Профилирование java в стиле linux

Sergey Melnikov

Обо мне

- Работал в компиляторной команде Intel
- Работал в Raiffeisen bank Russia
- CTO в стартапе

О чем будем говорить

1. Постановка проблемы
2. Профилирование java с помощью perf
3. Виды профилировщиков и отличие семплирующих от трассирующих профилировщиков
4. Трасса и профиль из нее для программы на Java
5. Резюме

Постановка проблемы

1. Постановка проблемы

- Хочется профилировать очень небольшой отрезок времени, который происходит относительно редко
- Хочется профилировать его максимально точно с минимальным оверхедом
- Хочется, чтобы профилировщик был максимально простым и распространенным, мог конфигурироваться с другими тулами

Возможные варианты

- async-profiler
  - Хороший вариант, но сложно интегрировать с другими тулами
- Что-то проприетарное
  - Не понятно что и как измеряют
- perf
  - Хороший вариант. Но про Java не знает вообще ничего

Фокусируемся на perf

Используем perf для java

2. Профилировщик perf

- Стандартный профилировщик в linux
- Часть репозитория linux kernel

Perf ничего не знает про java

```
$ perf record -F 1001 -a -g -- java -jar my.jar
$ perf report
```

```
+     3.95%     1.36%  java        [JIT] tid 666328      [.] 0x0000714cb83e898b
+     3.89%     2.86%  java        [JIT] tid 666328      [.] 0x0000714cb83e894a
+     3.77%     3.44%  java        [JIT] tid 666328      [.] 0x0000714cb83e891d
+     3.63%     3.53%  java        [JIT] tid 666328      [.] 0x0000714cb83e890f
+     3.62%     3.01%  java        [JIT] tid 666328      [.] 0x0000714cb83e892d
+     3.58%     0.14%  java        [JIT] tid 666328      [.] 0x0000714cb83e8920
+     3.58%     0.05%  java        [JIT] tid 666328      [.] 0x0000714cb83e8913
+     3.58%     3.53%  java        [JIT] tid 666328      [.] 0x0000714cb83e88e5
+     3.58%     0.05%  java        [JIT] tid 666328      [.] 0x0000714cb83e88e8
```

Как perf’у рассказать про java

1. perf-map-agent
2. Опция JVM DumpPerfMapAtExit
3. Встроенный в perf jvmti агент

(1) perf-map-agent

- GitHub repo: https://github.com/jvm-profiling-tools/perf-map-agent
- Сергей Мельников — Профилирование со сверхсветовой скоростью

https://www.youtube.com/watch?v=tTgHsmxofeU

(2) Опция jvm: DumpPerfMapAtExit

- Появился в Java 16
- Пользоваться максимально просто:
  - «-XX:+UnlockDiagnosticVMOptions -XX:+DumpPerfMapAtExit»
- Не позволяет заглянуть «внутрь» java методов
- Позволяет получить профиль в терминах методов в perf'e:

```
+   90.11%    90.11%   java         [JIT] tid 666409       [.] int HW.do_smth(int)
+    5.08%     0.00%   swapper      [kernel.kallsyms]      [k] secondary_startup_64_no_verify
+    5.08%     0.01%   swapper      [kernel.kallsyms]      [k] cpu_startup_entry
+    5.05%     0.01%   swapper      [kernel.kallsyms]      [k] do_idle
```

https://bell-sw.com/announcements/2022/04/07/how-to-use-perf-to-monitor-java-performance/

(3) perf jvmti агент

- В perf есть встроенная поддержка jit-кода
- В perf есть встроенная поддержка java
- Вероятно, может отсутствовать в вашем дистрибутиве – но perf очень просто собрать.
  - Надо только обратить внимание на опциональные зависимости

Использование jvmti агента

```
$ perf record -F 1001 -k 1 -a -g -- \
 java -agentpath:path/to/libperf-jvmti.so -cp my.jar
  java: jvmti: jitdump in /home/…/.debug/jit/...

$ perf inject --jit –i perf.data -o perf.data.jitted

$ perf report –i perf.data.jitted
```

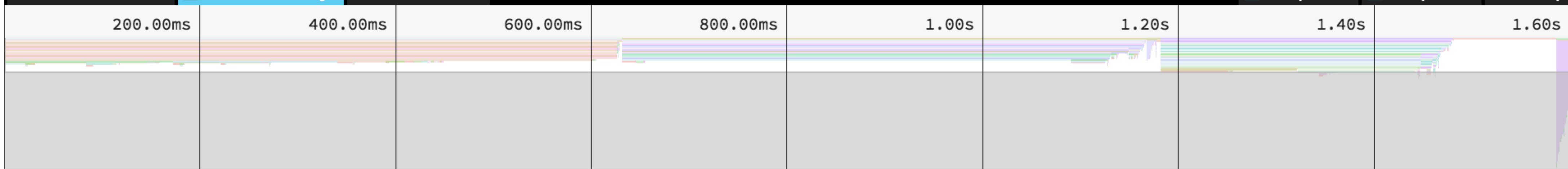
```
       │      public static int do_smth(int start_idx) {
       │      int result = start_idx;
       │      for (int i = 0; i < 50000; i += 1) {
       │ 1a:    mov    $0x1,%ebx
       │        mov    $0x85340854,%r14d
       │        mov    $0xa6810a69,%eax
       │      ↓ jmp    134
       │        nop
  2.38 │ 30:    mov    %r11d,%ebx
       │      result += div(i, 123);
  0.03 │ 33:    movslq %ebx,%r10
  3.30 │        imul   $0x214d0215,%r10,%r8
       │      return a % b;
  0.08 │        lea    (%r8,%rax,1),%r10
  1.98 │        lea    (%r8,%r14,1),%r9
  0.01 │        mov    %r8,%rdi
  2.99 │        mov    $0xe91b0e93,%r11d
  0.03 │        add    %r11,%rdi
  2.07 │        mov    %r8,%rdx
  0.03 │        mov    $0xc7ce0c7e,%r11d
  3.11 │        add    %r11,%rdx
  0.03 │        sar    $0x24,%r10
```

Удобный графический интерфейс - speedscope

https://github.com/jlfwong/speedscope

https://www.speedscope.app/

Time Order Left Heavy Sandwich importing-sucrase.json Export Import Help



200.00ms 400.00ms 600.00ms 800.00ms 1.00s 1.20s 1.40s 1.60s

fulfilled | CanvasContext.onBeforeFrame | rerender | (pr…am)
next | FlamechartMinimapView.onBeforeFrame | renderComponent
(anonymous) | renderRects | setComponentProps
importProfileGroup | renderBehind | renderComponent
__awaiter | setViewport | diff
Promise | props.canvasContext.renderBehind | idiff
(anonymous) | render | innerDiffNode
next | writeToAtlasIfNeeded | idiff
(anonymous) | renderInto | innerDiffNode
_importProfileGroup | (anonymous) | idiff
__awaiter | rowAtlas.writeToAtlasIfNeeded | buildComponentFromVNode
Promise | forEachLeafNodeWithinBounds | createComponent
(anonymous) | forEachLeafNodeWithinBounds | Connect
next | layers.(anonymous funct…afNodeWithinBounds.leaf | initSelector
(anonymous) | render | runComponentSelector
importFromChromeCPUProfile | getBuffer … | pureFinalPropsSelector
enterFrame | leaveFrame | f…e | … | handleFirstCall
_e…me | … | _…e | … | … | mergePropsProxy
mergeProps
exports.Chr…mechartView
args
(anonymous) | expo…hart
Flame…derer | Flamechart
(g…r) | forE…Call

Что может speedscope

- Интерактивный графический интерфейс
- Перемещение по профилю или мышью или WASD’ом
- Фильтрование по приложению/потоку
- Позволяет отображать как статистический профиль, так и «трассу»
  - Классический flamechart отображает просто соотношение весов фреймов, из него невозможно понять порядок выполнения

Time Order | Left Heavy | Help | simple-time-profile.trace | speedscope

500.00ms 1.00s 1.50s 2.00s 2.50s 3.00s 3.50s 4.00s

500.00ms 1.00s 1.50s 2.00s 2.50s 3.00s 3.50s 4.00s

_dyld_start | start
dyldbootst…ned long*) | main
dyld::_mai…ned long*) ...
dyld::link…igned int) ...
ImageLoade…ar const*) ...
ImageLoade…ool, bool) ...
ImageLoade…ool, bool) ...
ImageLoade…ool, bool) ...
ImageLoade…ool, bool) ...
ImageLoade…ool, bool) ...
ImageLoade…ool, bool)
ImageLoade…ool, bool)
ImageLoade…ool, bool)
ImageLoade…ool, bool)
ImageLoade…ool, bool)
ImageLoade…ool, bool)
ImageLoade…ool, bool)
ImageLoade…st&, bool)
ImageLoade…xt const&)

Made with Gifox

Небольшое резюме

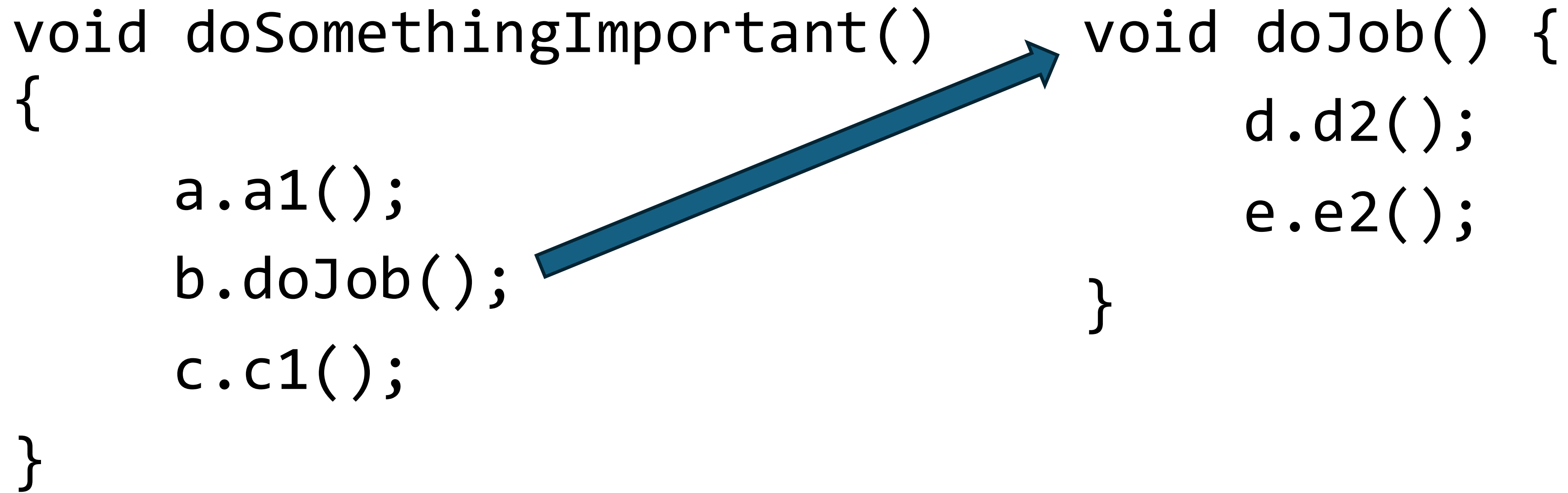
- Научились использовать perf для профилирования java
- Знаем целых 3 способа «подружить» java и perf
- Можем пользоваться красивым и удобным UI

Perf'ом можно пользоваться.

Это существенно проще, чем кажется

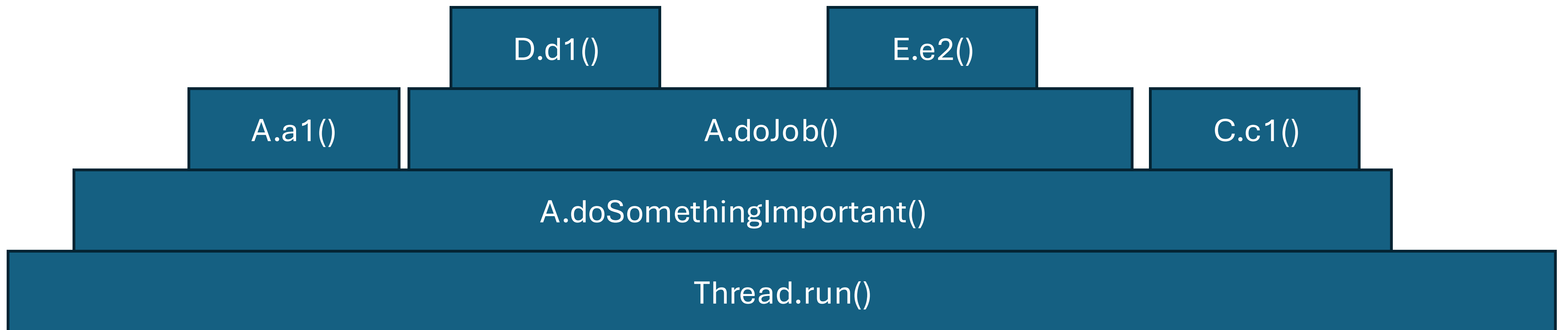
Виды профилировщиков

Как работает (почти) любой профилировщик

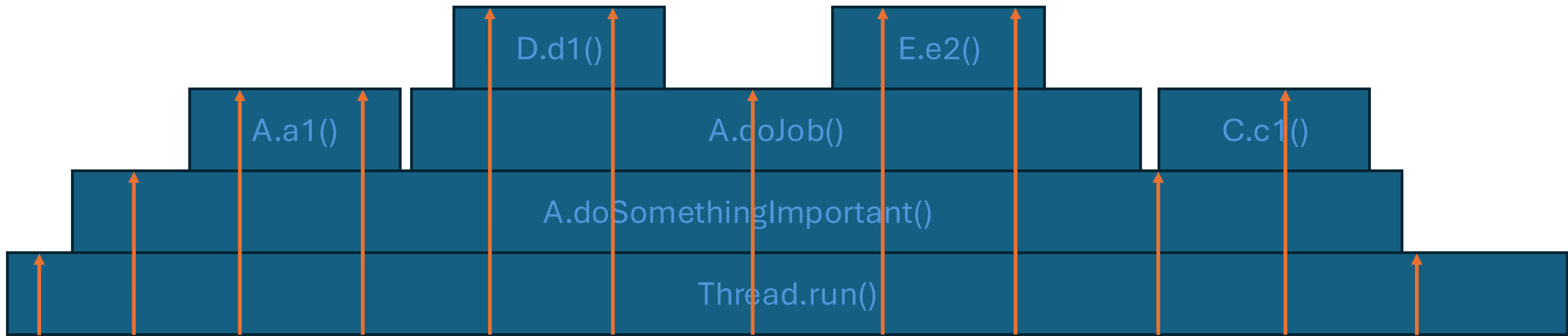

void doSomethingImportant()
{
a.a1();
b.doJob();
c.c1();
}
void doJob() {
d.d2();
e.e2();
}

```
void doSomethingImportant() {
     a.a1();
     b. doJob();
     c.c1();
}
```

```
void doJob() {
     d.d2();
     e.e2();
}
```



время




```
Thread.run
Thread.run -> A.doSomethingImportant
Thread.run -> A.doSomethingImportant -> A.a1
Thread.run -> A.doSomethingImportant -> A.a1
Thread.run -> A.doSomethingImportant -> A.doJob -> D.d1
Thread.run -> A.doSomethingImportant -> A.doJob -> D.d1
Thread.run -> A.doSomethingImportant -> A.doJob
Thread.run -> A.doSomethingImportant -> A.doJob -> E.e2
Thread.run -> A.doSomethingImportant -> A.doJob -> E.e2
Thread.run -> A.doSomethingImportant
Thread.run -> A.doSomethingImportant -> C.c1
Thread.run
```

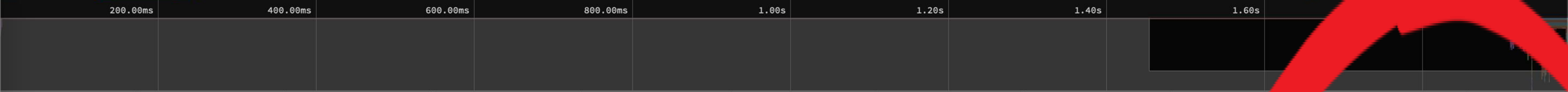
Статистический профиль из семплов

Стек	Количество семплов
`Thread.run`	2
`Thread.run -> A.doSomethingImportant`	2
`Thread.run -> A.doSomethingImportant -> A.a1`	2
`Thread.run -> A.doSomethingImportant -> A.doJob -> D.d1`	2
`Thread.run -> A.doSomethingImportant -> A.doJob`	1
`Thread.run -> A.doSomethingImportant -> A.doJob -> E.e2`	2
`Thread.run -> A.doSomethingImportant -> C.c1`	1

Статистический профиль из семплов

Стек	Количество семплов	% времени выполнения
`Thread.run`	2	**16%**
`Thread.run -> A.doSomethingImportant`	2	**16%**
`Thread.run -> A.doSomethingImportant -> A.a1`	2	**16%**
`Thread.run -> A.doSomethingImportant -> A.doJob -> D.d1`	2	**16%**
`Thread.run -> A.doSomethingImportant -> A.doJob`	1	**8%**
`Thread.run -> A.doSomethingImportant -> A.doJob -> E.e2`	2	**16%**
`Thread.run -> A.doSomethingImportant -> C.c1`	1	**8%**

Time Order | Left Heavy | Sandwich | java tid: 667547 (42/65) | Export | Help



1.50s 1.55s 1.60s 1.65s 1.70s 1.75s 1.80s 1.85s 1.95s

HW.do_smth(int)

??? (/usr/lib/x86_…nux-gnu/libc.so.6)
??? (/usr/lib/x86_…nux-gnu/libc.so.6)
ThreadJavaMain
JavaMain
JNI_CreateJavaVM | jni_Cal…Method
Threads::c…*, bool*) | jni_inv…rop.1]
init_…ls() | JavaCal…read*)
unive…it() | call_stub
Me…) | Interpreter
Me…) | In…e)
Me…) | In…*)
Me…) | Li…*)
Fi…) | Li…*)
Fi…) | Sy…*)
a…t | Ja…)
e…t | Ja…)
d…t | ca…b
h…t | In…r
_…t | In…r
h…t | In…r
In…r

This Instance		All Instances	
Total	Self	Total	Self
5.99ms	0.00ns	5.99ms	0.00ns

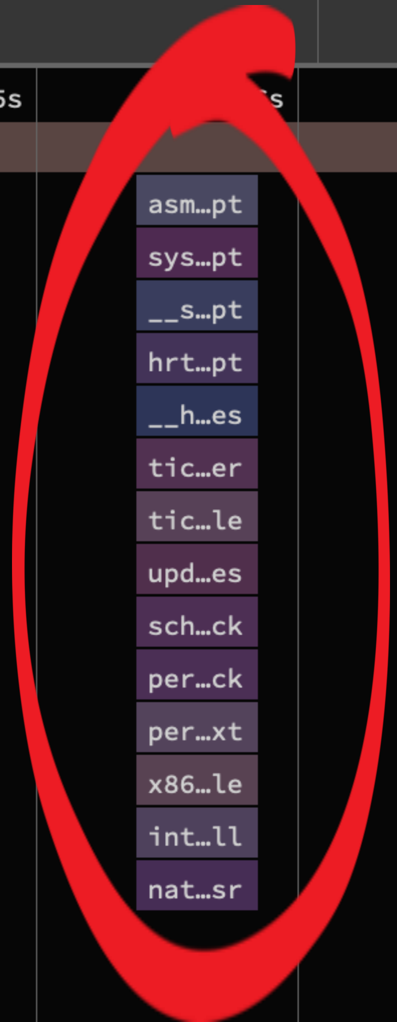
Threads::initialize_java_lang_classes(JavaThread*, JavaThread*) (/usr/lib/jvm/java-21-openjdk-amd64/lib/server/libjvm.so)
> Threads::create_vm(JavaVMInitArgs*, bool*) (/usr/lib/jvm/java-21-openjdk-amd64/lib/server/libjvm.so)
> JNI_CreateJavaVM (/usr/lib/jvm/java-21-openjdk-amd64/lib/server/libjvm.so)
> JavaMain (/usr/lib/jvm/java-21-openjdk-amd64/lib/libjli.so)
> ThreadJavaMain (/usr/lib/jvm/java-21-openjdk-amd64/lib/libjli.so)
> ??? (/usr/lib/x86_64-linux-gnu/libc.so.6) (/usr/lib/x86_64-linux-gnu/libc.so.6)

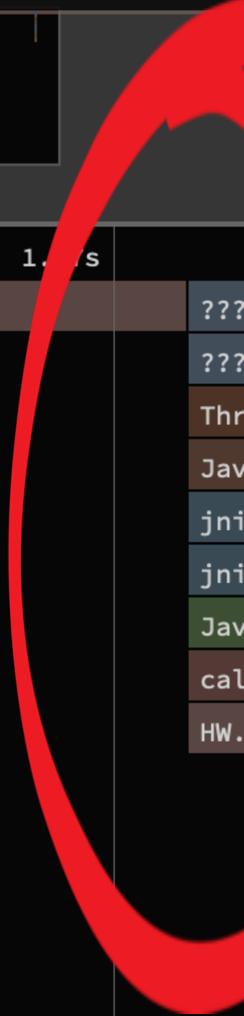
Sandwich

java tid: 667547 (42/65)

400.00ms 600.00ms 800.00ms 1.00s 1.20s

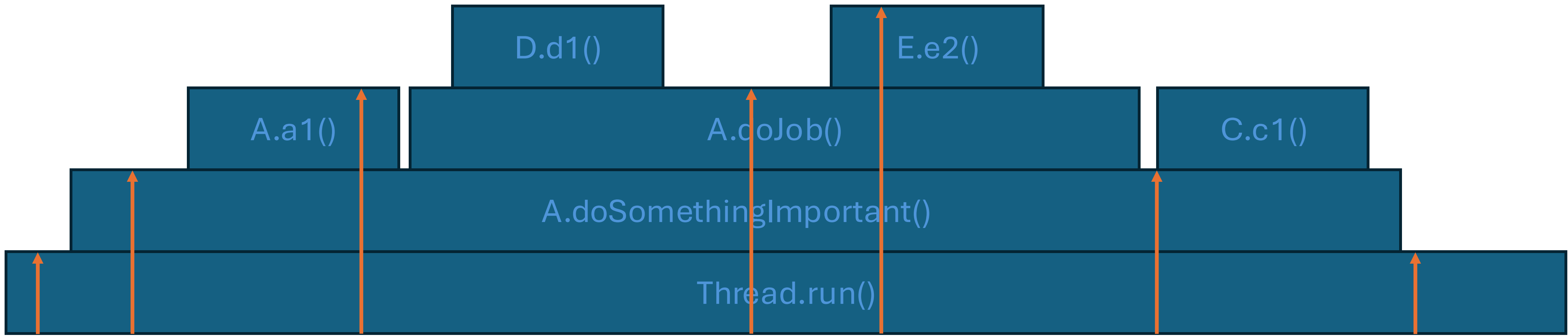
.25s 1.26s 1.26s 1.26s 1.26s 1.27s 1.27s 1.27s





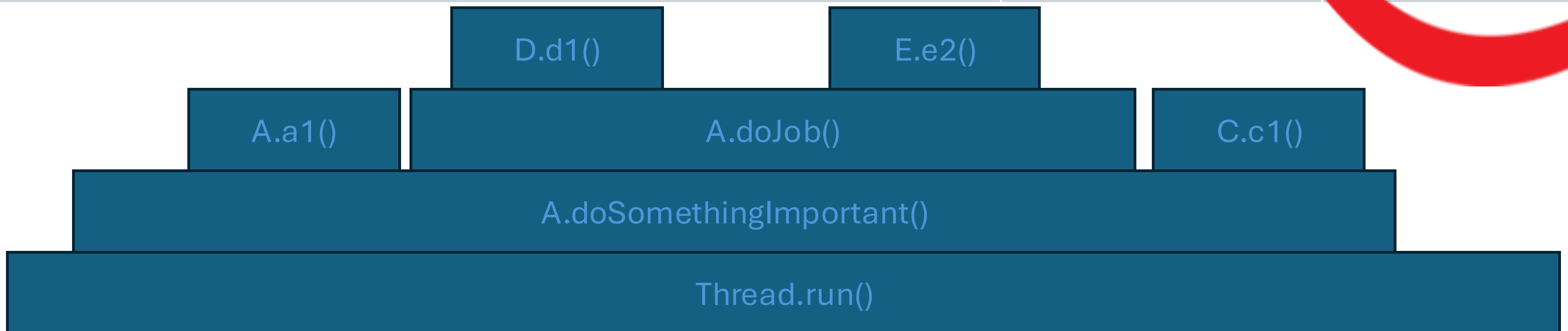
Если семпилирование частое, его результаты можно использовать как эрзац-трассу

Именно это позволяет увидеть speedscope


D.d1()
E.e2()
A.a1()
A.doJob()
C.c1()
A.doSomethingImportant()
Thread.run()

Получившийся профиль

Стек	Количество семплов	% времени выполнения
Thread.run	2	**28%**
Thread.run -> A.doSomethingImportant	2	**28%**
Thread.run -> A.doSomethingImportant -> A.a1	1	**14%**
Thread.run -> A.doSomethingImportant -> A.doJob	1	**14%**
Thread.run -> A.doSomethingImportant -> A.doJob -> E.e2	1	**14%**



В некоторых случаях семплирующие профилировщики бесполезны

- Coordinated omission
- Safepoint bias (и любой другой bias)
- Высокие квантили

Gil Tene
How NOT to Measure Latency

Nitsan Wakart
Profilers are lying hobbitses

Сергей Мельников
Профилируем черного лебедя

А можно получить настоящую трассу?

- Исторически, профилировщики делятся на семплирующие и трассирующие. Но сейчас применяются семплирующие
- Вторые применяются редко из-за огромных накладных расходов (~10x)

Резюме по профилировщикам

- Поняли отличия профилирующих и семплирующих профилировщиков
- Иногда статистические результаты не позволяют профилировать то, что интересует. И об этом надо помнить
- Давайте все-таки попробуем собрать трассу!

Трасса программы на java

Как же все-таки собирают трассу?

```
void doSomething() {
        // do something
}
```

```
void doSomething() {
        startT = System.nanoTime();
        try {
                // do something
        } finally {
                endT = System.nanoTime();
                writeExecution()
        }
}
```

Накладные расходы – огромные (~10x)

Но если очень хочется – то ведь можно?

Intel Processor Trace

1. Записывает для каждой инструкции условного ветвления, taken или нет эта инструкция
2. Периодически записывает rdtsc – тики железа
3. Имея (1) и ассемблер, можно восстановить полную трассу выполнения программы с наносекундной точностью

Особенности Intel processor trace

- Полностью реализовано в «железе»
- Низкие накладные расходы (до 15%)
- Генерирует настоящую трассу выполнения программы

Поддержка железа

- Впервые появилось в Broadwell (2014-2015 года)
- Полноценно реализована в Skylake (2015-2017 годы)

Фактически, любой современный процессор поддерживается

Поддержка ОС (linux)

- Первые коммиты попали в linux 4.1
- Полностью реализовано к linux 4.11

- Ubuntu 20.04 основана на 5.4
- Ubuntu 22.04 основана на 5.15

Можно использовать!

Ложка дегтя

- Похоже, не виртуализуется, т.е. нужно «голое железо»
- Может приводить к зависанию ОС из-за объема генер ируемых данных – нужно пользоваться осторожно

От статистического профиля к трассе

- Perf уже поддерживает использование processor trace для профилирования, правки сделаны давно
- Но этот режим похоже никто даже не планировал использовать с java – мы будем первыми!

Как это работает

- Запускаем JVM с правильным jvmti агентом
  - `$ java -agentpath:PATH_TO_libperf-jvmti.so -cp .`
- Очень аккуратно пробуем запускать perf в режиме processor trace
  - `$ perf record -e intel_pt// -a -- sleep 0.1`

Инжектим JIT данные в профиль

```
$ perf inject --jit -i perf.data -o perf.data.jittes
error, jitted code must be sampled with perf record -k 1
```

Выключаем проверку использования указанных точных часов.
Никто никогда не рассчитывал, что jit-код будет профилироваться с помощью processor trace

Рекомендации

- Perf любит записывать данные сразу на диск
- Processor trace генерирует ОООЧЕНЬ много данных
- Возможны «зависания» ОС, когда данные получаются perf’ом

```
# echo $[100*1024*1024] > /proc/sys/kernel/perf_event_mlock_kb

$ perf record -m 512,100000 -e intel_pt// ...
```

Но уж больно плохие данные

```
Samples: 88K of event 'psb', Event count (approx.): 88327
  Overhead  Command          Shared Object          Symbol
+   94.75%  java             jitted-13345-438.so    [.] HW2.doSmth(int)
+    1.72%  swapper          [unknown]              [k] 0000000000000000
+    0.85%  perf             [unknown]              [k] 0000000000000000
+    0.58%  sleep            [unknown]              [k] 0000000000000000
+    0.57%  java             [unknown]              [k] 0000000000000000
```

Хочется больше детализации

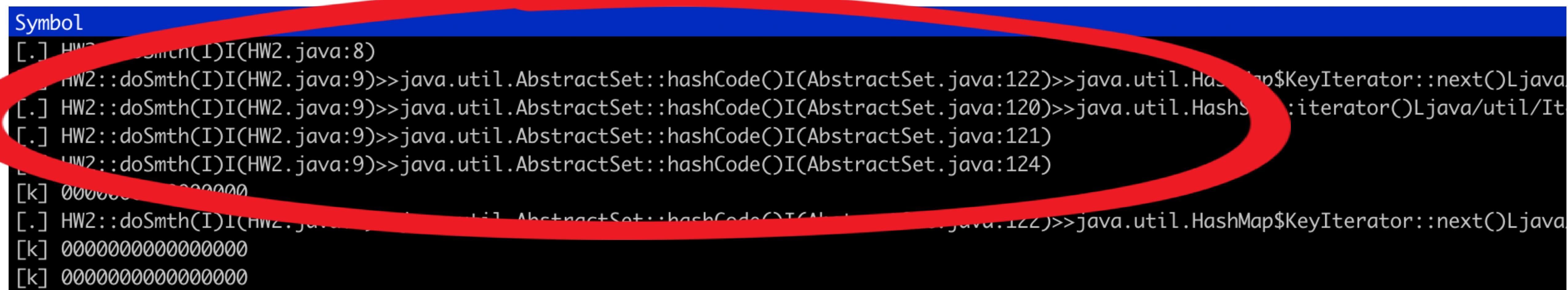
Сейчас отображается только тот метод, в который выполнялся инлайнинг

А хочется получать информацию и про заинлайненные методы

Мои правки

- Лежат здесь:
  - https://github.com/RainM/joker-perf-diff/blob/master/joker.diff
- Модифицируют libperf-jvmti.so – jvmti агента
- Позволяют отображать стек инлайнинга

Overhead
Shared Object
+ 44.55% java jitted-13506-2062.so
+ 17.01% java jitted-13506-2061.so
+ 16.65% java jitted-13506-2049.so
+ 12.60% java jitted-13506-2057.so
jitted-13506-2055.so
+ 1.38% swapper [unknown]
+ 1.01% java jitted-13506-2063.so
+ 0.95% perf [unknown]
+ 0.63% java [unknown]

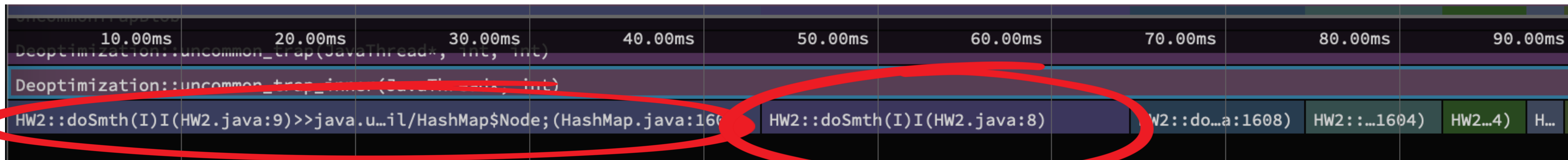

Symbol
[.] HW2::doSmth(I)I(HW2.java:9)>>java.util.AbstractSet::hashCode()I(AbstractSet.java:120)>>java.util.Hash
[.] HW2::doSmth(I)I(HW2.java:9)>>java.util.AbstractSet::hashCode()I(AbstractSet.java:121)
[k] 0000000000000000
[k] 0000000000000000

Визуализация трассы

- Формат трассы, генерируемый perf script уникален. Т.е. из коробки доступен только perf report
- Трассу в семплы можно преобразовать так:

```
$ perf script --itrace=i100usg
```

Полученную трассу смотреть в speedscope



Правки работают и для «обычного» режима профилирования

А мне предстоит длительный процесс апстрима изменений

Время подводить итоги

Итого

- Помотрели как использовать самый распространенный в linux профилировщик (perf) с java – 3 способами!
- Мы научились использовать perf с intel processor trace для профилирования java – похоже первые!
- Научились собирать профиль, отображает структуру инлайнинга
- Узнали удобный интерфейс для отображения профиля

Q&A?

Дискуссионная зона!

Sergey Melnikov

@SergeyM12

Dijkstra-Markets