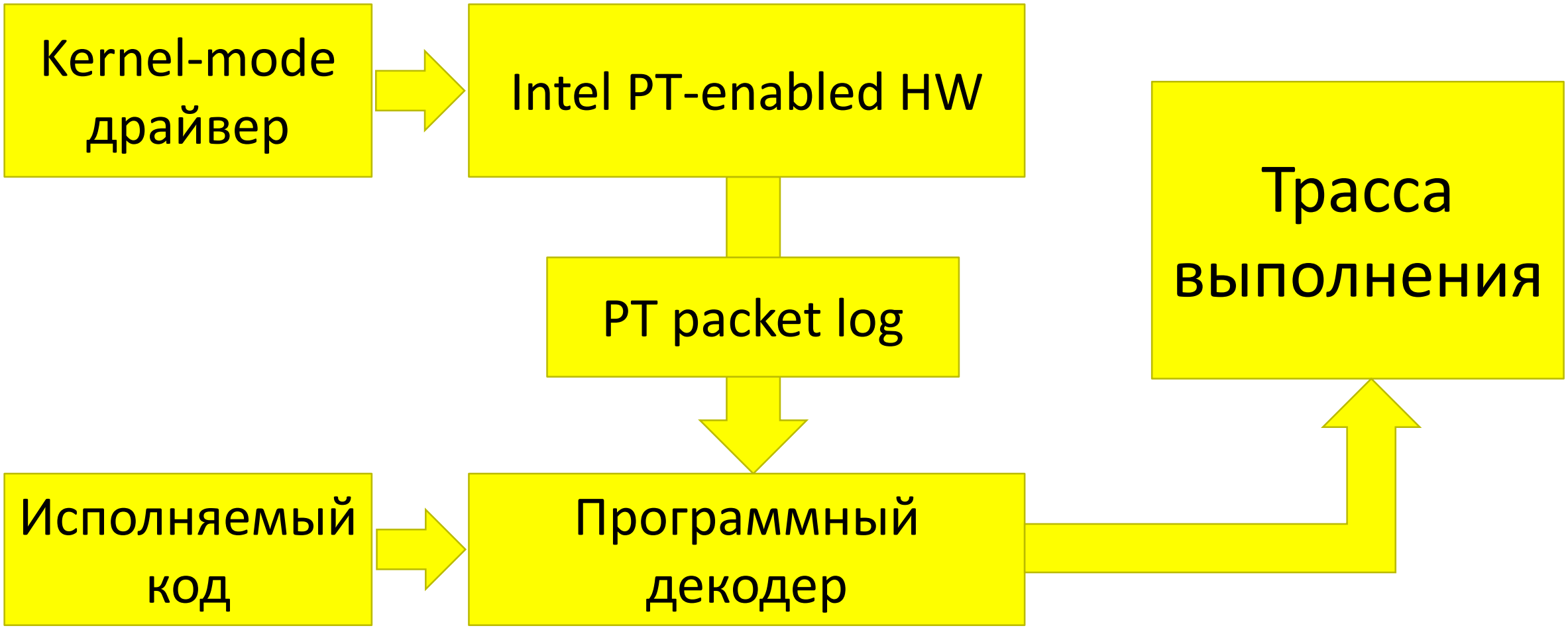
Теперь **control-flow** всей программы можно реконструировать!

Особенности Intel Processor Trace

- Полностью реализовано в «железе»
- Низкие накладные расходы (15%)
- Генерация трассы выполения

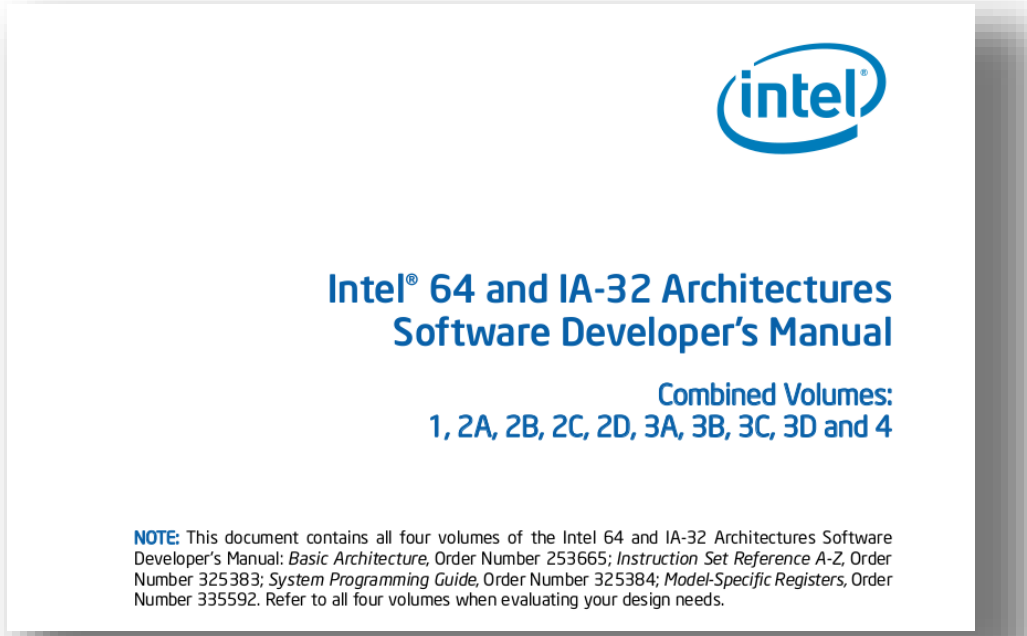
Источник: Harnessing Intel Processor Trace On Windows For Vuln Discovery - Richard Johnson

Intel Processor Trace – как это работает



Пакеты Processor Trace

- Пакет **TNT** (Taken/Not-Taken)
  - Перешли или нет по инструкции условного ветвления?
- Пакет **TIP** (Target Instruction Pointer)
  - Какую инструкцию сейчас начнем выполнять?
- Пакет **TSC** (Time Stamp Counter)
  - Текущее время в тактах процессора
- И многие другие…



Декодер ProcessorTrace уже есть в perf

Также, есть референсная реализация декодера от Intel:

https://github.com/01org/processor-trace

https://software.intel.com/en-us/articles/intel-sdm

Поддержка железа

Generation	Status	Release date
Broadwell 5th generation Core (i7-5XXX), Xeon v4	More overhead. No fine grained timing	2014 – 2015
Skylake or newer 6th generation Core (i7-6XXX), Xeon Scalable	Fine grained timing. Address filtering.	2015/2017
Goldmont (Apollo Lake, Denverton)	Fine grained timing. Address filtering.	2016

А что с поддержкой ОС (linux) ?

Версия ядра	Статус
4.1	Initial PT driver
4.2	Support for Skylake and Goldmont
4.3	Initial user tools support in Linux perf
4.5	Support for JIT decoding using agent
4.6	Bug fixes. Support address filtering
4.8	Bug fixes
4.10	Bug fixes. Support for PTWRITE and power tracing

Попробуем запустить!

```
$ perf record -e intel_pt/.../ java -cp . Test
[ perf record: Captured ... 8.313 MB perf.data ]
```

```
class Test {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

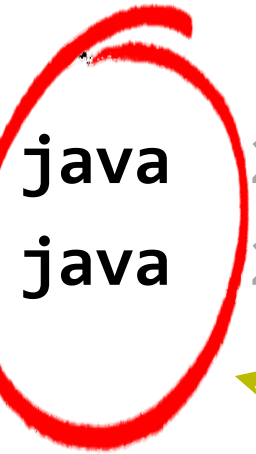
Пример запуска на Skylake-X (Xeon or i9)

$ perf script --ns

```
java  2926 ... 5658.917868168 ... 7f505544c0f0 free@plt ... => 7f50562834c0 free ...
java  2926 ... 5658.917868190 ... 7f50562834f1 free ... =>     7f50562834ff free ...
java  2926 ... 5658.917868190 ... 7f5056283504 free ... =>     7f505627f120 _int_free ...
java  2926 ... 5658.917868395 ... 7f505627f174 _int_free ... => 7f505627f250 _int_free ...
java  2926 ... 5658.917868395 ... 7f505627f25e _int_free ... => 7f505627f3e0 _int_free ...
java  2926 ... 5658.917868395 ... 7f505627f3fc _int_free ... => 7f505627f408 _int_free ...
java  2926 ... 5658.917868395 ... 7f505627f40b _int_free ... => 7f505627f264 _int_free ...
```

Разберем подробнее

```
java  2926 ... 9178190 ... 83504 free ...       => 7f120 _int_free
java  2926 ... 7868395 ... 7f174 _int_free ... => 7f250 _int_free
```



Имя процесса (например, java)

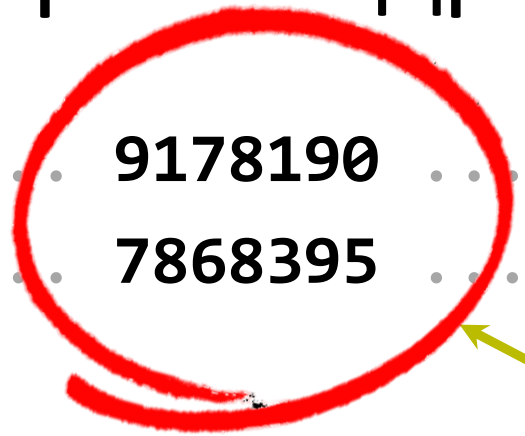
Разберем подробнее

```
java  2926 ... 9178190 ... 83504 free ...       => 7f120 _int_free
java  2926 ... 7868395 ... 7f174 _int_free ... => 7f250 _int_free
```

Идентификатор процесса (PID)

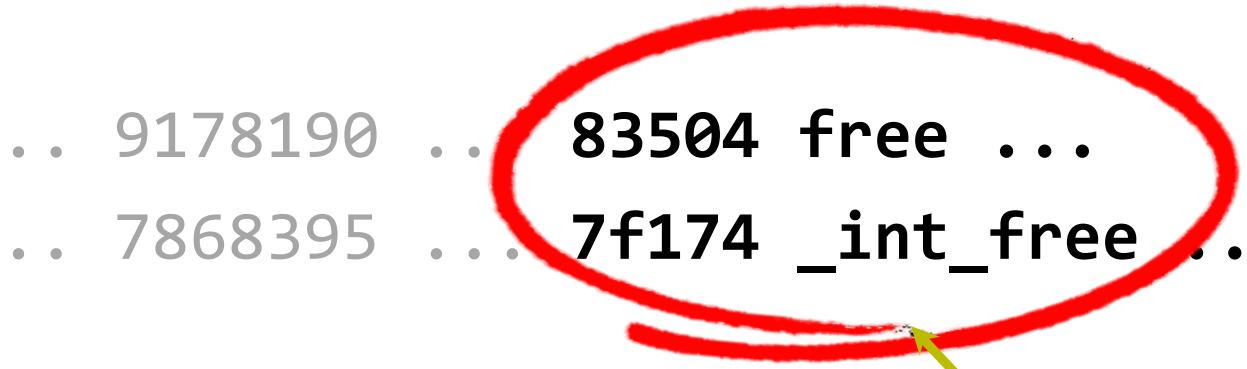
Разберем подробнее

```
java  2926 ... 9178190 ... 83504 free ...       => 7f120 _int_free
java  2926 ... 7868395 ... 7f174 _int_free ... => 7f250 _int_free
```



Разберем подробнее

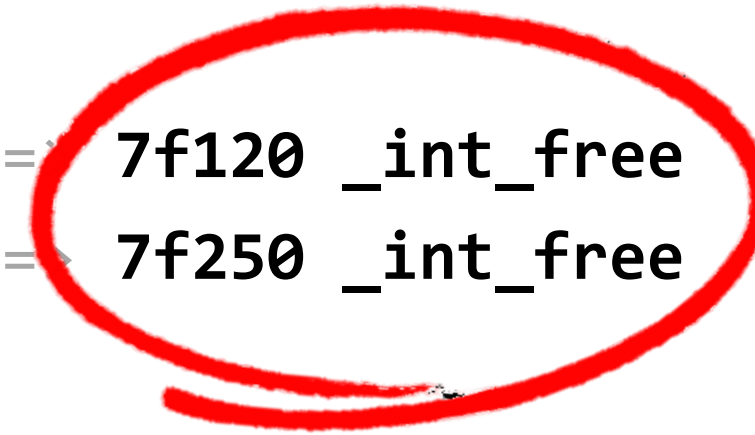
```
java  2926 ... 9178190 ... 83504 free ...          => 7f120 _int_free
java  2926 ... 7868395 ... 7f174 _int_free ... => 7f250 _int_free
```



Адрес инструкции – источника и название метода

Разберем подробнее

```
java  2926 ... 9178190 ... 83504 free ...       => 7f120 _int_free
java  2926 ... 7868395 ... 7f174 _int_free ... => 7f250 _int_free
```



Адрес инструкции **назначения** и название метода

Но есть нюанс…

```
$ perf script --ns > out.data

$ ls -lah out.data

    ... 1.9G ... out.data
```

Perf.... Как много в этом звуке

Perf отличный тул, но в таком режиме данных будет *слишком* много

Решение: нужно запускать perf только в нужные моменты времени

Сделаем libperf.so из perf!

И будем запускать профилирование через jni!

Публичный JNI-API libperf.so

```
package ru.raiffeisen;

public class PerfPtProf {
  public static native void init(int cntdwn);
  public static native void start();
  public static native void stop();
}
```

Интерфейс libperf

- **`init()`** – создает поток P, в котором живет профилировщик

Интерфейс libperf

- `init()` – создает поток P, в котором живет профилировщик
- **`start()`** – инициирует начало профилирования, пробуждая поток P

Интерфейс libperf

- `init()` – создает поток P, в котором живет профилировщик
- `start()` – инициирует начало профилирования, пробуждая поток P
- **`stop()`** – останавливает профилирование, устанавливая флаг остановки S

Получение реалистичных данных

```
class ClassToProfile {
   void profileIt() {
      PerfPtProf.start();
      Executor.doSomething();
      PerfPtProf.stop();
   }
}
```

Получение реалистичных данных

```
class Executor {
  static void doSomething() { ... }
  static {
    SomethingReallyHeavy();
  }
}
```

Получение реалистичных данных

Необходимо уметь пропускать несколько первых вызовов start()

=> Число пропускаемых вызовов конфигурируется через аргумент метода init

А вызовы jni будем вставлять с помощью java-агента!

Агент читает аргументы, передаваемые через -DXXX=YYYY, и вставляет вызовы профилировщика в нужный места

```
$ java                                          \
    -javaagent:path_tp_agent.jar                \
    -DPROFILE_CLASS=ru.raiffeisen.OrderSender   \
    -DPROFILE_METHOD=sendFewOrders      ...
```

А как (lib)perf будет узнавать о java-методах?

Perf ищет файл /tmp/perf-PID.map, в который записаны адреса и имена функций

Решение:

Добавляем JVMTI-агент, который записывает информацию по адресах методов

Итого, full-stack (без JavaScript ☺)

1. **libperf.so** – урезанная версия perf, профилирующая через intel processor trace

Итого, full-stack (без JavaScript ☺)

1. **libperf.so** – урезанная версия perf, профилирующая через intel processor trace
    1. Суммирует время выполнения каждой функции и печатает TOP
    2. Записывает файл perf.data.log с трассой выполнения
    3. JVMTI-агент для записи информации о именах функций

Итого, full-stack (без JavaScript ☺)

1. **libperf.so** – урезанная версия perf, профилирующая через intel processor trace
    1. Суммирует время выполнения каждой функции и печатает TOP
    2. Записывает файл perf.data.log с трассой выполнения
    3. JVMTI-агент для записи информации о именах функций
2. **Java-агент** для вставки кода вызова профилировщика

Тренируемся на кошках

Профилируем декодирование JSON’а библиотекой GSON от Google

```
private static Gson gson = new Gson();
private static String generate_json(int len) { ... }
private static Map decode(String json) {
    Map result = gson.fromJson(json, Map.class);
    return result;
}
public static void main(String[] args ) {
    for (int i = 0; i < 20000; ++i) {
        String json = generate_json(i);
        Map res = decode(json);
        if (res.size() != i) { System.out.println("???"); }
    }
}
```

Как запустить профилирование?

```
$ java -agentlib:perf=                                          \
       -javaagent:perf-instrumenter-....jar                     \
       -DTRIGGER_COUNTDOWN=10000                                \
       -DTRIGGER_METHOD=decode -DTRIGGER_CLASS=ru/raiffeisen/App  \
-cp target/json-tester-....jar:.../gson-2.8.2.jar               \
    ru.raiffeisen.App
```

Как запустить профилирование?

```
$ java -agentli                                                  \
       -javaage                                                  \
       -DTRIGGE                                                  \
       -DTRIGGE                                  ru/raiffeisen/App  \
-cp target/json                                  ar              \
    ru.raiffeis
```

Что за параметры?

```
$ java -agentlib:perf=                                          \
       -javaagent:perf-instrumenter-....jar                     \
       -DTRIGGER_COUNTDOWN=10000                                \
       -DTRIGGER_METHOD=decode -DTRIGGER_CLASS=ru/raiffeisen/App  \
-cp target/json-tester-....jar:.../gson-2.8.2.jar               \
    ru.raiffeisen.App
```

Подключение libperf/JVMTI-агента

Что за параметры?

```
$ java -agentlib:perf=                                          \
       -javaagent:perf-instrumenter-....jar                     \
       -DTRIGGER_COUNTDOWN=10000                                \
       -DTRIGGER_METHOD=decode -DTRIGGER_CLASS=ru/raiffeisen/App  \
-cp target/json-tester-....jar:.../gson-2.8.2.jar               \
    ru.raiffeisen.App
```

Подключаем инструментирующий агент

Что за параметры?

```
$ java -agentlib:perf=                                    \
       -javaagent:perf-instrumenter-....jar               \
       -DTRIGGER_COUNTDOWN=10000                          \
       -DTRIGGER_METHOD=decode -DTRIGGER_CLASS=ru/raiffeisen/App  \
-cp target/json-tester-....jar:.../gson-2.8.2.jar         \
    ru.raiffeisen.App
```

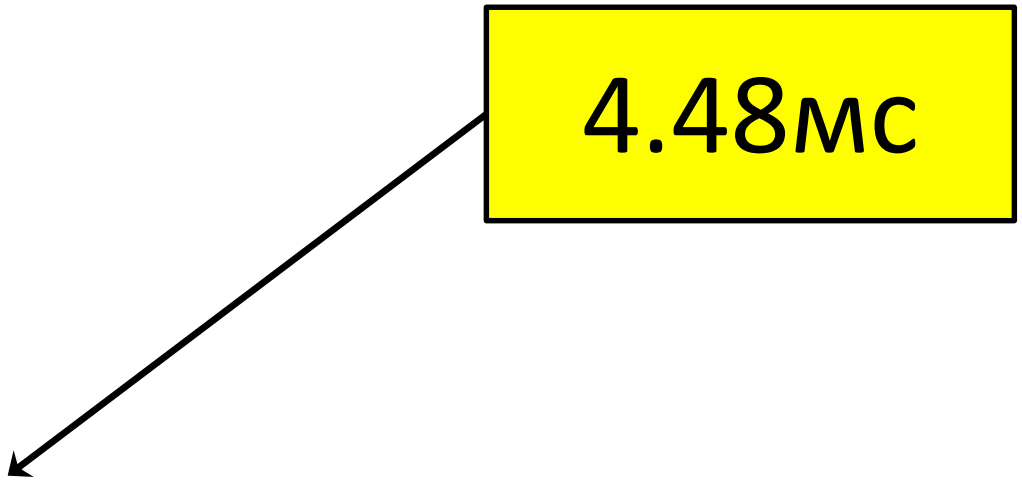
Указываем какой метод класса профилировать

Что за параметры?

```
$ java -agentlib:perf=                                          \
       -javaagent:perf-instrumenter-....jar                     \
       -DTRIGGER_COUNTDOWN=10000                                \
       -DTRIGGER_METHOD=decode -DTRIGGER_CLASS=ru/raiffeisen/App  \
-cp target/json-tester-....jar:.../gson-2.8.2.jar               \
    ru.raiffeisen.App
```

Все как обычно - ничего интересного ☺

```
$ java ...

[ perf record: Woken up 63 times to write data ]
[ perf record: Captured and wrote 3.986 MB perf.data ]
Record done
Dumping symbols
Processed: 423 symbols
Processing top
…
Total for all functions: 4,482,542ns
```



Два слова про корректность профиля

```
$ head -n1 perf.data.log
      java ... 337915.451443257 ...

$ tail -n1 perf.data.log
      java ... 337915.455925799 ...
```

455,925,799 – 451,443,257 =

= 4,482,542нс

TOP в деталях

```
1 ... L.../MapTypeAdapterFactory$Adapter;::read@...->
                                                  2,456,964ns
2 ... L.../TypeAdapterRuntimeTypeWrapper;::read@...->
                                                  865,848ns
3 ... L.../LinkedTreeMap;::rebalance@...->        324,172ns
4 ... unknown@unknown  ->                         296,655ns
5 ... L.../JsonReader;::doPeek@... ->             189,315ns
6 ... L.../JsonReader;::nextNonWhitespace@... ->  147,243ns
7 ... jshort_disjoint_arraycopy@... ->            141,356ns
```

TOP в деталях

```
1 ... L.../MapTypeAdapterFactory$Adapter;::read@...->          2,456,964ns

2 ... L.../TypeAdapterRuntimeTypeWrapper;::read@...->          865,848ns

3 ... L.../LinkedTreeMap;::rebalance@...->                     324,172ns

4 ... unknown@unknown  ->                                      296,655ns

5 ... L.../JsonReader;::doPeek@... ->                          189,315ns

6 ... L.../JsonReader;::nextNonWhitespace@... ->               147,243ns

7 ... jshort_disjoint_arraycopy@... ->                         141,356ns
```

Теперь можно точно увидеть, что выполнялось!

…

... 280388529: ... ...53f2cf Lcom/...;::doPeek (...) => ... Lcom/...;::nextNonWhitespace (...)

... 280388542: ... ...4e23fc Lcom/...;::nextNonWhitespace (...) => ... Lcom/...;::doPeek (...)

... 280388542: ... ...53f2d7 Lcom/...;::doPeek (...) => ... Lcom/...;::doPeek (...)

…

... 280388562: ... ...53f749 Lcom/...;::doPeek (...) => ... Lcom/...;::doPeek (...)

... 280388562: ... ...53f2b4 Lcom/...;::doPeek (...) => ... Lcom/…;::read (...)

... 280388566: ... ...5a39f9 Lcom/…;::read (...) => ... Lcom/…;::read (...)

... 280388569: ... ...5a2aeb Lcom/…;::read (...) => ... Lcom/...;::nextString (...)

... 280388582: ... ...587b1b Lcom/...;::nextString (...) => ... Lcom/...;::nextString (...)

…

Теперь можно точно увидеть, что выполнялось!

…

… 280388529: … …53f2cf **Lcom/…;::doPeek (…) => … Lcom/…;::nextNonWhitespace** (…)

… 280388542: … …4e23fc Lcom/…;::nextNonWhitespace (…) => … Lcom/…;::doPeek (…)

… 280388542: … …53f2d7 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

…

… 280388562: … …53f749 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

… 280388562: … …53f2b4 Lcom/…;::doPeek (…) => … Lcom/…;::read (…)

… 280388566: … …5a39f9 Lcom/…;::read (…) => … Lcom/…;::read (…)

… 280388569: … …5a2aeb Lcom/…;::read (…) => … Lcom/…;::nextString (…)

… 280388582: … …587b1b Lcom/…;::nextString (…) => … Lcom/…;::nextString (…)

…

Теперь можно точно увидеть, что выполнялось!

…

… 280388529: … …53f2cf Lcom/…;::doPeek (…) => … Lcom/…;::nextNonWhitespace (…)

… 280388542: … …4e23fc **Lcom/…;::nextNonWhitespace (…) => … Lcom/…;::doPeek** (…)

… 280388542: … …53f2d7 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

…

… 280388562: … …53f749 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

… 280388562: … …53f2b4 Lcom/…;::doPeek (…) => … Lcom/…;::read (…)

… 280388566: … …5a39f9 Lcom/…;::read (…) => … Lcom/…;::read (…)

… 280388569: … …5a2aeb Lcom/…;::read (…) => … Lcom/…;::nextString (…)

… 280388582: … …587b1b Lcom/…;::nextString (…) => … Lcom/…;::nextString (…)

…

Теперь можно точно увидеть, что выполнялось!

…

… 280388529: … …53f2cf Lcom/…;::doPeek (…) => … Lcom/…;::nextNonWhitespace (…)

… 280388542: … …4e23fc Lcom/…;::nextNonWhitespace (…) => … Lcom/…;::doPeek (…)

… 280388542: … …53f2d7 **Lcom/…;::doPeek (…) => … Lcom/…;::doPeek** (…)

…

… 280388562: … …53f749 **Lcom/…;::doPeek (…) => … Lcom/…;::doPeek** (…)

… 280388562: … …53f2b4 Lcom/…;::doPeek (…) => … Lcom/…;::read (…)

… 280388566: … …5a39f9 Lcom/…;::read (…) => … Lcom/…;::read (…)

… 280388569: … …5a2aeb Lcom/…;::read (…) => … Lcom/…;::nextString (…)

… 280388582: … …587b1b Lcom/…;::nextString (…) => … Lcom/…;::nextString (…)

…

Теперь можно точно увидеть, что выполнялось!

…

… 280388529: … …53f2cf Lcom/…;::doPeek (…) => … Lcom/…;::nextNonWhitespace (…)

… 280388542: … …4e23fc Lcom/…;::nextNonWhitespace (…) => … Lcom/…;::doPeek (…)

… 280388542: … …53f2d7 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

…

… 280388562: … …53f749 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

… 280388562: … …53f2b4 **Lcom/…;::doPeek (…) => … Lcom/…;::read** (…)

… 280388566: … …5a39f9 Lcom/…;::read (…) => … Lcom/…;::read (…)

… 280388569: … …5a2aeb Lcom/…;::read (…) => … Lcom/…;::nextString (…)

… 280388582: … …587b1b Lcom/…;::nextString (…) => … Lcom/…;::nextString (…)

…

Теперь можно точно увидеть, что выполнялось!

…

… 280388529: … …53f2cf Lcom/…;::doPeek (…) => … Lcom/…;::nextNonWhitespace (…)

… 280388542: … …4e23fc Lcom/…;::nextNonWhitespace (…) => … Lcom/…;::doPeek (…)

… 280388542: … …53f2d7 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

…

… 280388562: … …53f749 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

… 280388562: … …53f2b4 Lcom/…;::doPeek (…) => … Lcom/…;::read (…)

… 280388566: … …5a39f9 **Lcom/…;::read (…) => … Lcom/…;::read** (…)

… 280388569: … …5a2aeb Lcom/…;::read (…) => … Lcom/…;::nextString (…)

… 280388582: … …587b1b Lcom/…;::nextString (…) => … Lcom/…;::nextString (…)

…

Теперь можно точно увидеть, что выполнялось!

…

… 280388529: … …53f2cf Lcom/…;::doPeek (…) => … Lcom/…;::nextNonWhitespace (…)

… 280388542: … …4e23fc Lcom/…;::nextNonWhitespace (…) => … Lcom/…;::doPeek (…)

… 280388542: … …53f2d7 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

…

… 280388562: … …53f749 Lcom/…;::doPeek (…) => … Lcom/…;::doPeek (…)

… 280388562: … …53f2b4 Lcom/…;::doPeek (…) => … Lcom/…;::read (…)

… 280388566: … …5a39f9 Lcom/…;::read (…) => … Lcom/…;::read (…)

… 280388569: … …5a2aeb **Lcom/…;::read (…) => … Lcom/…;::nextString (…)**

… 280388582: … …587b1b Lcom/…;::nextString (…) => … Lcom/…;::nextString (…)

…

Какую же частоту мы получили?

Точность сборки таймингов порядка десятков **наносекунд**

Т.е. миллионы синтетических сэмплов в секунду!

Какую же частоту мы получили?

С таким тулом можно **анализировать** производительность крайне **точно**!

Теперь можно реконструировать control-flow участка кода с точными таймингами

Резюме

1. Оценили важность 100мкс
2. Выбрали профилировщик
3. Научили perf собирать подробный профиль java-приложения
4. Воспользовались событиями PMU/PEBS в perf
5. Применили Intel Processor Trace для профилирования Java

А ведь это может быть полезно не только нам!

Уже выложил на github!

http://github.com/RainM/rperf

Пока еще скорее R&D, чем Production-ready

Если нужно профилировать относительно небольшие участки приложения (скажем, до десятков миллисекунд), использование Intel Processor Trace будет крайне полезным

Q&A

Sergey Melnikov

Sergey.V.Melnikov@raiffeisen.ru

Профилируем с точностью до инструкций и микросекунд