

podlodka.

Java Crew

26 – 30 мая

Java и производительность

Владимир Плизга

Tibbo Systems

Доклад
Профилирование
Java-приложений:
подходы и инструменты



ПРИВЕТ!

- Я – Владимир Плизгá
- Пишу на Java с 2011 г.
 - ФинТех (интернет-банк)
 - Промышленный ІоТ
- Люблю помогать людям
(особенно разработчикам)



КОНТЕКСТ



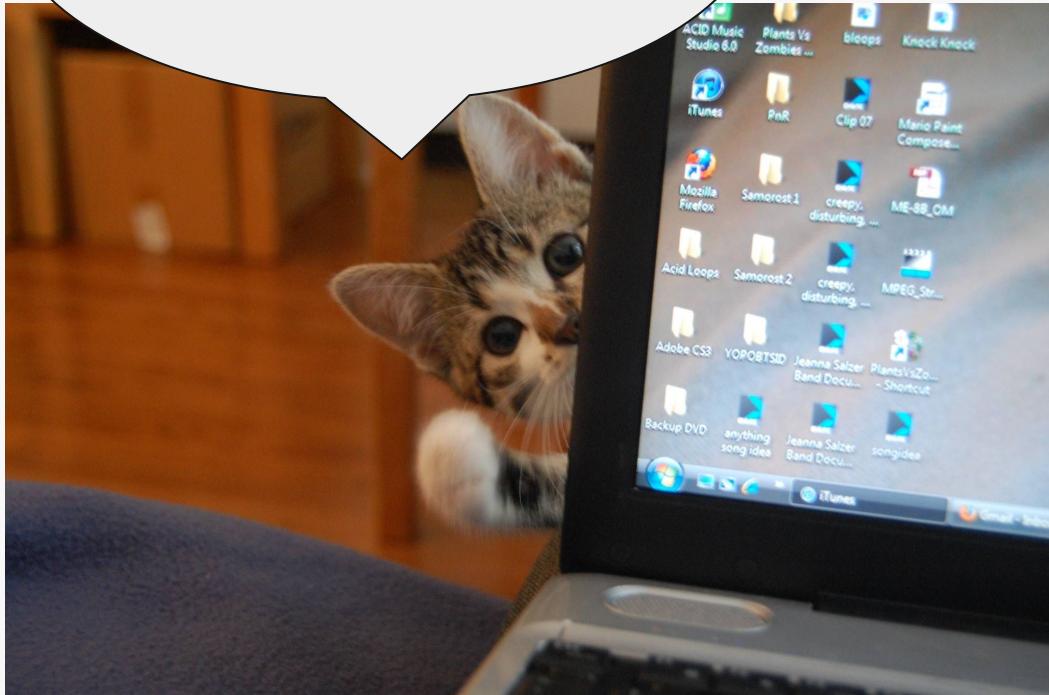
“Когда происходят ошибки, у нас есть стектрейсы”



“А когда падает производительность – их нет :("

КОНТЕКСТ

“Логи и метрики
не помогают!”



“Хоть на работу
не ходи!”



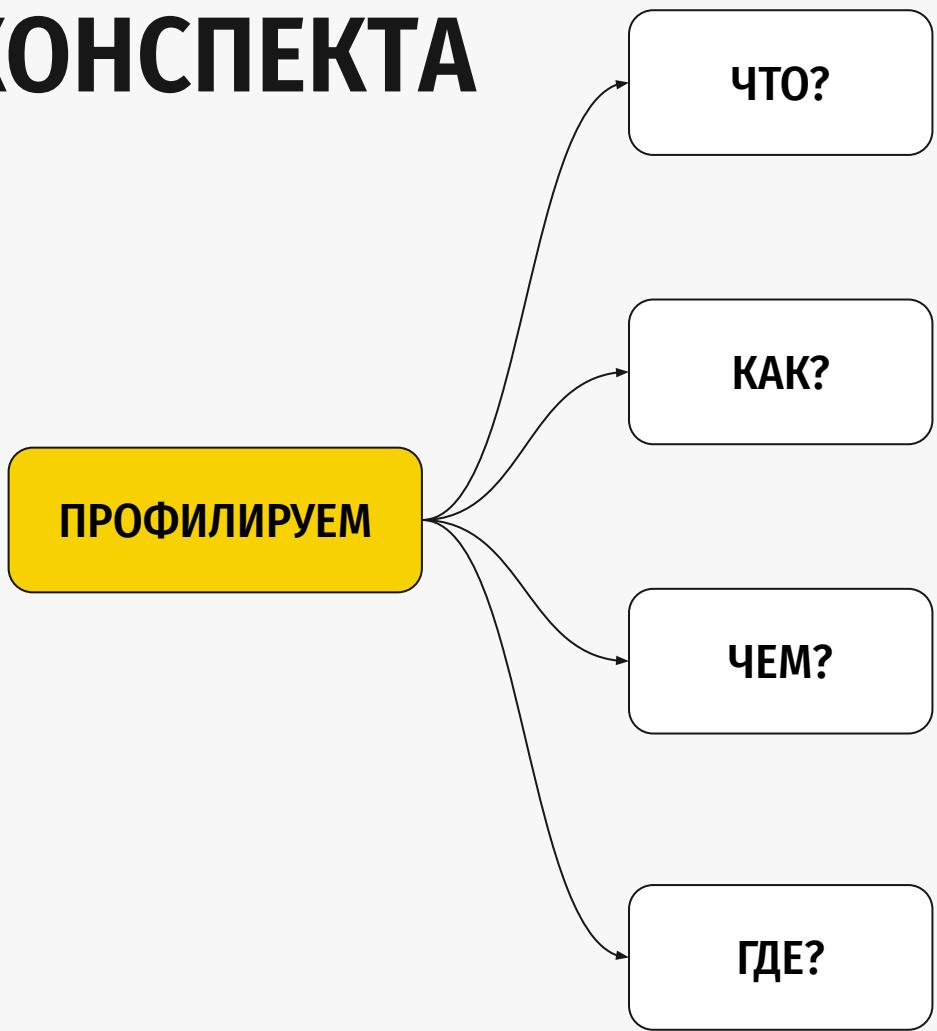
ЧТО ДЕЛАТЬ?

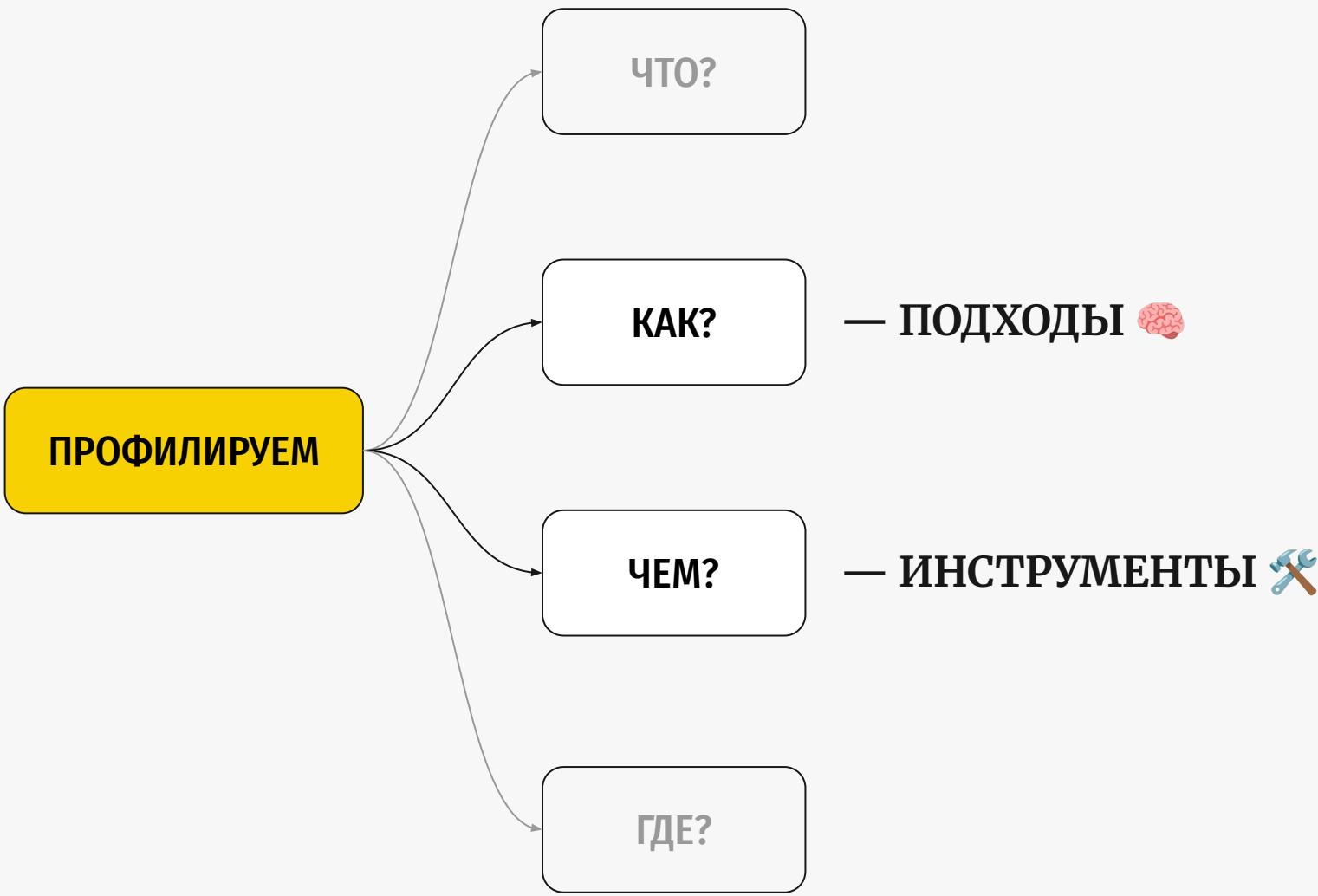
- Выявлять проблемные участки в коде
- Доставать их стектрейсы
- Оценивать продолжительность

PROFILING TO THE RESCUE

ПРОФИЛАГА ДО ШЕГ ИГСОГ

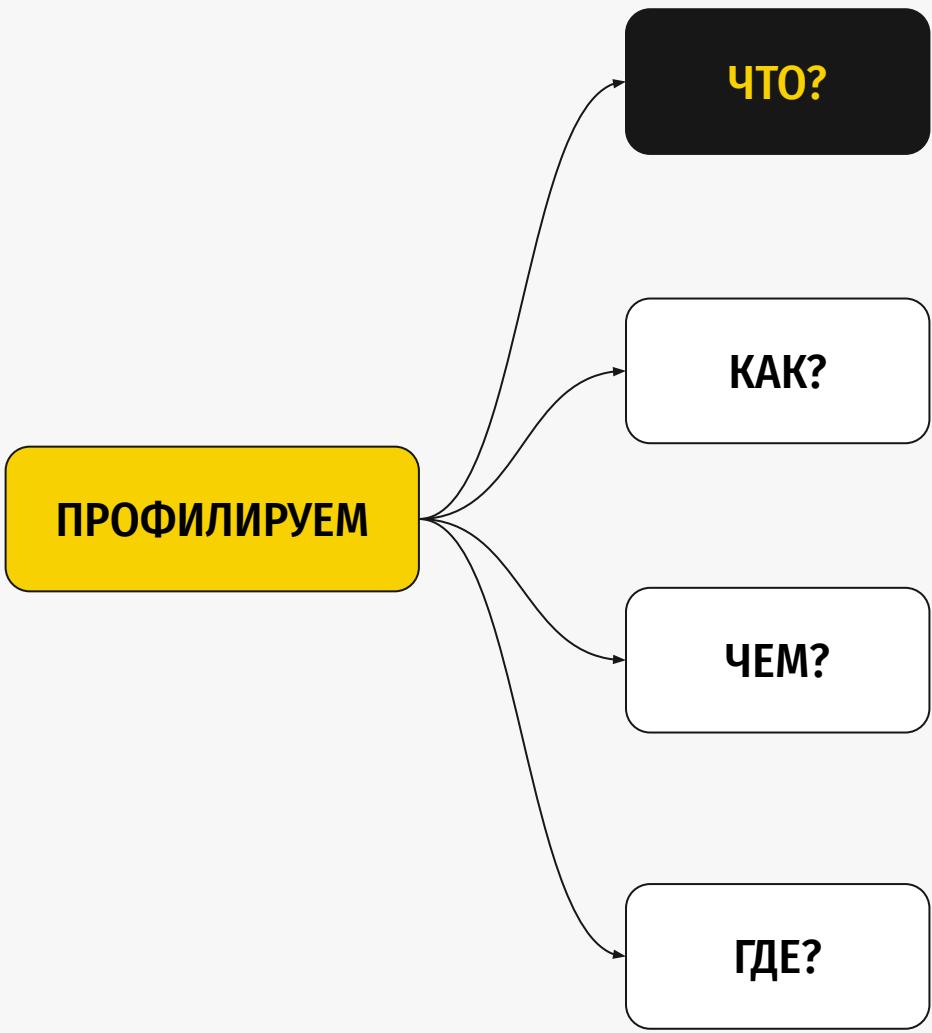
ПЛАН КОНСПЕКТА





{

Основная часть доклада



- Вглубь
- Вширь

ЧТО МЫ ПРОФИЛИРУЕМ?



ЧТО МЫ ПРОФИЛИРУЕМ?

Наш фокус

Цепочка
следствий

Прикладной
код

JVM
OS

...

ЕСЛИ ХОЧЕТСЯ ЗАНЫРНУТЬ

The slide features a dark background with red wavy lines. In the top left, the text "Joker<?>" is written in white, with "2024" below it. In the center, the title "Профилирование Java в стиле Linux" is displayed in large white font. To the right is a circular portrait of a man with long hair, framed by a pink border. Below the portrait, the text "Сергей Мельников" is written in white, followed by "Dijkstra Markets". In the bottom left, there is a QR code.

Joker<?>
2024

Профилирование
Java в стиле Linux

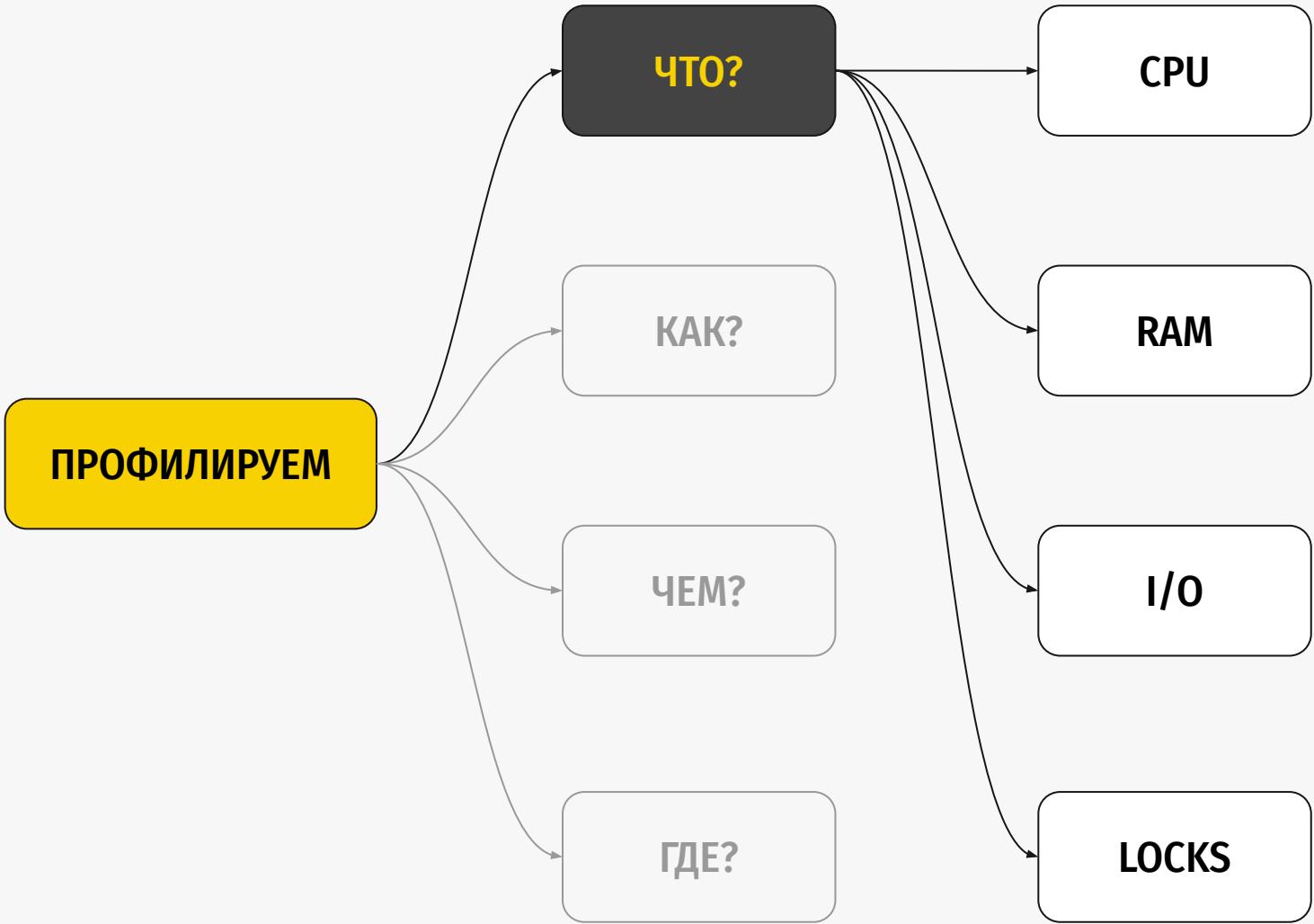
Сергей
Мельников
Dijkstra Markets

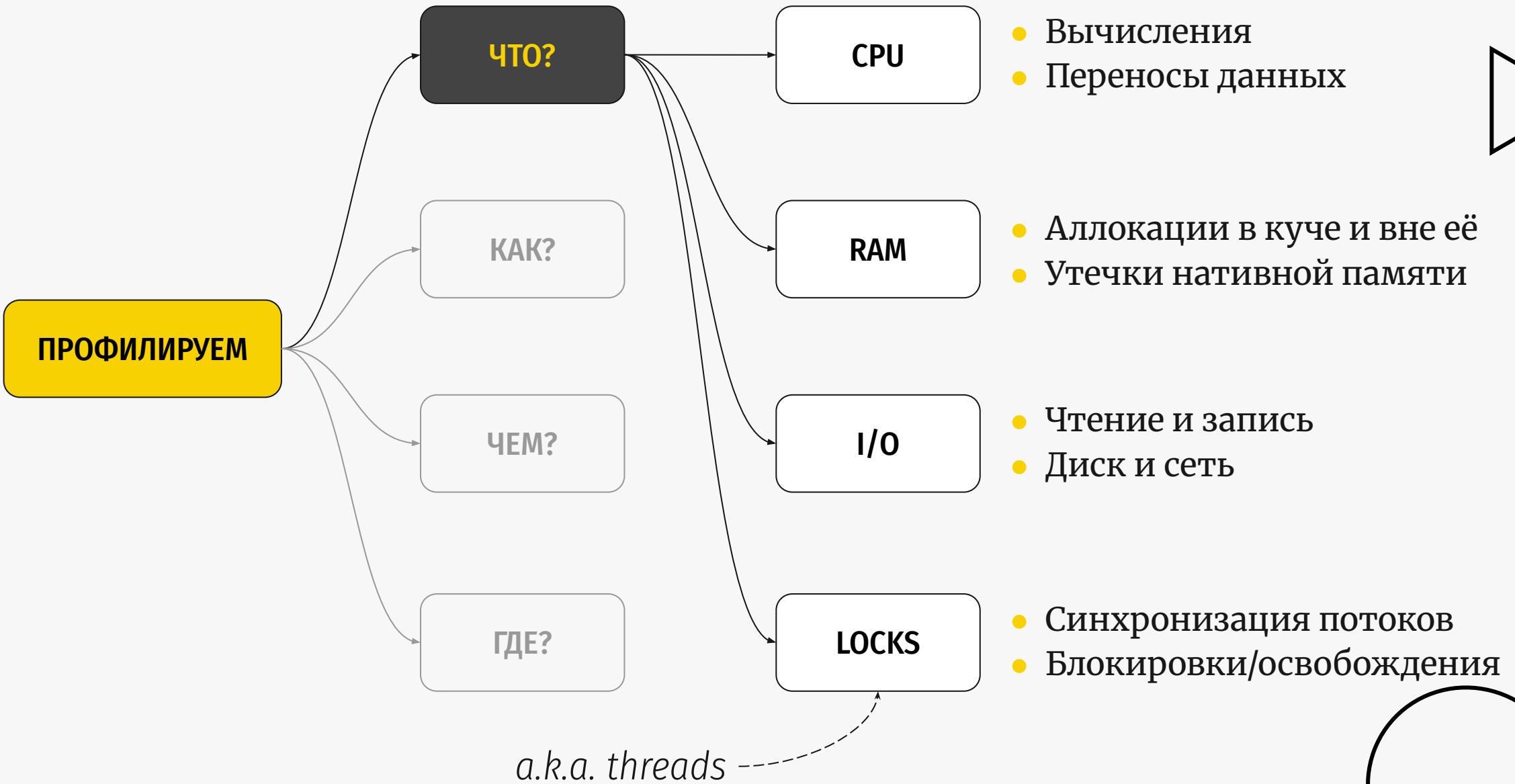
QR code

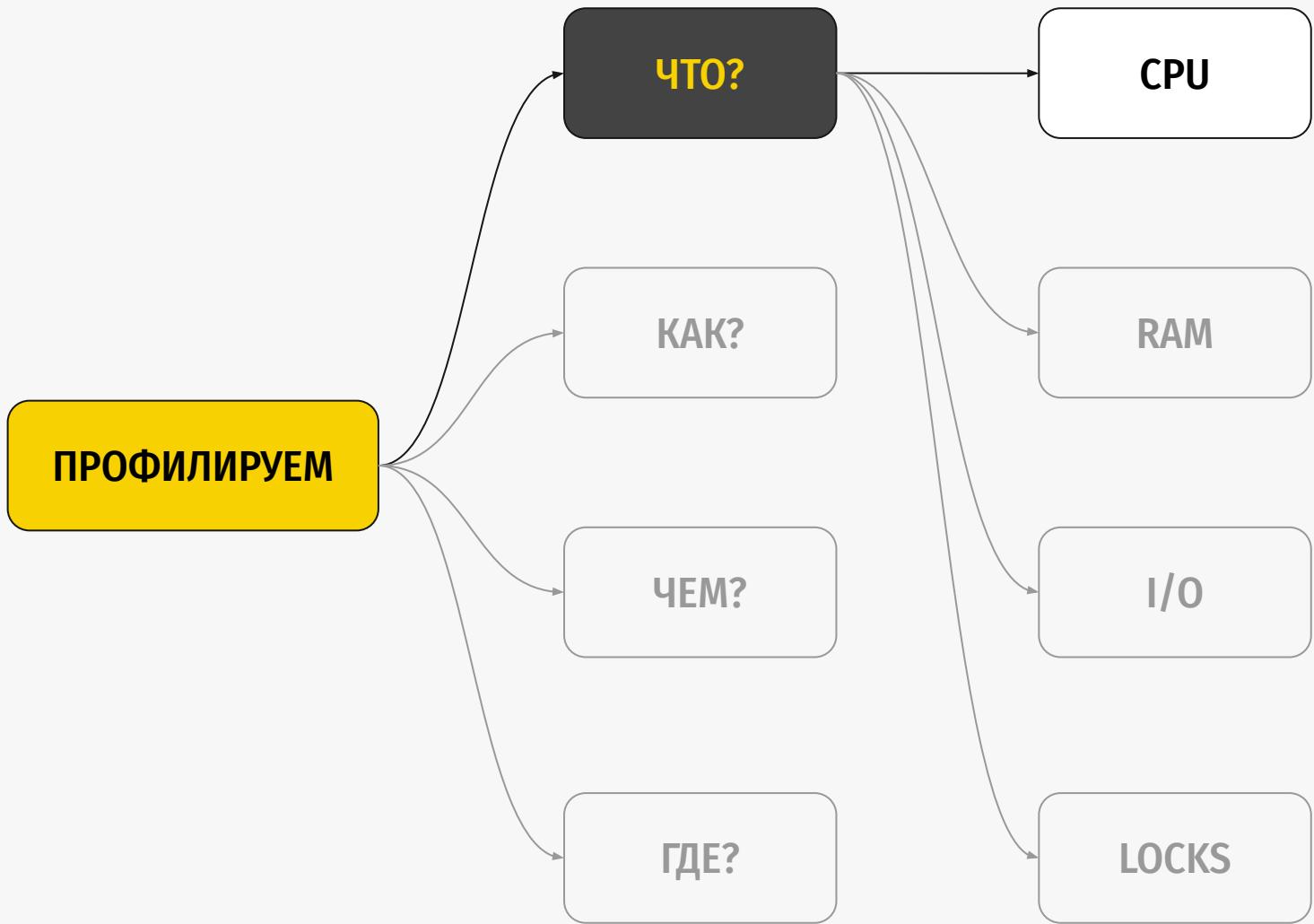
<https://www.youtube.com/watch?v=EZA5zcfFA4w>



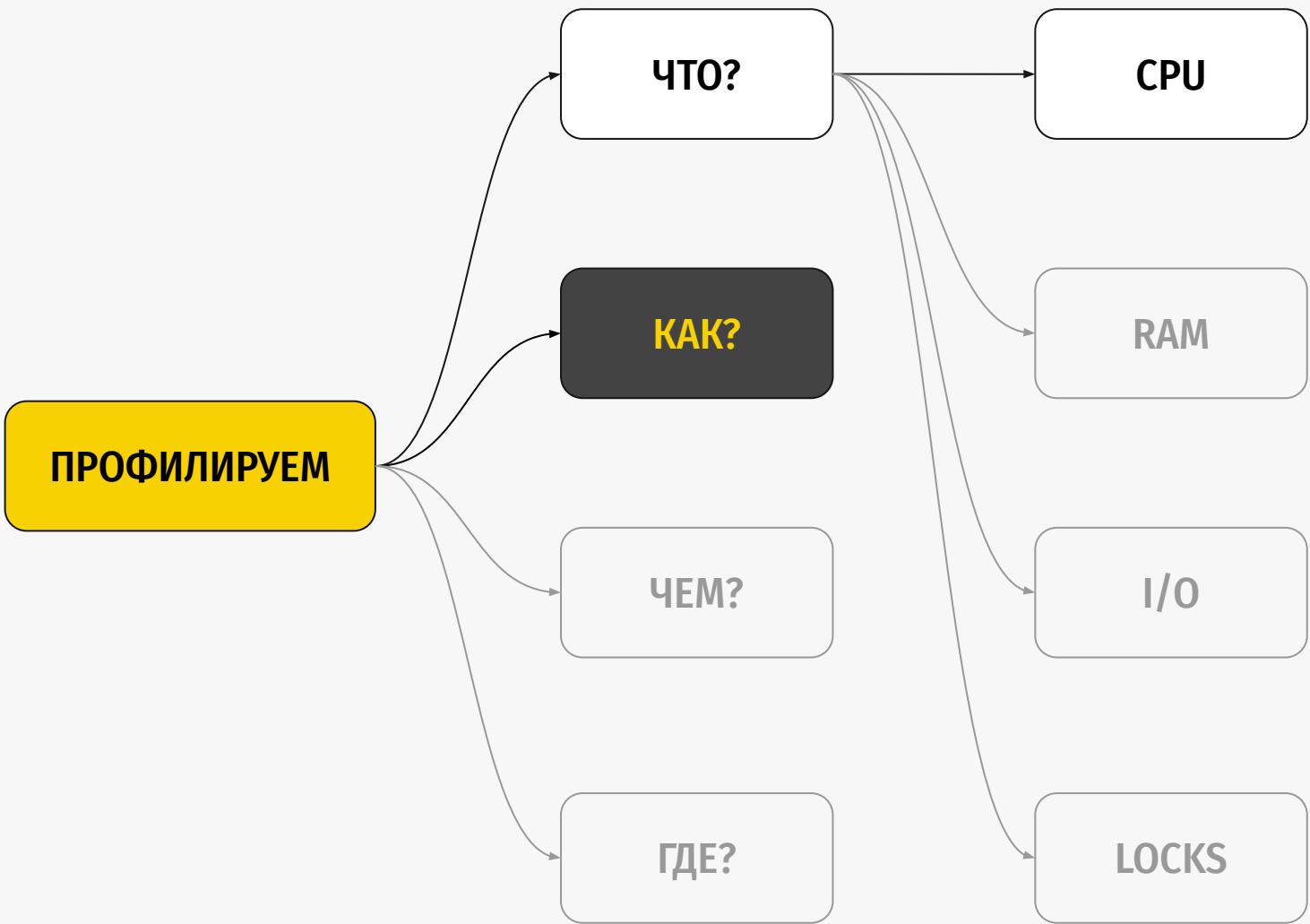
- В глубь
- Вширь





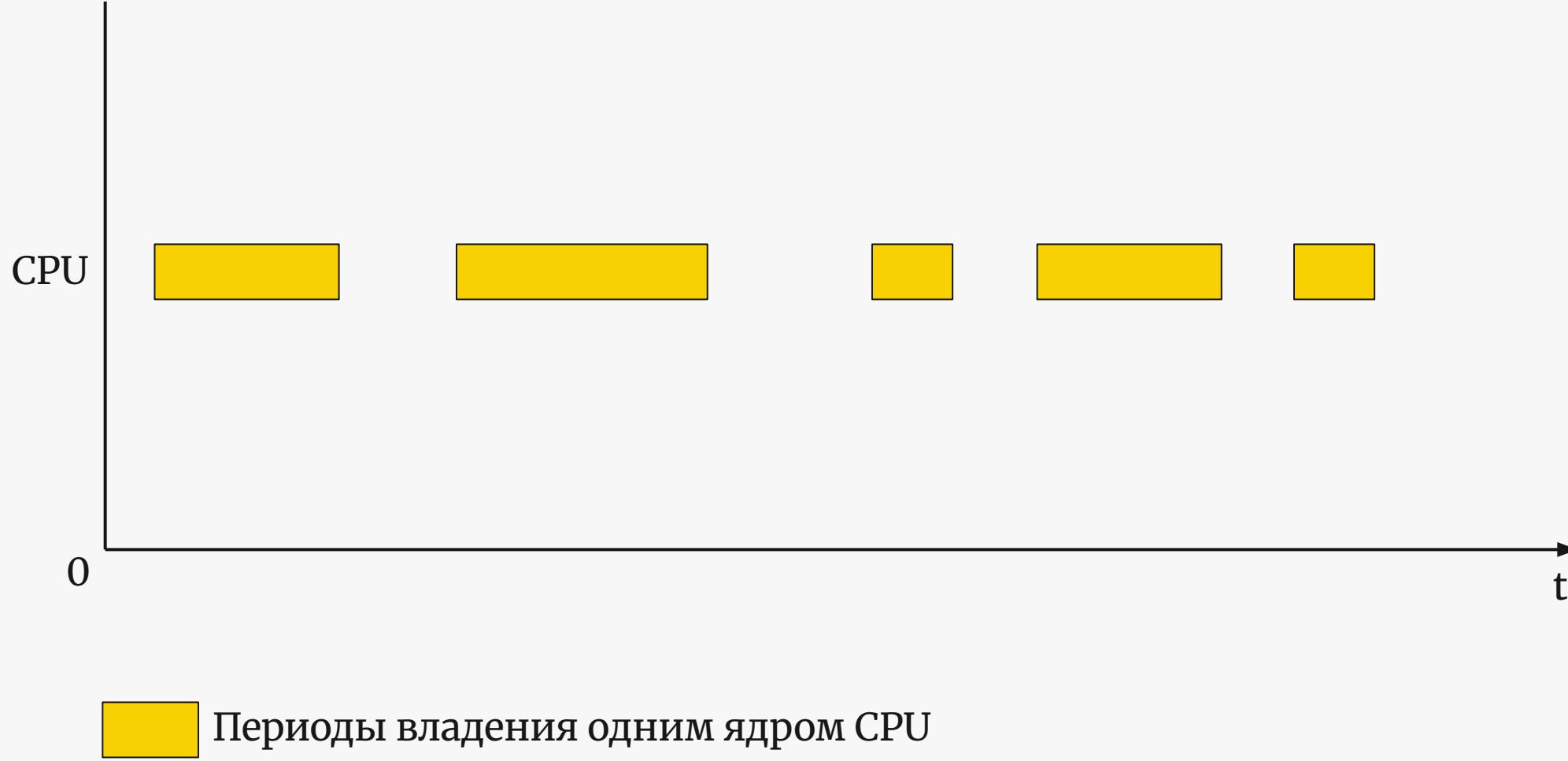


- Вычисления
- Переносы данных



- Вычисления
- Переносы данных

РЕАЛЬНЫЙ ПРОФИЛЬ МЕТОДА foo()

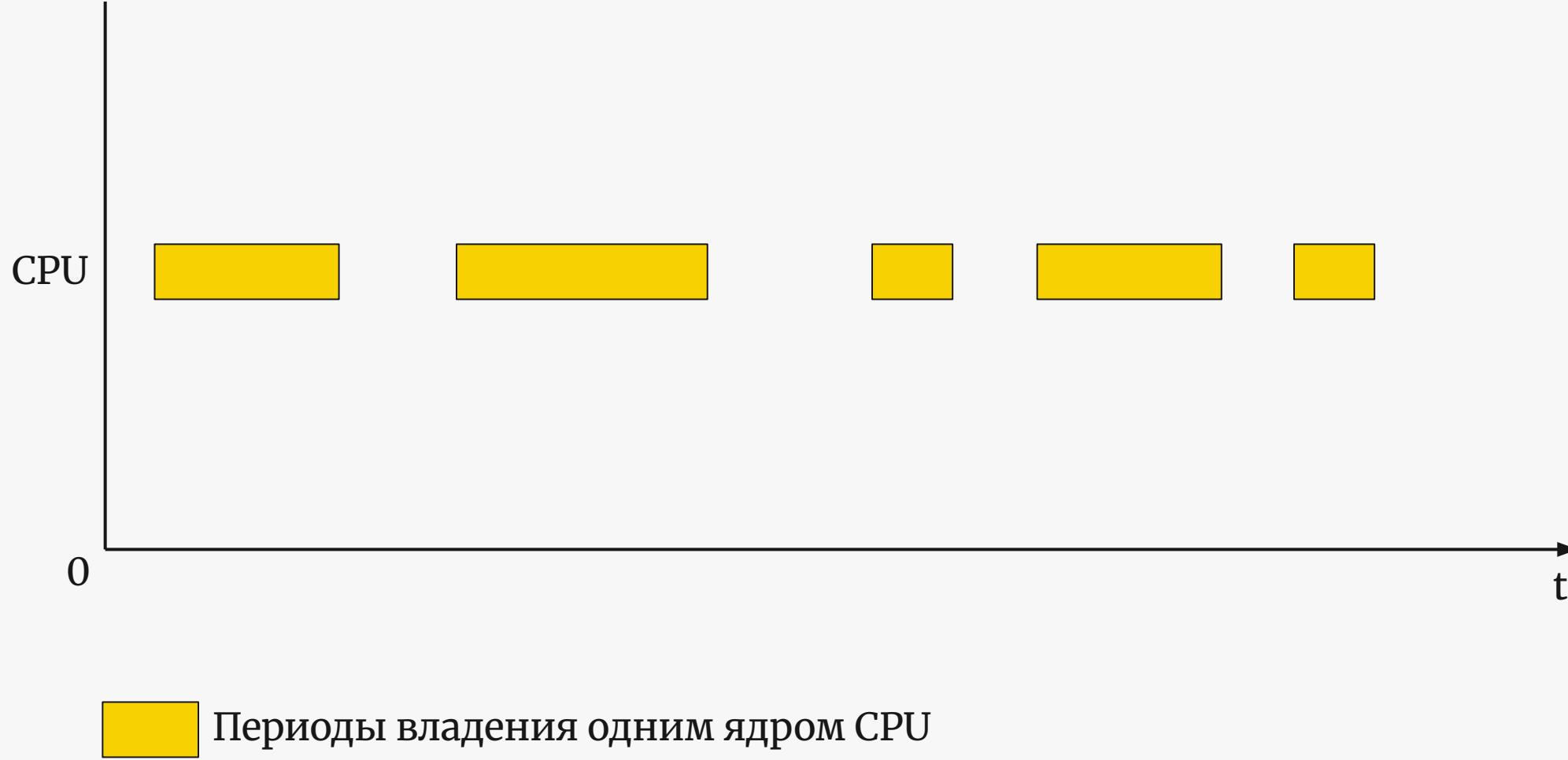


КАК ПРОФИЛИРОВАТЬ CPU

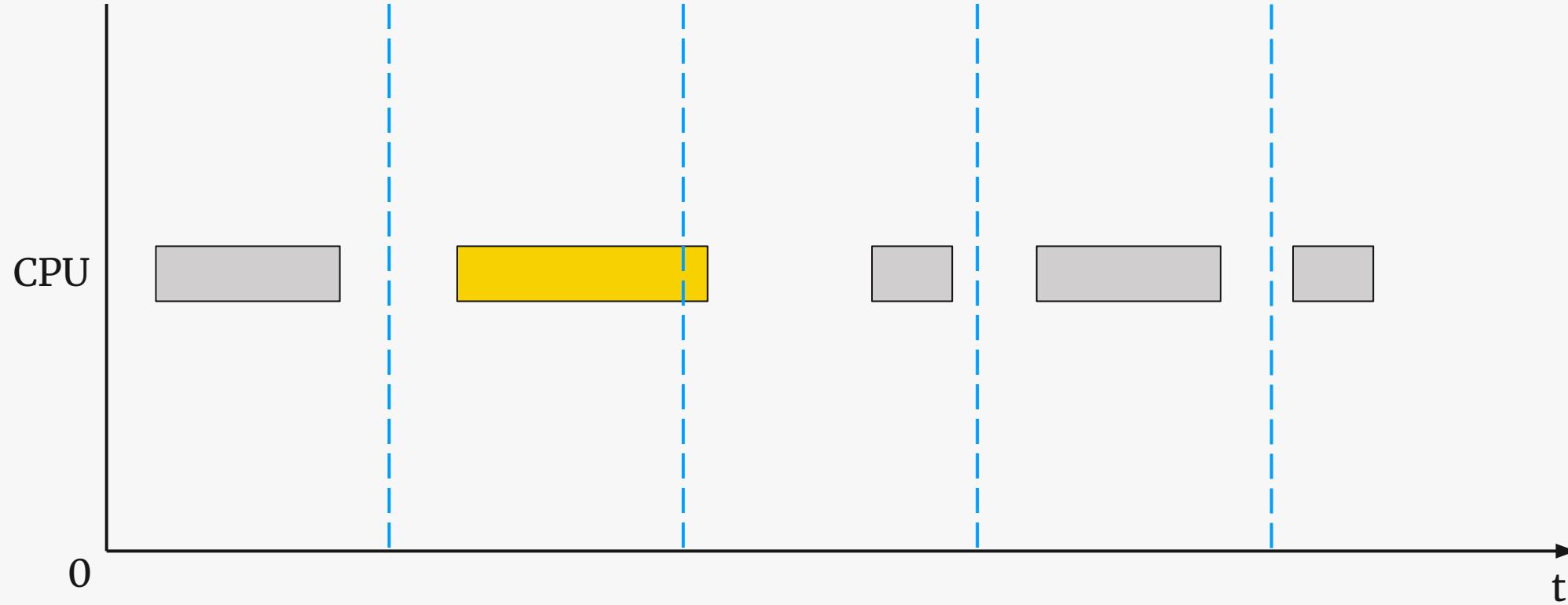
Подход 1. Sampling

1. Снимаем дампы потоков с какой-то частотой
2. Склеиваем их
3. Чем чаще встречается метод, тем больше он занимает CPU

РЕАЛЬНЫЙ ПРОФИЛЬ МЕТОДА `foo()`

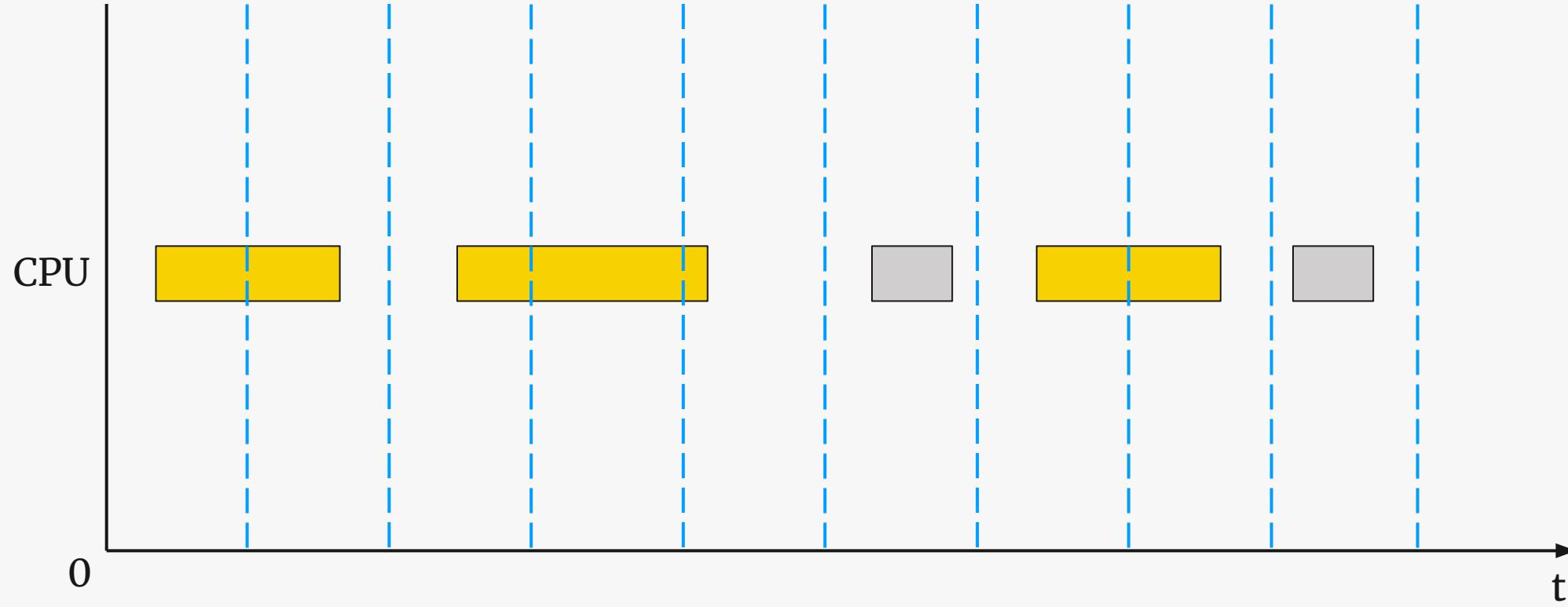


ДЕЛАЕМ СНИМКИ ПОТОКОВ ПО ТАЙМЕРУ



----- Моменты снятия дампов потоков (сэмплы)

НЕ ХВАТАЕТ – ДЕЛАЕМ ЧАЩЕ



НЕ ОТХОДИ
ОТ СТАНКА
ДО ПОЛНОЙ ОСТАНОВКИ
СЕРДЦА

SAMPLING

Плюсы

- ✓ Управляемый overhead
- ✓ Легко реализовать
- ✓ Просто использовать

Минусы

- ✗ Не всегда точен
- ✗ Подвержен safepoint bias*

* SAFEPOINT BIAS (НА ПАЛЬЦАХ)

- Поток JVM нельзя остановить в любом месте
- Но можно в т.н. safepoint'ах
 - например, на выходе из метода и повторении циклов
- Оптимизации в JVM могут влиять на исполнение:
 - методы становятся inline
 - циклы превращаются в последовательности
- Результат: safepoint'ы прореживаются, sampling теряет в точности

ГДЕ УЗНАТЬ БОЛЬШЕ

The diagram illustrates a memory model issue at a safepoint. It shows two threads, Thread 1 and Thread 2, represented as orange rectangles. Thread 1 is labeled "Поток 1" (Thread 1) and contains the text "запросили остановку" (requested stop) with an arrow pointing to its top, "работаем с памятью" (working with memory) in the middle, and "return" at the bottom. Thread 2 is labeled "Поток 2" (Thread 2) and contains "ЧТО-ТО ВЫЧИСЛЯЕМ" (something we're calculating) with a curved arrow pointing to it, and "loop" at the bottom. A horizontal line connects the two threads, representing a safepoint.

JUG.RU

Safepoint

запросили остановку

Поток 1

работаем с памятью

return

Поток 2

ЧТО-ТО ВЫЧИСЛЯЕМ

loop

одноклассники

alm works

SEMrUSH

Андрей Паньгин — Искусство Java профилирования

<https://www.youtube.com/watch?v=QiGrTvsCZmA>



SAMPLING

Плюсы

- ✓ Управляемый overhead
- ✓ Легко реализовать
- ✓ Просто использовать

Минусы

- ✗ Не всегда точен
- ✗ Подвержен safepoint bias*

SAMPLING

Плюсы

- ✓ Управляемый overhead
- ✓ Легко реализовать
- ✓ Просто использовать

Минусы

- ✗ Не всегда точен
- ✗ Подвержен safepoint bias*
- ✗ Не умеет считать вызовы**

** ПОДСЧЕТ ЧИСЛА ВЫЗОВОВ

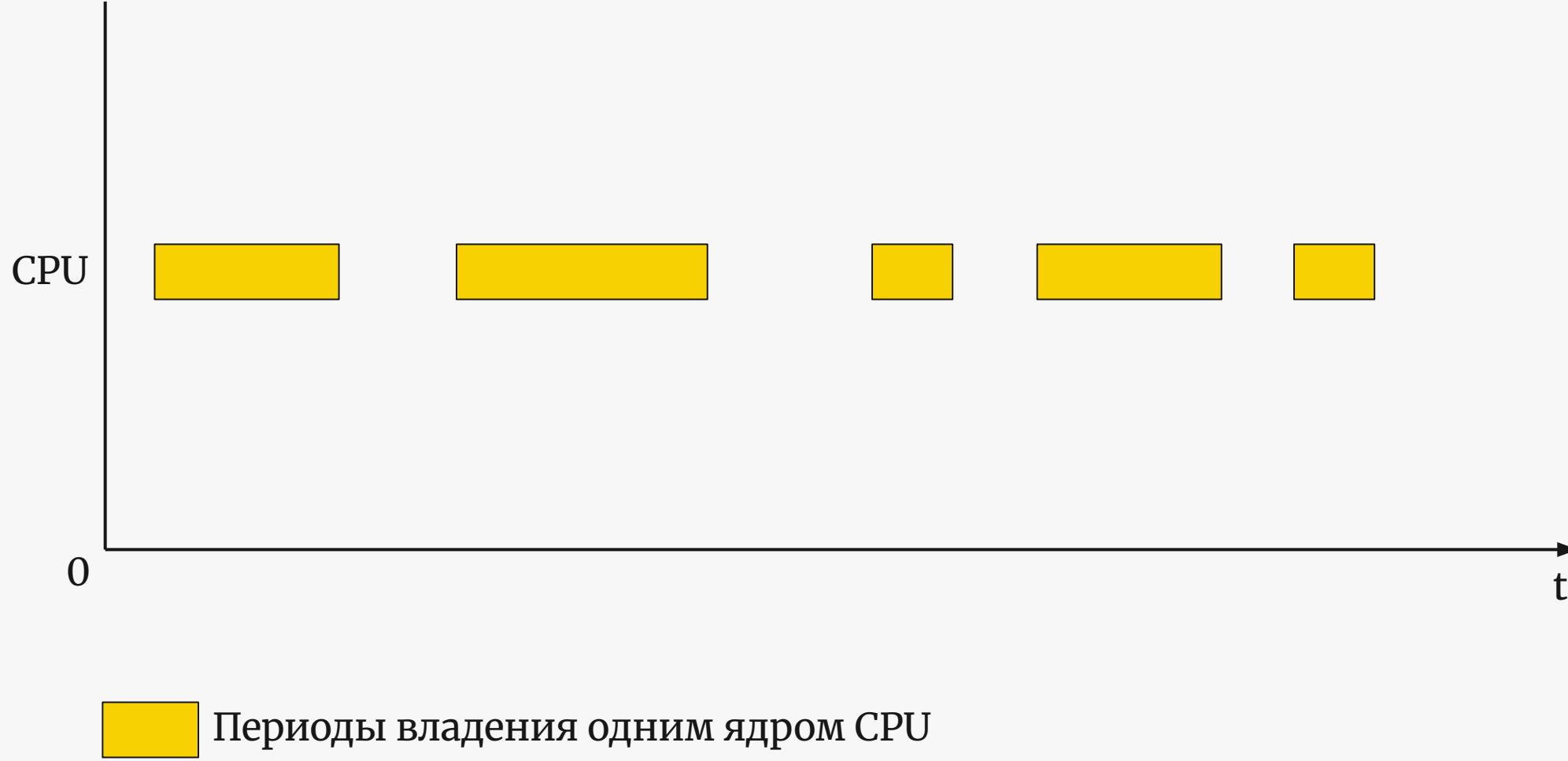
- Пример тормозящей цепочки методов:
 - `findUsersByIds` → `fetchDataFromDB` → `executeSQL`
- Возможные причины:
 - метод `executeSQL()` долго работает **с большим IN**
 - метод `findUsersByIds()` делает запросы **по каждому ID**
- Sampling укажет **только на executeSQL()**
- Различить можно **по числу вызовов**

КАК ПРОФИЛИРОВАТЬ СРУ

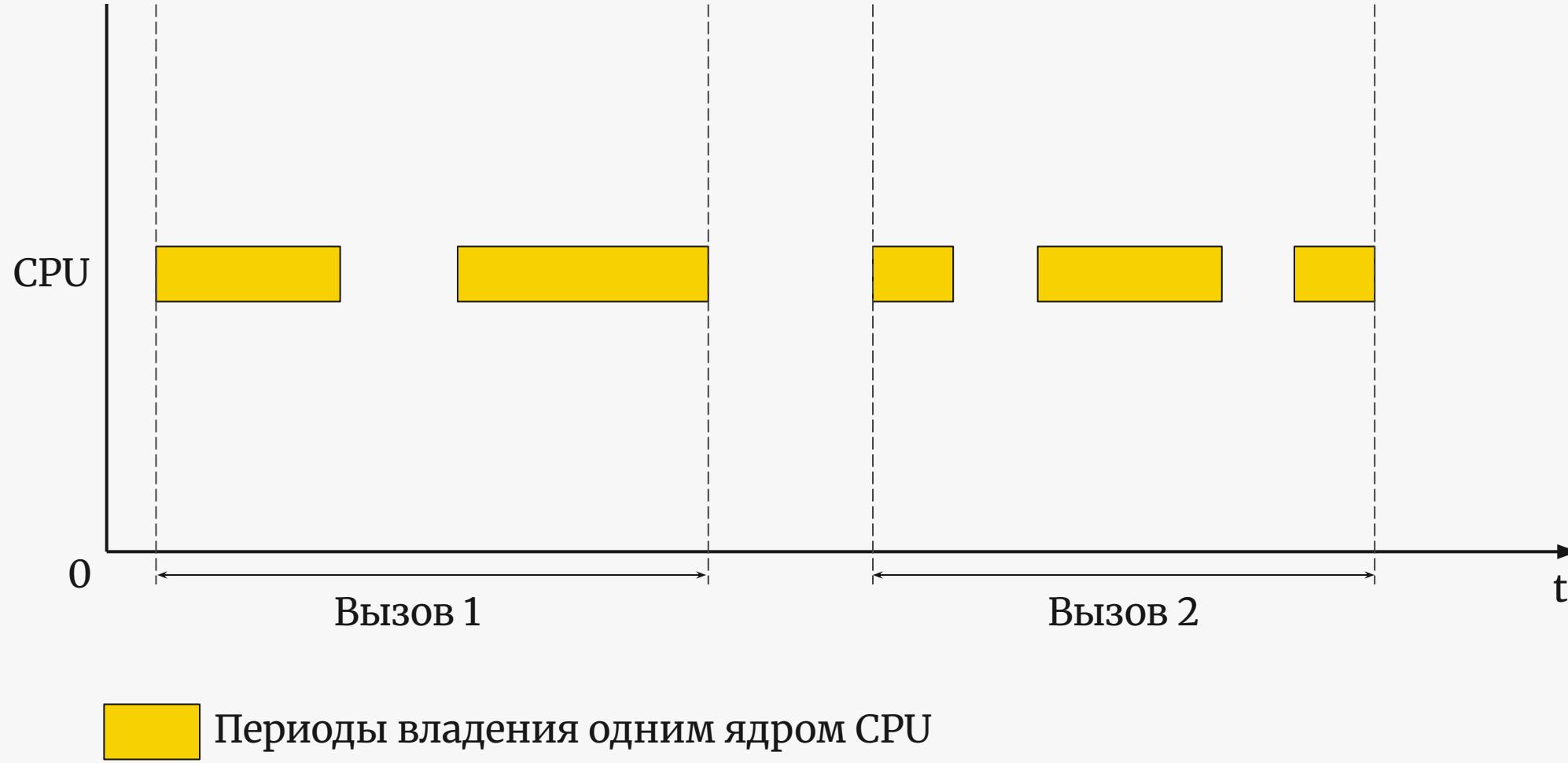
Подход 2. Tracing a.k.a. instrumentation

1. Внедряем в байт-код метки вызова методов
2. Строим стектрейсы по ним
3. Получаем точные временные рамки работы методов

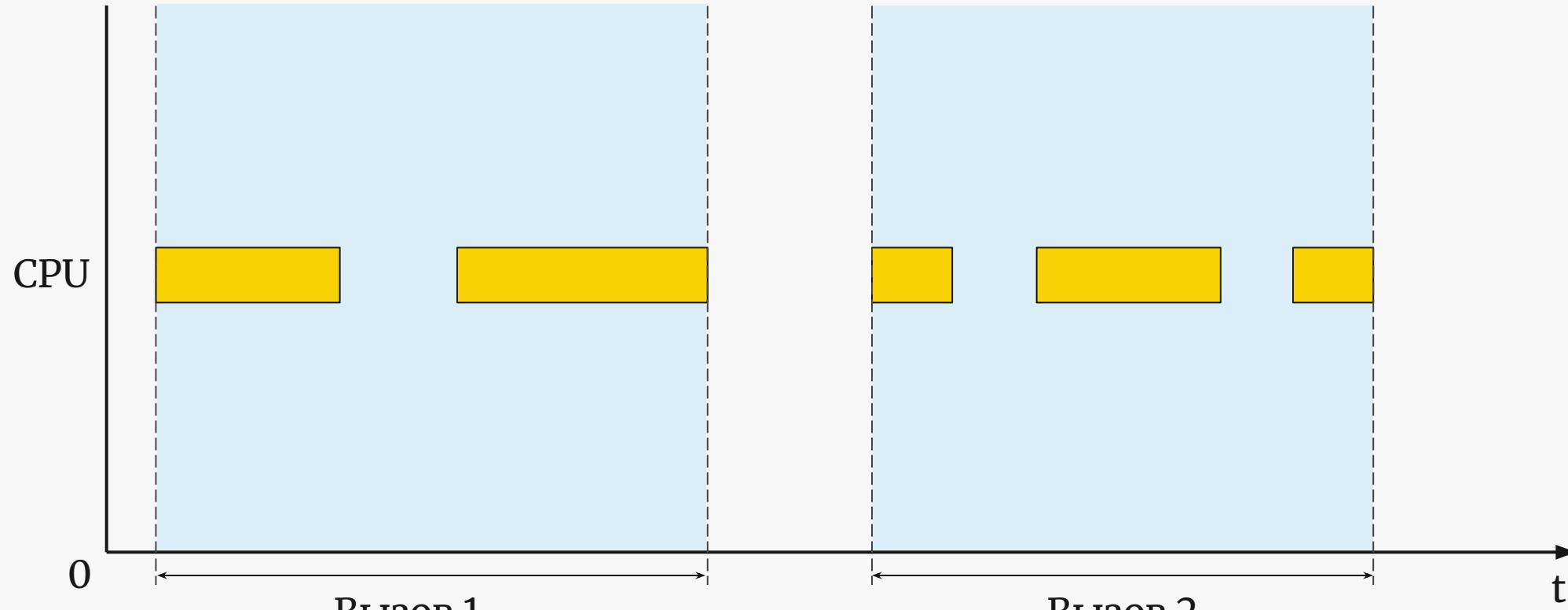
РЕАЛЬНЫЙ ПРОФИЛЬ МЕТОДА foo()



РЕАЛЬНЫЙ ПРОФИЛЬ МЕТОДА foo()



STACK-TRACING



Периоды трассировки метода (от входа до выхода)

TRACING

Плюсы

- ✓ Высокая точность
- ✓ Подсчет числа вызовов
- ✓ Путь к high-level метрикам



TRACING

Плюсы

- ✓ Высокая точность
- ✓ Подсчет числа вызовов
- ✓ Путь к high-level метрикам

Минусы

- ✗ Лютый overhead
- ✗ Сложность настройки
- ✗ Может влиять на профиль*

* (ДЕ)ОПТИМИЗАЦИЯ (НА ПАЛЬЦАХ)

- Наиболее “горячие” методы оптимизируются самой JVM
- Их исполняемый код может не соответствовать исходному
- Включение трейсинга меняет исходный класс
- Это приводит к:
 - (не)применению других оптимизаций
 - деоптимизации ранее сгенерированного кода
- Результат: профиль меняется, **трейсинг может врать**

ГДЕ УЗНАТЬ БОЛЬШЕ

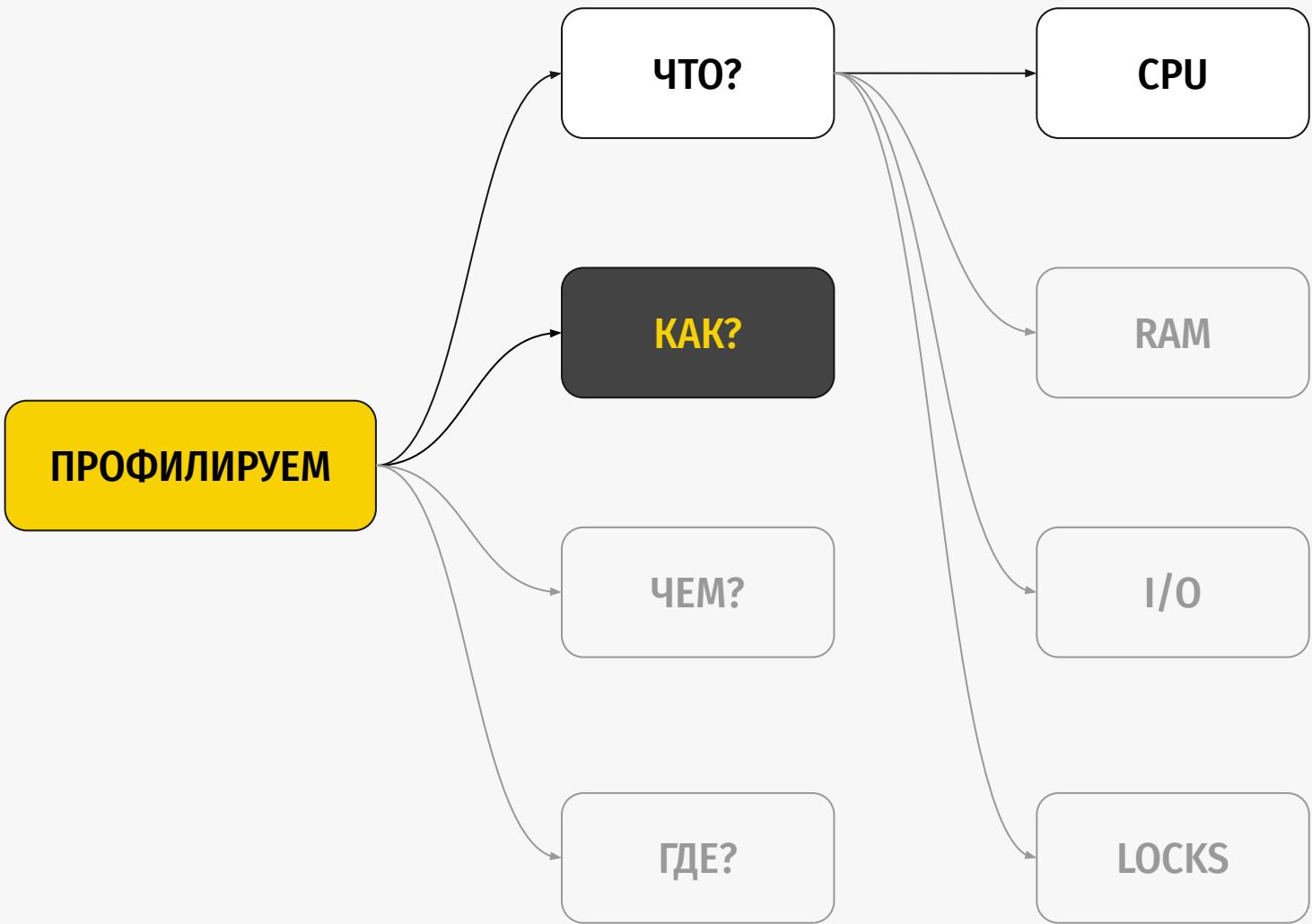
A presentation slide featuring a purple background with white and teal accents. At the top left is the jbreak; logo, which includes a stylized white wolf head icon and the text "jbreak;" followed by "Новосибирск 2017". Below the logo is the speaker's name, "Иван Крылов", and his affiliation, "Azul Systems". A large pink rectangular box contains the title of the talk, "Жизненный цикл JIT кода". At the bottom of the slide is a URL: <https://www.youtube.com/watch?v=9valOxgDbI>.

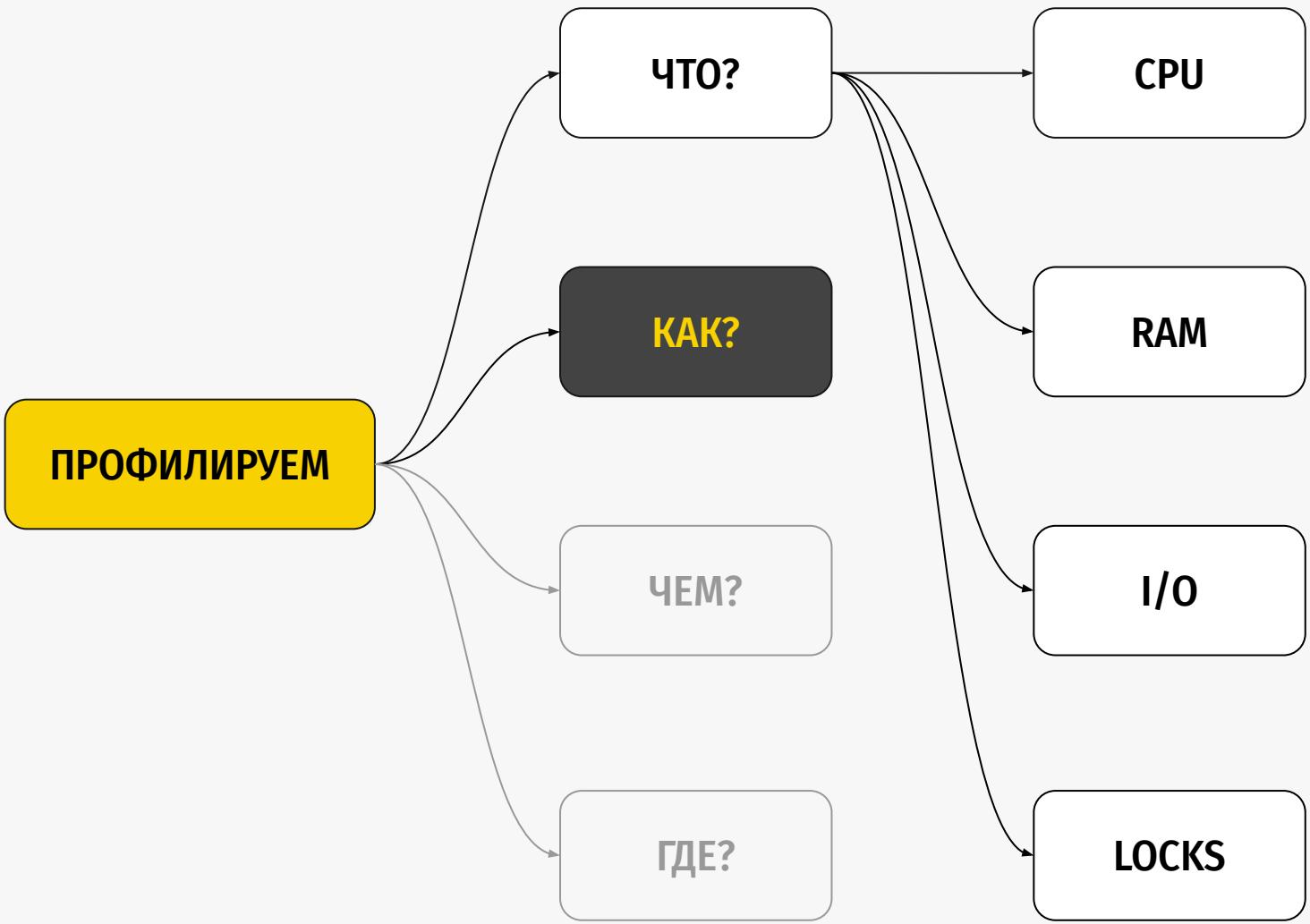
Иван Крылов
Azul Systems

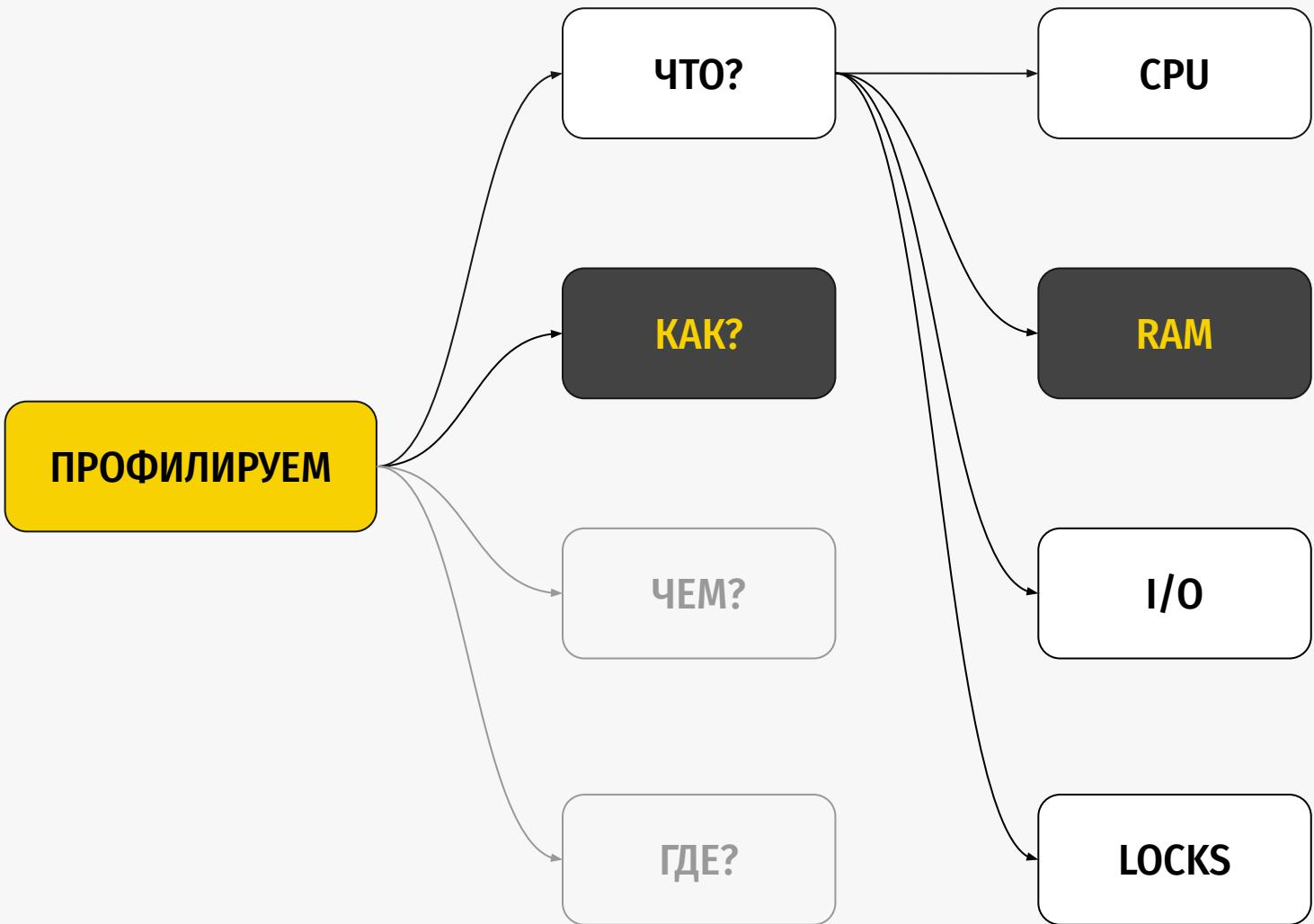
Жизненный цикл
JIT кода

<https://www.youtube.com/watch?v=9valOxgDbI>









АЛЛОКАЦИЯ ПАМЯТИ В КУЧЕ (НА ПАЛЬЦАХ)

- Новые объекты создаются в поточно-локальных буферах
 - TLAB - Thread-Local Allocation Buffer
- Если объект не влезит в текущий TLAB, создастся новый
- Если объект огромный, он может быть создан вне TLAB
 - это требует синхронизации, поэтому называется **slow path**
- На оба этих случая в JVM есть события, на них можно подписаться
- Подробнее: [Как JVM аллоцирует объекты?](#) (Хабр)

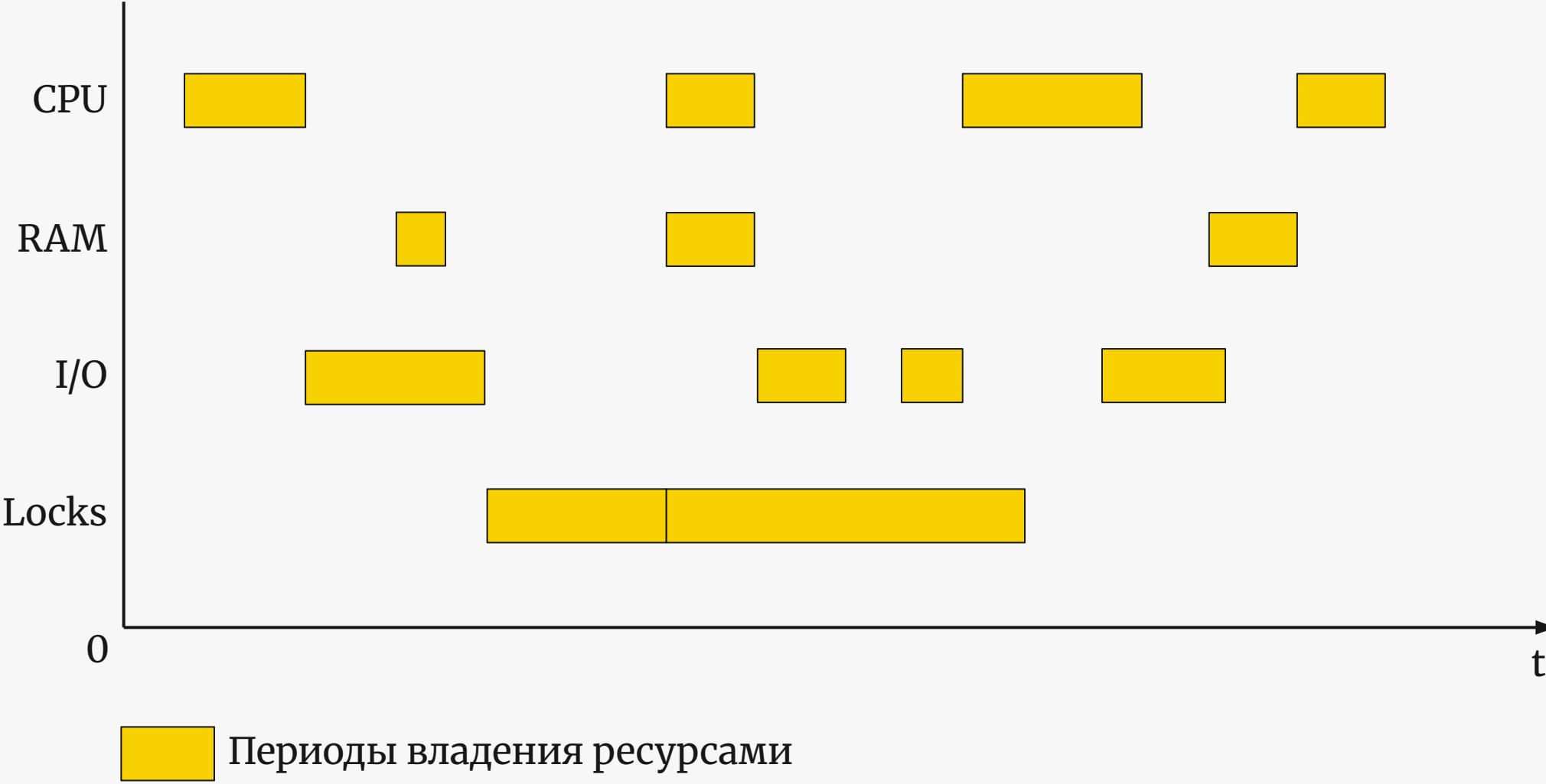


КАК ПРОФИЛИРОВАТЬ ДРУГИЕ РЕСУРСЫ

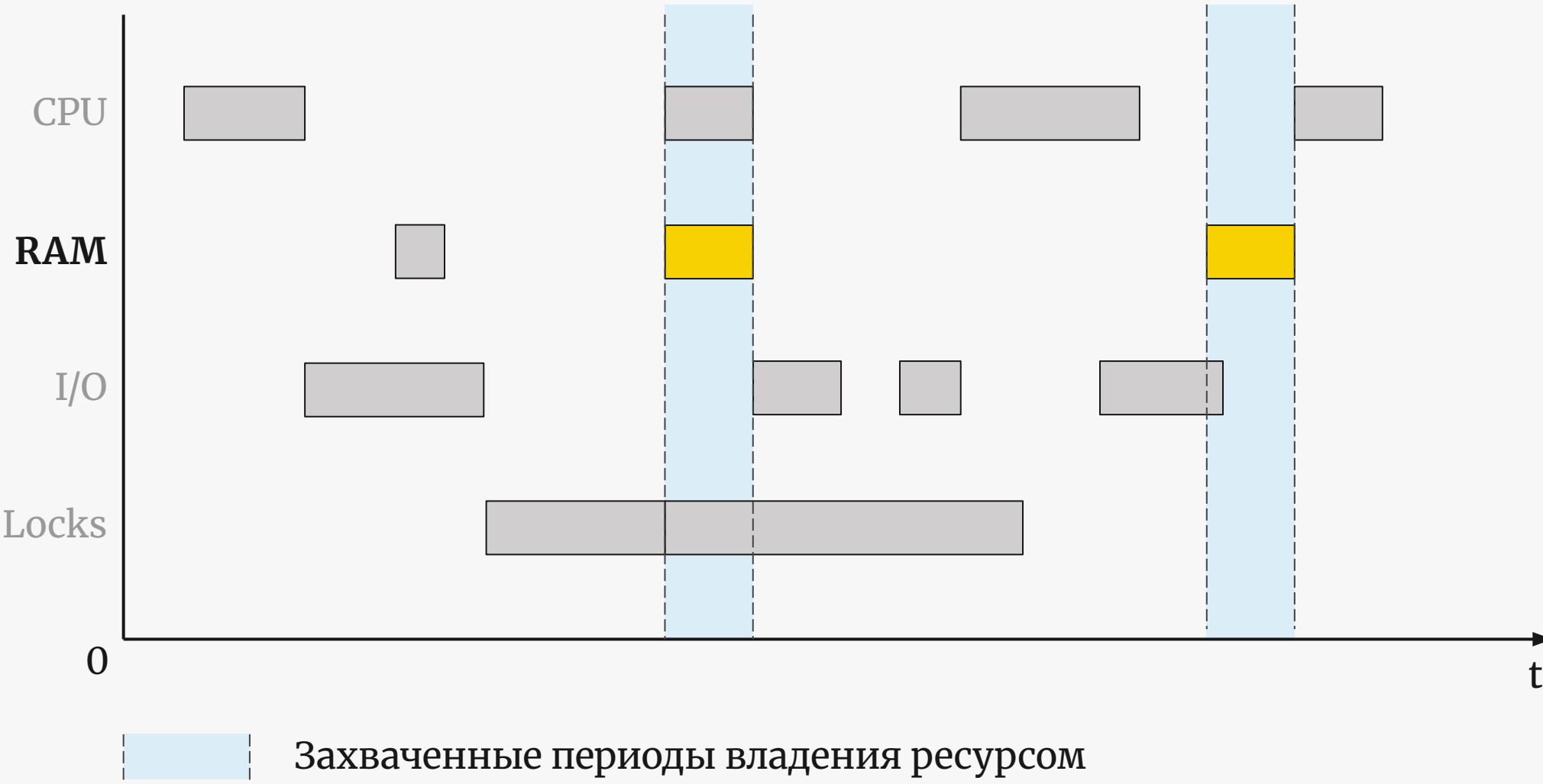
Подход 3. Event-based

1. Полагаемся на события обращения к ресурсам **внутри JVM**
2. Собираем стектрейсы с них
3. Получаем точную картину по ресурсам

ТИПА РЕАЛЬНЫЙ ПРОФИЛЬ МЕТОДА `foo()`

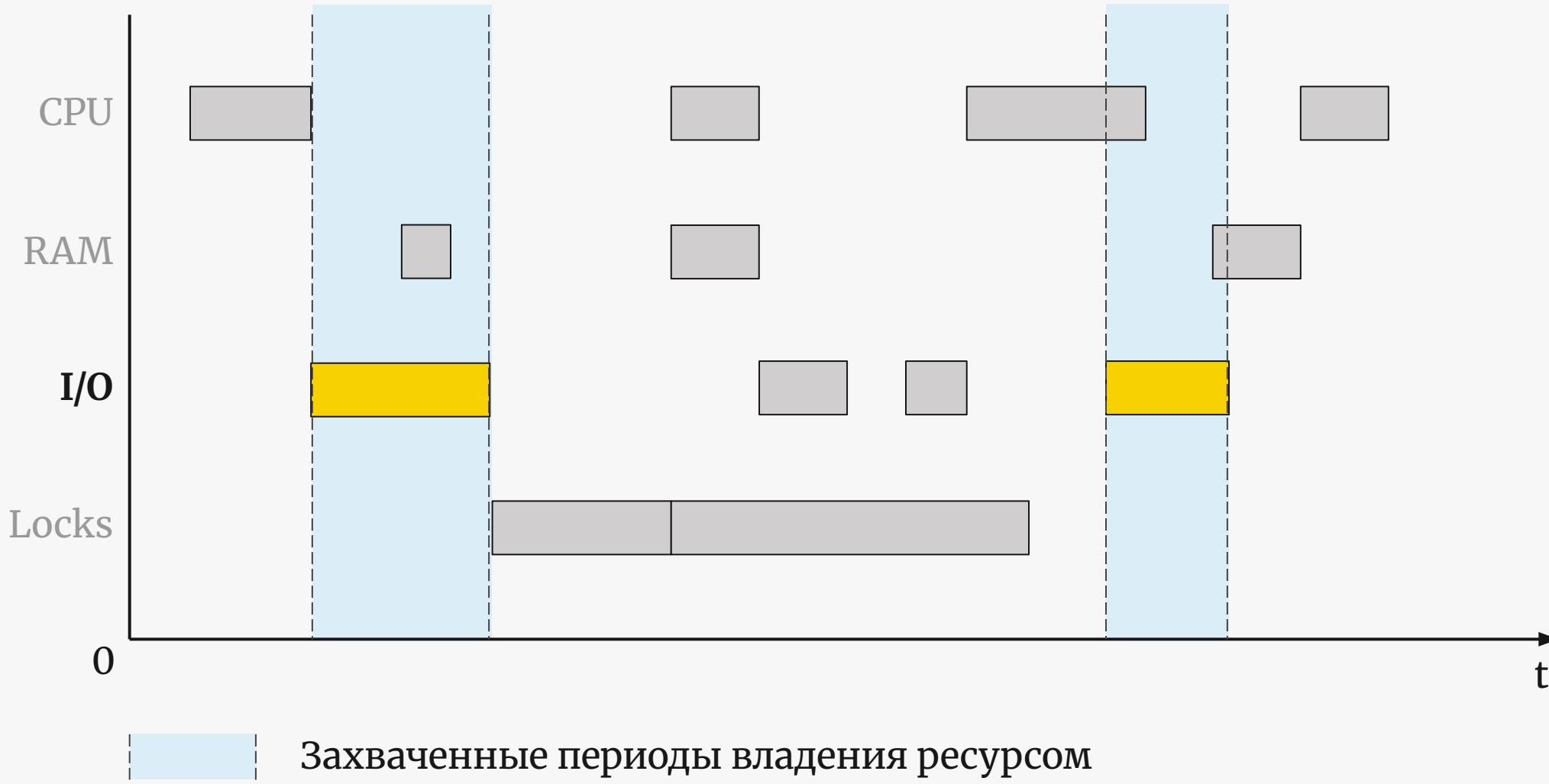


ЗАХВАТ АЛЛОКАЦИЙ ЧЕРЕЗ СОБЫТИЯ

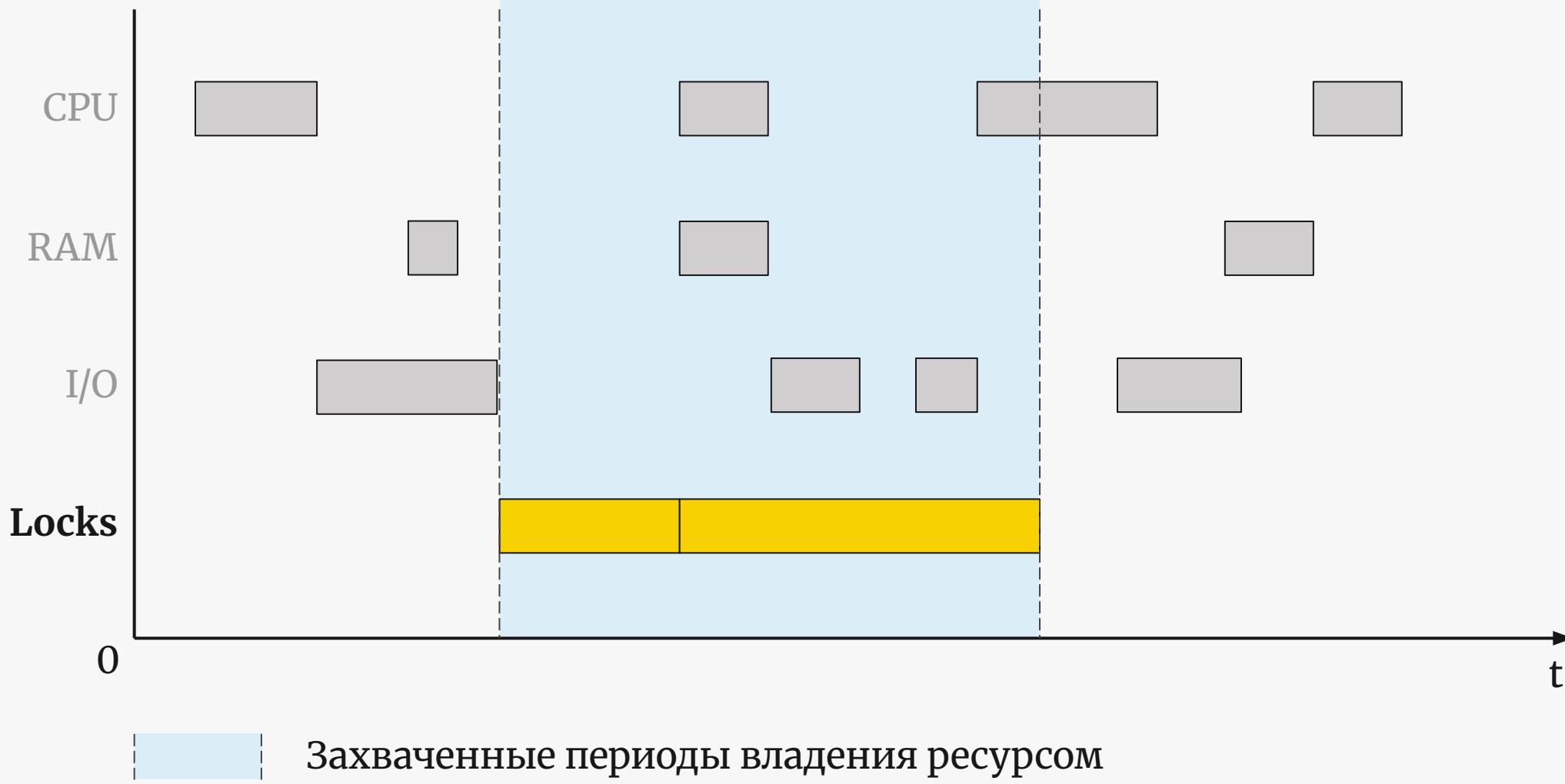


Захваченные периоды владения ресурсом

ЗАХВАТ ВВОДА-ВЫВОДА ЧЕРЕЗ СОБЫТИЯ



ЗАХВАТ БЛОКИРОВОК ЧЕРЕЗ СОБЫТИЯ



Захваченные периоды владения ресурсом

EVENTS

Плюсы

- ✓ Высокая точность
- ✓ Умеренный overhead
- ✓ Нативная поддержка

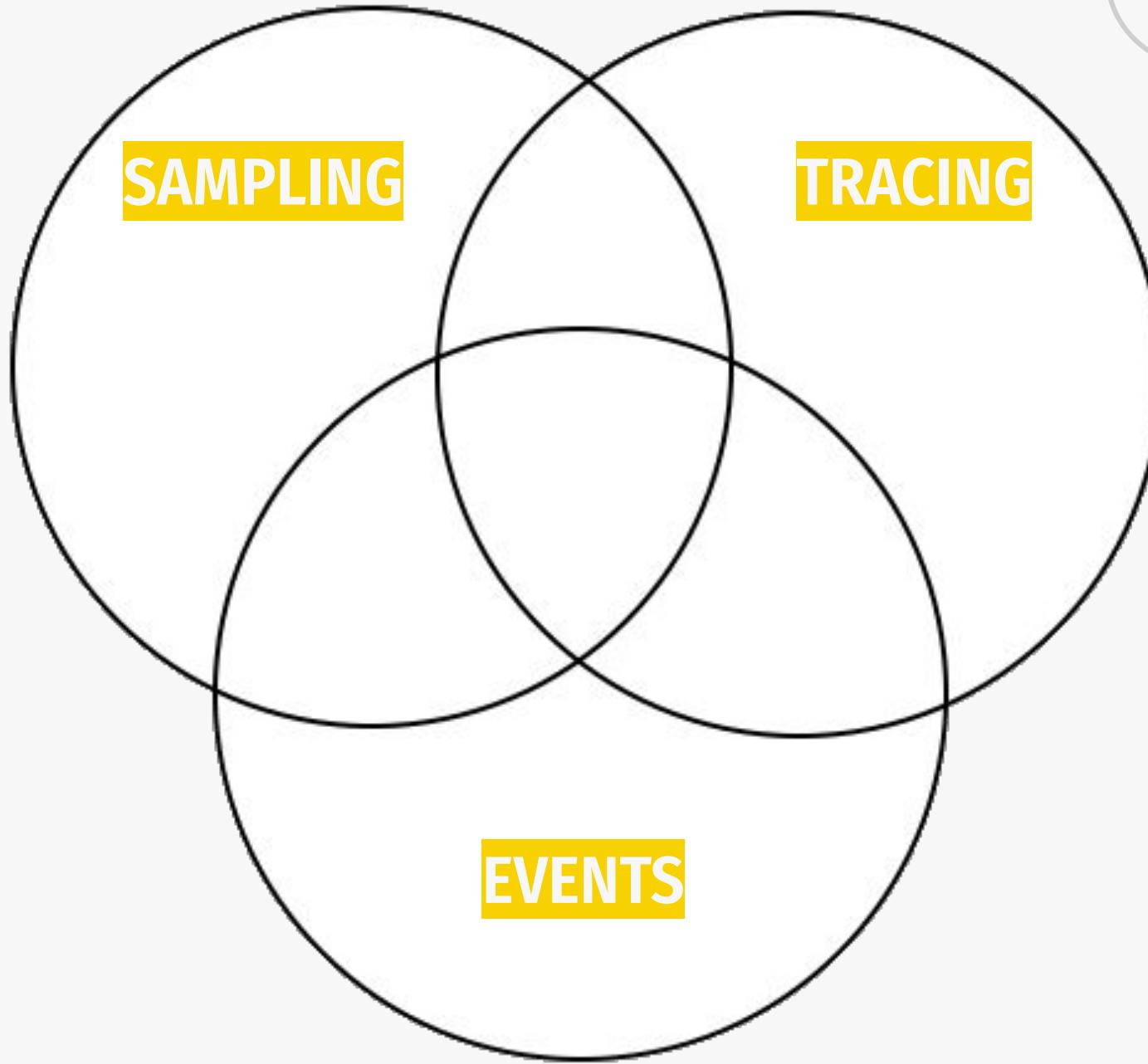
Минусы

- ✗ Слабая связь с бизнес-логикой приложения
- ✗ Ограниченнная применимость

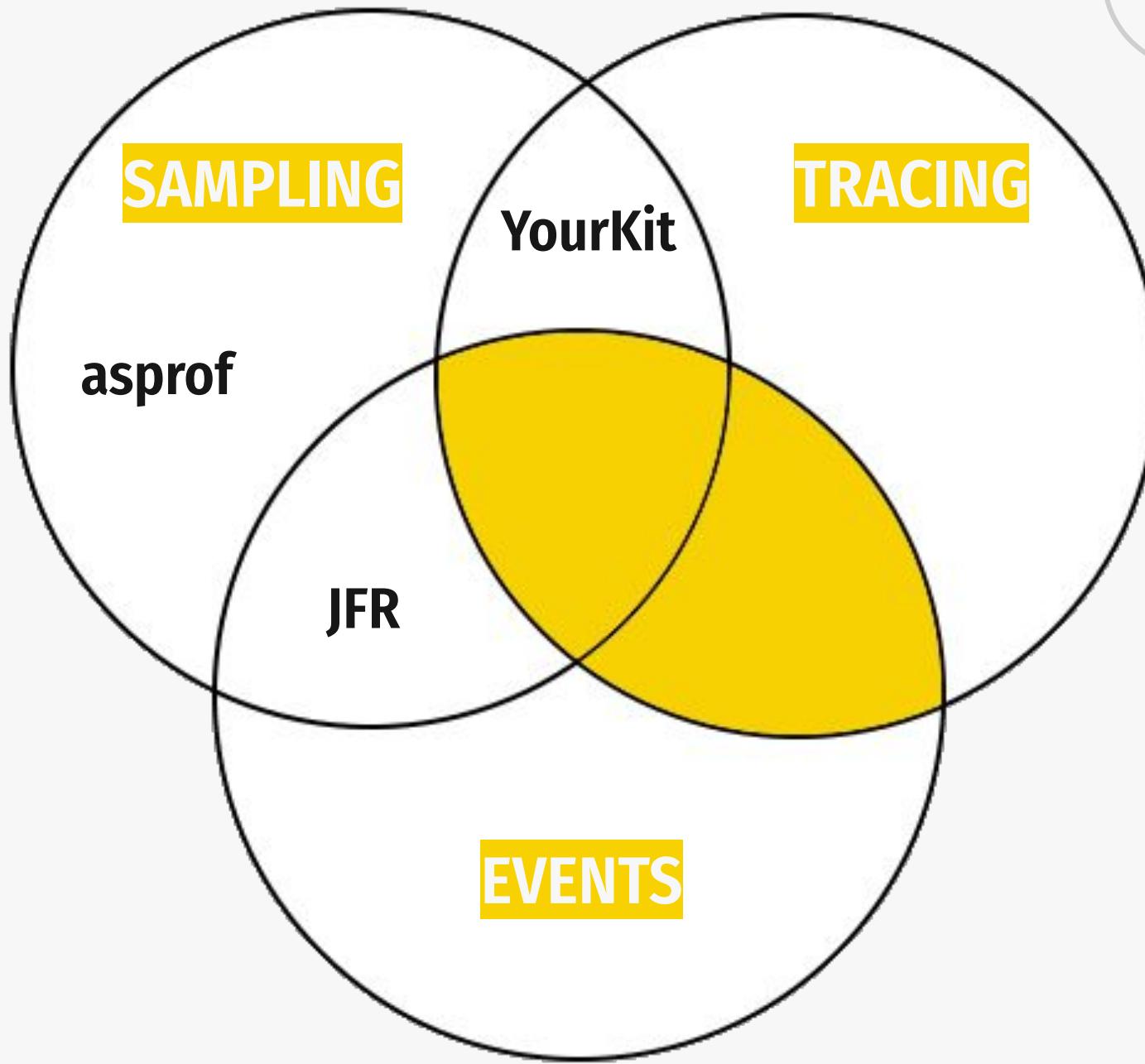
ИНСТРУМЕНТЫ-ПРЕДСТАВИТЕЛИ

- Sampling
 - Async-profiler a.k.a. asprof
- Tracing
 - YourKit Java Profiler a.k.a. YourKit
- Events
 - IDK Flight Recorder a.k.a. JFR

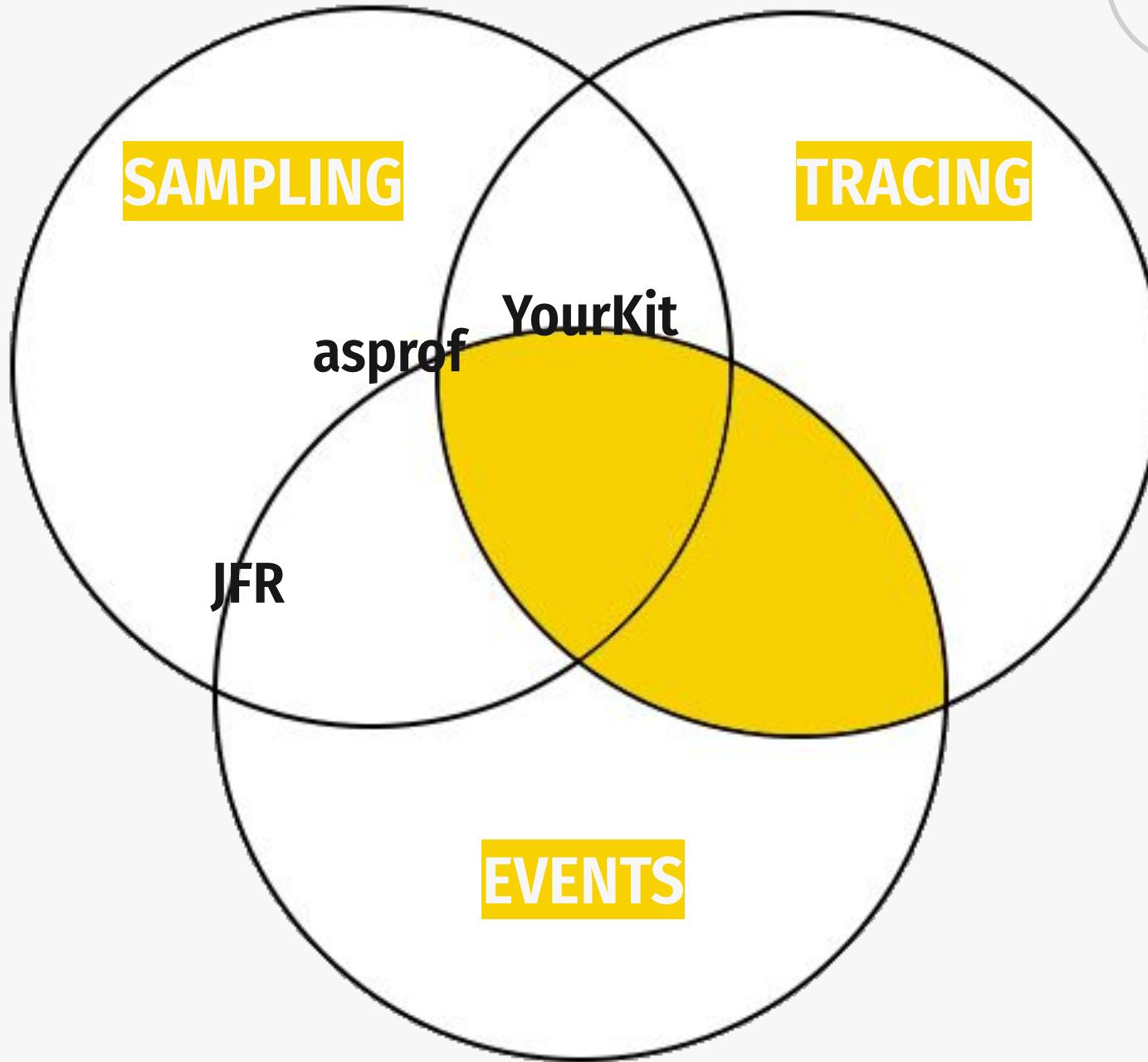
А ЕСЛИ
ТОЧНЕЕ

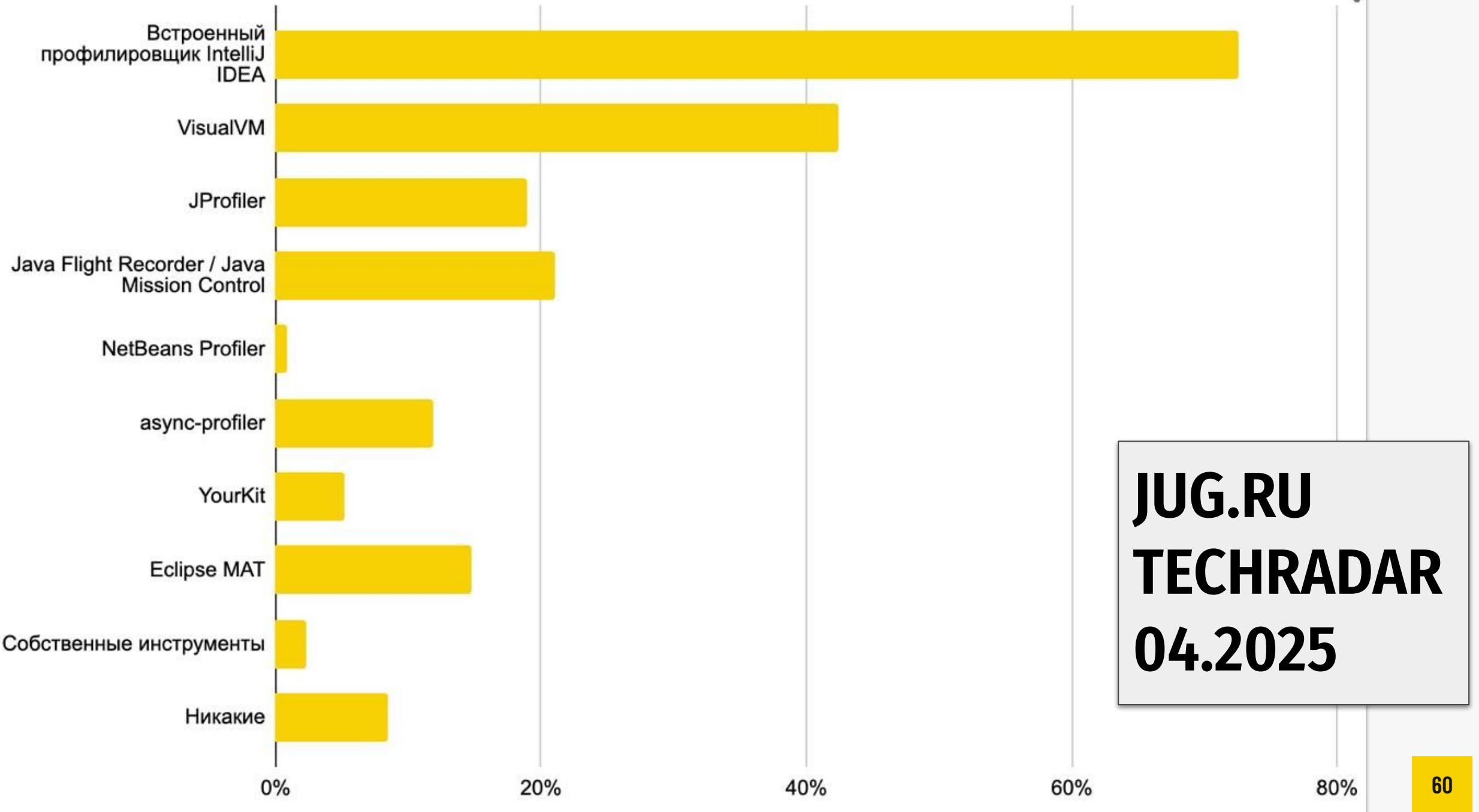


А ЕСЛИ
ТОЧНЕЕ

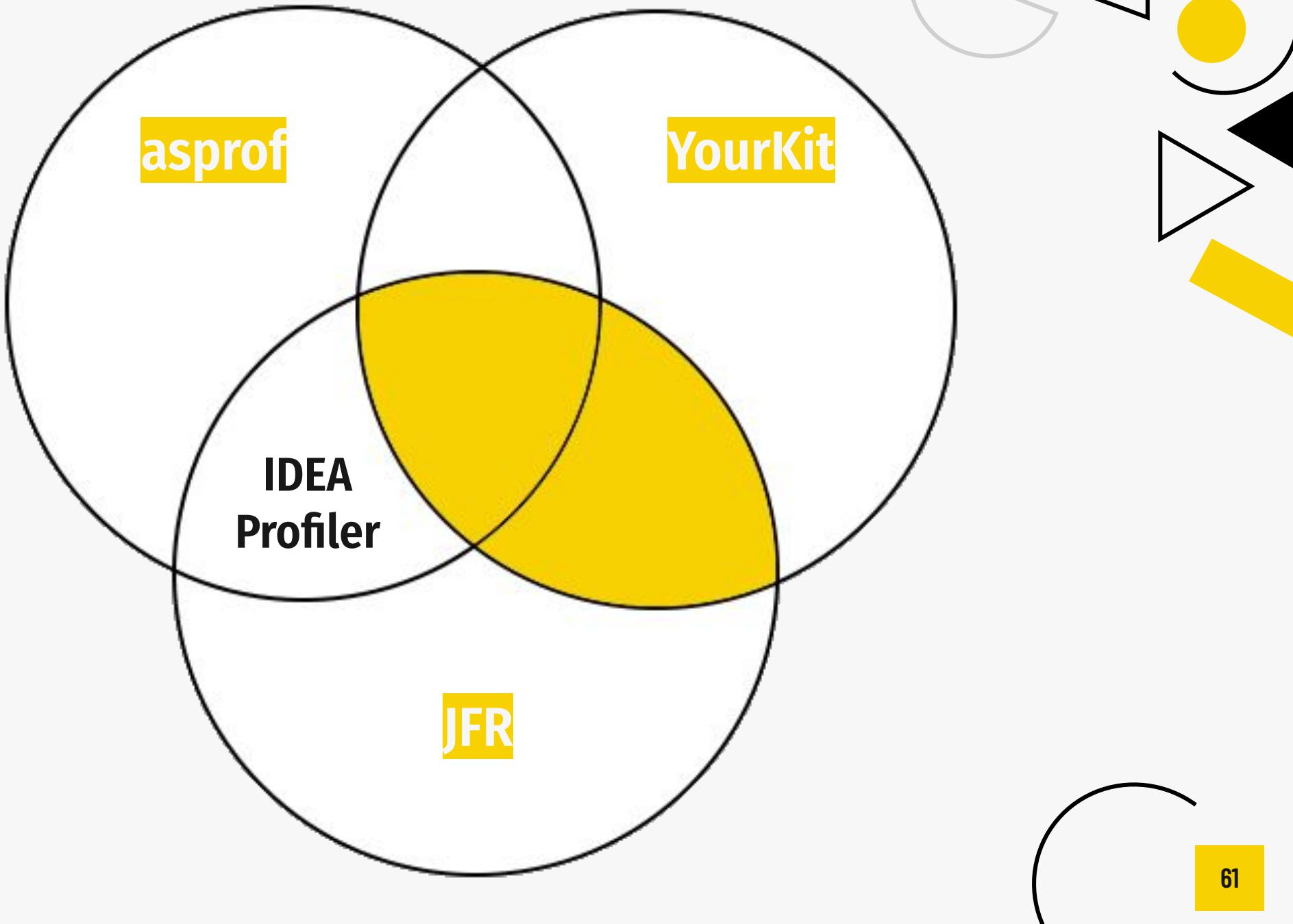


А ЕСЛИ
ЕЩЕ
ТОЧНЕЕ





А ГДЕ ЖЕ ПРОФАЙЛЕР ИЗ IDEA?



А ГДЕ ЖЕ ПРОФАЙЛЕР <XXX>?

- VisualVM
- JProfiler
- Digma.AI
- Alibaba Arthas
- NetBeans Profiler
- ...
- OTEL Agent
- DataDog
- NewRelic
- Micrometer
- ...

O – Observability



А ГДЕ ЖЕ ПРОФАЙЛЕР <XXX>?

- VisualVM
- JProfiler
- Digma.AI
- Alibaba Arthas
- NetBeans Profiler
- ...
- OTEL Agent
- DataDog
- NewRelic
- Micrometer
- ...
- JIT log
- GC Log
- println()
- ...

Вне фокуса доклада

ПОЧЕМУ НЕТ ЕДИНОГО СУПЕР ПРОФАЙЛЕРА

Joker<?> 2017

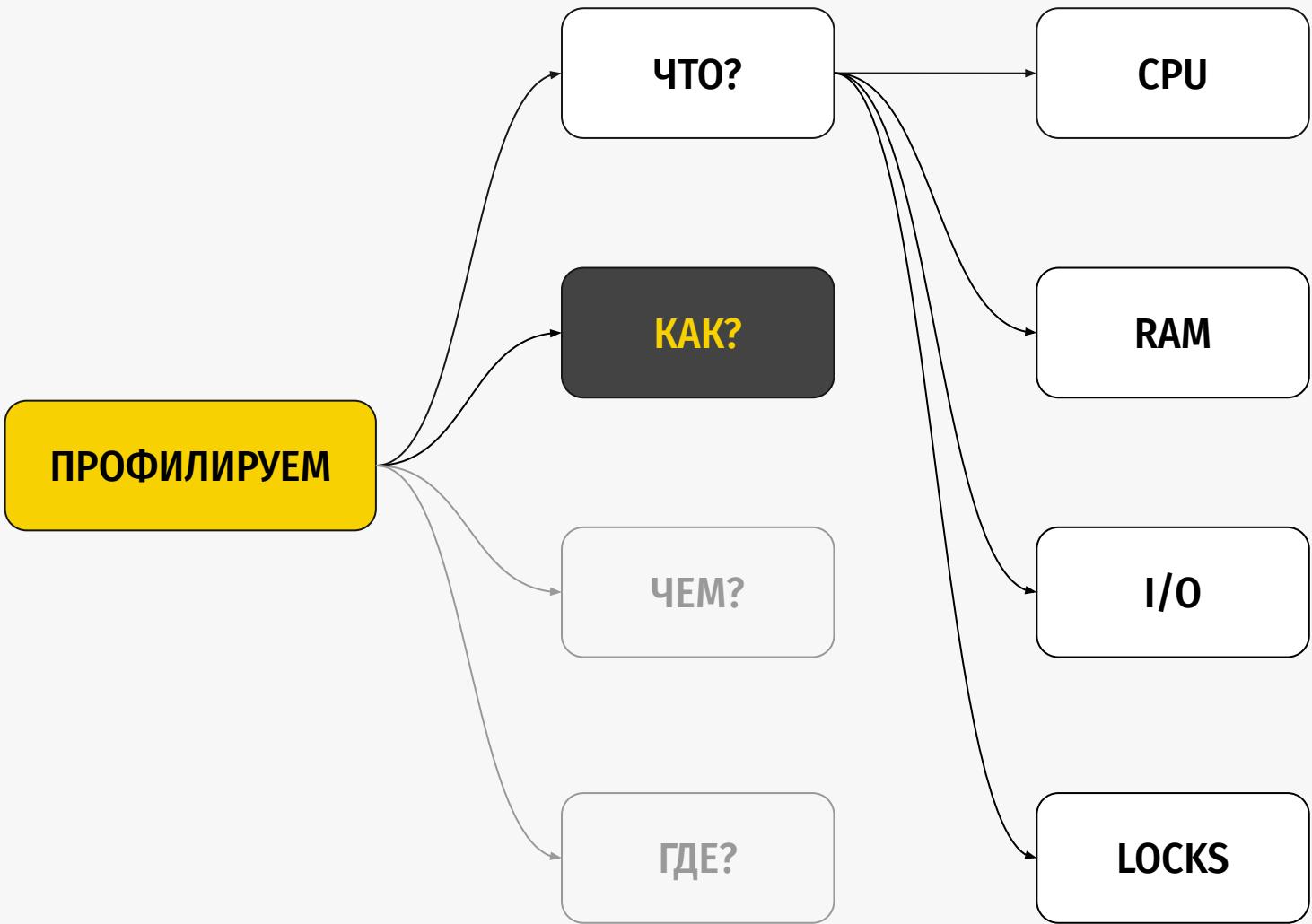
Nitsan Wakart

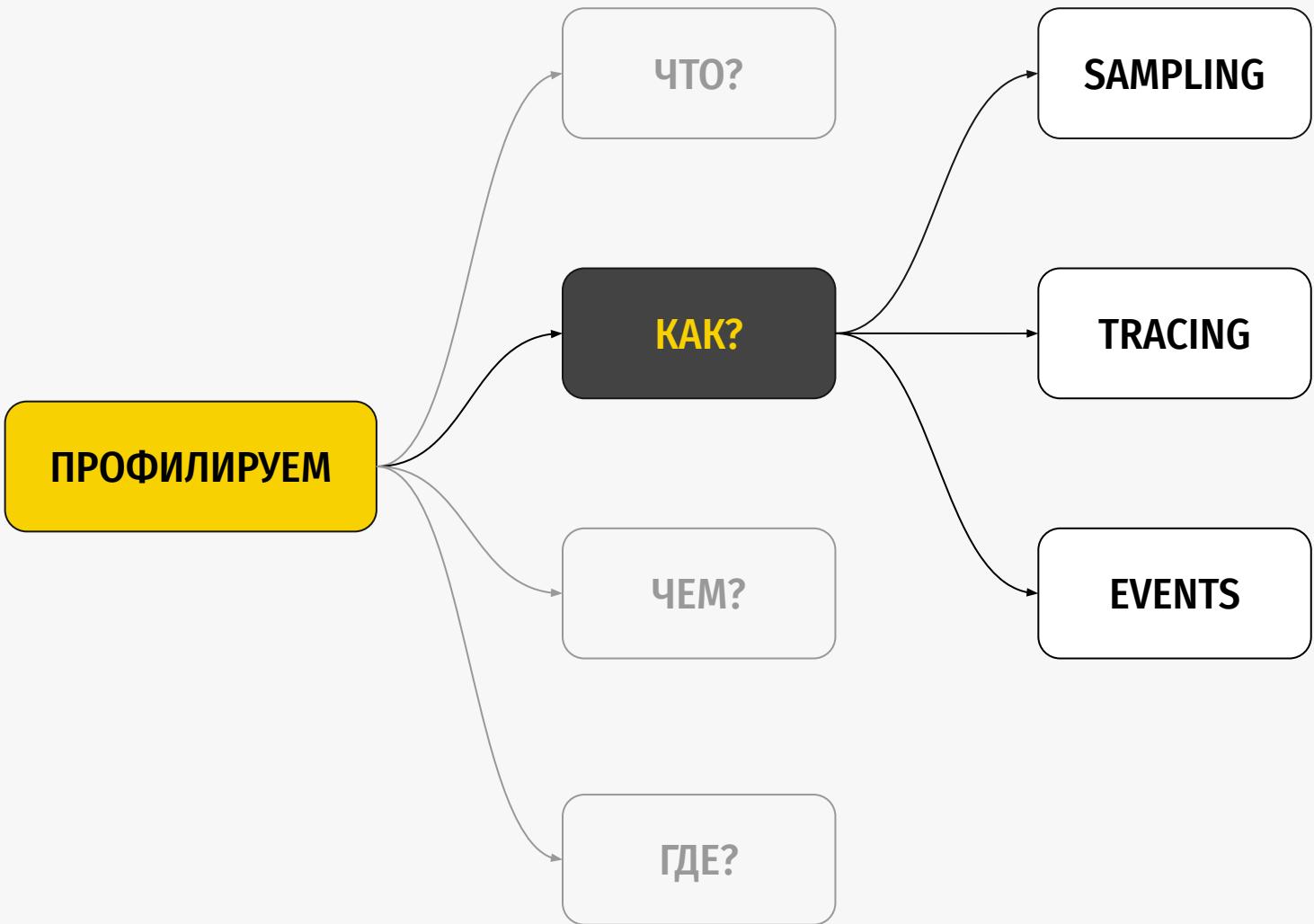
TTNR Labs

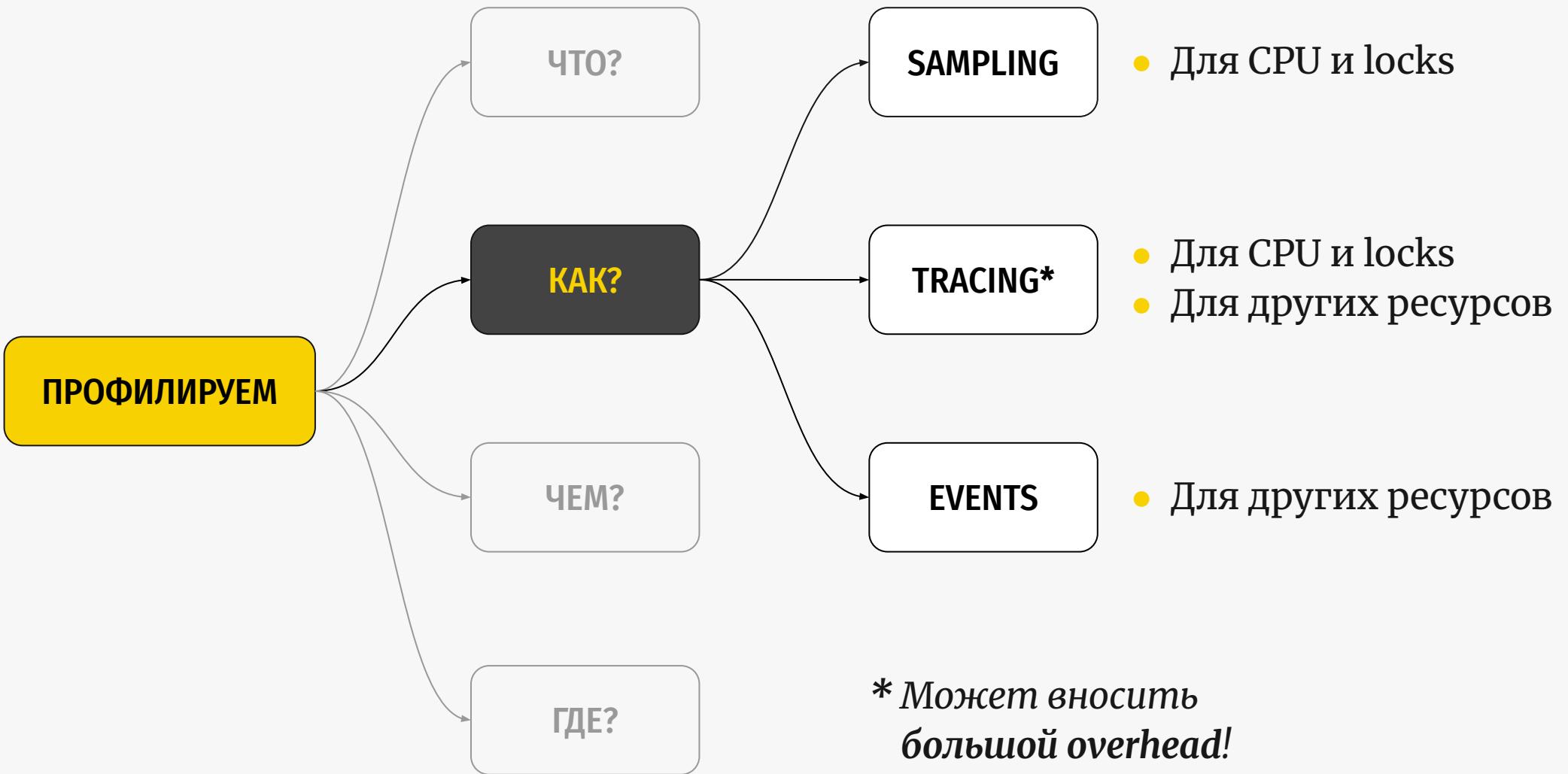
Profilers are lying
hobbitses

<https://www.youtube.com/watch?v=7IkHlqPeFjY>









- Для CPU и locks
- Для CPU и locks
- Для других ресурсов
- Для других ресурсов

* Может вносить
большой overhead!

ОБЩИЙ ПОДХОД

1. Подключить профайлер к приложению
2. Запустить профилирование
3. Выполнить проблемное действие
4. Остановить профилирование
5. Сохранить результаты

← *если знаешь какое*



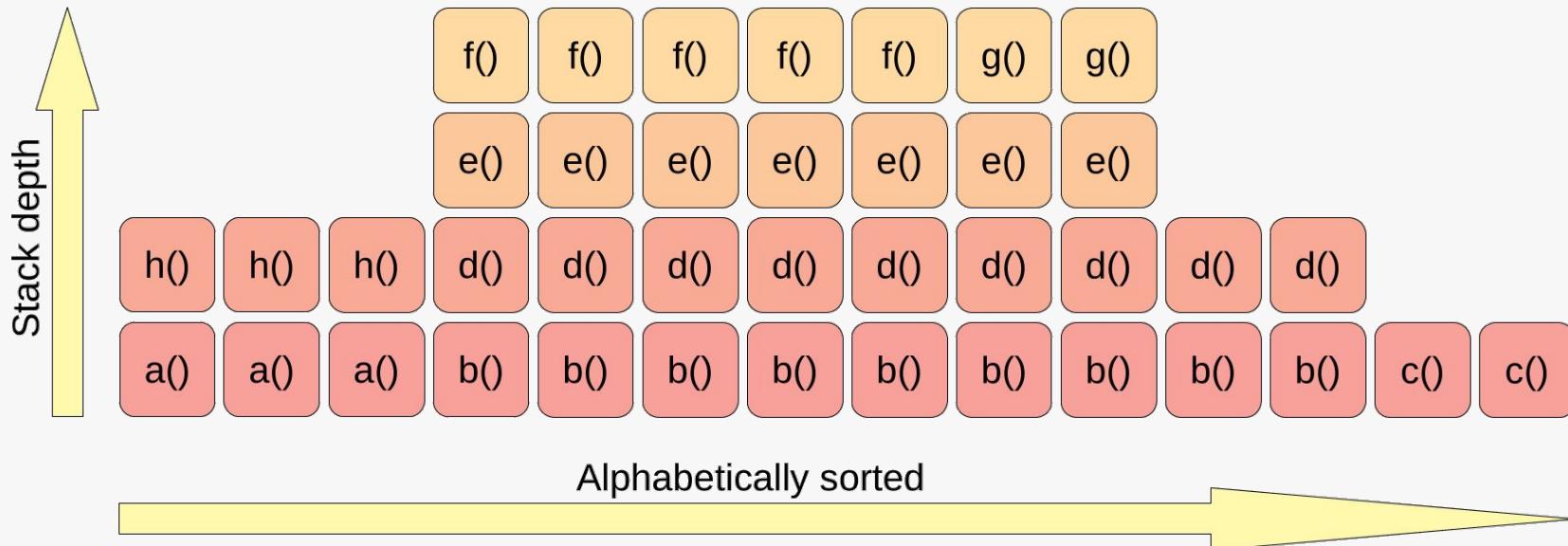
ВАРИАНТЫ РЕЗУЛЬТАТОВ ПРОФИЛИРОВАНИЯ

- Обычный текст (в т. ч. CSV)
- Записи JFR
 - Бинарные “логи” со встроенным сжатием
 - Поддерживают разнотипные события в одном файле
- Flame Graphs 🔥
 - Интерактивная визуализация множества стектрейсов
 - Как правило, в SVG

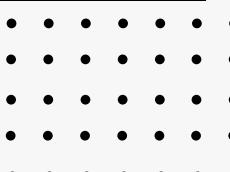


КАК СТРОИТСЯ FLAME GRAPH

1. Выписать стектрейсы в алфавитном порядке нижнего этажа



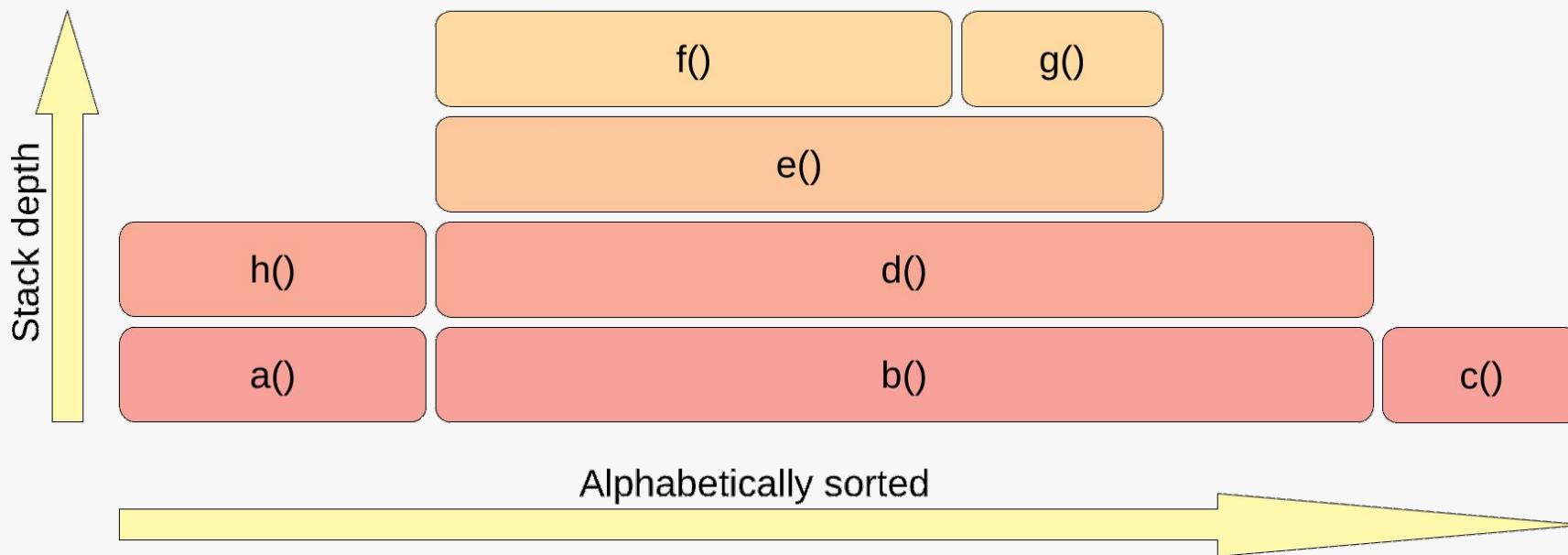
<https://krzysztofslusarski.github.io/2022/12/12/async-manual.html#flames>





КАК СТРОИТСЯ FLAME GRAPH

2. Объединить одинаковые соседние этажи

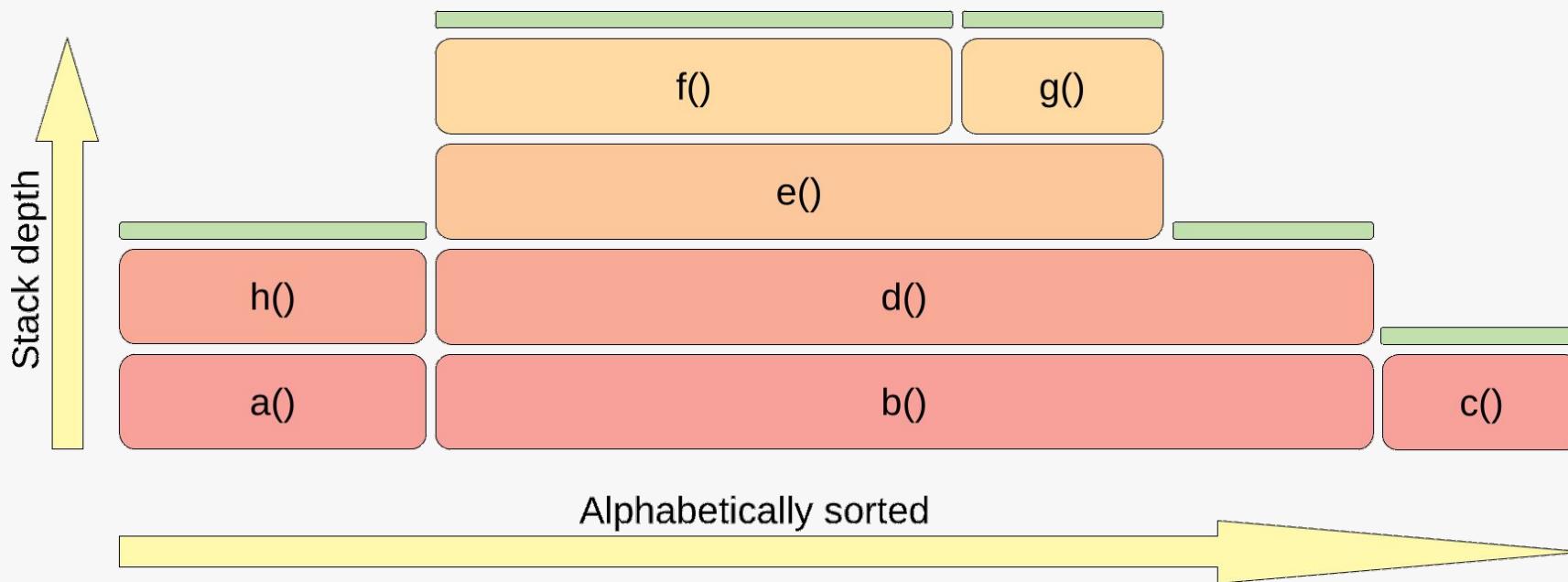


<https://krzysztofslusarski.github.io/2022/12/12/async-manual.html#flames>



КАК СТРОИТСЯ FLAME GRAPH

3. Задать ширину верхнего этажа по измеряемому ресурсу



<https://krzysztofslusarski.github.io/2022/12/12/async-manual.html#flames>



CPU profile

Produced by [async-profiler](#)



ПРИМЕРЫ ИЗМЕРЯЕМЫХ РЕСУРСОВ

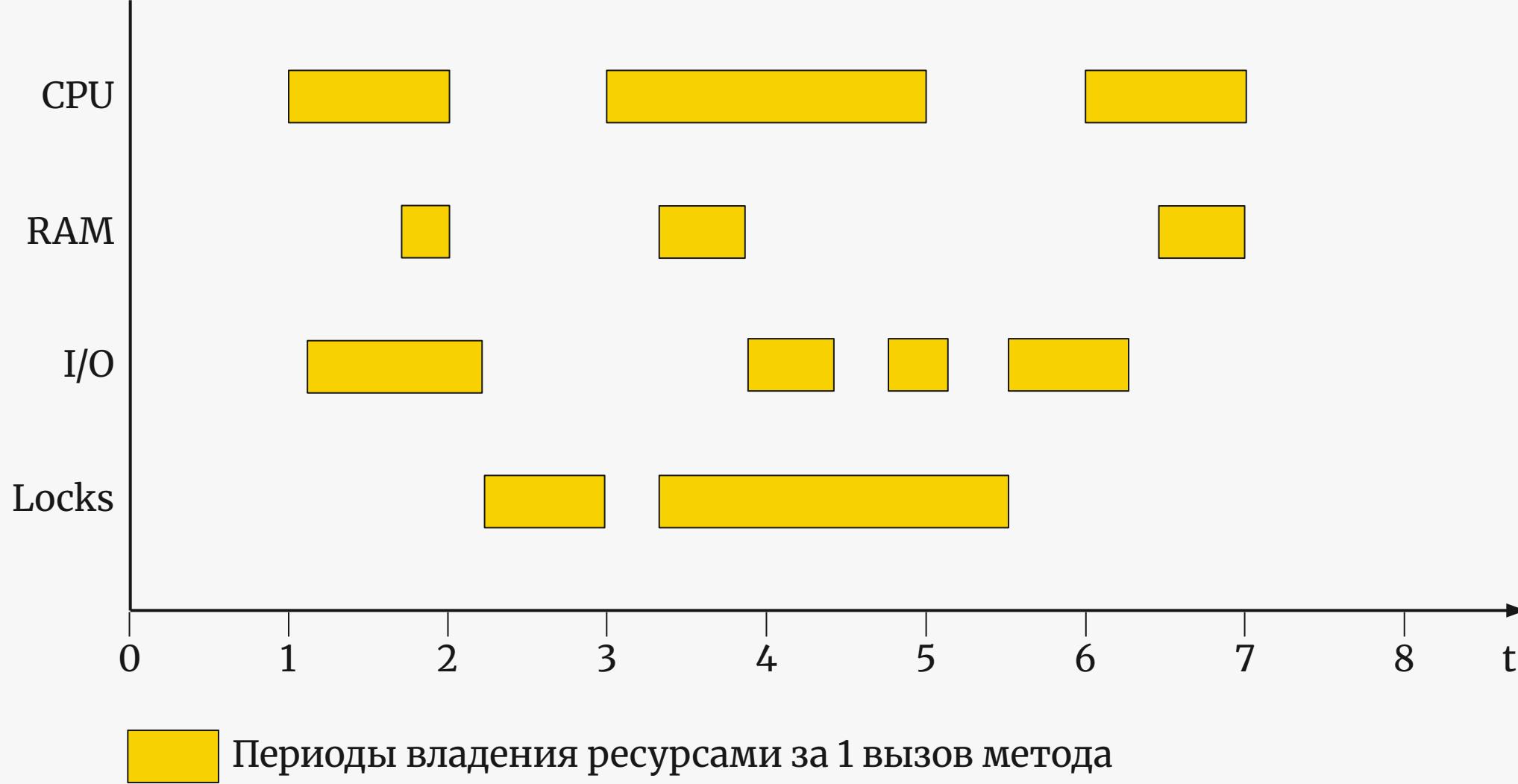
- Размеры/количество алоцированных объектов в куче
- Объем переданных/записанных данных
- Время, проведенное на процессоре
- Время, прошедшее за наблюдение

} чё? o_0

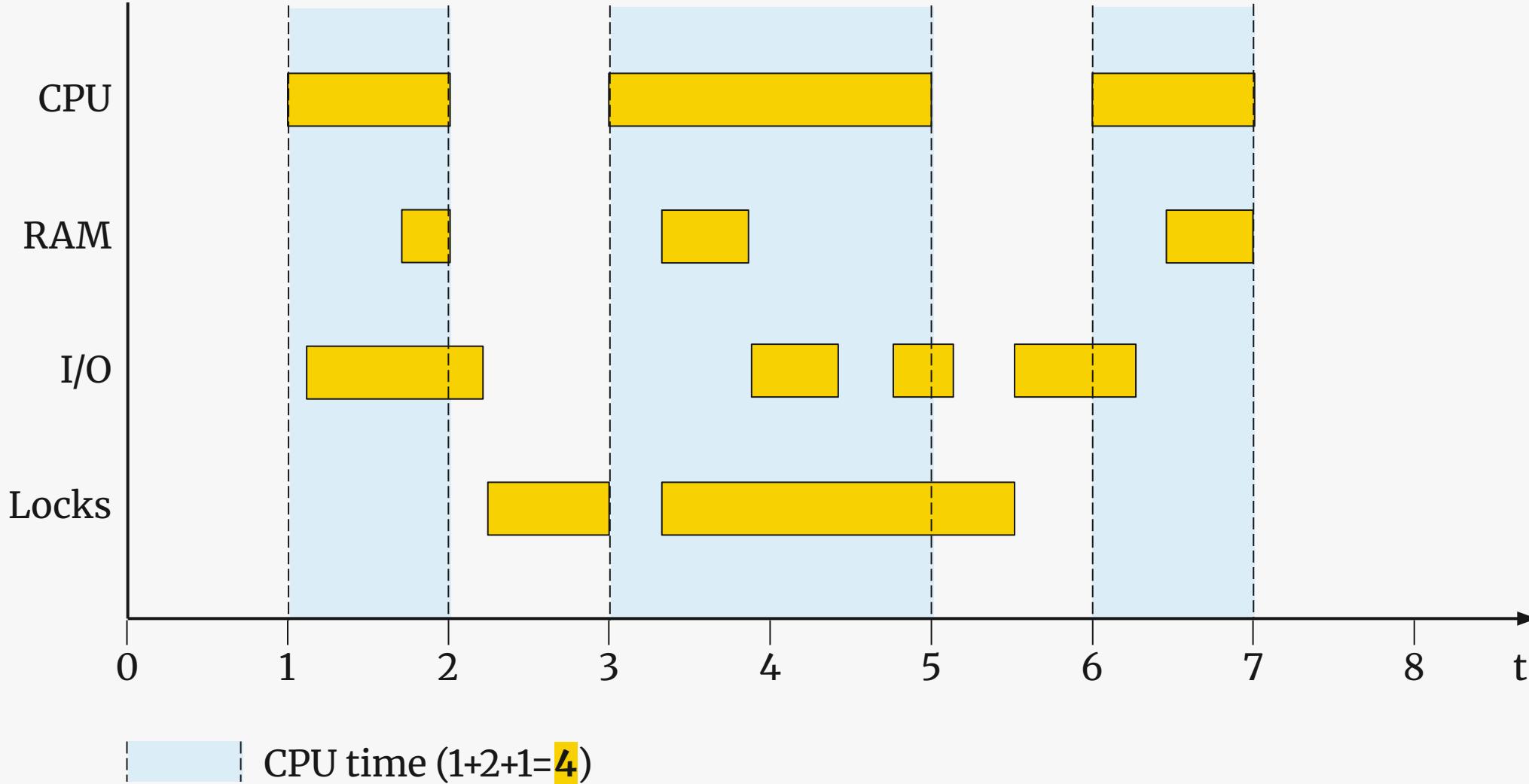
CPU TIME **VS** WALL CLOCK TIME

- CPU time – время, проведенное потоком **на процессоре**
- Wall clock time – **общее** время потока
 - Включая ожидание (в т. ч. I/O)
 - Включая блокировки
 - Включая сон

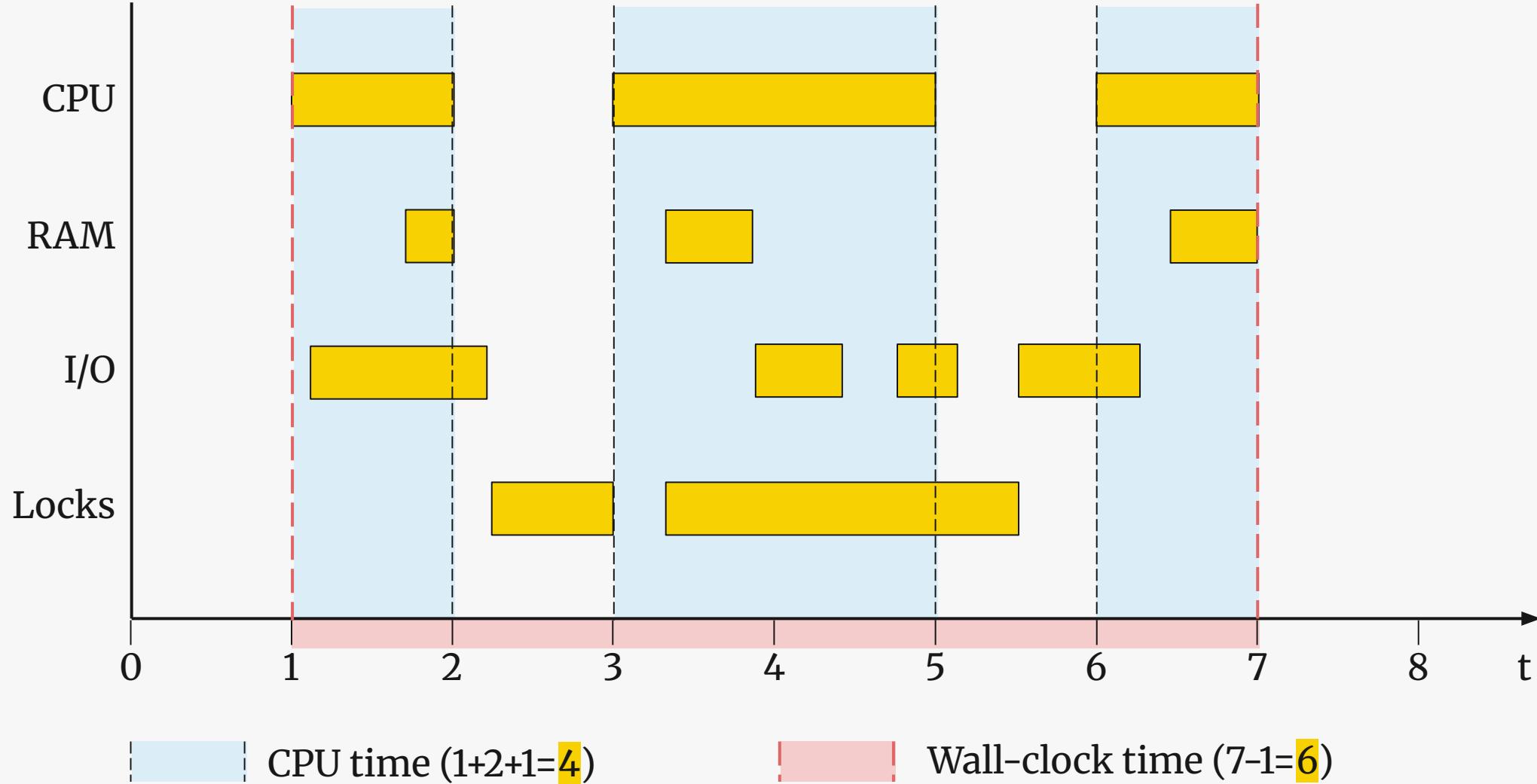
ПРОФИЛЬ ОДНОГО ВЫЗОВА МЕТОДА foo()



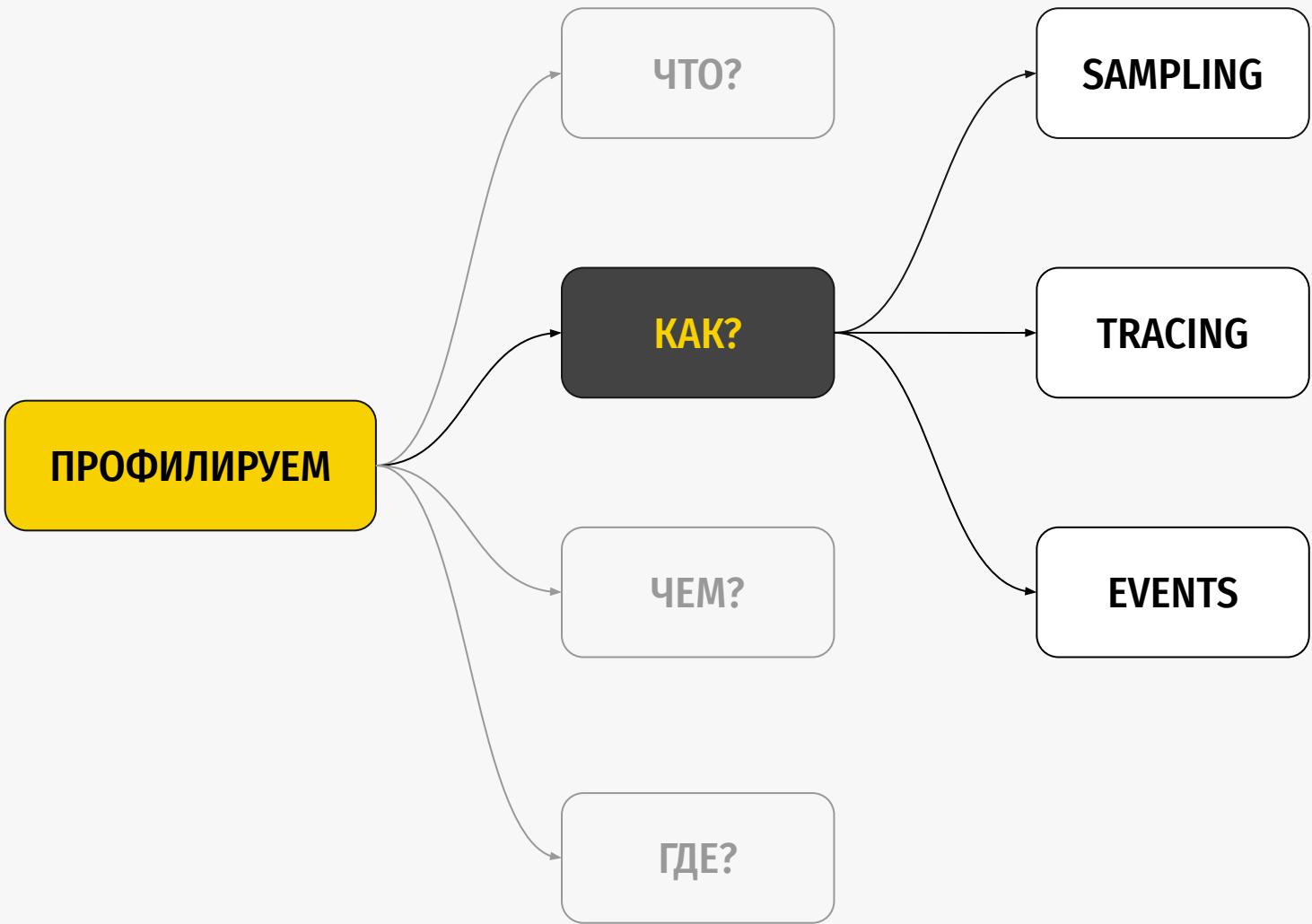
CPU TIME



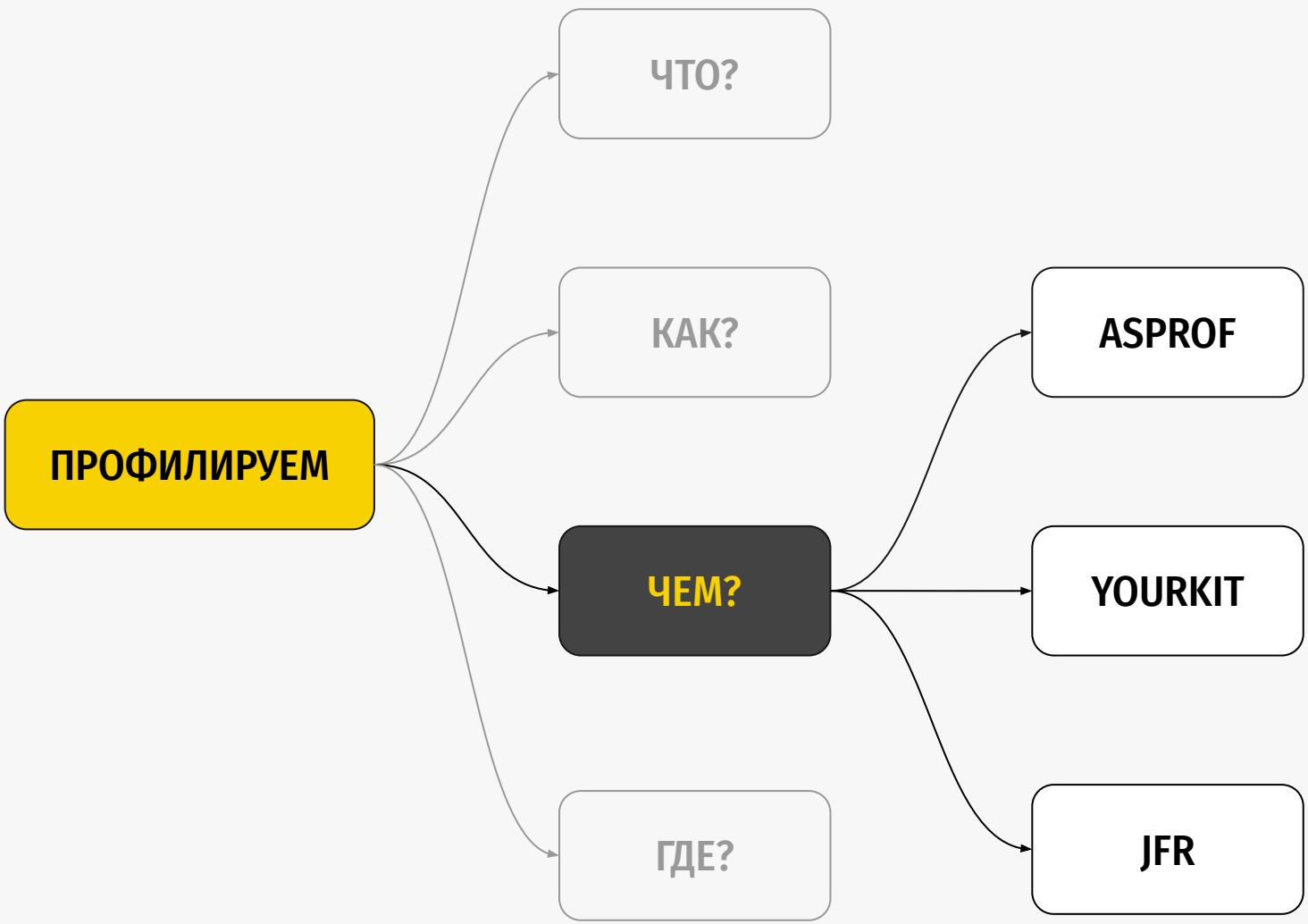
CPU TIME VS WALL CLOCK TIME

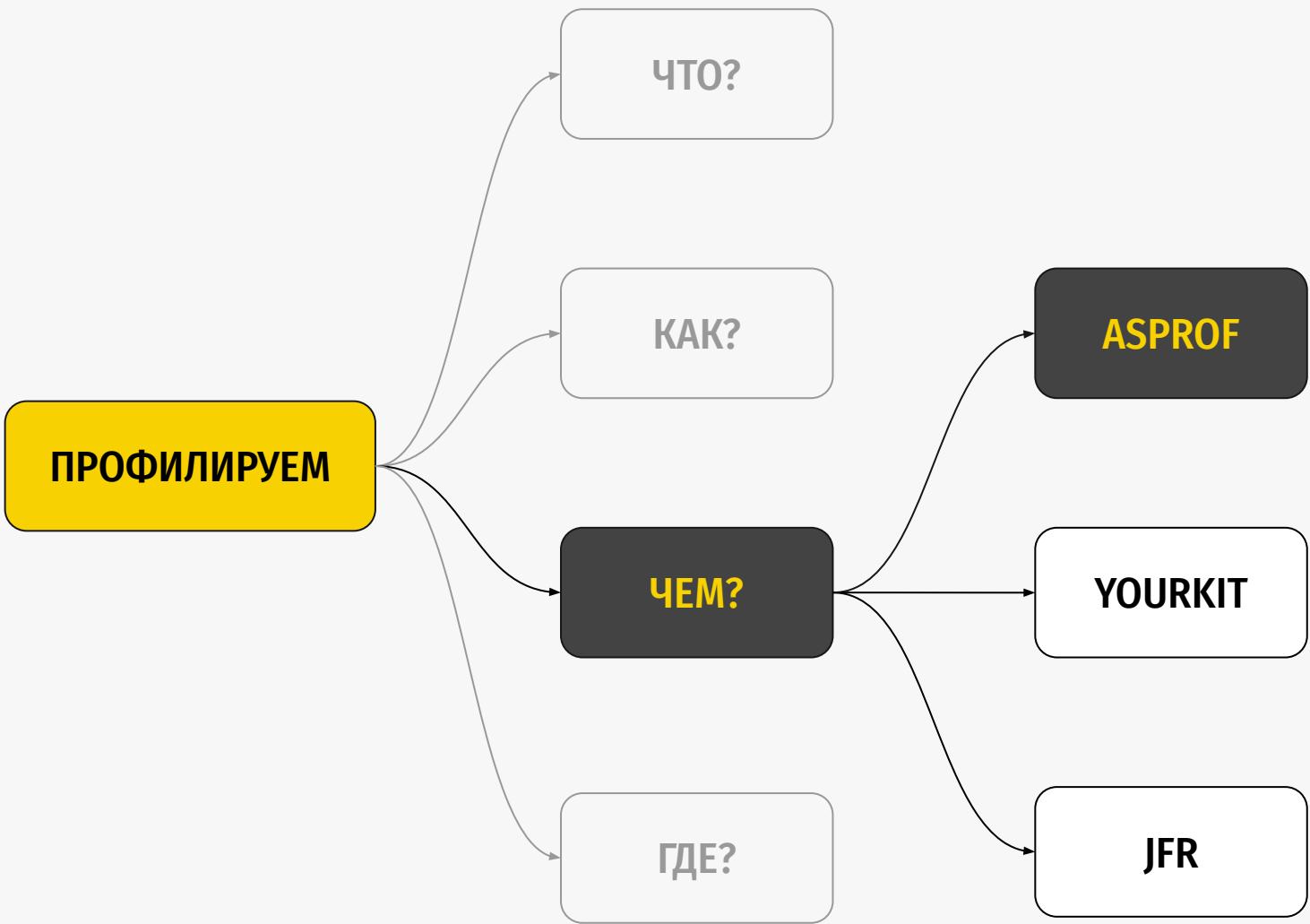












ASYNC-PROFILER: ОБЩЕЕ

- Доступность: open source
- Разработчик: Андрей Паньгин и контрибьюторы
- Тип: sampling (в основном)
- Интерфейсы:
 - CLI
 - Java API
 - GUI (IDEA Ultimate)

**AsyncGetCallTrace (AGCT) is a non-standard
extension of HotSpot JVM ...**

**async-profiler ... got its name
after this function.**

[async-profiler GitHub](#)

ASYNC-PROFILER: ПОДКЛЮЧЕНИЕ (CLI)

- При старте JVM:
 - `java -agentpath:/path/to/asprof.so=start,event=cpu ...`
- На лету:
 - `asprof -d 30 -e cpu -f profile.html <pid>`
⚠ Для этого режима JVM лучше запускать с опциями:
`-XX:+UnlockDiagnosticVMOptions -XX:+DebugNonSafepoints`
- В составе JMH:
 - Воркшоп «JMH: вводный курс по микробенчмаркам»

ASYNC-PROFILER: ОСОБЕННОСТИ РАБОТЫ

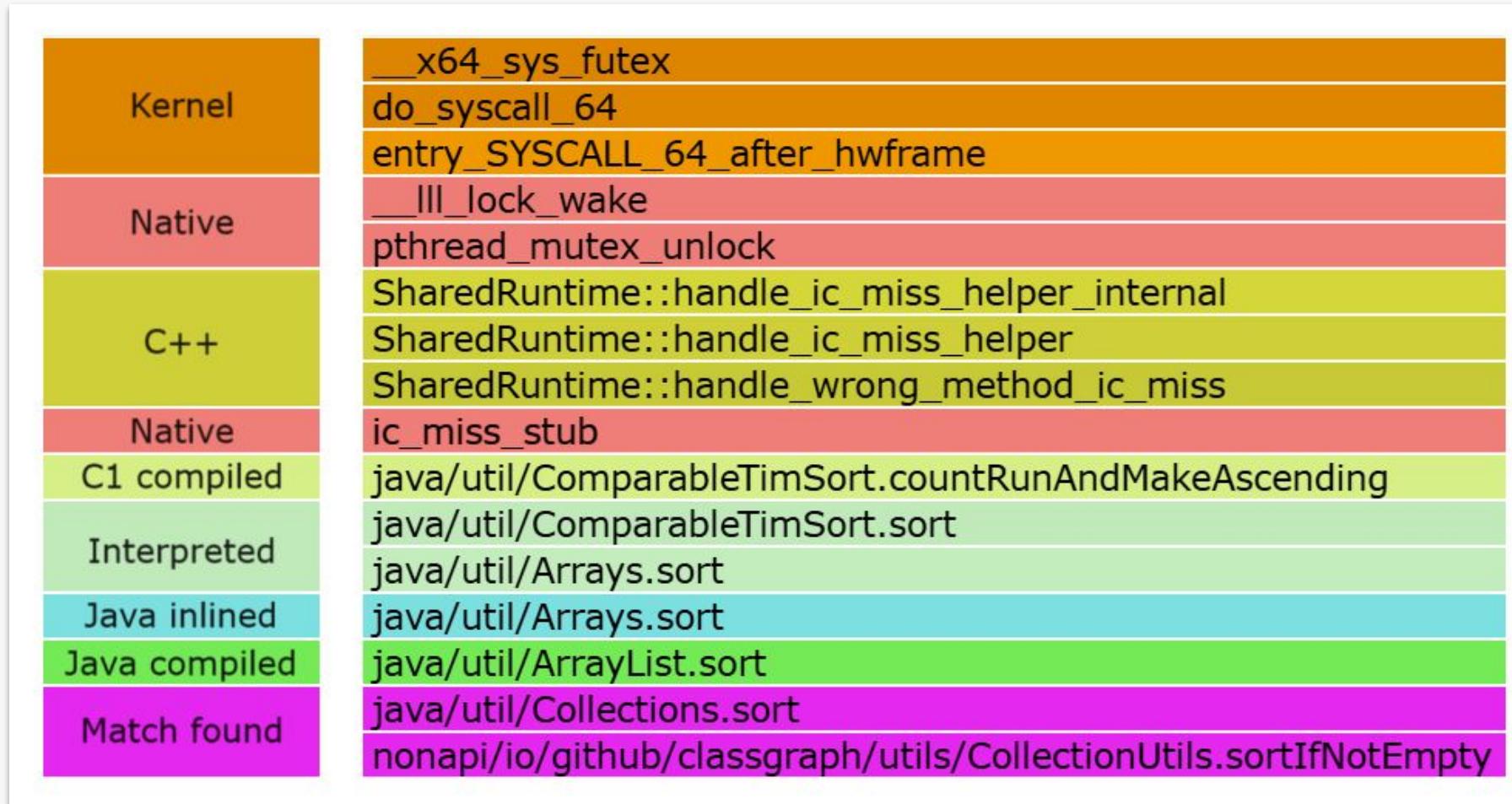
- Async-profiler собирает callstack'и вне safepoint'ов
- Ему нужно сопоставлять их с исходным кодом
- Он берет данные у JVM
- Если она запущена без `+DebugNonSafepoints`, их может не быть
- Результат: **профайлер теряет в точности**
 - Например, не видит простые inlined-методы
- Подробнее: [Why JVM modern profilers are still safepoint biased?](#)



ASYNC-PROFILER: CPU TIME

- Включается опцией `-e cpu`
- Почти не подвержен safepoint bias
- Частота сэмплирования: `-i 10ms`
- Включает нативные фреймы в стектрейсы

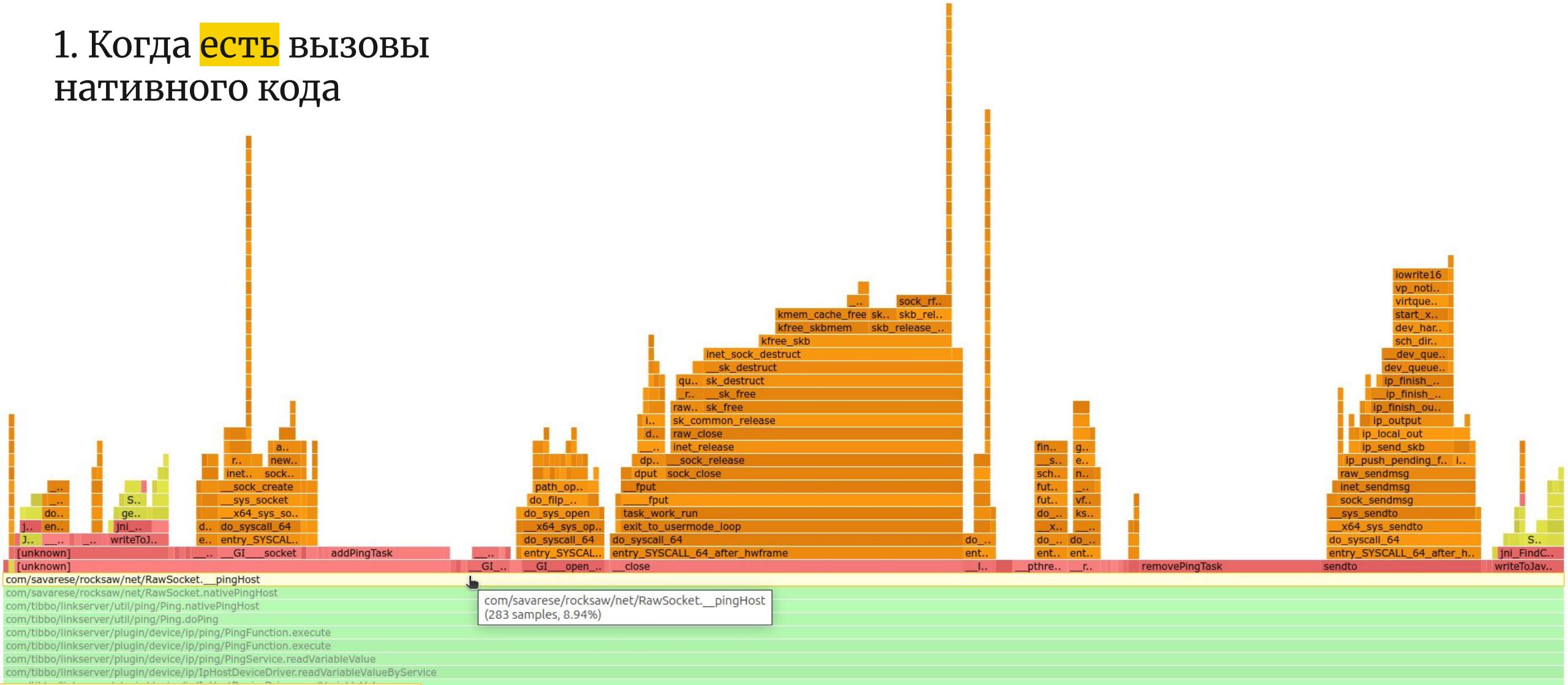
ПАЛИТРА FLAME GRAPH В ASPROF



<https://github.com/async-profiler/async-profiler/blob/master/docs/FlamegraphInterpretation.md>

КОГДА НУЖНЫ НАТИВНЫЕ ФРЕЙМЫ

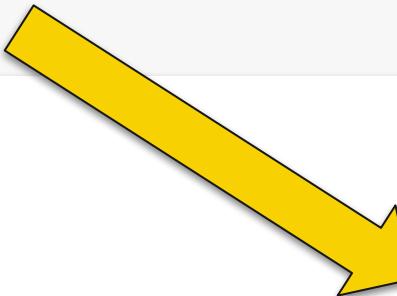
1. Когда есть вызовы нативного кода



КОГДА НУЖНЫ НАТИВНЫЕ ФРЕЙМЫ

2. Когда **нет** вызовов
нативного кода

```
java/util/regex/Pattern$CharPredicate.lambda$negate$3
java/util/regex/Pattern$CharPredicate$$Lambda$39.0x00000008001129a0.is
java/util/regex/Pattern$CharProperty.match
java/util/regex/Pattern$StartS.match
java/util/regex/Matcher.search
java/util/regex/Matcher.find
java/util/regex/Pattern.split
java/lang/String.split
java/lang/String.split
com/tibbo/aggregate/common/datatable/field/DateFormat.dateFromString
com/tibbo/aggregate/common/datatable/field/DateFormat.valueFromString
com/tibbo/aggregate/common/datatable/field/DateFormat.valueFromString
com/tibbo/aggregate/common/datatable/FieldFormat.valueFromEncodedString
com/tibbo/aggregate/common/datatable/DataRecord.setData
com/tibbo/aggregate/common/datatable/DataRecord.<init>
```



```
__JL_LOCK_WAIT
Mutex::lock
G1CollectedHeap::attempt_allocation_slow
G1CollectedHeap::allocate_new_tlab
M G1CollectedHeap::allocate_new_tlab
M G1CollectedHeap::allocate_new_tlab
(5 samples, 0.01%)
TypeArrayKlass::allocate_common
OptoRuntime::new_array_C
java/util/regex/Matcher.<init>
java/util/regex/Pattern.matcher
```

ASYNC-PROFILER: ALLOCATION

- Включается опцией `-e alloc`
- Захватывает аллокации памяти в куче
- Не все, а выше порога: `--alloc 500k`
- Можно использовать с флагом `--live`
 - сохранит только те объекты, которые не были удалены к концу сеанса профилирования
(для обнаружения утечек)

ПРИМЕР КРУПНОЙ АЛЛОКАЦИИ

byte[]

```
java.io.ByteArrayOutputStream.<init>
org.springframework.web.util/ContentCachingRequestWrapper.<init>
org.springframework.web.filter/AbstractRequestLoggingFilter.doFilterInternal
org.springframework.web.filter/OncePerRequestFilter.doFilter
org.apache.catalina.core/ApplicationFilterChain.internalDoFilter
org.apache.catalina.core/ApplicationFilterChain.doFilter
org.springframework.web.filter/RequestContextFilter.doFilterInternal
org.springframework.web.filter/OncePerRequestFilter.doFilter
org.apache.catalina.core/ApplicationFilterChain.internalDoFilter
org.apache.catalina.core/ApplicationFilterChain.doFilter
org.springframework.web.filter/FormContentFilter.doFilterInternal
org.springframework.web.filter/OncePerRequestFilter.doFilter
org.apache.catalina.core/ApplicationFilterChain.internalDoFilter
org.apache.catalina.core/ApplicationFilterChain.doFilter
org.springframework.web.filter/ServerHttpObservationFilter.doFilterInternal
org.springframework.web.filter/OncePerRequestFilter.doFilter
```

byte[]

(5,242,896,000 samples, 99.99%)

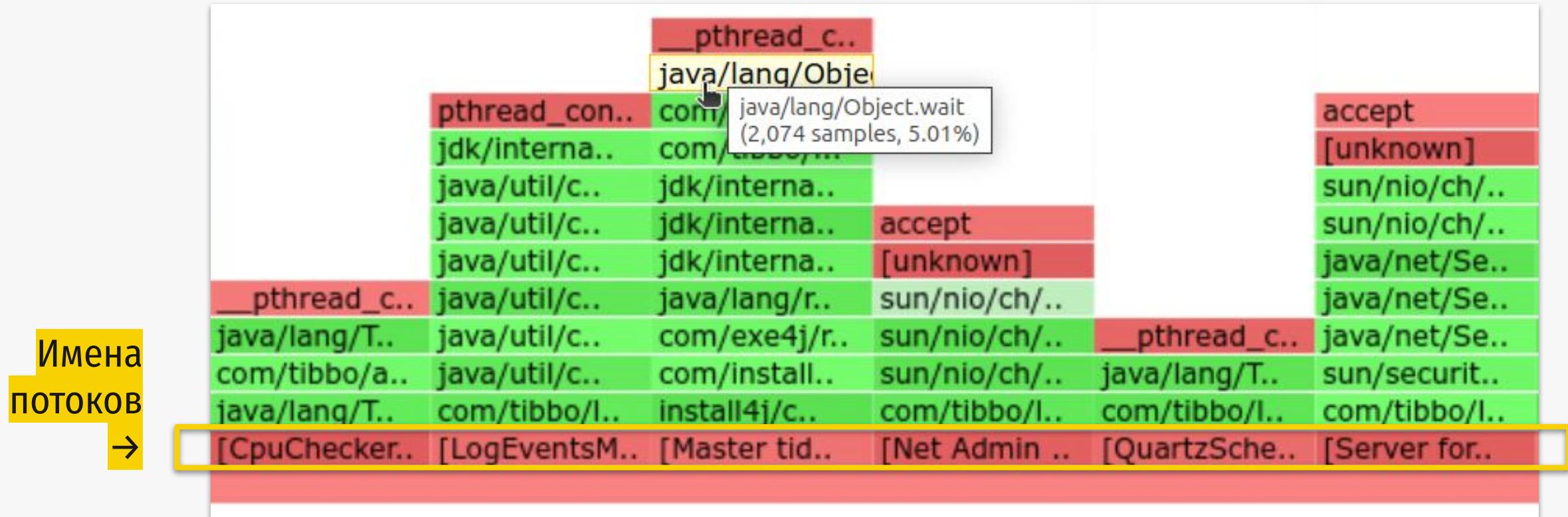
**КАКОЙ РЕЖИМ лучше,
если непонятно,
куда смотреть?**



ASYNC-PROFILER: WALL-CLOCK

- Включается опцией `-e wall`
- Захватывает потоки в любом состоянии
- Подходит для выяснения общих затрат времени
- Частота сэмплирования: `-i 10ms`
- Лучше использовать с флагом `-t` (threads)

ПРИМЕР РАЗДЕЛЕНИЯ НА ПОТОКИ

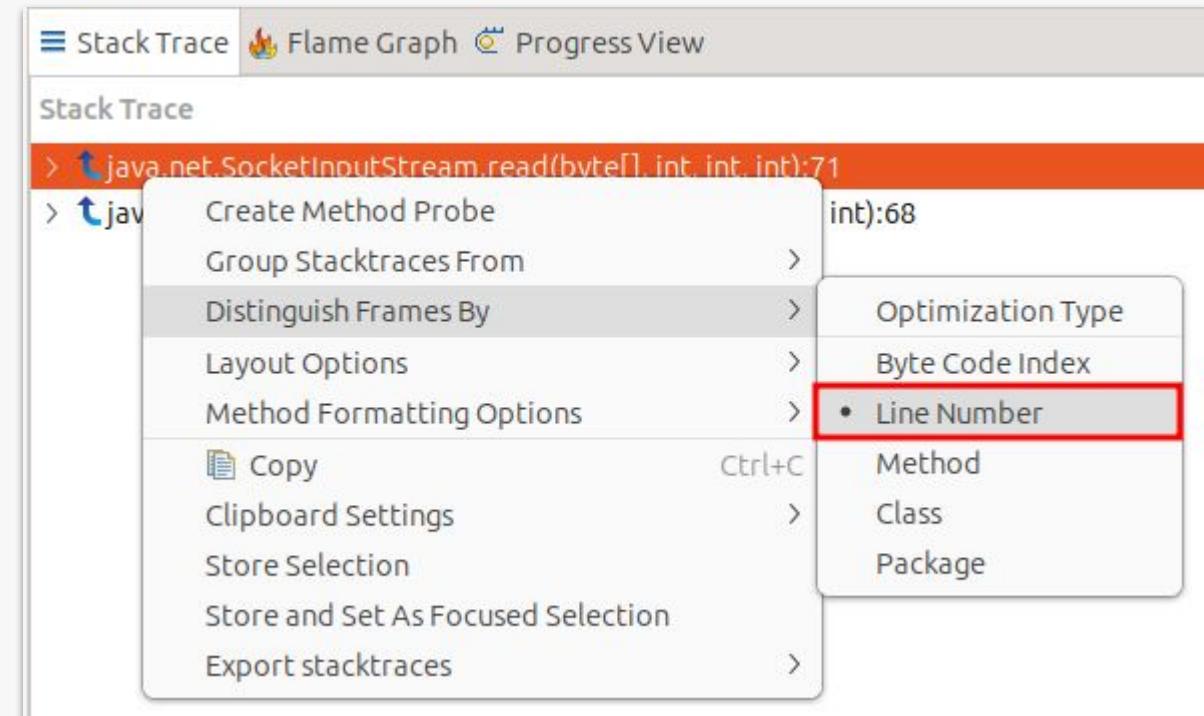


ASYNC-PROFILER: ДРУГИЕ РЕЖИМЫ

- Аллокации нативной памяти (`-e nativemem`)
- Блокировки (`-e lock`)
- Отдельные Java-методы (`-e className.methodName`)
- И не Java тоже:
 - `G1CollectedHeap::humongous_obj_allocate`
 - `Java_java_lang_ClassLoader_defineClass1`
 - `Java_java_lang_Throwable_fillInStackTrace`
 - `JVM_StartThread`
- Прочие

ASYNC-PROFILER: ГДЕ НОМЕРА СТРОК?

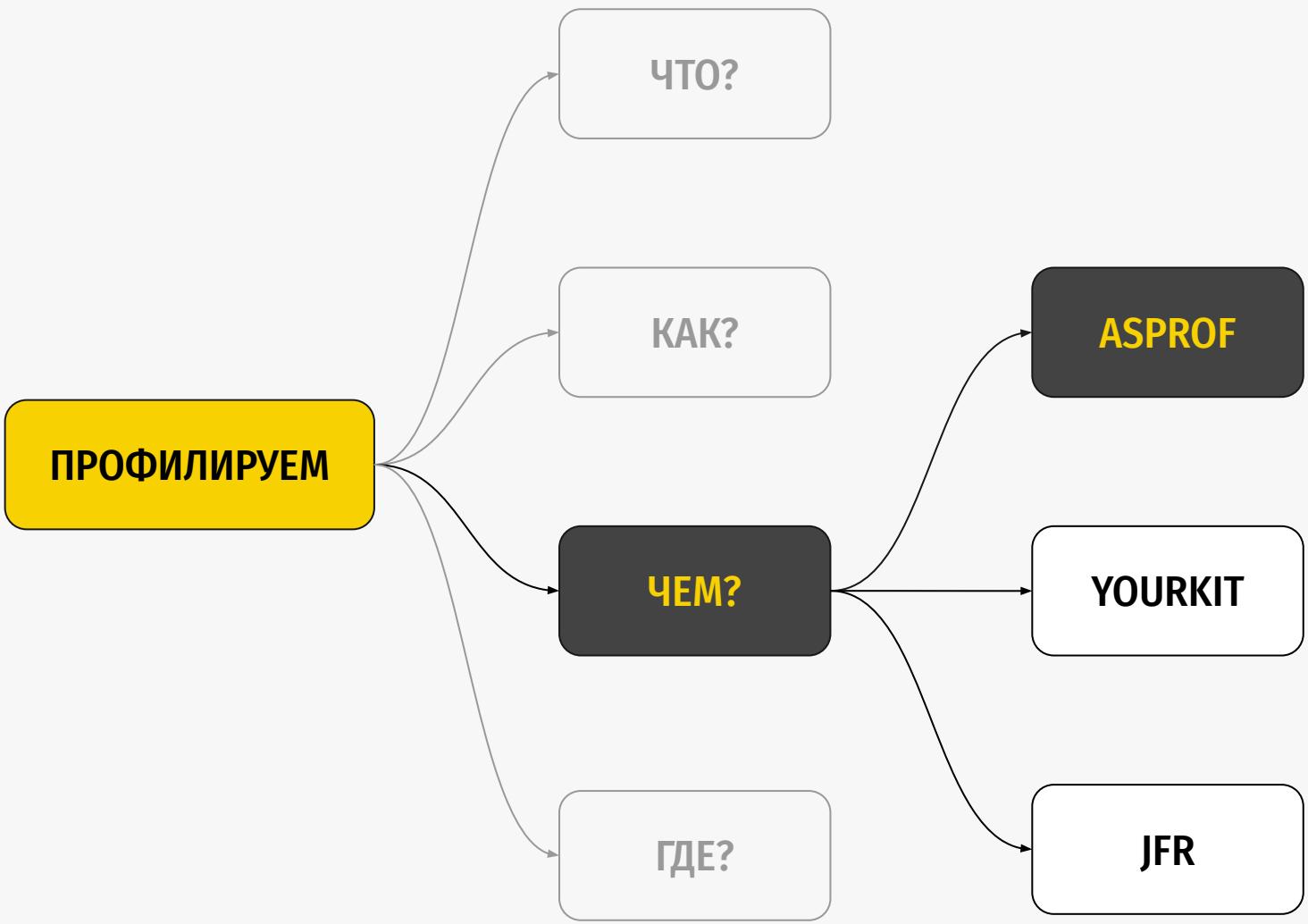
Формат экспорта	Номера строк
Plain text	✗
Деревья вызовов (HTML)	✗
Flame Graphs (SVG)	✗
JFR-записи (бинарный)	✓

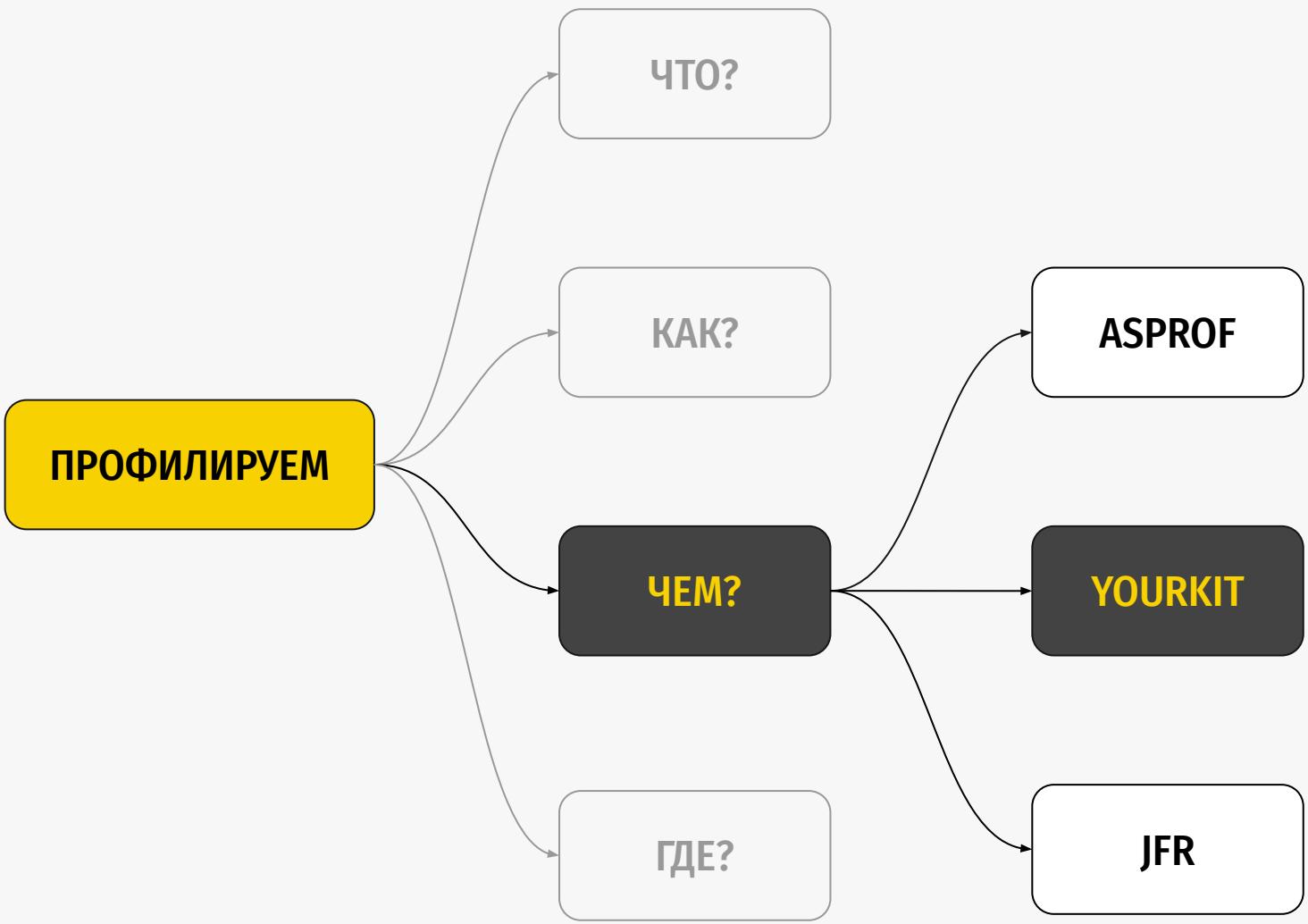




ASYNC-PROFILER: ЧЕГО НЕ ХВАТАЕТ

- Как получить **количества** вызовов методов?
- Как профилировать **ввод-вывод**?
- Как подключаться **удаленно**?
- Как профилировать действия **верхнего уровня**?
 - Например HTTP и SQL запросы







YOURKIT: ОБЩЕЕ

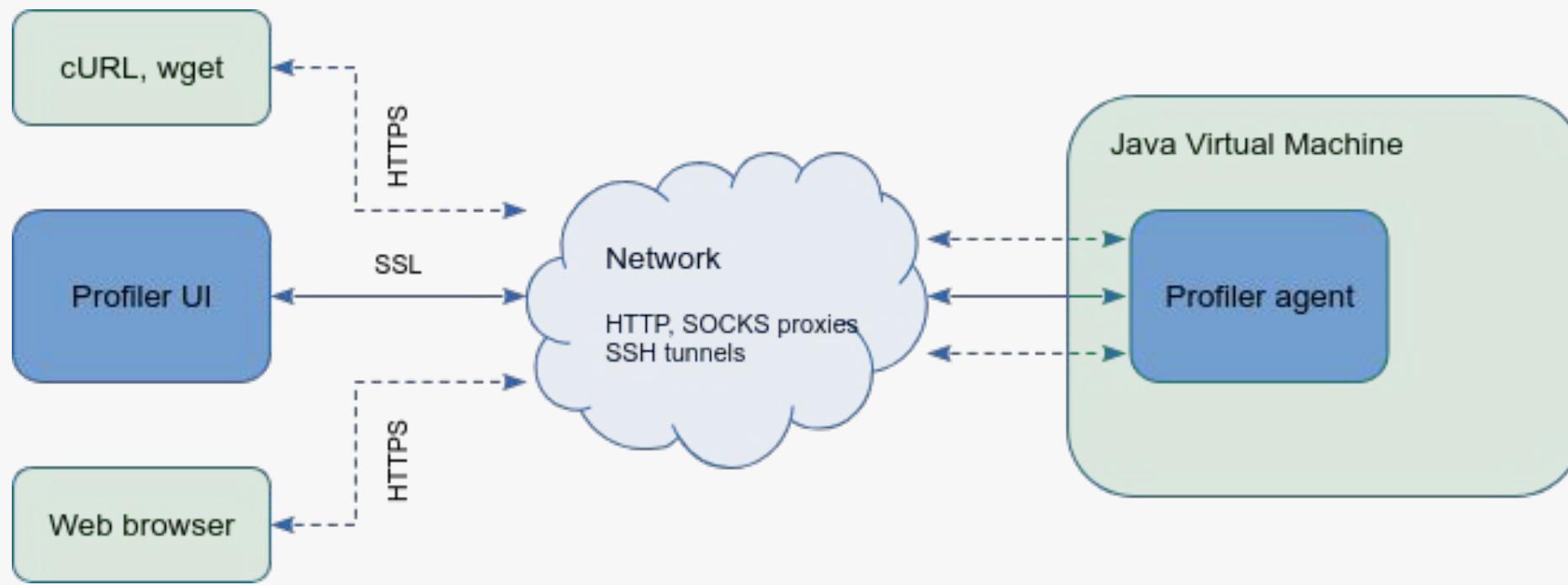
- Доступность: платные лицензии
- Разработчик: YourKit GmbH
- Тип: sampling & tracing
- Интерфейсы:
 - GUI
 - CLI
 - HTTP API
 - Java API

**The [open source] license is granted
to developers of non-commercial
Open Source projects, with an established
and active community.**

yourkit.com



YOURKIT: АРХИТЕКТУРА



<https://www.yourkit.com/docs/java-profiler/latest/help/architecture.jsp>



YOURKIT: CPU TRACING

- Захватывает моменты входа и выхода в/из методов
- Сохраняет точные **длительности** и **количества** вызовов
- Может применяться с **CPU time** и **wall-clock time**
- По умолчанию – **адаптивный**

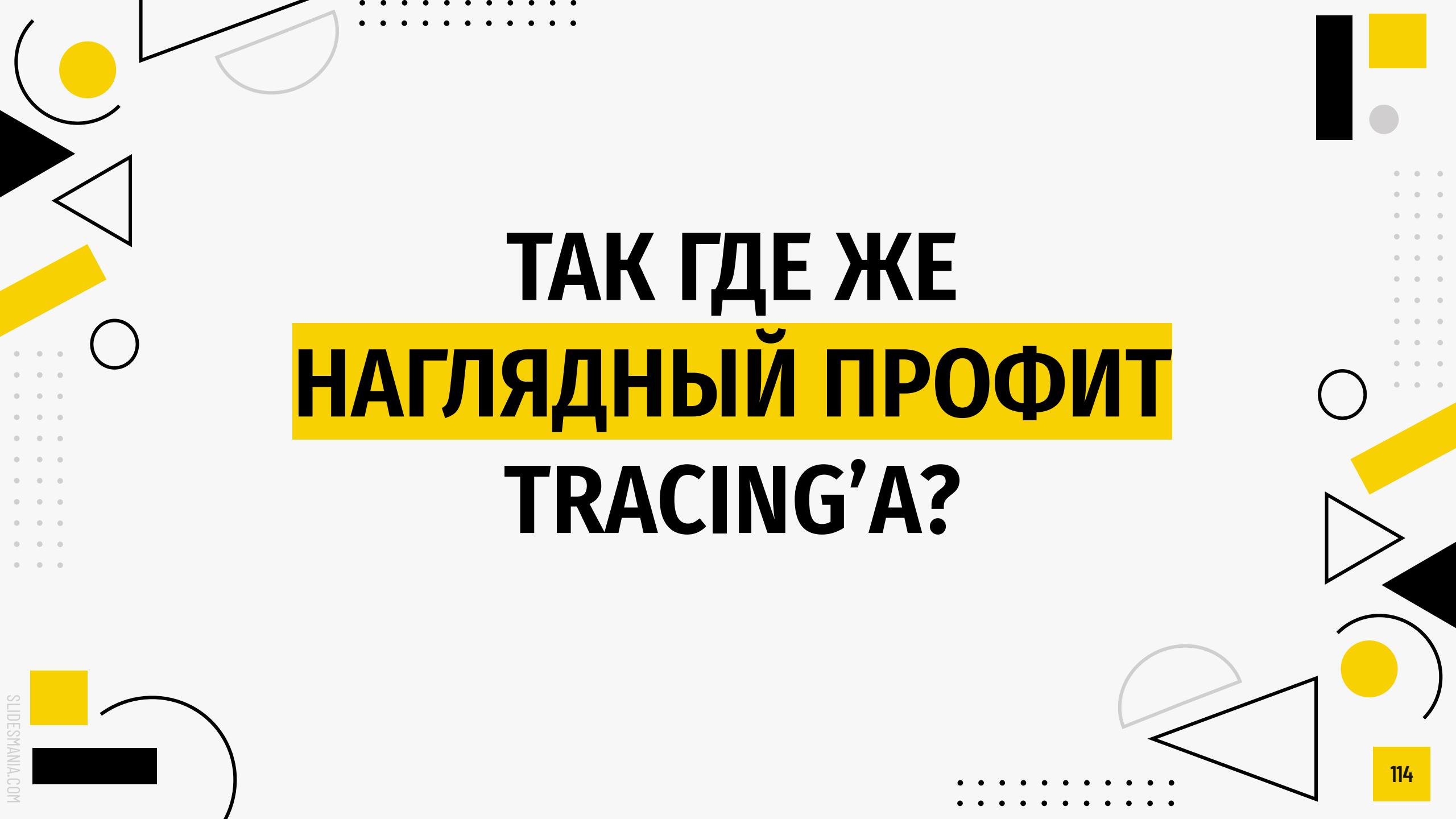
YOURKIT: CPU ADAPTIVE TRACING



- Исключает короткие, но часто вызываемые методы
- Основывается на статистике по ходу профилирования
- Отражается в результатах:

DemoApp.java:87	↓	DemoApp.drawDemo(Graphics2D)	30,041	6 %	493
DemoApp.java:72	↓	sun.java2d.SunGraphics2D.clip(Shape)	28,336	6 %	493
SunGraphics2D.java:2059	↓	sun.java2d.SunGraphics2D.intersectShapes(Shape, Shape, boolean,	28,221	6 %	493
SunGraphics2D.java:463	↓	sun.java2d.SunGraphics2D.intersectRectShape(Rectangle2D, Sha	28,221	6 %	493
SunGraphics2D.java:509	↓	sun.java2d.SunGraphics2D.intersectByArea(Shape, Shape, bc	28,218	6 %	350
SunGraphics2D.java:542	↓	java.awt.geom.Area.<init>(Shape)	26,271	5 %	350
Area.java:126	↓	java.awt.geom.Area.pathToCurves(PathIterator)	26,270	5 %	350
Area.java:195	↓	<...> sun.awt.geom.AreaOp.calculate(Vector, Vector)	26,209	5 %	350
Area.java:169	↓	<...> - sun.awt.geom.Curve.insertQuad(Vector, double, double, c	≥ 26	0 %	≥ 35,272
Area.java:176	↓	<...> sun.awt.geom.Curve.insertCubic(Vector, double, double, do	7	0 %	5,600

https://www.yourkit.com/docs/java-profiler/latest/help/tracing_settings.jsp



**ТАК ГДЕ ЖЕ
НАГЛЯДНЫЙ ПРОФИТ
TRACING'A?**

SAMPLING

	Call Tree	▼ Time (ms)	Samples
⊖	<All threads>	40,643 100 %	147,212
⊕	com.tibbo.linkserver.Server.main(String[])	30,979 76 %	2,843
⊕	com.tibbo.linkserver.Server.<clinit>()	6,745 17 %	111,367
⊕	com.tibbo.linkserver.Server.<clinit>() ↴<...> sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, String)	2,752 7 %	163
	↴<...> com.intellij.rt.execution.application.AppMainV2\$1.run()	69 0 %	6
⊕	com.tibbo.aggregate.common.context.EventDispatcher.run()	33 0 %	3,012
⊕	com.sun.management.internal.GarbageCollectorExtImpl.createGCNotification(long, String, String, String)	22 0 %	1,411
⊕	com.sun.management.GcInfo.<init>(GcInfoBuilder, long, long, long, MemoryUsage[], MemoryUsage[])	17 0 %	7
⊕	com.sun.management.GcInfo.<init>(GcInfoBuilder, long, long, long, MemoryUsage[], MemoryUsage[])	9 0 %	3

TRACING

	Call Tree	▼ Time (ms)	Avg. Time (ms)	Count
⊖	<All threads>	4,736,405 100 %		
⊕	java.lang.Thread.run()	4,146,780 88 %	37,358	111
⊕	Thread.java:840 ↴<...> java.util.concurrent.ThreadPoolExecutor\$Worker.run()	3,078,869 65 %	32,753	94
⊕	Thread.java:840 ↴ org.apache.tomcat.util.threads.TaskThread\$WrappingRunna	775,699 16 %	64,641	12
⊕	Thread.java:63 ↴ org.apache.tomcat.util.threads.ThreadPoolExecutor\$V	632,256 13 %	63,225	10
⊕	ThreadPoolExecutor.java:659 ↴ org.apache.tomcat.util.threads.ThreadPc	632,256 13 %	63,225	10
⊕	ThreadPoolExecutor.java:1176 ↴<...> org.apache.tomcat.util.threads.T	623,010 13 %	1,384	450
⊕	ThreadPoolExecutor.java:1191 ↴ org.apache.tomcat.util.net.SocketProce	9,238 0 %	20	450
	ThreadPoolExecutor.java:1192 ↴<...> org.apache.tomcat.util.threads.Th	2 0 %	< 0.1	450



YOURKIT: HEAP PROFILING

≡ Memory profiling

All objects (reachable and unreachable) [Group objects by reachability scope](#)
Objects: 1,260,263 / shallow size: 71 MB / retained size: 71 MB [Strong reachable](#) [amortized](#) [Calculate exact retained sizes](#)

Class	Objects	Shallow Size	Retained Size
<All>	1,260,263 100 %	74,905,256 100 %	≈ 74,905,256 100 %
java	886,439 70 %	27,799,120 37 %	≈ 74,905,256 100 %
sun	11,725 1 %	586,032 1 %	≈ 71,481,998 95 %
com	79,804 6 %	4,986,808 7 %	≈ 69,138,496 92 %
char[]	203,000 16 %	20,315,104 27 %	≈ 20,315,104 27 %
javax	66,707 5 %	3,200,528 4 %	≈ 20,271,176 27 %
int[]	8,091 1 %	16,946,608 23 %	≈ 16,946,608 23 %
byte[]	1,116 0 %	729,536 1 %	≈ 729,536 1 %
int[][]	502 0 %	199,760 0 %	≈ 492,416 1 %
org	1,361 0 %	38,368 0 %	≈ 271,048 0 %
byte[][]	78 0 %	3,480 0 %	≈ 103,552 0 %
long[]	537 0 %	29,152 0 %	≈ 29,152 0 %
double[]	205 0 %	23,872 0 %	≈ 23,872 0 %
boolean[]	452 0 %	18,016 0 %	≈ 18,016 0 %

А МОЖНО ПОПРОЩЕ?

Joker<?>
2024

Путеводитель
по анализу памяти
JVM-приложений

QR code

Владимир
Плизга
Tibbo Systems

<https://www.youtube.com/watch?v=fPns2O-cnYQ>



YOURKIT: ДРУГИЕ РЕЖИМЫ

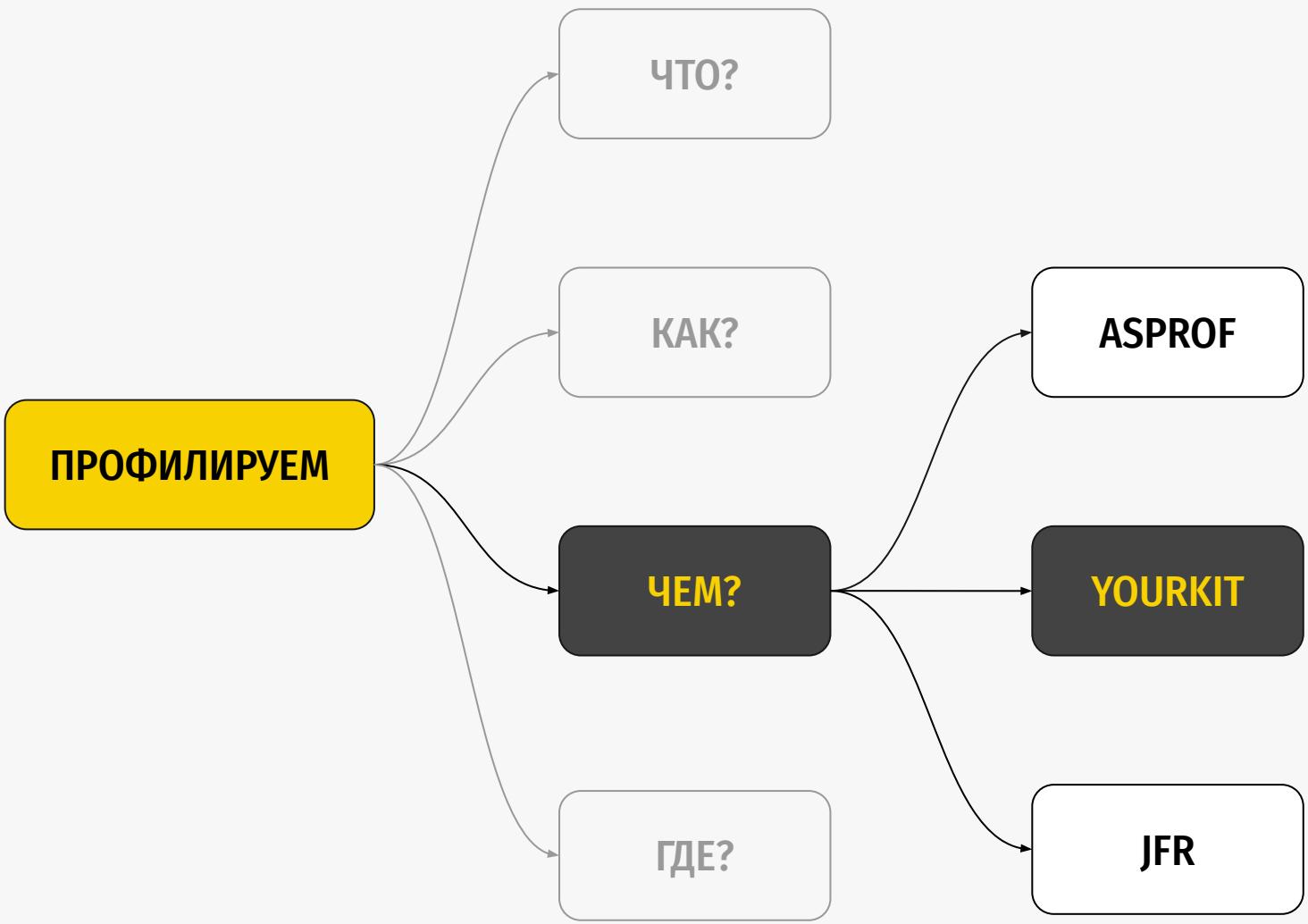
- CPU time
- Wall clock time
- Аллокации памяти
- Исключения
- Блокировки
- События

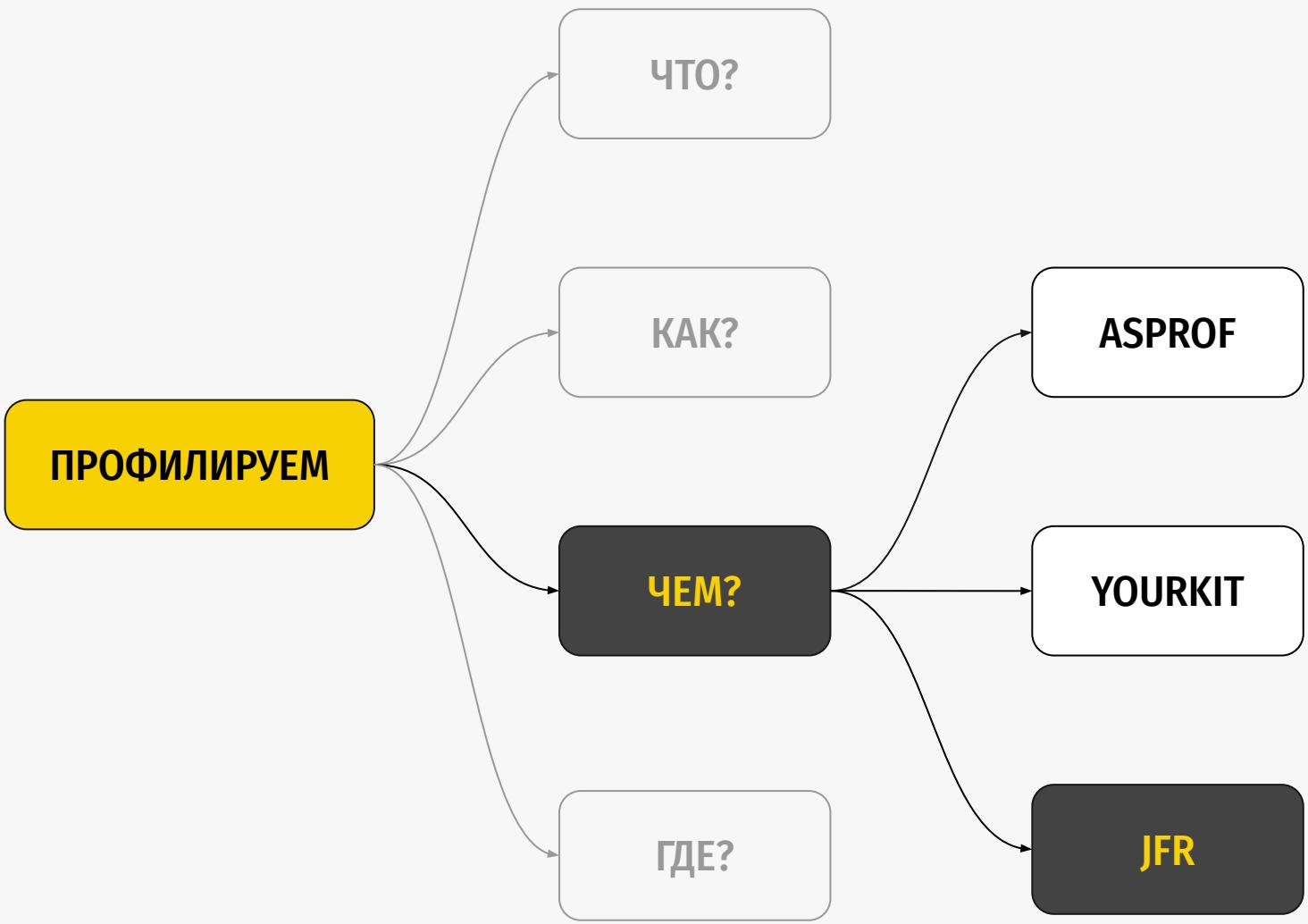
Events:

JSP/Servlet (2,303)

- ▶ SQL (4)
- ▶ Socket (163)
- ▶ File (348)
- ▶ Process (12)
- ▶ Thread (1,804)
- Thread park (0)
- Class Loading (16,228)
- Message (8)
- ▶ Directory Stream (0)
- JNDI (3)









JFR: ОБЩЕЕ

- **Распространение:** в составе HotSpot® JVM
- **Разработчик:** Sun/Oracle
- **Тип:** events + sampling
- **Интерфейсы:**
 - CLI
 - Java API
 - GUI: Java Mission Control

**The JDK Flight Recorder was designed
to minimize the Observer Effect
in the profiled system, and is meant
to be always on in production systems.**

[Wikipedia: JDK Flight Recorder](#)

JFR: CPU SAMPLING PROFILING

- У JFR свой механизм получения стектрейсов
- Особенности:
 - Не поддерживает Wall-clock time
 - Выводит только Java-код, не нативный
 - Пропускает некоторые методы, например, `System.arraycopy`
 - Сильно зависит от `-XX:+DebugNonSafepoints`
- Подробнее: [Alexey Ragozin - Lies, darn lies and sampling bias](#)





JFR: EVENTS

- На версию JDK 21 в HotSpot поддерживается 500+ событий
- Бывают с длительностью и без
- Сохраняются в бинарные файлы *.jfr
- Есть предустановленные наборы:
 - **default** – легковесный, малоинформационный
 - **profile** – увесистый, подробный (профилирующий)
- По умолчанию все **отключены**



JFR: ПОДКЛЮЧЕНИЕ

- При старте:
 - `java -XX:StartFlightRecording=filename=record.jfr ...`
- На лету:
 - `jccmd <pid> JFR.start filename=record.jfr`
 - Либо через GUI (например, Java Mission Control)
 - Либо удаленно по JMX:
 - `com.sun.management.DiagnosticCommand.jfrStart`



JFR: ВЫБОР СОБЫТИЙ В ІМС

Filter: File

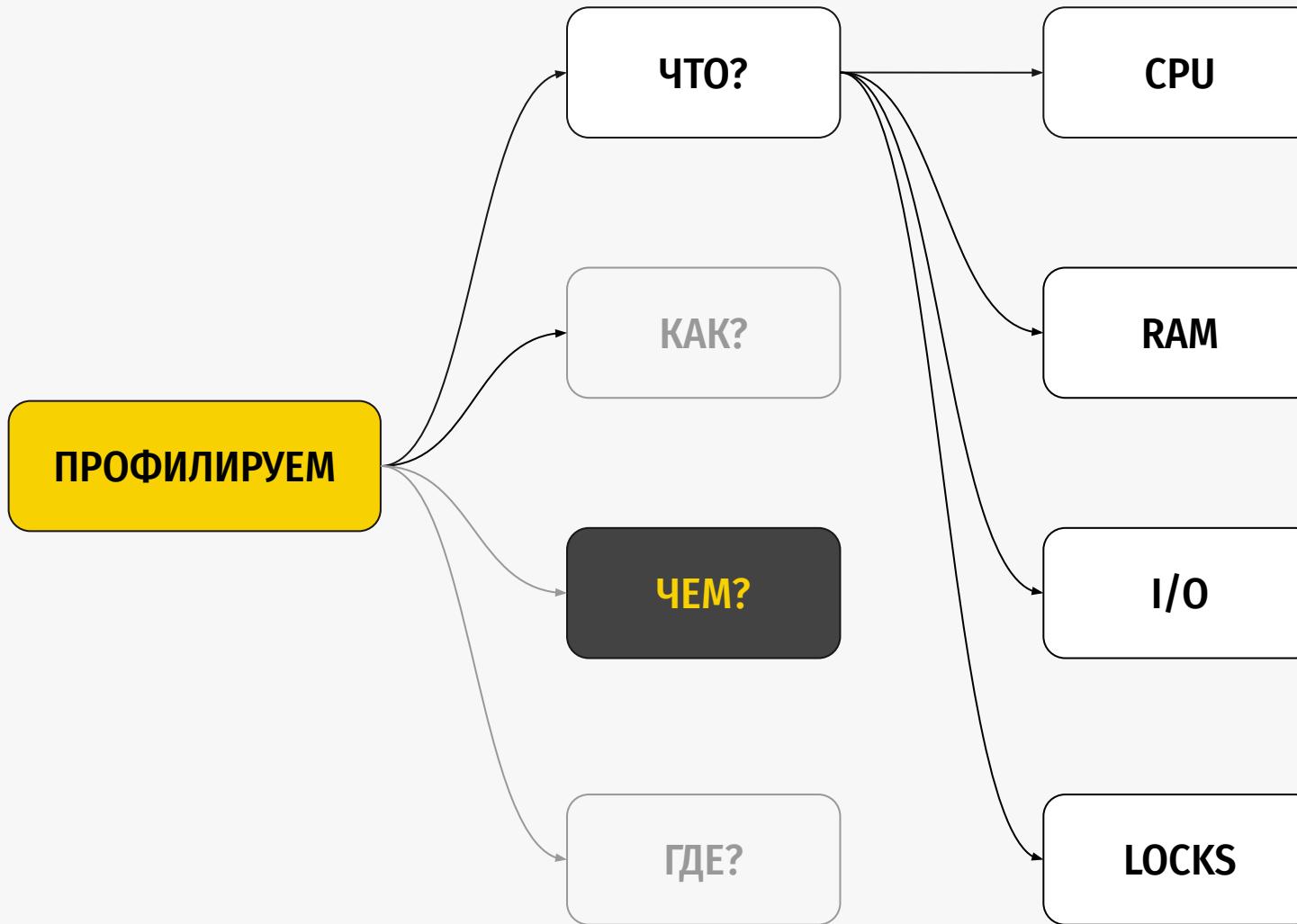
- ✓ **Java Application**
 - > **File Force**
 - > **File Read**
 - > **File Write**
- ✓ **Operating System**
 - ✓ **File System**
 - > **Container IO Usage**

Enabled

Stack Trace

Threshold **10 ms**

JFR СОБЫТИЯ ДЛЯ РЕСУРСОВ



- **jdk.ExecutionSample**
- **jdk.NativeMethodSample**

- **jdk.ObjectAllocationSample**
- **jdk.ObjectAllocationInNewTLAB**
- **jdk.ObjectAllocationOutsideTLAB**

- **jdk.SocketWrite (Read)**
- **jdk.FileWrite (Read)**

- **jdk.JavaMonitorWait (Enter)**
- **jdk.ThreadPark (Sleep)**

СЕМЬ РАЗ ОТМЕРЬ

ОПИШЬ ВСЁ ОТРЕШЬ



ДЕРЕВО РЕЗУЛЬТАТОВ В ЈМС

- Содержит записанные события
- Если чего-то нет, см. Event Browser
- Почти везде wall-clock time



Подробности:
[Alexey Ragozin – Hunting down code hotspots with JDK Flight Recorder](#)

Automated Analysis Results

- Java Application
 - Threads
 - Memory
 - Lock Instances
 - File I/O
 - Socket I/O
 - Method Profiling
 - Exceptions
 - Thread Dumps
- JVM Internals
 - Garbage Collections
 - GC Configuration
 - GC Summary
 - Compilations
 - Class Loading
 - VM Operations
 - TLAB Allocations
- Environment
- Event Browser

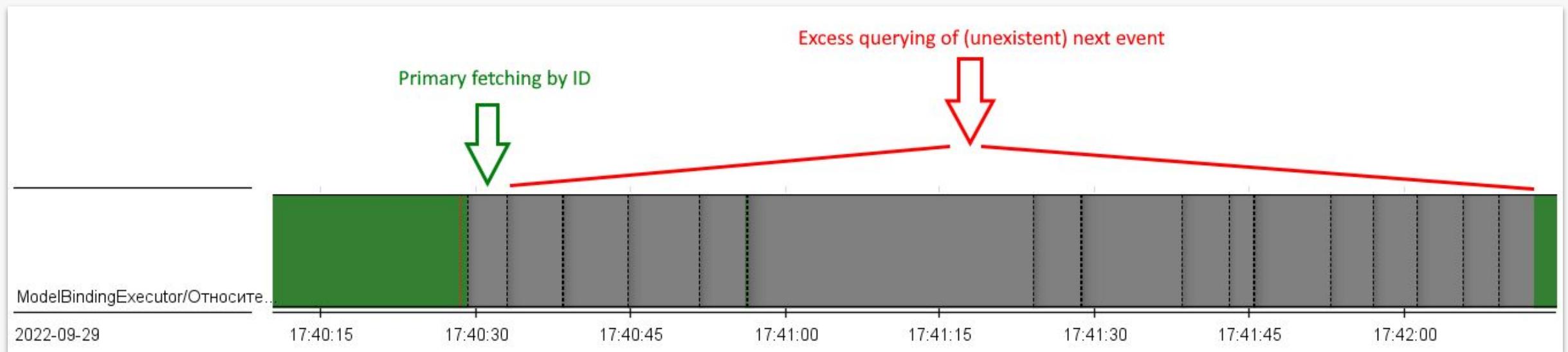
Результаты профилирования sampling'ом (CPU time)

WALL-CLOCK: ПРИМЕР ИЗ ЖИЗНИ

Projects /  Aggregate /  Add parent /  AGGREGATE

Eliminate excess load on Cassandra upon fetching events by ID

```
iterator.hasNext() ? iterator.next() : null
```



Events:

- JSP/Servlet (2,303)
 - ▶ SQL (4)
 - ▶ Socket (163)
 - ▶ File (348)
 - ▶ Process (12)
 - ▶ Thread (1,804)
- Thread park (0)
- Class Loading (16,228)
- Message (8)
- ▶ Directory Stream (0)
- JNDI (3)

JVM Browser Outline

Automated Analysis Results

- Java Application
 - Threads
 - Memory
 - Lock Instances
 - File I/O
 - Socket I/O
 - Method Profiling
 - Exceptions
 - Thread Dumps
- JVM Internals
 - Garbage Collections
 - GC Configuration
 - GC Summary
 - Compilations
 - Class Loading
 - VM Operations
 - TLAB Allocations
- Environment
- Event Browser

Java Mission Control

JVM Browser Outline

Automated Analysis Results

- Java Application
 - Threads
 - Memory
 - Lock Instances
 - File I/O
 - Socket I/O
 - Method Profiling
 - Exceptions
 - Thread Dumps
- JVM Internals
 - Garbage Collections
 - GC Configuration
 - GC Summary
 - Compilations
 - Class Loading
 - VM Operations
 - TLAB Allocations
 - Environment
 - Event Browser

Events:

- JSP/Servlet (2,303)
- SQL (4)
- Socket (163)
- File (348)
- Process (12)
- Thread (1,804)
- Thread park (0)
- Class Loading (16,228)
- Message (8)
- Directory Stream (0)
- JNDI (3)

```
graph LR; A[Java Application] --> B[Threads]; A --> C[Memory]; A --> D[Lock Instances]; A --> E[FileIO]; A --> F[SocketIO]; A --> G[MethodProfiling]; A --> H[Exceptions]; A --> I[ThreadDumps]; B --> J[JVM Internals]; C --> J; D --> J; E --> J; F --> J; G --> J; H --> J; I --> J; J --> K[GarbageCollections]; J --> L[GCConfiguration]; J --> M[GCSummary]; J --> N[Compilations]; J --> O[ClassLoading]; J --> P[VMOperations]; J --> Q[TLABAllocations]; J --> R[Environment]; J --> S[EventBrowser];
```

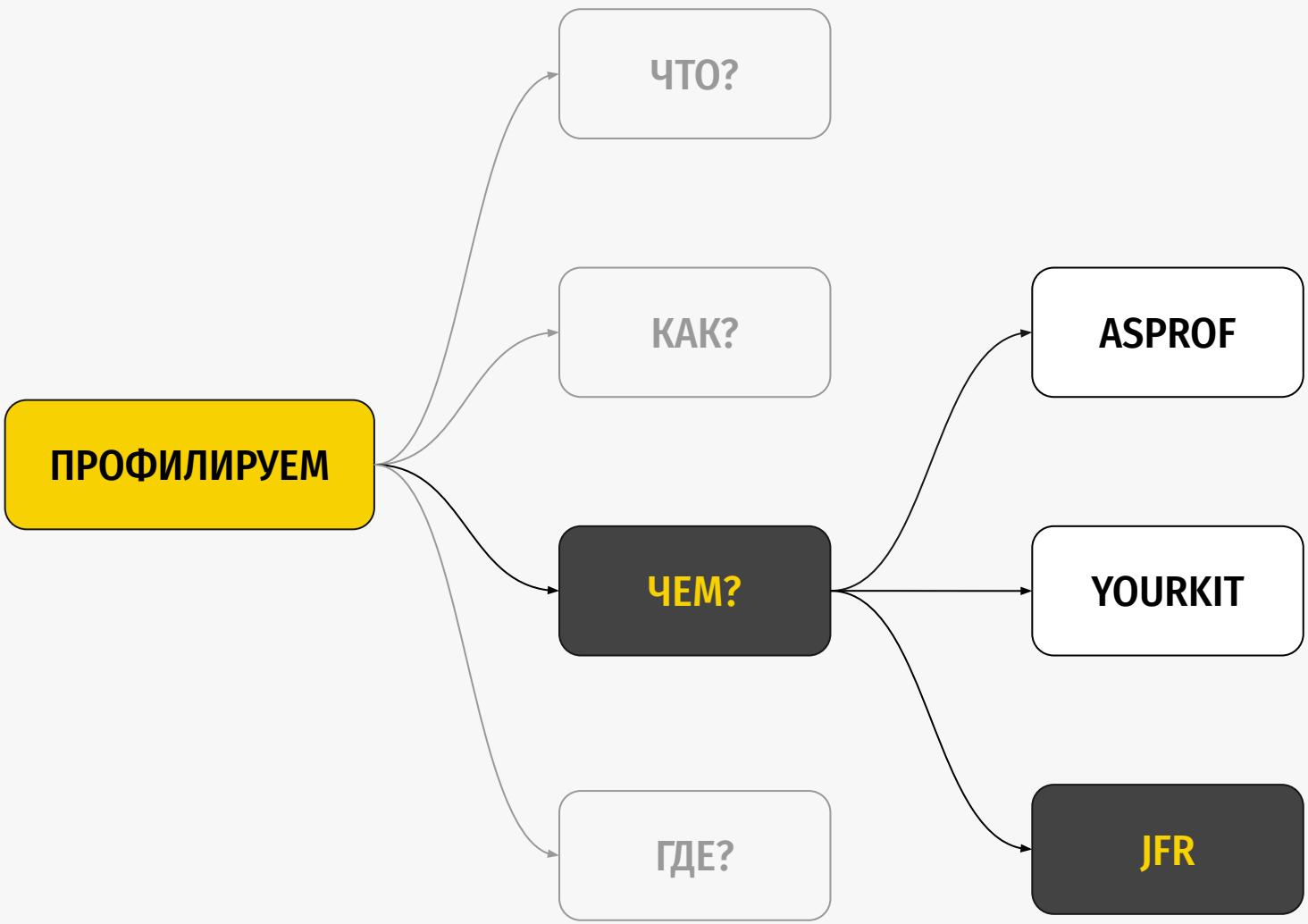
JFR: КАК ДОБАВИТЬ СВОЕ СОБЫТИЕ

1. Объявить в коде как Java класс
2. Выпускать в нужных местах
3. Искать в Event Browser'е в JMC



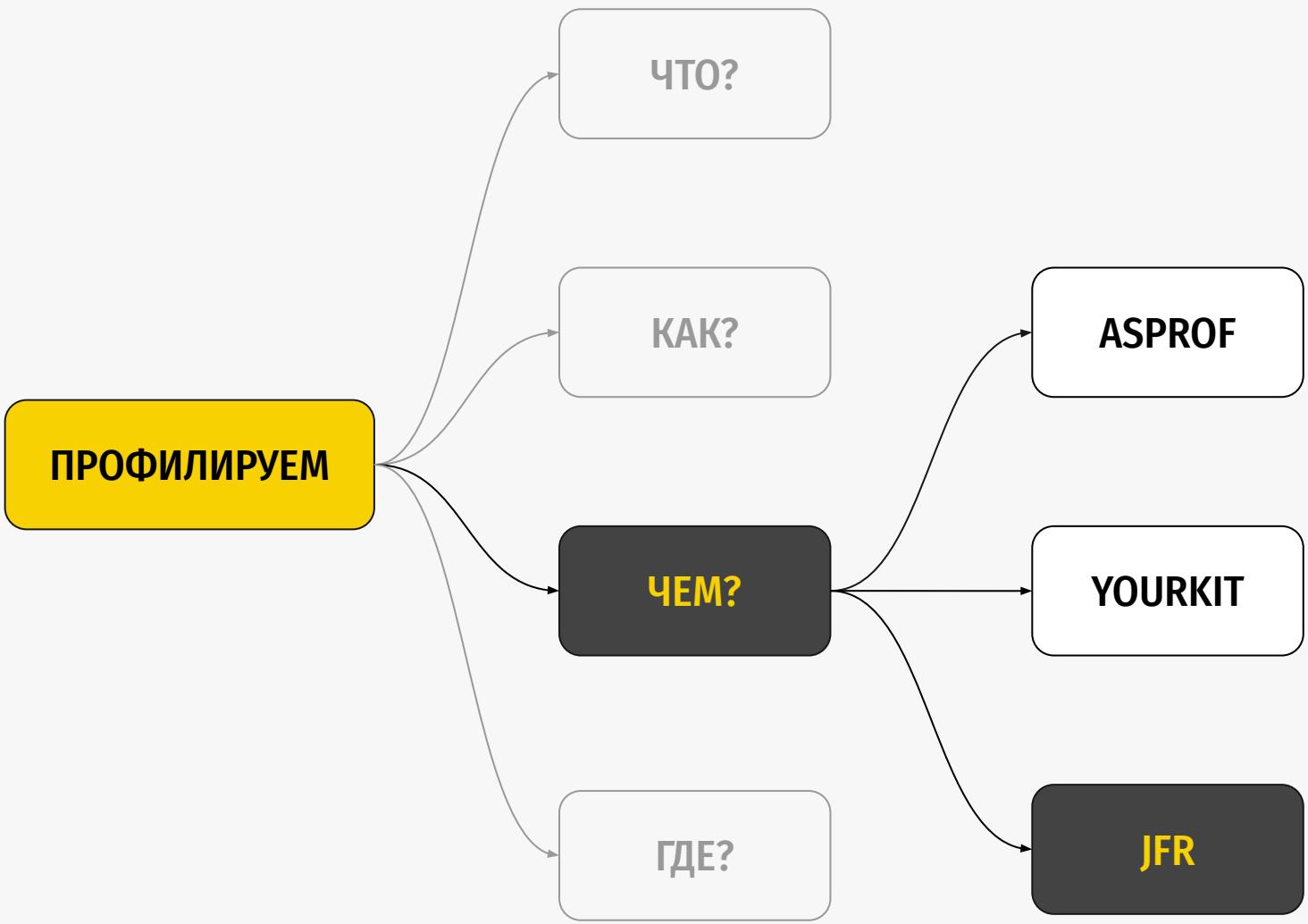
Примеры и объяснения:

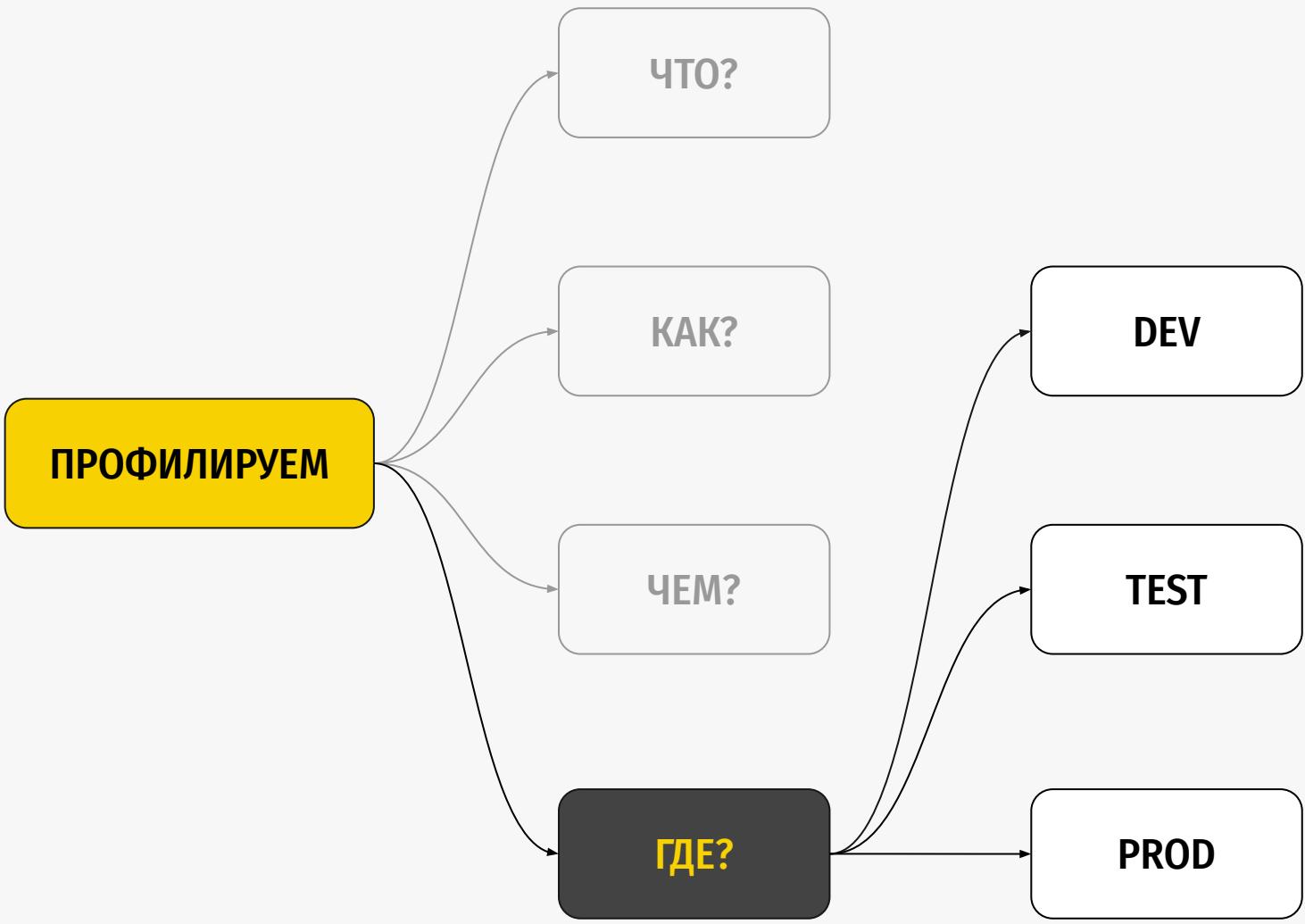
- [Monitoring REST APIs with Custom JDK Flight Recorder Events](#)
- [Программные возможности JFR](#)



СВОДКА

Фича \ Инструмент	ASPROF	YOURKIT	JFR
Sampling/Tracing/Events	S+T	S+T	E+S
Собственный GUI	✗	✓	✓
Open source	✓	✗	✓
Удаленное профилирование	✗	✓	✓
Стектрейсы с native-частью	✓	✗	✗
Поддержка high-level метрик	✗	✓	✗





- **IDEA Profiler** (asprof+JFR)
 - Или другой с интеграцией в IDE
- **YourKit**
 - Или другой с tracing'ом
- async-profiler
 - JFR
- OpenTelemetry

CONTINUOUS PROFILING @ PRODUCTION

- **async-profiler:**
 - `asprof --loop 1h -f profile-%t.jfr <pid>`
- **JFR:**
 - `jccmd <pid> JFR.start filename=long.jfr \
maxage=24h maxsize=2G settings=profile.jfc`

свежей
свеклы
сырой
сыром



atm food

ПОПРОБУЙ МАМИНЫХ
ЛЮЛЕЙ!*

*Люля кебаб от 99 рублей



А ЕСЛИ В МАСШТАБАХ...



**Непрерывное
профилирование
в облаке
с помощью eBPF**



**Андрей
Паньгин**

Одноклассники



**Леонид
Талалаев**

Одноклассники



**Артем
Дроздов**

Одноклассники



<https://www.youtube.com/watch?v=EcmR2BOeVZE>



ВЫВОДЫ

СОВЕРШЕННОГО ПРОФАЙЛЕРА ПОКА НЕТ

И не нужен...

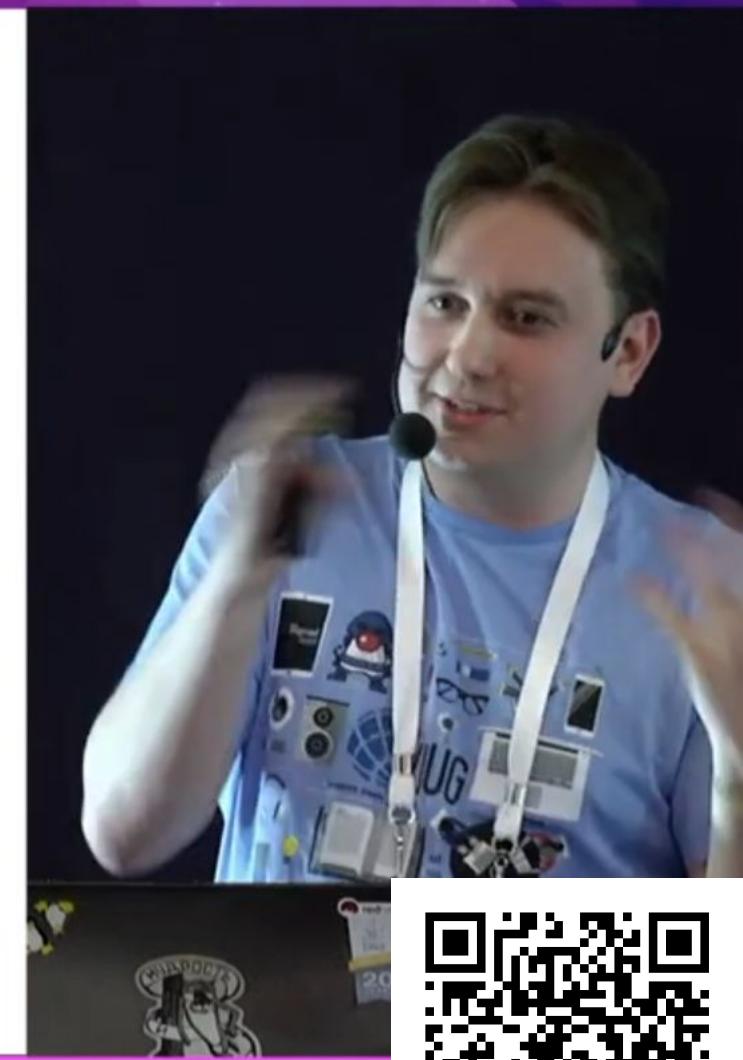
Зелёная зона: подитог

Профилирование – необходимая часть ежедневной разработки

Наблюдения:

- >95% проблем находится на первых же заходах
- >90% проблем тривиально разрешимы
- Чёткие инструкции по запуску профилировки **сильно** помогают: отлично, если есть однстрочник, или однокнопочник, или APM
- Возьмёте девелопера за руку, и с ним один раз попрофилируете – это **уверенно** купирует боязнь базовой перформансной работы¹

¹ «Нет-нет, не надо закрывать это окно, оно боится тебя больше, чем ты его»



НА ЗАМЕТКУ

- Освойте какой-нибудь профайлер сегодня (пока не пригорело)
 - Не знаете, какой выбрать – берите async-profiler
- Начинайте с sampling'a и в режиме wall-clock time
 - Сомневаетесь? Повышайте частоту/длительность
- Профилируйте заранее: dev, test, stage
 - Не всё то bottleneck, что только на production

НЕУПОМЯНУТЫЕ ССЫЛКИ

Статьи

- Why (Most) Sampling Java Profilers Are F*king Terrible
- Предрелизная оценка производительности на YourKit

Доклады

- Safepoint – и пусть весь мир подождёт
- Профайлер в каждый дом

СПАСИБО!

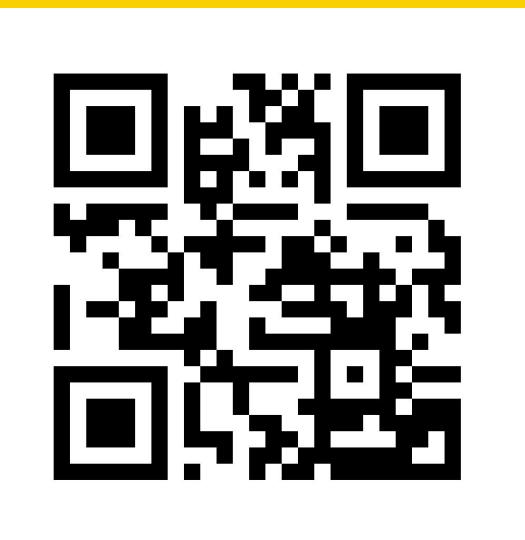
Вопросы?

😊 Владимир Плизгá

💼 Tibbo Systems

👉 [@stopshelf](https://twitter.com/stopshelf)

👉 toparvion.pro



[Слайды](#)