Профилируем черного лебедя с помощью Intel Processor Trace

или

что делать если иногда код выполняется 20 мс вместо 2мс

Сергей Мельников

Райффайзен БАНК

Хардкора будет много!

Райффайзен БАНК

Обо мне

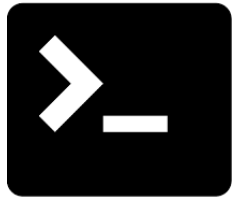
- Совсем **не Enterprise** Java-разработчик, только **Java SE**
- В прошлом Compiler **Performance** Engineer @ **Intel** Compiler Lab
- Контрибьютил в Android (AOSP), FreeBSD, GCC, ocperf, jmh, …

Agenda

1. Профилируем небольшой участок кода с помощью perf’a
2. Intel Processor Trace – что это такое и как профилировать Java-приложения
3. Практика - профилируем разбор JSON’a
4. А что если иногда код работает медленнее? Почему такое бывает? И как это профилировать?
5. И при чем здесь черные лебеди? ☺

4 Языка





1 секунда = 1000 миллисекунд = 1000000 микросекунд

Предметная область

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

Но некоторые участки кода “более критичны” к быстродействию!

Самый критичный к скорости код

Выполняется сотни микросекунд и напрямую влияет на финансовый результат

100мкс могут стоить миллион рублей на каждой сделке

Пример: Московская Биржа

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

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

MOSCOW EXCHANGE

Итак, задача

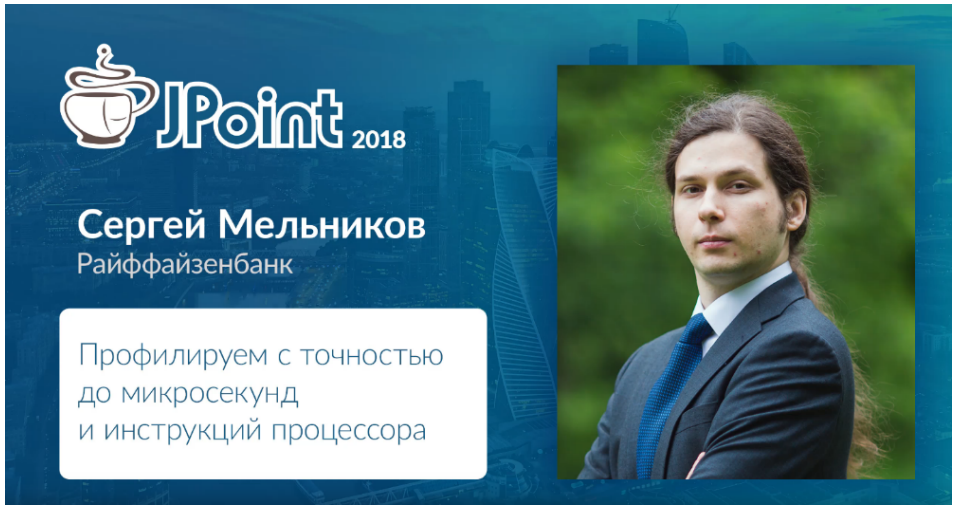
Нужно понять, на что тратится время при выполнении метода `sendOrdersToMoex`

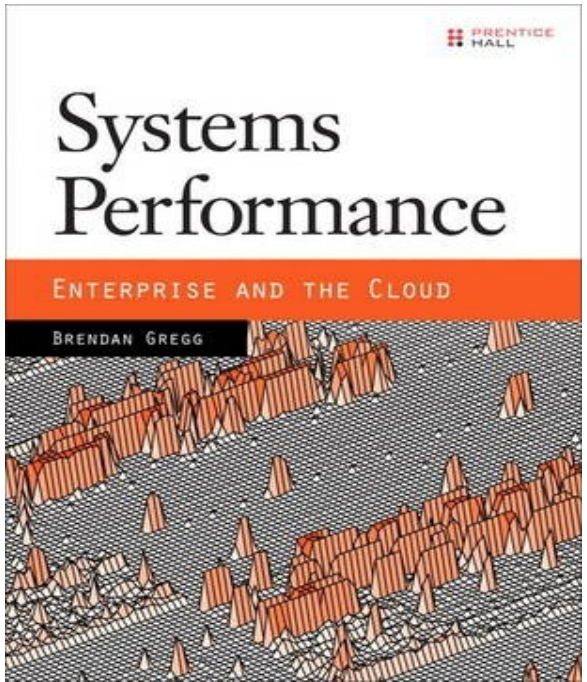
Характерное время выполнение метода 100мкс

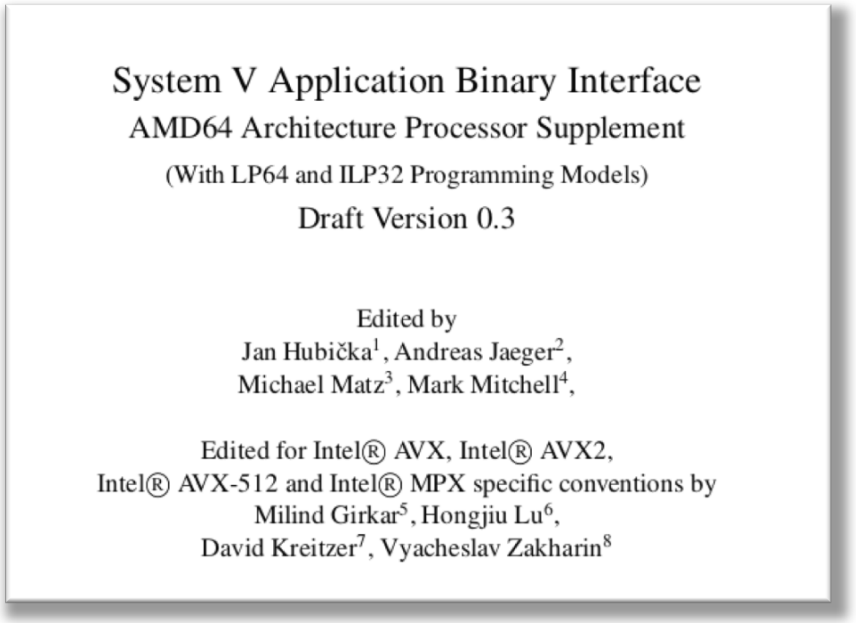
1. Учимся собирать подробный профиль небольшого участка кода с помощью perf’a

Подробности

1. Конференция JPoint 2018, Сергей Мельников – Профилируем с точностью до инструкций и микросекунд
2. Brendan Gregg – Systems performance
3. 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]

Что осложняет профилирование?

- Выполняется редко и очень быстро
- Участок кода сложно выделить в микробенчмарк
- Код затрагивает сеть
- Хочется получить профиль максимально похожий на реальный

Сэмплирующие профилировщики – наш выбор

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

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)
```

Perf – ничего не знает про Java

Perf мыслит *исключительно* в терминах стандартов, принятых в разработке на C/C++

Что нам нужно от perf’а

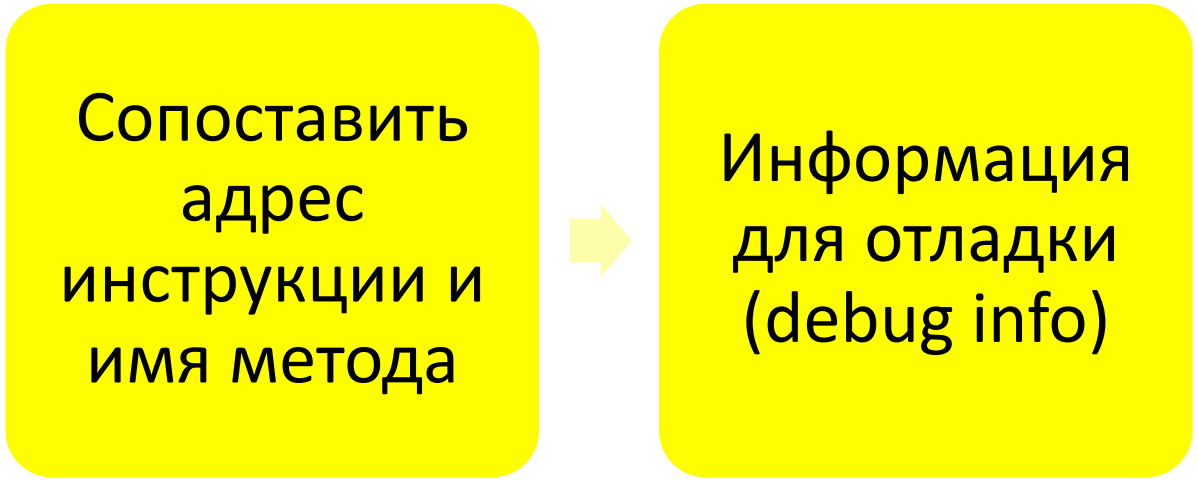
1. Сопоставить адрес инструкции названию метода
2. Получить стек вызовов в терминах имен методов

1. Сопоставление адресов инструкций

Сопоставить адрес инструкции и имя метода

Проблема

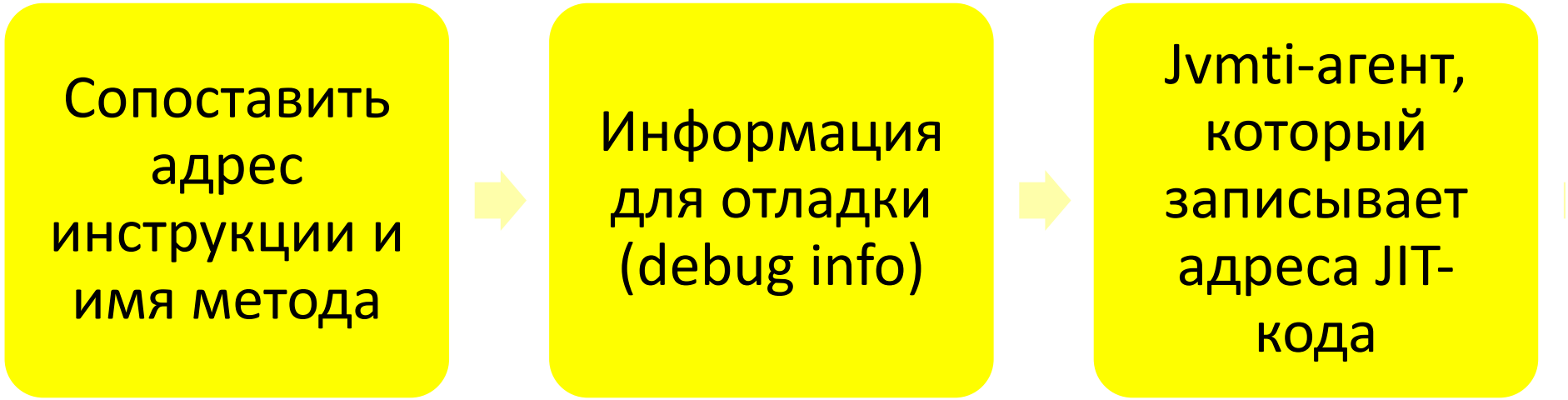
1. Сопоставление адресов инструкций



Проблема

Как обычно решается в мире C/C++

1. Сопоставление адресов инструкций

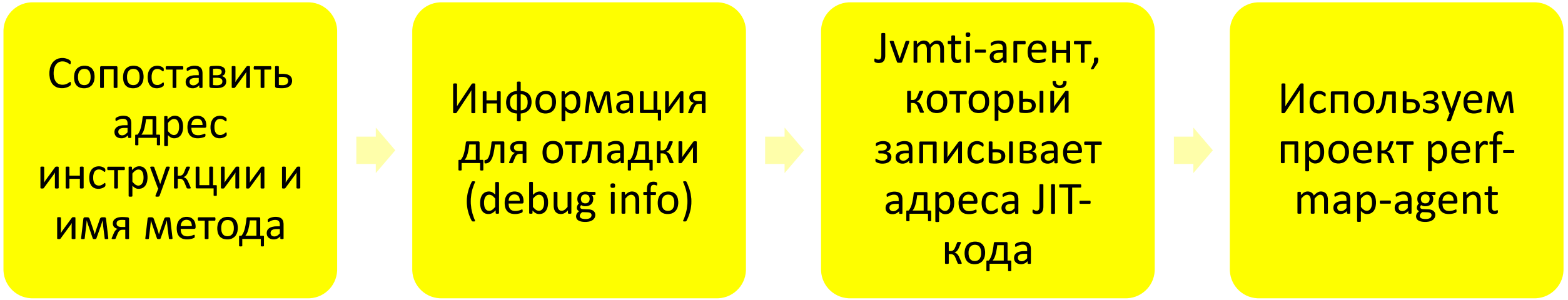


Проблема

Как обычно решается в мире C/C++

Что там делать в Java

1. Сопоставление адресов инструкций



Проблема

Как обычно решается в мире C/C++

Что там делать в Java

Конкретное действие

2. Получение стека вызовов

Получить стек вызовов

Проблема

2. Получение стека вызовов

Получить стек вызовов

Регистр rbp позволяет понять адрес вызвавшего метода

Проблема

Как обычно решается в мире C/C++

2. Получение стека вызовов

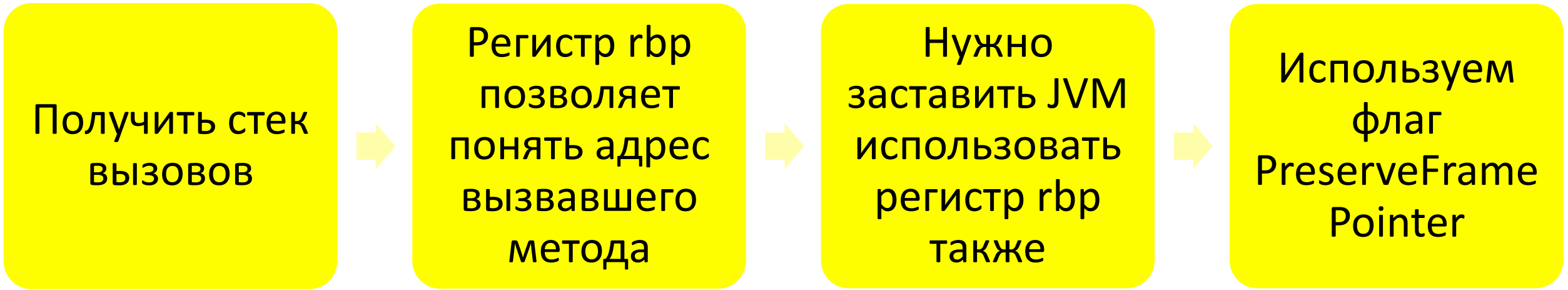
Получить стек вызовов → Регистр rbp позволяет понять адрес вызвавшего метода → Нужно заставить JVM использовать регистр rbp также

Проблема

Как обычно решается в мире C/C++

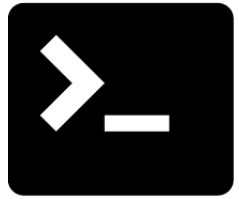
Что там делать в Java

2. Получение стека вызовов



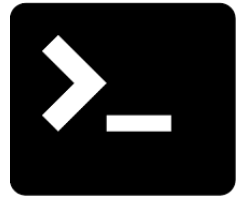
Запускаем perf

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

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

$ perf record ...

$ perf script ...
```



Смотрим профиль

```
$ 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)
```



А теперь максимальная скорость!

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

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

В нашем случае: сигнализируем каждую **2.000.003** (период) такт процессора

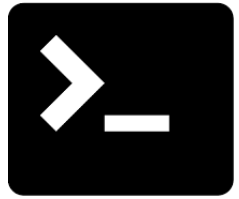
Период задается с помощью флага “-c“

Добавляем количество событий

```
...
sudo perf record -e cpu-cycles -c 10007 ...
...
```

Не забываем про

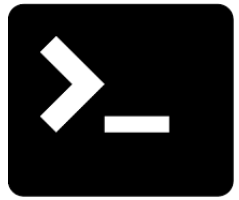
```
$ sudo sysctl kernel.perf_cpu_time_max_percent=70
$ sudo sysctl kernel.perf_event_max_sample_rate=300000
```



Получилось лучше!

```
$ perf record -e cpu-cycles -c 10007 –p PID – sleep 15
...
[ 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 1000 сэмплов ~= **30 выполнений профилируемого кода** – можно жить!

Собираем нужную 1000 сэмплов. А кроме них, мы собираем 9,000,000 сэмплов. Из которых *нужна лишь 1000 сэмплов*

99.98% собранных данных – **мусор**

А это замедляет работу всех последующих тулов и добавляет накладные расходы

I have a dream

А вот бы железо само собирало профиль!

«Железо» умеет само собирать профиль!

2. 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!

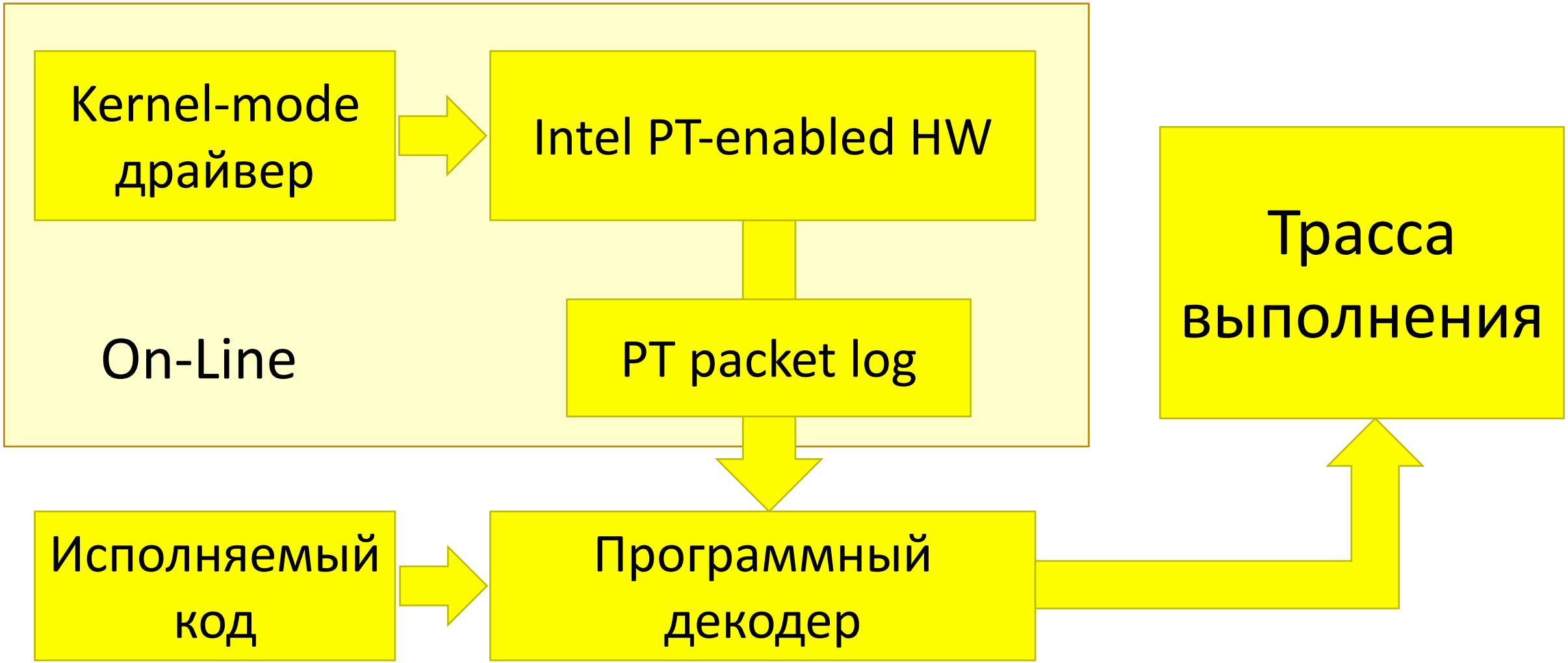
Поддержка железа

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

Intel Processor Trace – как это работает



Perf уже поддерживает Intel Processor Trace

Может, будем использовать готовое?

Perf.... Как много в этом звуке

Perf отличный тул, но это отдельная *большая* зависимость, да еще и с лицензией GPL

Решение: нужно сделать свой небольшой perf с blackjack’ом и хорошей лицензией

Сделать свой perf? А что для этого нужно?

1. Запустить профилировщик
2. Сохранить профиль/трассу
3. Декодировать трассу в терминах адресов
4. Декодировать имена методов
5. Декодировать время элементов трассы

Магия Linux

Можно посмотреть все исходные коды и сделать лучше!

Perf – обычное приложение. Никакой магии. Просто добавь... *правильный* интерфейс ядра

Делаем свой perf

Используем вызов ядра perf_event_open

```
int perf_event_open(
        ① struct perf_event_attr *attr,
        ② pid_t pid,
        ③ int cpu,
        ④ int group_fd,
        ⑤ unsigned long flags);
```

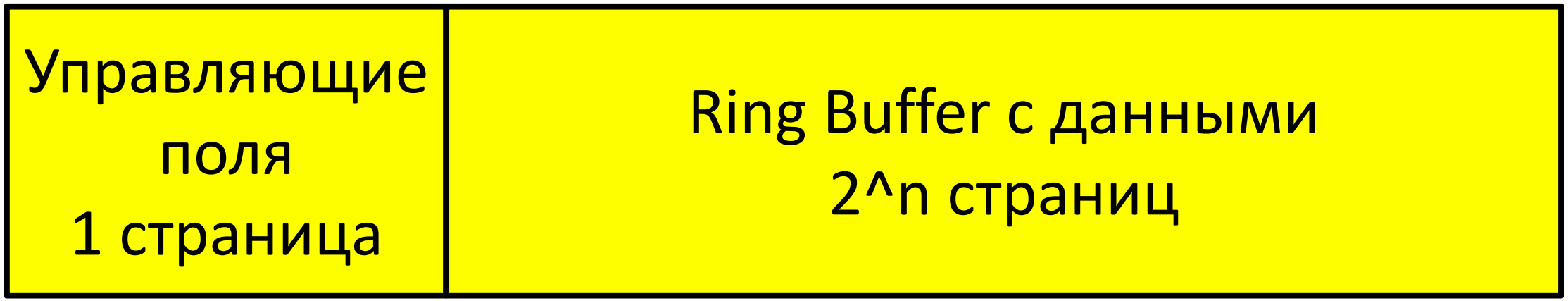
1. Запустить профилировщик - **умеем**
2. Сохранить профиль/трассу
3. Декодировать трассу в терминах адресов
4. Декодировать имена методов
5. Декодировать время элементов трассы

Профилировщик создали. А откуда получать данные?

Для этого нужно выделить память, в которую ядро будет писать данные

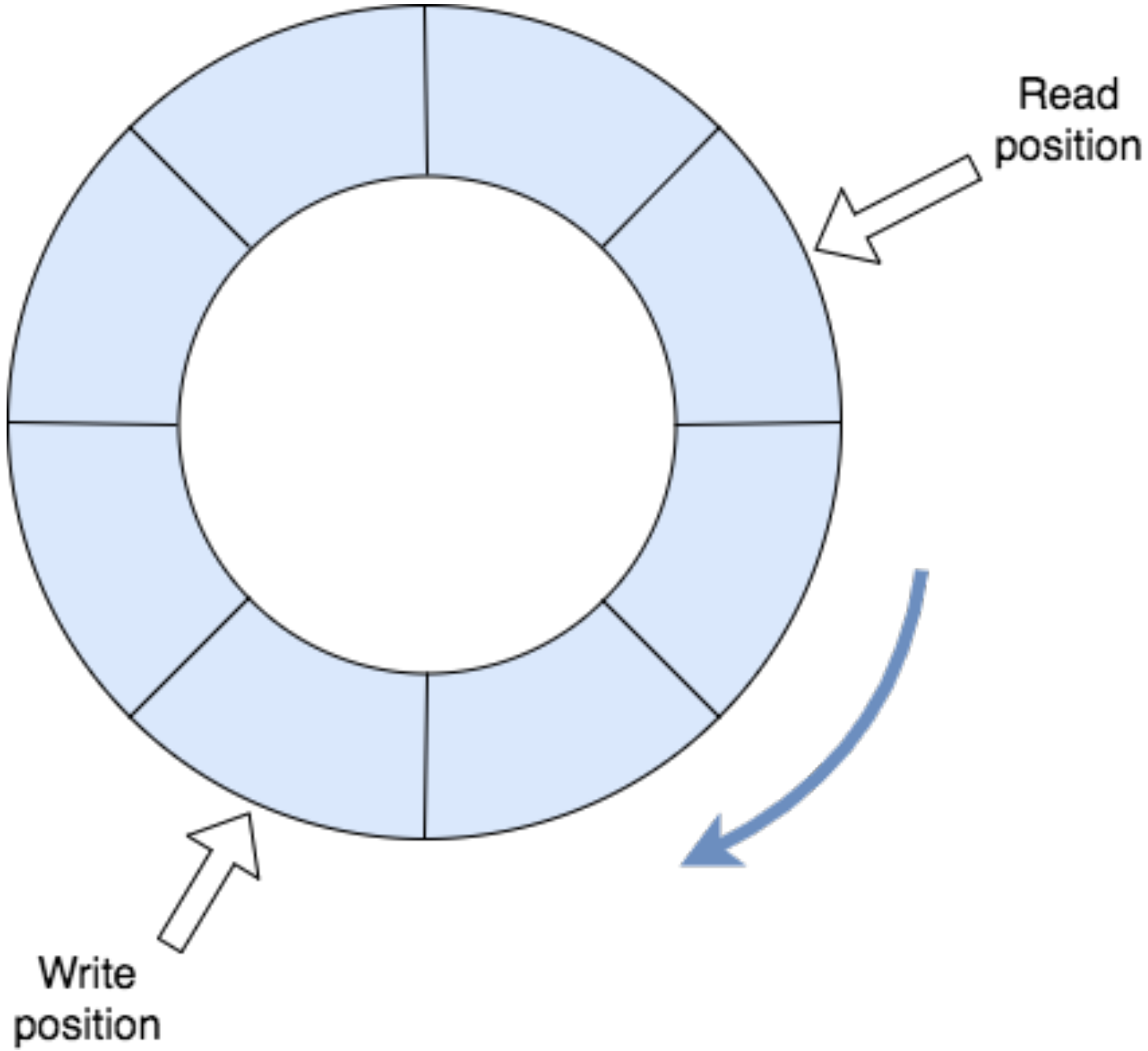
```
perf_fd = ::perf_event_open(...);

size_of_buffer = (1 + num_of_pages) * page_size;

perf_event_mmap_page* buffer =

    ::mmap(NULL, size_of_buffer, ..., perf_fd, ...);
```

Что будет в выделенной памяти?

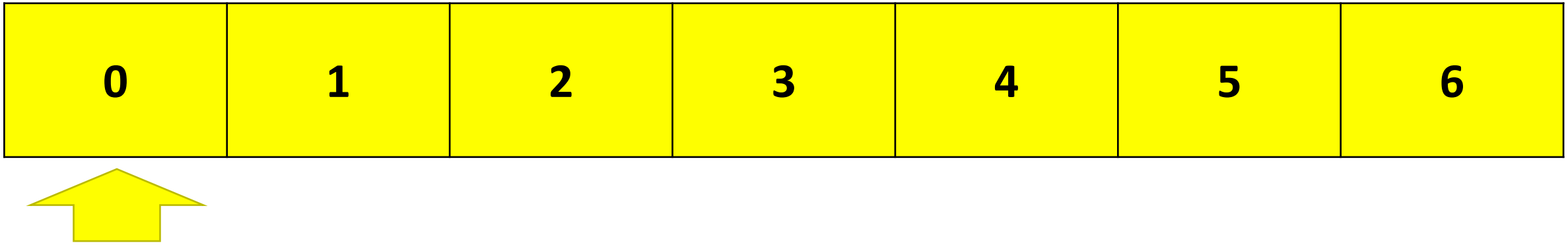


Ring Buffer – что это такое?

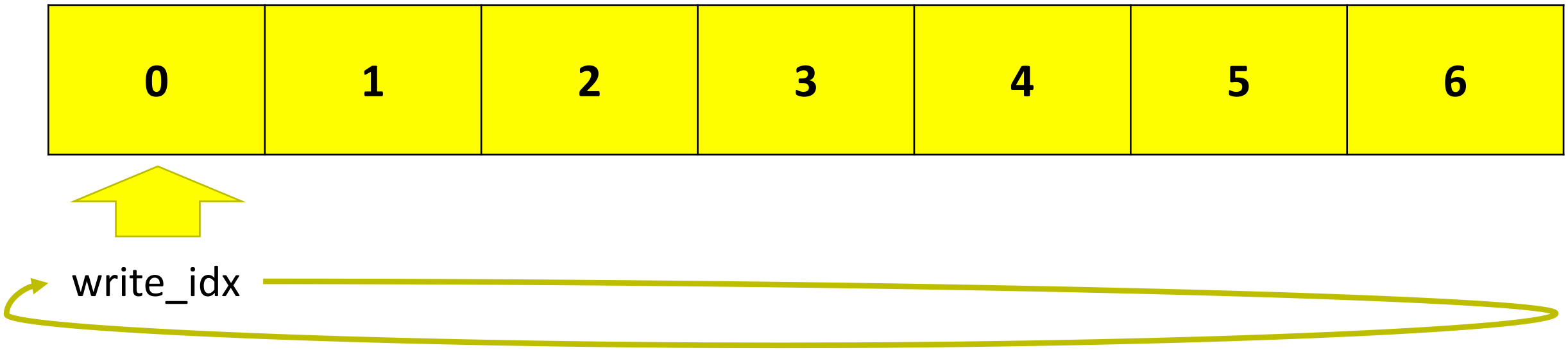
- Буфер фиксированного размера, ведущий себя, будто после последнего элемента сразу идет первый
- Используется для буферизации потоков данных
- Может быть толерантен к тому, что читатель пропустит данные


Read
position
Write
position

Пишем данные в массив



Пишем данные в массив



Позиция записи

0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 0 -> 1...

Как определить позицию в массиве?

Арифметика по модулю

```
0   % 7 = 0
  ...
6   % 7 = 6
7   % 7 = 0
8   % 7 = 1
  ...
100 % 7 = 2
  ...
```

Если сделать отдельные переменные для позиции чтения и позиции записи, можно будет понять, пропустил ли читатель данные

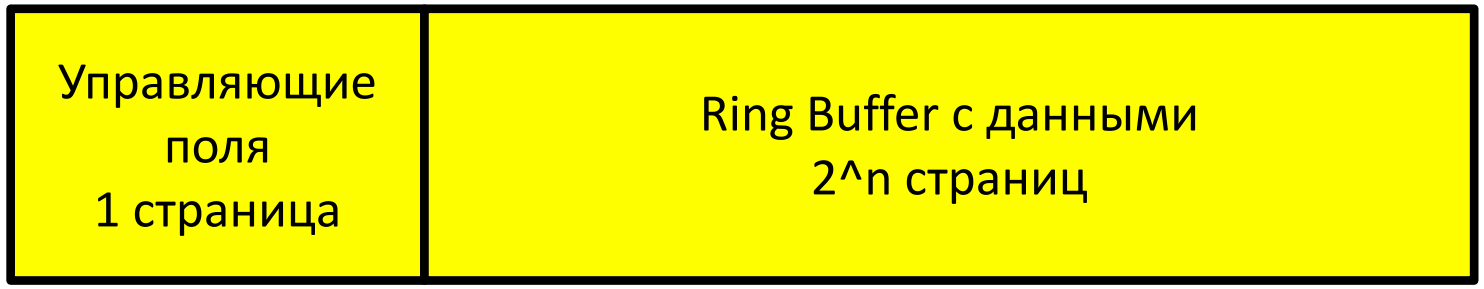
Ring Buffer

- https://en.wikipedia.org/wiki/Circular_buffer
- CircularFifoBuffer from Apache Collections
- Disruptor

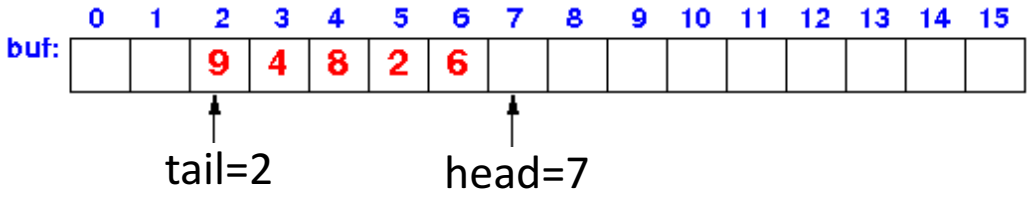
Ну что, читаем данные Processor Trace?

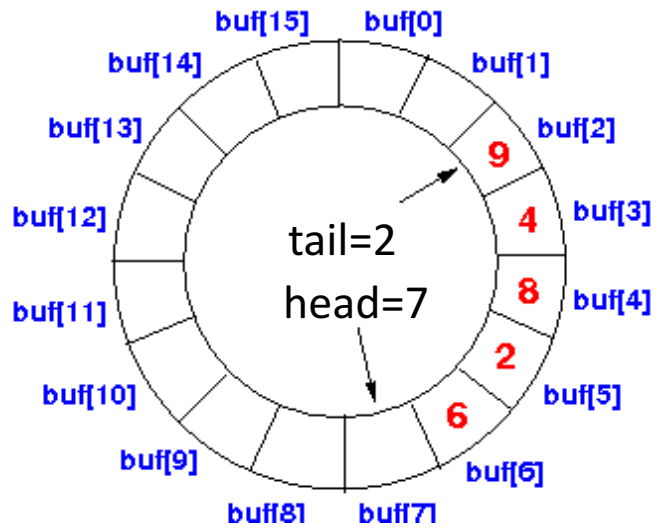
Ведь теперь мы знаем, как работать с RingBuffer’ом

- long aux_head;
- long aux_tail;
- long aux_size;



```
__u64 size     = hdr->aux_size;
__u64 head_pos = hdr->aux_head;
__u64 tail_pos = hdr->aux_tail;
__u64 head_idx = head_pos % size;
__u64 tail_idx = tail_pos % size;


if (head_idx > tail_idx)
  // copy data [tail_idx; head_idx)
else
  // copy data [tail_idx; size)
  // copy data [0; head_idx)
hdr->aux_tail = head_pos;
```





1. Запустить профилировщик - **умеем**
2. Сохранить профиль/трассу - **умеем**
3. Декодировать трассу в терминах адресов
4. Декодировать имена методов
5. Декодировать время элементов трассы

Профиль/трасса, который мы собрали состоит из пакетов Intel Processor Trace

Пакеты? А они бывают?

- Пакеты описывают все детали, что выполняло «железо»
- Суммарно 25 разных пакетов
- ~25 страниц описания

Пакеты, описывающие «что выполнялось»

- Пакет **TNT** (Taken/Not-Taken)
  - Перешли или нет по инструкции условного ветвления?
  - Описывает все возможные ветвления: if, else, switch, ...
- Пакет **TIP** (Target Instruction Pointer)
  - Какую инструкцию сейчас начнем выполнять?
  - Позволяет понять какая реализация виртуального метода была выполнена
- И другие пакеты

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

Как все это декодировать?

1. Написать декодер самостоятельно
2. Взять декодер Processor Trace из perf
3. Использовать референсную реализация декодера от Intel: https://github.com/01org/processor-trace

```
pt_config config = {};
config.begin = <pt buffer begin>;
config.end   = <pt buffer end>;
config.cpu   = <cpu identifier>;
...
... = ::pt_<layer>_alloc_decoder(&config);
...
... = ::pt_<layer>_sync_forward(decoder);
```

Level – один из packet, query, instruction, block

Что мы будем получать из декодера?

1. Instructions – поток исполненных инструкций

   => Получаем конкретные инструкции
2. Block – поток исполненных «блоков» – линейных последовательностей инструкций

   ⇒Получаем линейные последовательности инструкций – фундаментальные блоки, из которых строятся методы
3. Др. (packets, query)

```
59:    jle     0x6a
5b:    mov     -0x4(%rbp),%eax   => время t6, метод B.doo
5e:    sub     $0x64,%eax
61:    mov     %eax,%edi
63:    callq   0x87
87:    push    %rbp              => время t10
```

T(B.doo) += (t10 – t6)

x86

1. Запустить профилировщик - **умеем**
2. Сохранить профиль/трассу - **умеем**
3. Декодировать трассу в терминах адресов - **умеем**
4. Декодировать имена методов
5. Декодировать время элементов трассы

А откуда брать названия методов?

- Получаем информацию о скомпилированных методах через `jvmti`-агент, который получает нотификации о событиях:
  - `JVMTI_EVENT_COMPILED_METHOD_LOAD`
  - `JVMTI_EVENT_COMPILED_METHOD_UNLOAD`
  - `JVMTI_EVENT_DYNAMIC_CODE_GENERATED`
- Для «нативного» кода используем функцию `dladdr`, которая по адресу возвращает название метода

1. Запустить профилировщик - **умеем**
2. Сохранить профиль/трассу - **умеем**
3. Декодировать трассу в терминах адресов - **умеем**
4. Декодировать имена методов - **умеем**
5. Декодировать время элементов трассы

Пакеты о времени

- **TSC** – младшие биты значения, возвращаемого инструкцией `rdtsc`
- **MTC** - время в прерываниях Always Running Timer (ART)
- **CYC** - время в тактах процессора с момента предыдущего пакета CYC

Столько разных способов!

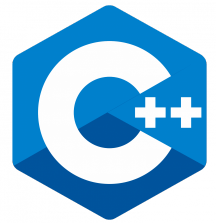
И потенциально они противоречивые…

Подробности доступны:

1. Файл intel-pt.txt из linux kernel
2. Глава 35 Intel SDM

Смотрим самое интересное – время!

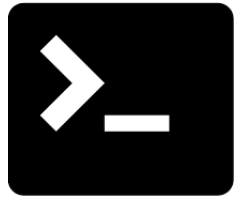
```
auto status = ::pt_blk_time(decoder, &now, ...);
if (prev_now > now) {
  printf(
    “Did you use time machine? %lx -> %lx\n“,
    prev_now,
    now);
}
prev_now = now;
```



Смотрим, что получилось!

```
$ grep “Did you use time machine” log | wc -l
615
```

Наш код пользовался машиной времени 615 раз…



Но у perf’а получается лучше…

Копаем дальше!

~~Чип и Дейл~~ GitHub спешит на помощь!

markus-metz... commented 5 days ago — Contributor

Hello Sergey,

The error says that the decoder does not have the necessary information to use the MTC packet to update its estimated TSC. See struct pt_config in intel-pt.h on what information you need to provide and how to get it.

When using ptdump and ptxed, you would use the options:

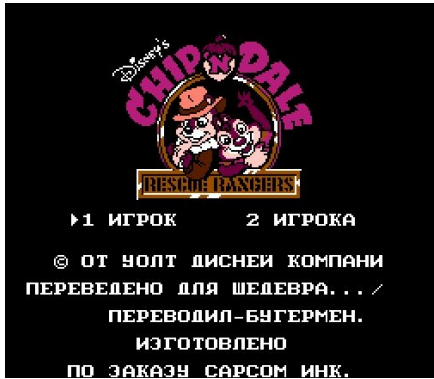
```
--mtc-freq <n>          set the MTC frequency (IA32_RTIT_CTL[17:14]) to <n>.
--nom-freq <n>          set the nominal frequency (MSR_PLATFORM_INFO[15:8]) to <n>.
--cpuid-0x15.eax        set the value of cpuid[0x15].eax.
--cpuid-0x15.ebx        set the value of cpuid[0x15].ebx.
```

In struct perf_event_attr.config you would enable MTC and CYC and set the MTC frequency and the CYC threshold. Except for the MTC frequency, the decoder does not need to know your settings. See chapter 35 in Vol 3 of the SDM for details on those settings.

If you enabled tick events it should suffice to use the TSC field in struct pt_event.

regards,
markus.

https://github.com/01org/processor-trace/issues/47

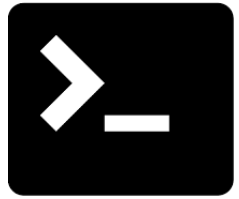


Итого, нам нужно выставить правильно 4 поля

1. `Mtc freq` – значение, которое мы передаем при конфигурации perf_event_open => нет проблем
2. `Cpuid[0x15].eax` – выполняем инструкцию cpuid и получаем значение => нет проблем
3. `Cpuid[0x15].ebx` – выполняем инструкцию cpuid и получаем значение => нет проблем
4. `Nominal freq` - значение части регистра MSR_PLATFORM_INFO.

Неужели получилось?

```
$ grep “Did you use time machine” log | wc -l
0
```



1. Запустить профилировщик - **умеем**
2. Сохранить профиль/трассу - **умеем**
3. Декодировать трассу в терминах адресов - **умеем**
4. Декодировать имена методов - **умеем**
5. Декодировать время элементов трассы - **умеем**

Чего-то не хватает?

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();
}
```

Итого: full-stack (без JavaScript ☺)

1. **librperf2.so** – наш профилировщик, который состоит из:
   1. Jvmti-агента для сбора информации об именах методов
   2. Jni-кода, запускающего профилирование
   3. Декодера трассы и агрегатор профиля
2. Java-агент для прозрачной инструментации кода вызовами профилировщика

3. Практика - профилируем разбор JSON’а

Тренируемся на кошках

Профилируем декодирование JSON’а библиотекой GSON от Google

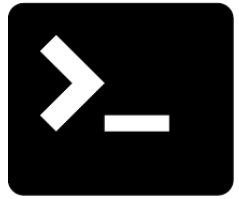
```
public static
Map decode(String json)
{
  Map result =
   gson.fromJson(json, Map.class);
  return result;
}
```

```
public static
void main( String[] args ) {
  for (int i = 0; ...) {
     Map m = decode_1(json, i);
  }
}
```

```
public static
Map decode_1(String json, int idx)
{
  ...
  return decode(json, idx);
}
```

Как запустить профилирование?

```
$ java -agentlib:rperf2=                                    \
  -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
```



TOP в деталях


```
151080ns [ 8%] ...LinkedTreeMap->rebalance
155472ns [ 8%] ...MapTypeAdapterFactory$Adapter->read
186508ns [ 9%] ...JsonReader->nextQuotedValue
355373ns [18%] ...JsonReader->nextUnquotedValue
987199ns [50%] ...LinkedTreeMap->put
Total time: 1954507ns
```

Теперь можно точно увидеть, что выполнялось!

```
...
Timestamp: 15912 JsonReader->nextUnquotedValue
Timestamp: 15912 MapTypeAdapterFactory$Adapter->read
...
Timestamp: 15914 MapTypeAdapterFactory$Adapter->read
Timestamp: 15914 JsonReader->doPeek
...
Timestamp: 15916 JsonReader->doPeek
Timestamp: 15916 JsonReader->nextNonWhitespace
...
```

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

Теперь можно реконструировать control-flow участка кода с точными таймингами

А что дальше?

Мы научились профилировать редкие события… Но не забываем про реальный мир:

мы оптимизируем не скорость, а количество заработанных денег!

Как это бывает в реальности?

Вариант	Медианное время выполнения, мкс	Прибыль, попугаев
1	100	**9**
2	110	**9,5**

Второй вариант более прибыльный!

Почему так получается?

Исполнение	1 вариант, мкс	Заработок, попугаев	2 вариант, мкс	Заработок, попугаев
1 раз	100	1	110	0,95
2 раз	100	1	110	0,95
		…		
9 раз	100	1	110	0,95
10 раз	200	0	110	0,95
Итого:		**9**		**9,5**

Нам нужна метрика для описания редких событий

Называется перцентиль или квантиль распределения

Например,
Худшее время из 10 запусков = квантиль 0.9
Худшее время из 1000 запусков = квантиль 0.999

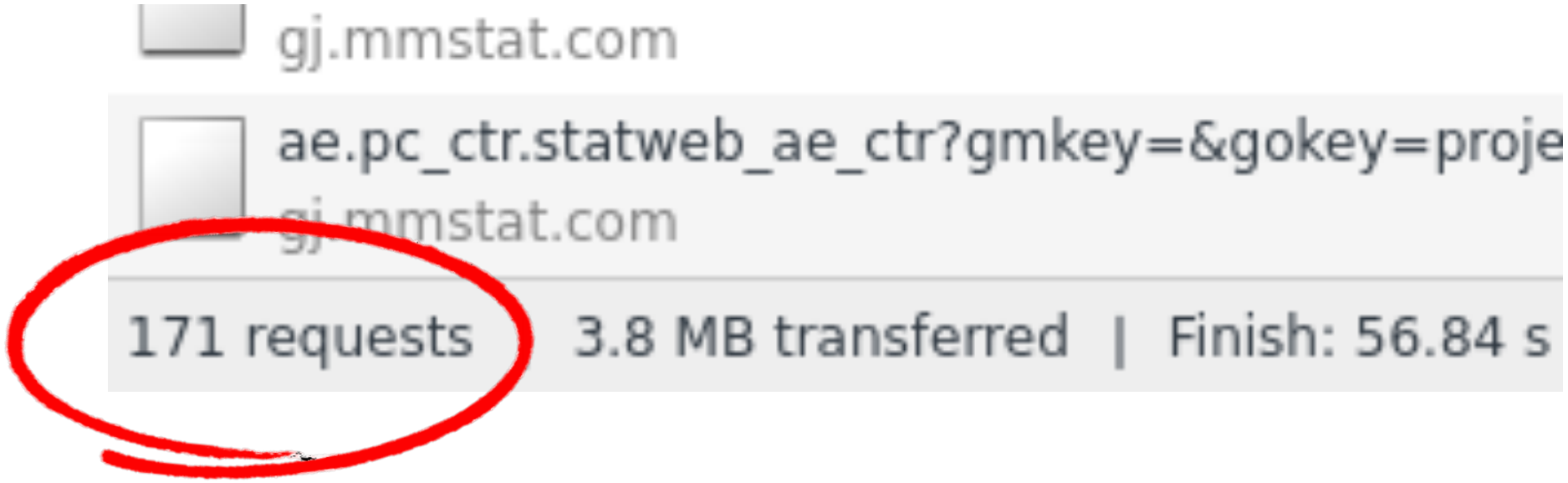
Недостающая метрика

Вариант	Медианное время выполнения, мкс	0.9 квантиль	Прибыль, попугаев
1	100	200	**9**
2	110	110	**9,5**

Да кому нужны эти высокие перцентили?

Name	Status	Type	Initiator	Size	T...	Waterfall
connectkeyword.aliexpress.com			Script	41 B	...	
entry.js?t=213573 assets.alicdn.com/g/alilog/oneplus	200	script	VM2141:5 Script	1.2 KB 1.5 KB	... ...	
aliexpress.ico ae01.alicdn.com/images/eng/wholesale/icon	200	x-icon	Other	4.4 KB 4.2 KB	... ...	
blk.html g.alicdn.com/alilog/oneplus	200	document	entry.js?t=213573:2 Script	5.9 KB 13.4 KB	... ...	
p.gif?memberSeq=126344325&isNewUser=false&pageId... perf.mmstat.com	200	gif	??loader.011e762a.js:1 Script	160 B 0 B	... ...	
HTB1gIGnaSzqK1RjSZFpq6ykSXXan.jpg ae01.alicdn.com/kf	200	webp	home-ru.6200e717.js:1 Script	32.1 KB 31.7 KB	... ...	
ae.pc_ctr.statweb_ae_ctr?gmkey=&gokey=project_id%3..... gj.mmstat.com	200	gif	VM2141:5 Script	97 B 43 B	... ...	
HTB1OSRRXizxK1RjSspjq6AS.pXa0.jpg ae01.alicdn.com/kf	200	jpeg	home-ru.6200e717.js:1 Script	359 KB 358 KB	... ...	
ae.pc_ctr.statweb_ae_ctr?gmkey=&gokey=project_id%3..... gj.mmstat.com	200	gif	VM2141:5 Script	97 B 43 B	... ...	
HTB1M2s_a7PoK1RjSZKb7601IXXat.png ae01.alicdn.com/kf	200	png	home-ru.6200e717.js:1 Script	117 KB 116 KB	... ...	
ae.pc_ctr.statweb_ae_ctr?gmkey=&gokey=project_id%3..... gj.mmstat.com	200	gif	VM2141:5 Script	97 B 43 B	... ...	
HTB1zlXzXh2rK1RkSnhJq6ykdpXa1.jpg ae01.alicdn.com/kf	200	webp	home-ru.6200e717.js:1 Script	37.6 KB 37.1 KB	... ...	
ae.pc_ctr.statweb_ae_ctr?gmkey=&gokey=project_id%3..... gj.mmstat.com	200	gif	VM2141:5 Script	97 B 43 B	... ...	
ae.pc_ctr.statweb_ae_ctr?gmkey=&gokey=project_id%3..... gj.mmstat.com	200	gif	VM2141:5 Script	97 B 43 B	... ...	

171 requests | 3.8 MB transferred | Finish: 56.84 s | DOMContentLoaded: 977 ms | Load: 31.96 s



Квантиль «каждый 171 раз» или 0.994 – с ним будет встречаться каждый пользователь!

Наши пользователи сталкиваются с редкими событиями намного чаще, чем нам хочется

Но что с этим делать?

4. А что если иногда код работает медленнее?

Почему такое бывает? **И как это профилировать?**

Почему так бывает?

Это заложено самой архитектурой компьютера!

- JIT-компилятор оптимизирует выполнение наиболее часто выполняемого паттерна. Зачастую, ценой пессимизации редко выполняемых путей выполнения
- Структуры данных: ре-хэш или ребалансировка дерева
- Применение кэширования на всех уровнях архитектуры: от управления памятью до микроархитектуры процессора
- И так далее!

Почему так бывает?

- емого
-
- ния

Опасности подстерегают повсюду!!!

Но как анализировать причины замедлений?

памятью до микроархитектуры процессора

- И так далее!

Что делать?

Perf не поможет, так как профилирует «средний случай»

Раз уж мы умеем собирать профиль за всего лишь одно выполнение профилируемого кода – нам останется всего лишь дождаться этого редкого случая и сохранить его профиль!

Как профилировать почему код работает медленнее обычного?

Т.е. именно те самые редкие медленные выполнения

1. Собираем статистику – запоминаем время выполнения на нужном квантиле
2. Профилируем и ждем, когда время выполнения будет больше запомненного
3. Выводим профиль именного этого редкого события!

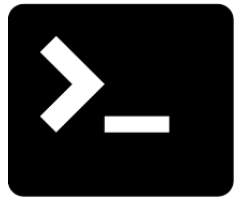
А что может произойти? Мой код работает стабильно быстро!

```
public static
Map decode(String json)
{
  Map result =
   gson.fromJson(json, Map.c
  return result;
}
public static
void main( String[] args ) {
  for (int i = 0; ...) {
     Map m = decode_1(json, 1);
  }
}
```

Почему метод decode может работать медленно?

Как запустить профилирование?

```
$ java -agentlib:rperf2=                                    \
  -javaagent:perf-instrumenter-....jar                      \
  -DPERCENTILE=0.999                                        \
  -DTRIGGER_COUNTDOWN=10000                                 \
  -DTRIGGER_METHOD=decode -DTRIGGER_CLASS=ru/raiffeisen/App \
  -cp target/json-tester-....jar:.../gson-2.8.2.jar         \
  ru.raiffeisen.App
```



TOP в деталях


```
Median: 1575682ns
Timing for 0.999 timings: 2168875ns
Processing trace with length 5284262ns > 2168875ns
...
```

TOP

```
136211ns  [3%]  MapTypeAdapterFactory$Adapter->read
153626ns  [3%]  LinkedTreeMap->rebalance
340685ns  [6%]  JsonReader->nextQuotedValue
526368ns  [10%] JsonReader->nextUnquotedValue
724595ns  [14%] LinkedTreeMap->put
2979060ns [56%] Interpreter
Total time: 5287159ns
```

```
public static
Map decode(String json)
{
  Map result =
   gson.fromJson(json, Map.class);
  return result;
}

public static
void main( String[] args ) {
  for (int i ...) {
     Map m = decode_1(json, i);
  }
}
```

```
public static
Map decode_1(String json, int idx)
{
  if (idx > 17000
       && idx < 17010)
  {
    json = "{\"qqq\": \"123\"}";
  }
  return decode(json, idx);
}
```

```
boolean condition = ...;
if(condition) {
  never (); // попали!
}
// теперь все исполняется
// в интерпретаторе
// до конца метода!
```

```
public static
Map decode(String json)
{
  Map result =
   gson.fromJson(json, Map.class);
  return result;
}

public static
void main( String[] args ) {
  for (int i ...) {
     Map m = decode_1(json, i);
  }
}
```

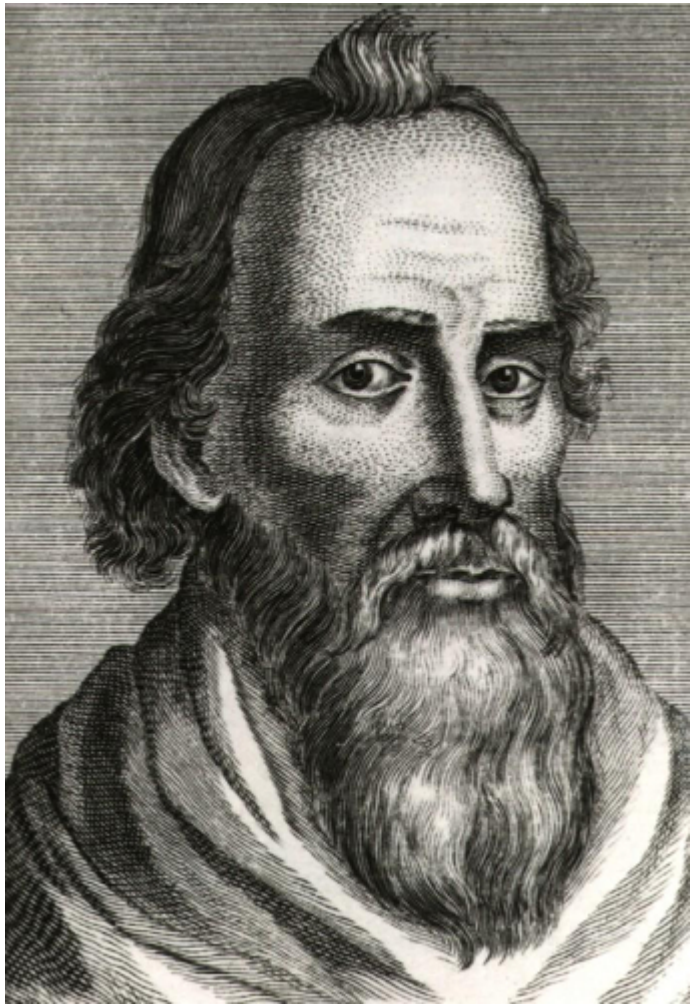
```
public static
Map decode_1(String json, int idx)
{
  if (idx > 17000
       && idx < 17010)
  {
    json = "{\"qqq\": \"123\"}";
  }
  return decode(json, idx);
}
```

Итого, что мы обнаружили?

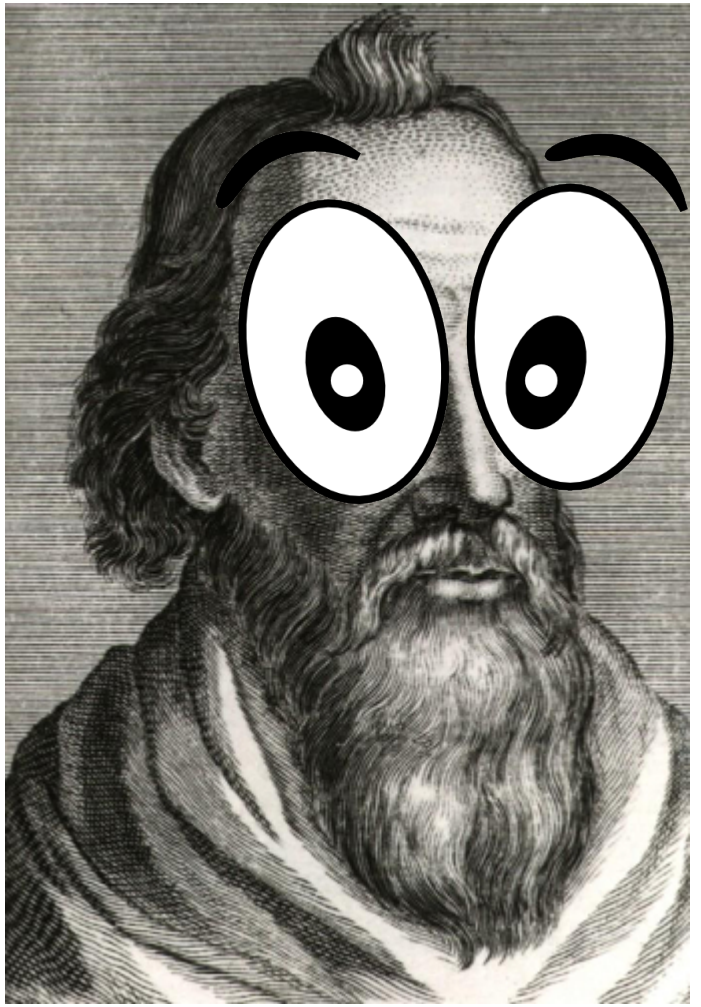
1. Замедление является неожиданным
2. Замедление наносит ощутимые финансовые потери
3. Пост-фактум: замедление имеет рационалистическое объяснение

Хм.. Напоминает черного лебедя

Черные лебеди?



«Редкая птица, подобная черному лебедю»



События типа «черный лебедь»

Термин введен Нассимом Талебом в 2007

- Событие является неожиданным
- Событие производит значительные последствия
- После наступления, в ретроспективе, событие имеет рационалистическое объяснение

Замедление работы софта – такой же черный лебедь

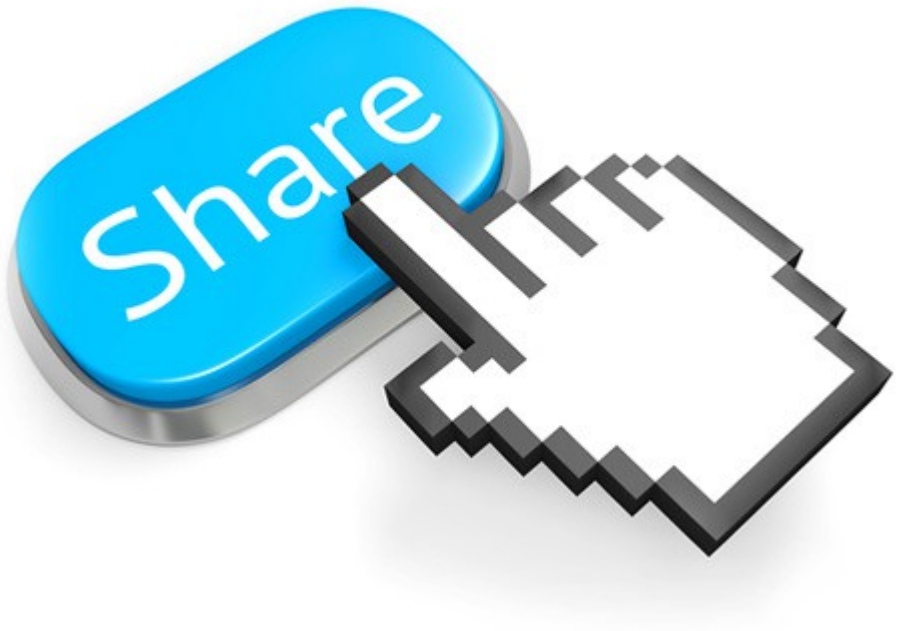
Но наших черных лебедей мы теперь умеем ловить за хвост!

Итоги

У нас получился интересный инструмент!

Теперь мы можем заглянуть не просто в то, что выполнялось в «среднем» случае, а понять, почему наш код *иногда* работает медленнее – поймать черного лебедя за хвост

Черных лебедей полезно ловить не только Райффайзенбанку!



https://github.com/RainM/rperf2

Пользуйтесь и ловите Ваших черных лебедей

Райффайзен БАНК

Сергей Мельников

Sergey.V.Melnikov@raiffeisen.ru

Профилируем «черного лебедя» с помощью Intel Processor Trace, или Что делать, если иногда код выполняется 20 мс вместо 2

https://github.com/RainM/rperf2