Mastering Java performance with async-profiler

2019

Andrei Pangin
Odnoklassniki

About me

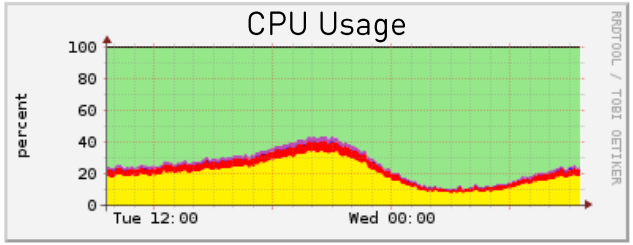
AndreiPangin

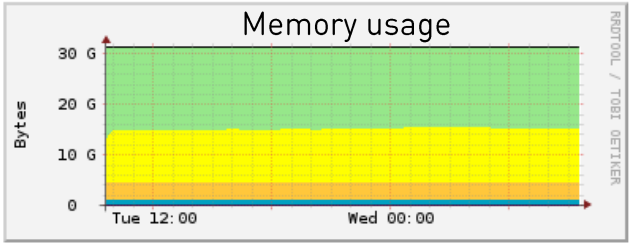
- Principal Engineer @ Odnoklassniki
- JVM enthusiast
- Top #jvm answerer on Stack Overflow
- Author of async-profiler
  https://github.com/jvm-profiling-tools/async-profiler

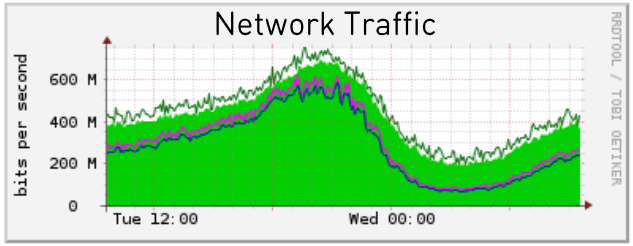
Agenda

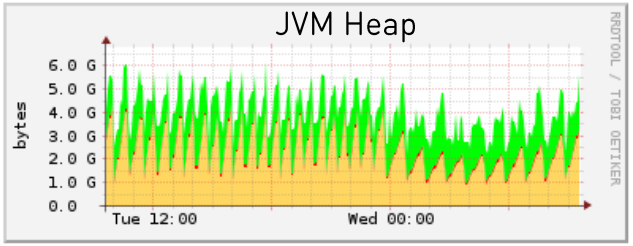
1. Profiling theory
2. What's wrong with the classical approach
3. Introduction to async-profiler
4. Allocation profiling
5. Advanced techniques

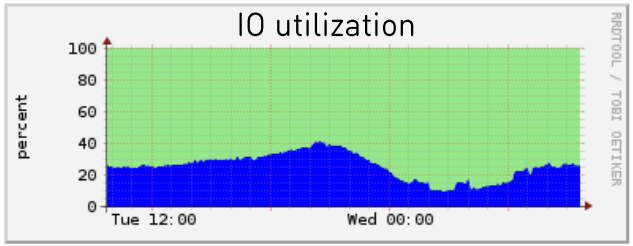
What to optimize?

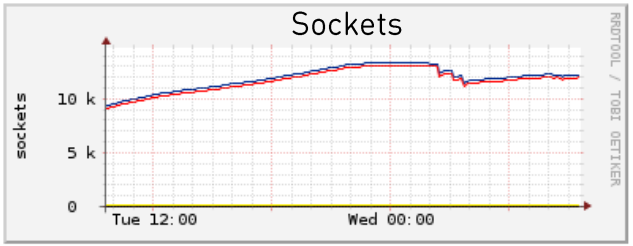




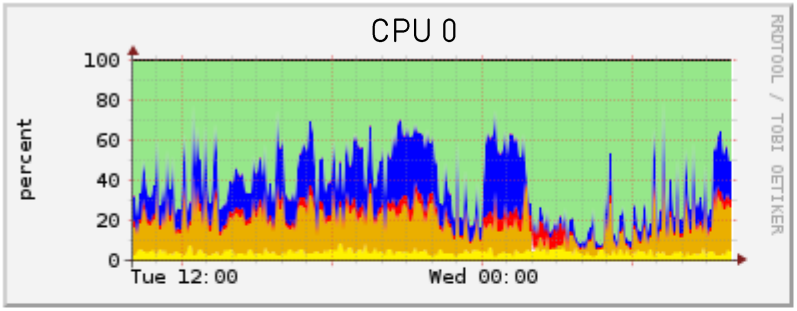


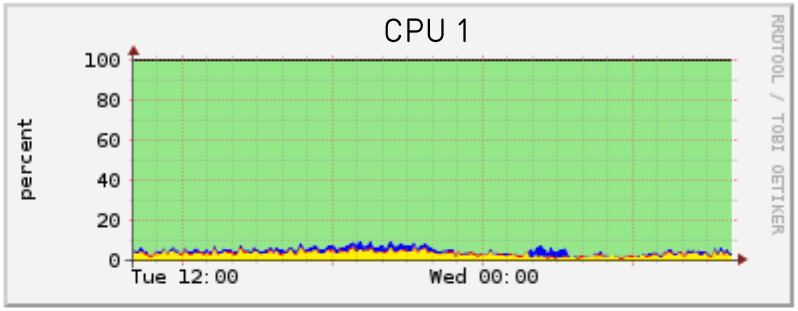


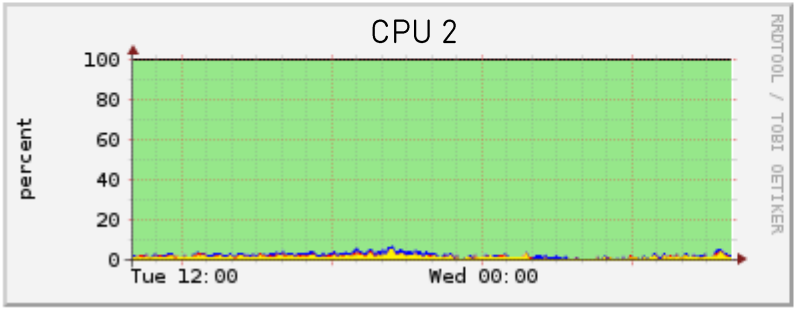


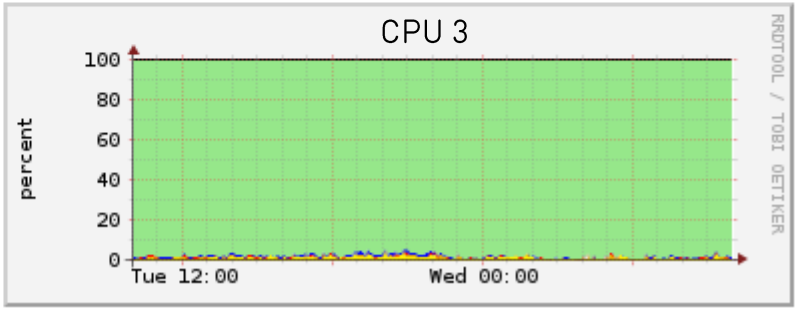


To profile or not to profile?









CPU profiling

How to profile

Instrumenting

Trace method transitions

- Sloooow

Sampling

Periodic snapshots

- Production ready

Thread Dump

Java API

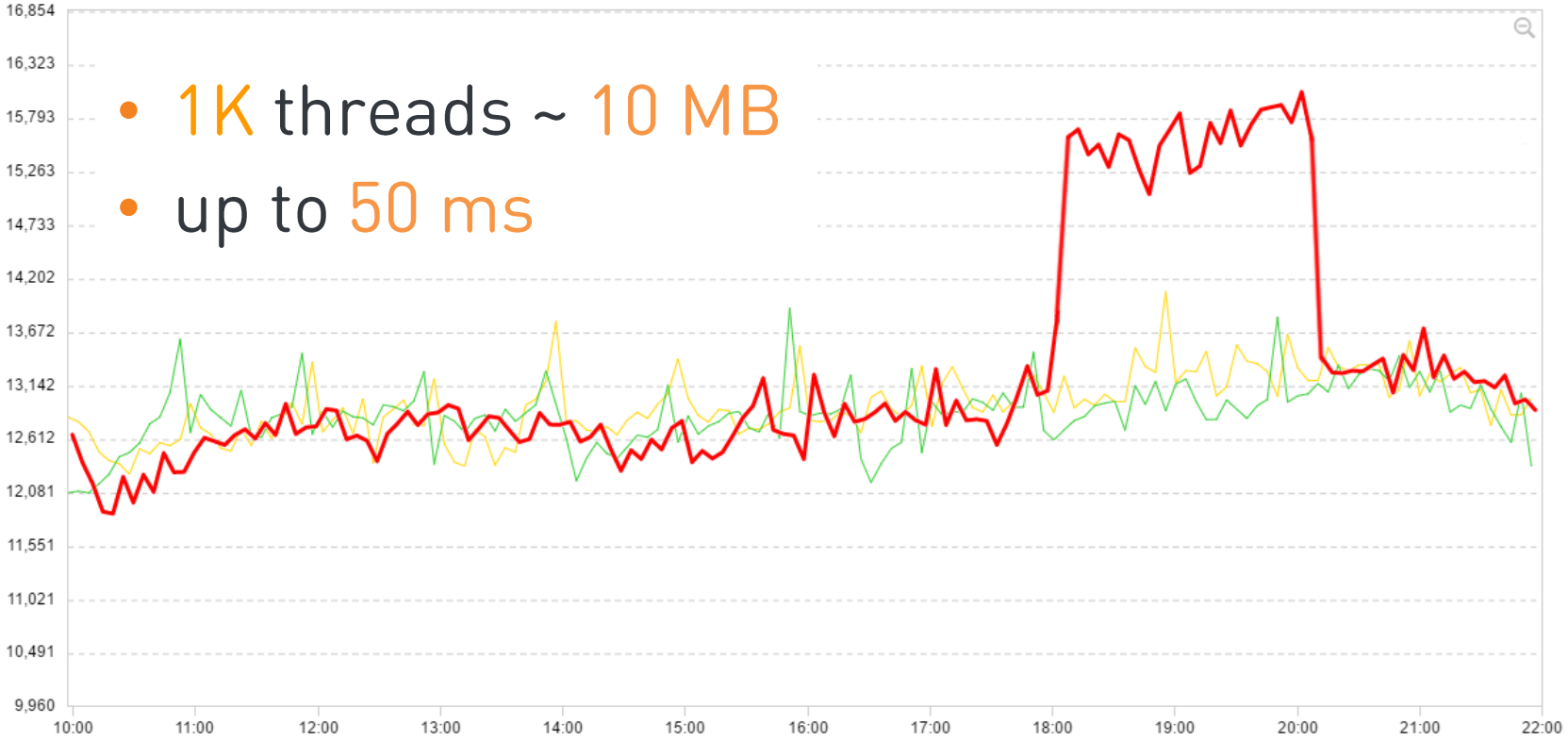
Thread.getAllStackTraces()

Map<Thread, StackTraceElement[]>

Native (JVMTI)

GetAllStackTraces()

Overhead


• 1K threads ~ 10 MB
• up to 50 ms
16,854
16,323
15,793
15,263
14,733
14,202
13,672
13,142
12,612
12,081
11,551
11,021
10,491
9,960
10:00
11:00
12:00
13:00
14:00
15:00
16:00
17:00
18:00
19:00
20:00
21:00
22:00

Advantages

- Simple
- All Java platforms
- No extra JVM options required

✓ Supported by many profilers

Replying to @AndreiPangin

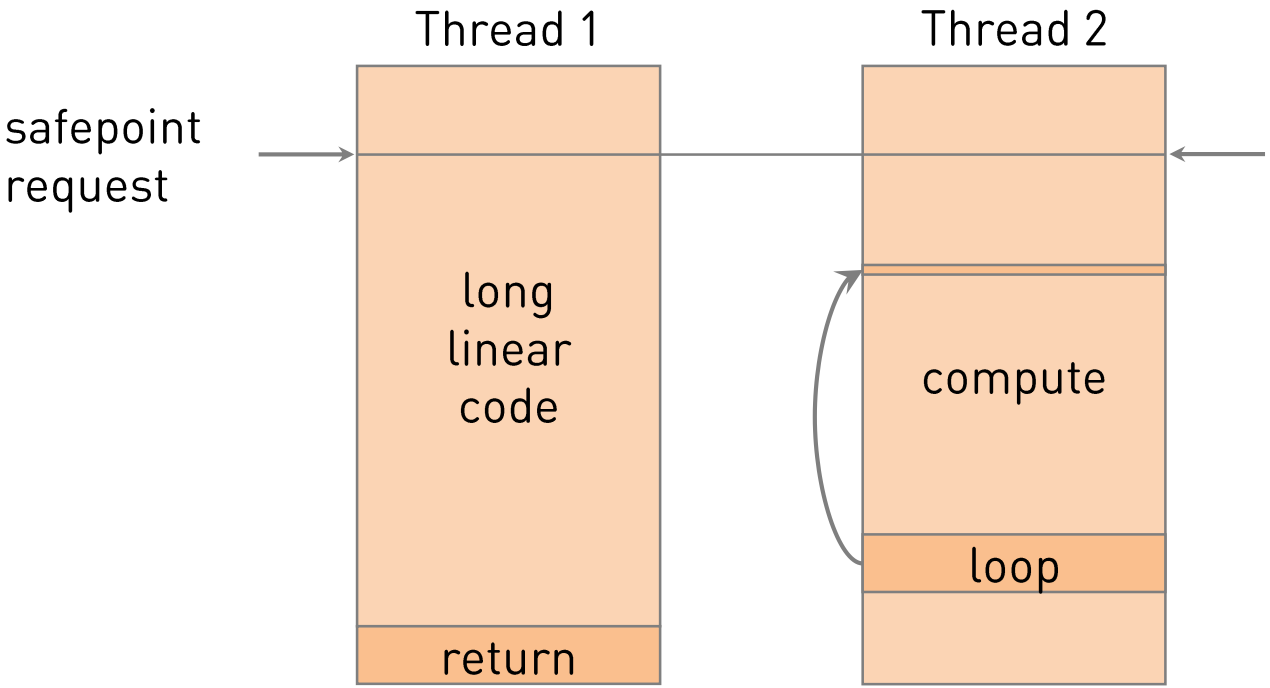
I'm happy with ✸Profiler ... no need for another tool

9:37 PM - 18 May 2018

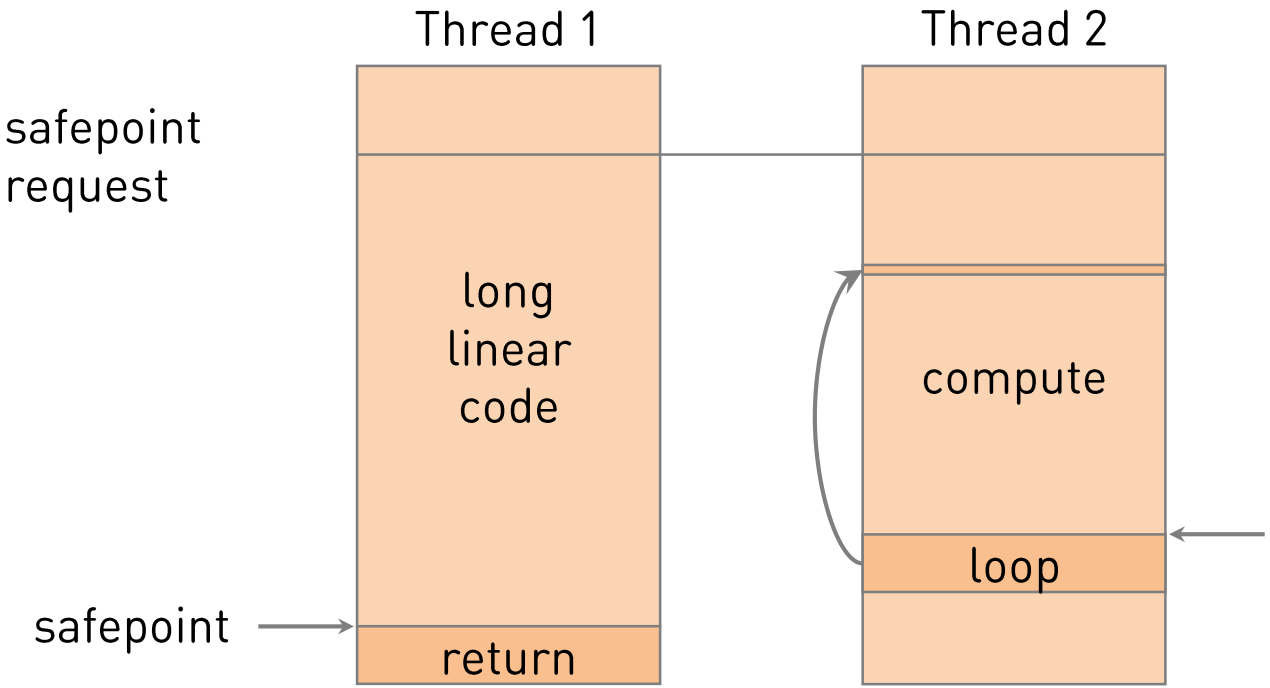



Demo

Safepoint


Thread 1
Thread 2
safepoint
request
long
linear
code
return
compute
loop

Safepoint

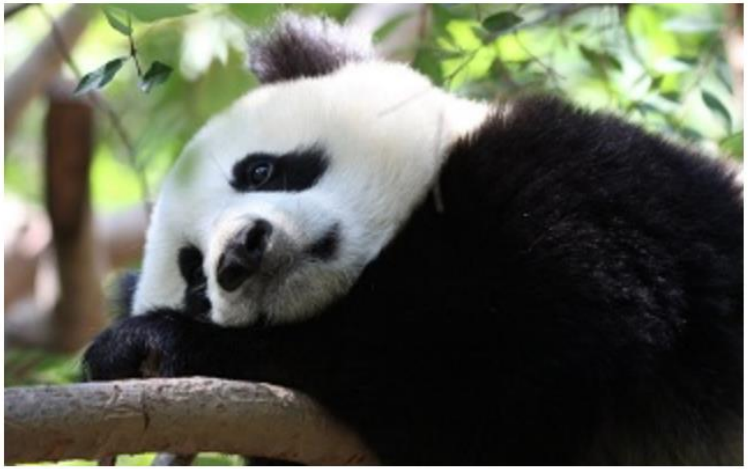

Thread 1
Thread 2
safepoint
request
long
linear
code
compute
loop
safepoint
return

Wednesday, 24 February 2016

Why (Most) Sampling Java Profilers Are F[redacted] Terrible

This post builds on the basis of a previous post on safepoints. If you've not read it you might feel lost and confused. If you have read it, and still feel lost and confused, and you are certain this feeling is related to the matter at hand (as opposed to an existential crisis), please ask away.
So, now that we've established what safepoints are, and that:

1. Safepoint polls are dispersed at fairly arbitrary points (depending on execution mode, mostly at uncounted loop back edge or method return/entry).
2. Bringing the JVM to a global safepoint is high cost



http://psy-lob-saw.blogspot.ru/2016/02/why-most-sampling-java-profilers-are.html

because of Safepoint bias

...not the profiler's fault...

http://psy-lob-saw.blogspot.ru/2016/02/why-most-sampling-java-profilers-are.html

Native methods

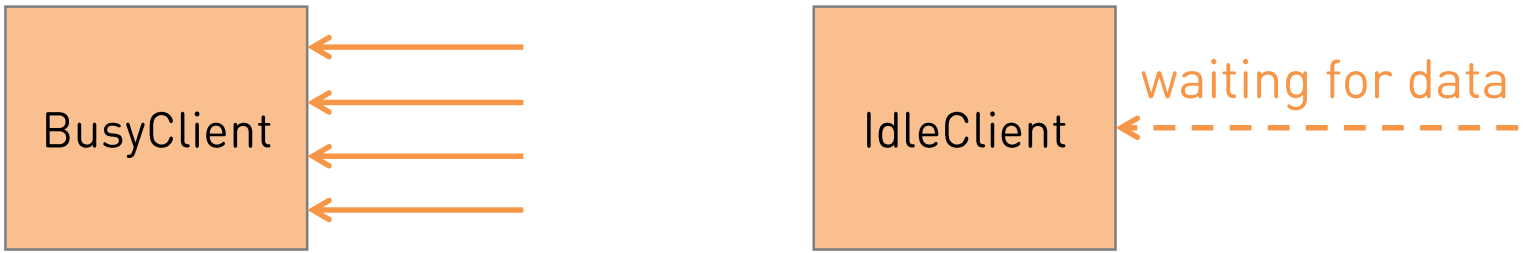
```
Socket s = new Socket(host, port);

InputStream in = s.getInputStream();
while (in.read(buf) >= 0) {
    // keep reading
}
```

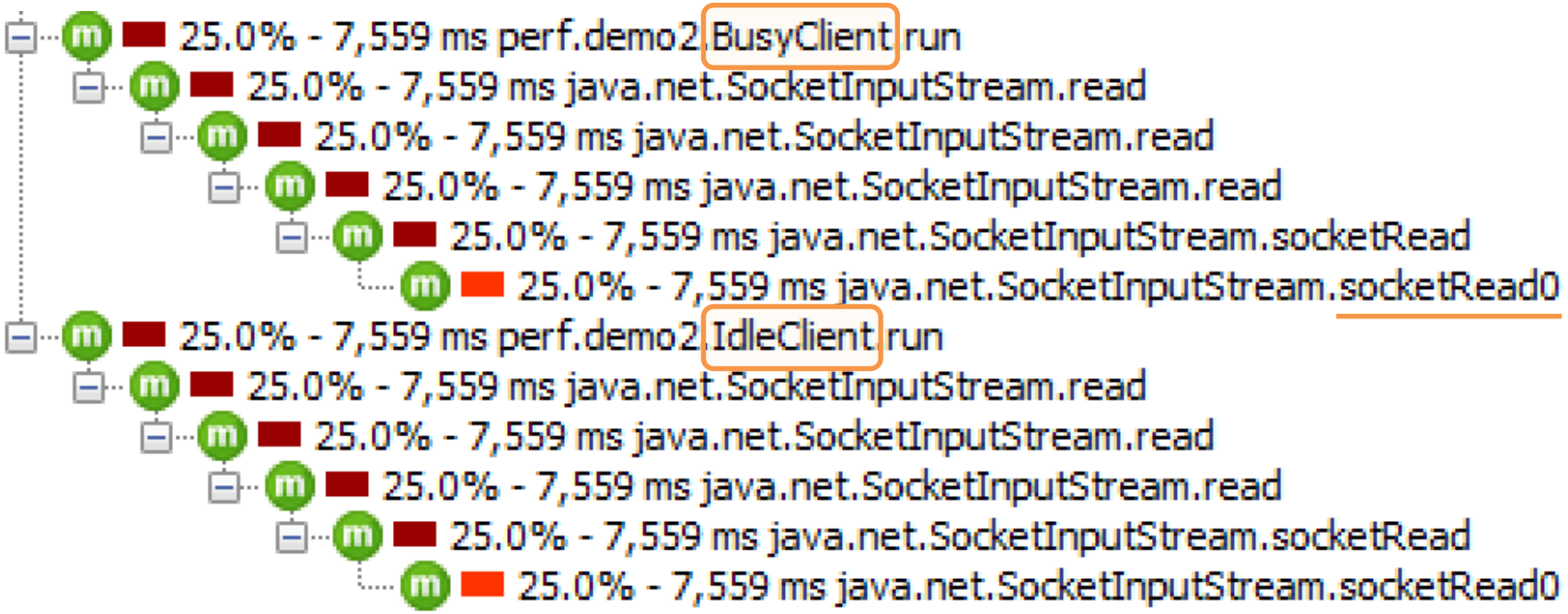
Native methods

```
Socket s = new Socket(host, port);

InputStream in = s.getInputStream();
while (in.read(buf) >= 0) {
    // keep reading
}
```



Native methods



Solving GetAllStackTraces problems

- Avoid safepoint bias
- Skip idle threads
- Profile native code

AsyncGetCallTrace

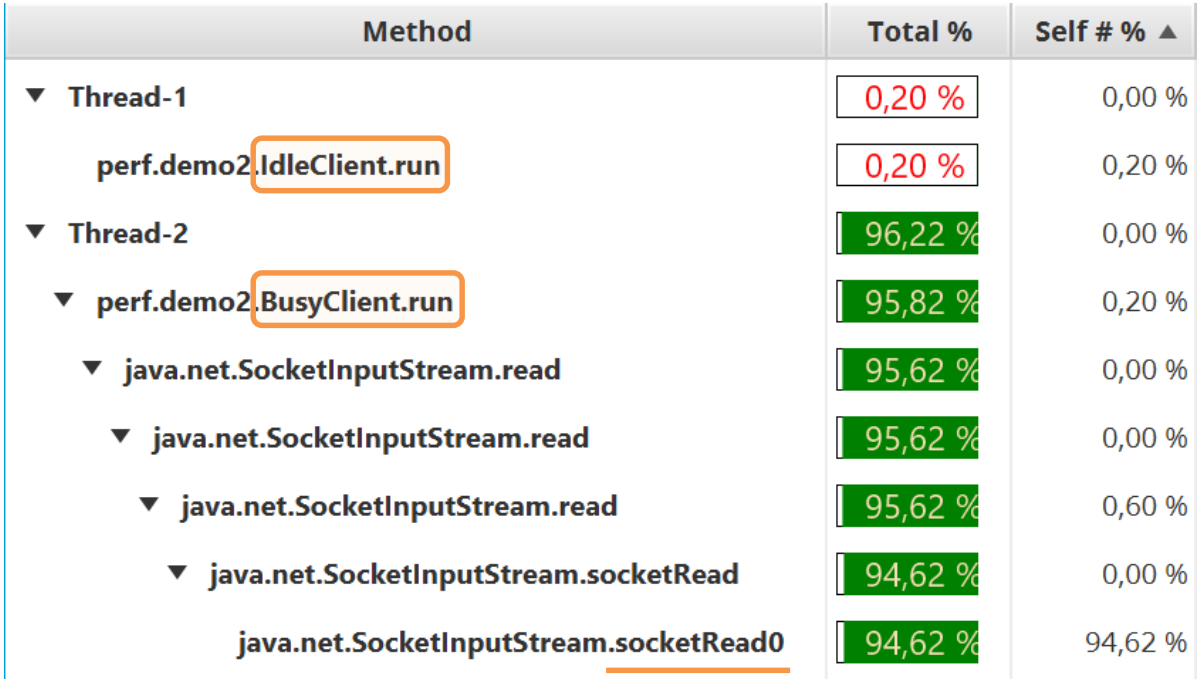
How it works

```
AsyncGetCallTrace(ASGCT_CallTrace *trace,
                  jint depth,
                  void* ucontext)
```

from itimer signal handler

- Oracle Developer Studio
- github.com/jvm-profiling-tools/honest-profiler
- github.com/jvm-profiling-tools/async-profiler

honest-profiler example



Method	Total %	Self # % ▲
▼ Thread-1	0,20 %	0,00 %
perf.demo2.IdleClient.run	0,20 %	0,20 %
▼ Thread-2	96,22 %	0,00 %
▼ perf.demo2.BusyClient.run	95,82 %	0,20 %
▼ java.net.SocketInputStream.read	95,62 %	0,00 %
▼ java.net.SocketInputStream.read	95,62 %	0,00 %
▼ java.net.SocketInputStream.read	95,62 %	0,60 %
▼ java.net.SocketInputStream.socketRead	94,62 %	0,00 %
java.net.SocketInputStream.socketRead0	94,62 %	94,62 %

AsyncGetCallTrace

- Pros
  - Only active threads
  - No safepoint bias
    -XX:+DebugNonSafepoints
- Cons
  - ✖ No Windows support
  - ✖ No native or JVM code

The bad part

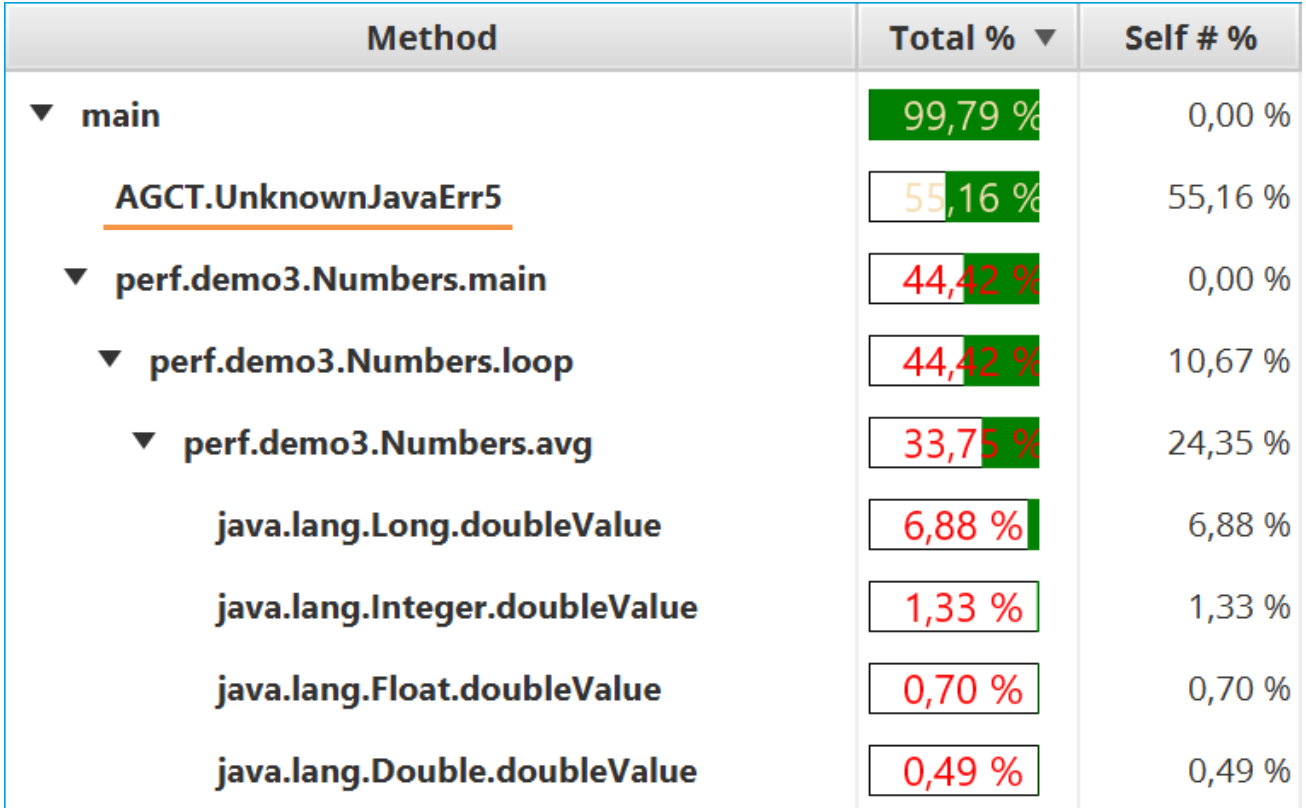
Method	Total %	Self # %
main	98,07 %	0,00 %
AGCT.UnknownJavaErr5	95,22 %	95,22 %
perf.demo1.ArrayListTest.main	2,64 %	0,29 %
sun.launcher.LauncherHelper.checkAndLoadMain	0,21 %	0,00 %
Unknown Thread(s)	1,85 %	0,00 %
AGCT.UnknownNotJava3	1,85 %	1,85 %

The bad part

```
double avg(Number... numbers) {
    double sum = 0;
    for (Number n : numbers) {
        sum += n.doubleValue();
    }
    return sum / numbers.length;
}


x = avg(123, 45.67, 890L, 33.3f, 999, 787878L);
```

The bad part



Method	Total %	Self # %
main	99,79 %	0,00 %
AGCT.UnknownJavaErr5	55,16 %	55,16 %
perf.demo3.Numbers.main	44,42 %	0,00 %
perf.demo3.Numbers.loop	44,42 %	10,67 %
perf.demo3.Numbers.avg	33,75 %	24,35 %
java.lang.Long.doubleValue	6,88 %	6,88 %
java.lang.Integer.doubleValue	1,33 %	1,33 %
java.lang.Float.doubleValue	0,70 %	0,70 %
java.lang.Double.doubleValue	0,49 %	0,49 %

Problems

```
enum {
  ticks_no_Java_frame          =  0,
  ticks_no_class_load          = -1,
  ticks_GC_active              = -2,
  ticks_unknown_not_Java       = -3,
  ticks_not_walkable_not_Java  = -4,
  ticks_unknown_Java           = -5,
  ticks_not_walkable_Java      = -6,
  ticks_unknown_state          = -7,
  ticks_thread_exit            = -8,
  ticks_deopt                  = -9,
  ticks_safepoint              = -10
};
```

src/share/vm/prims/forte.cpp

JDK / JDK-8178287

AsyncGetCallTrace fails to traverse valid Java stacks

Details

Type:	Bug	Status:	OPEN
Priority:	P4	Resolution:	Unresolved
Affects Version/s:	8u121, 9, 10	Fix Version/s:	tbd
Component/s:	hotspot		
Labels:	AsyncGetCallTrace analyzer		
Subcomponent:	svc		
CPU:	x86_64		

Description

There is a number of cases (observed in real applications) when profiling with AsyncGetCallTrace is useless due to HotSpot inability to walk through Java stack. Here is the analysis of such cases.

#1. An application performs many System.arraycopy() and spends a lot of time inside

People

Assignee: Unassigned

Reporter: Andrei Pangin

Votes: 0 Vote for this issue

Watchers: 3 Start watching this issue

Dates

Created: 2017-04-06 17:32

Perf Events

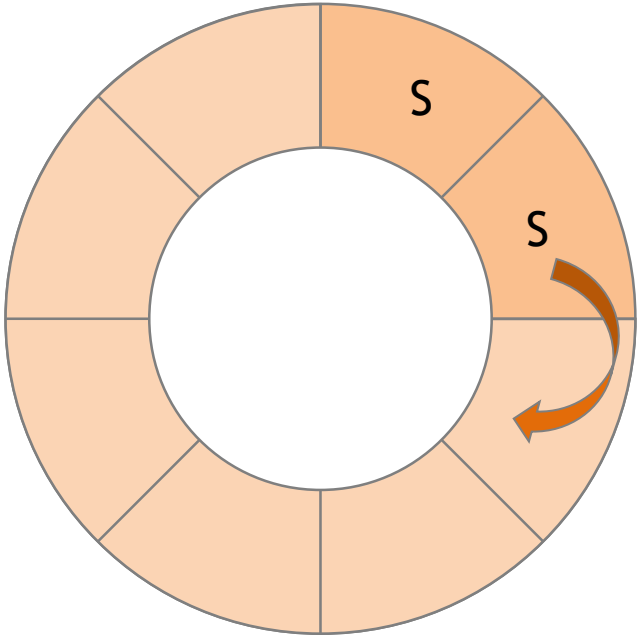
PMU

HW
interrupt

- Hardware counters
  - Cycles, instructions
  - Cache misses
  - Branch misses
  - etc.

perf_event_open

- Linux syscall
  - Subscribe to HW/OS events
- Samples
  - pid, tid
  - CPU registers
  - Call chain (user + kernel)



perf

```
$ perf record -F 1009 java ...
$ perf report
```

perf.wiki.kernel.org/index.php/Tutorial

perf

```
$ perf record -F 1009 java ...
$ perf report
```

```
  4.70%    java    [kernel.kallsyms]     [k] clear_page_c
  2.10%    java    libpthread-2.17.so    [.] pthread_cond_wait
  1.97%    java    libjvm.so             [.] Unsafe_Park
  1.40%    java    libjvm.so             [.] Parker::park
  1.31%    java    [kernel.kallsyms]     [k] try_to_wake_up
  1.31%    java    perf-18762.map        [.] 0x00007f8510e9e757
  1.21%    java    perf-18762.map        [.] 0x00007f8510e9e89e
  1.17%    java    perf-18762.map        [.] 0x00007f8510e9cc17
```

perf.wiki.kernel.org/index.php/Tutorial

perf-map-agent

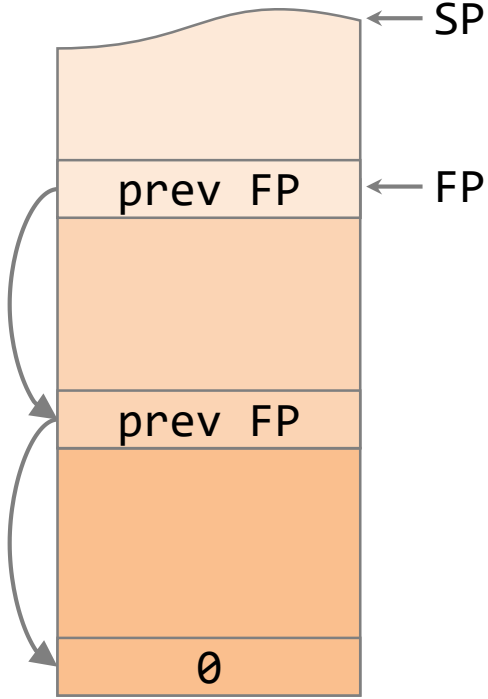
github.com/jvm-profiling-tools/perf-map-agent

```
$ java -agentpath:/usr/lib/libperfmap.so ...
```

```
7fe0e91175e0  140  java.lang.String::hashCode
7fe0e9117900   20  java.lang.Math::min
7fe0e9117ae0   60  java.lang.String::length
7fe0e9117d20  180  java.lang.String::indexOf
```

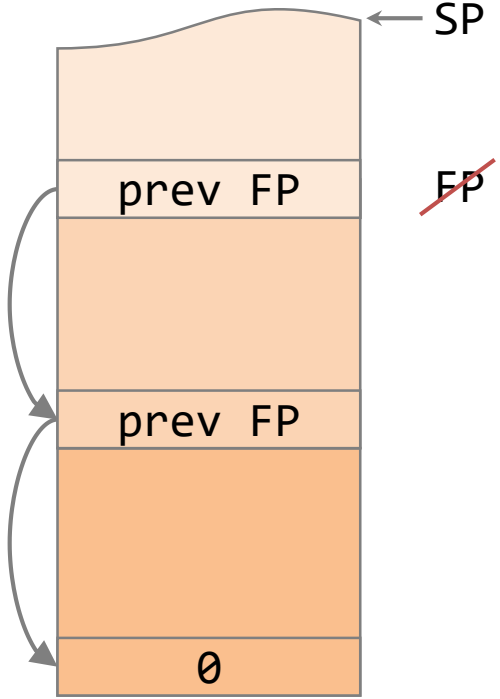
JVM support for perf

- -XX:+PreserveFramePointer



JVM support for perf

- -XX:+PreserveFramePointer
- JDK ⩾ 8u60
- Permanent overhead (< 5%)

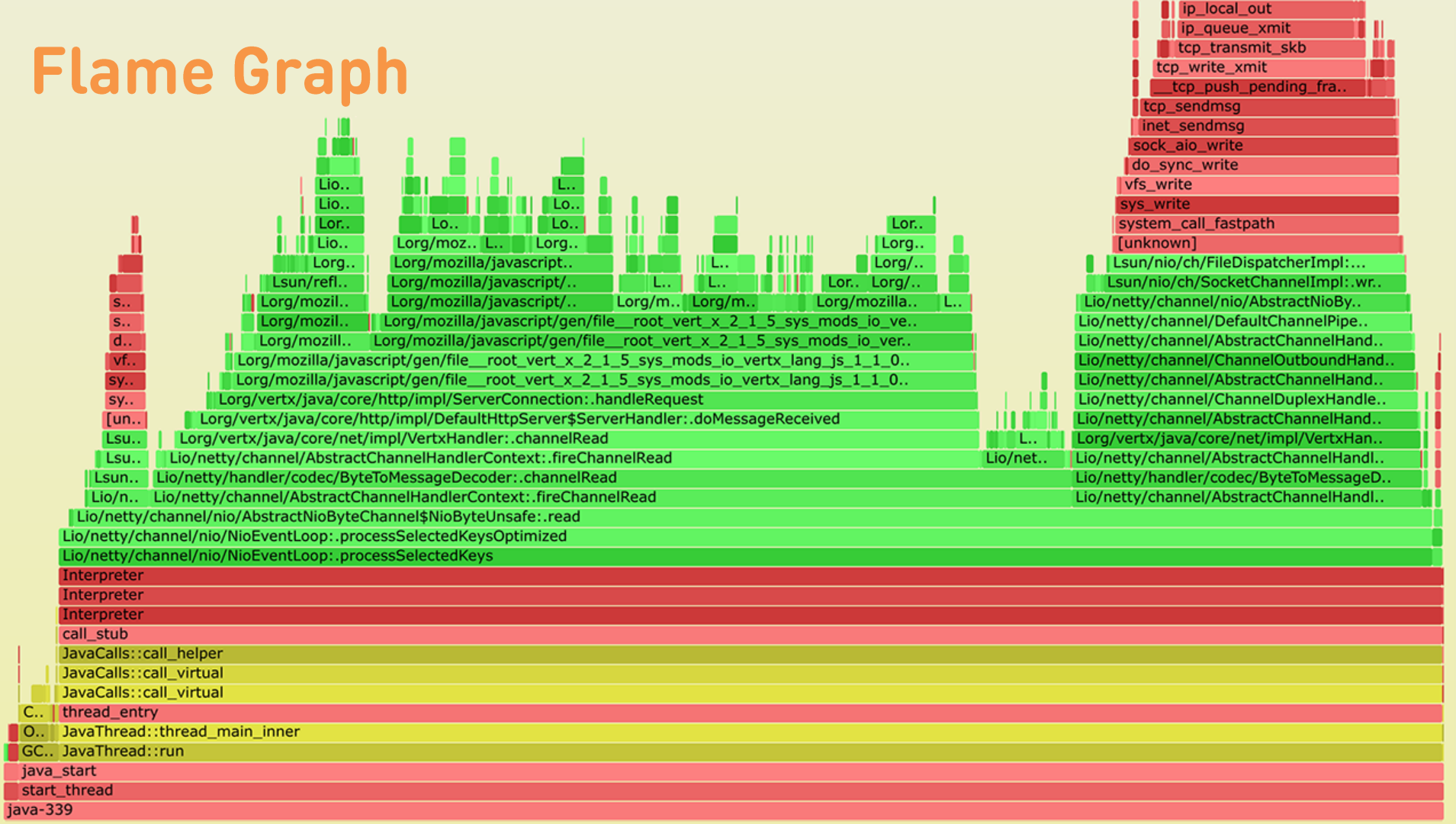


perf + FlameGraph

1. perf record
2. perf script
3. FlameGraph/stackcollapse-perf.pl
4. FlameGraph/flamegraph.pl

github.com/brendangregg/FlameGraph

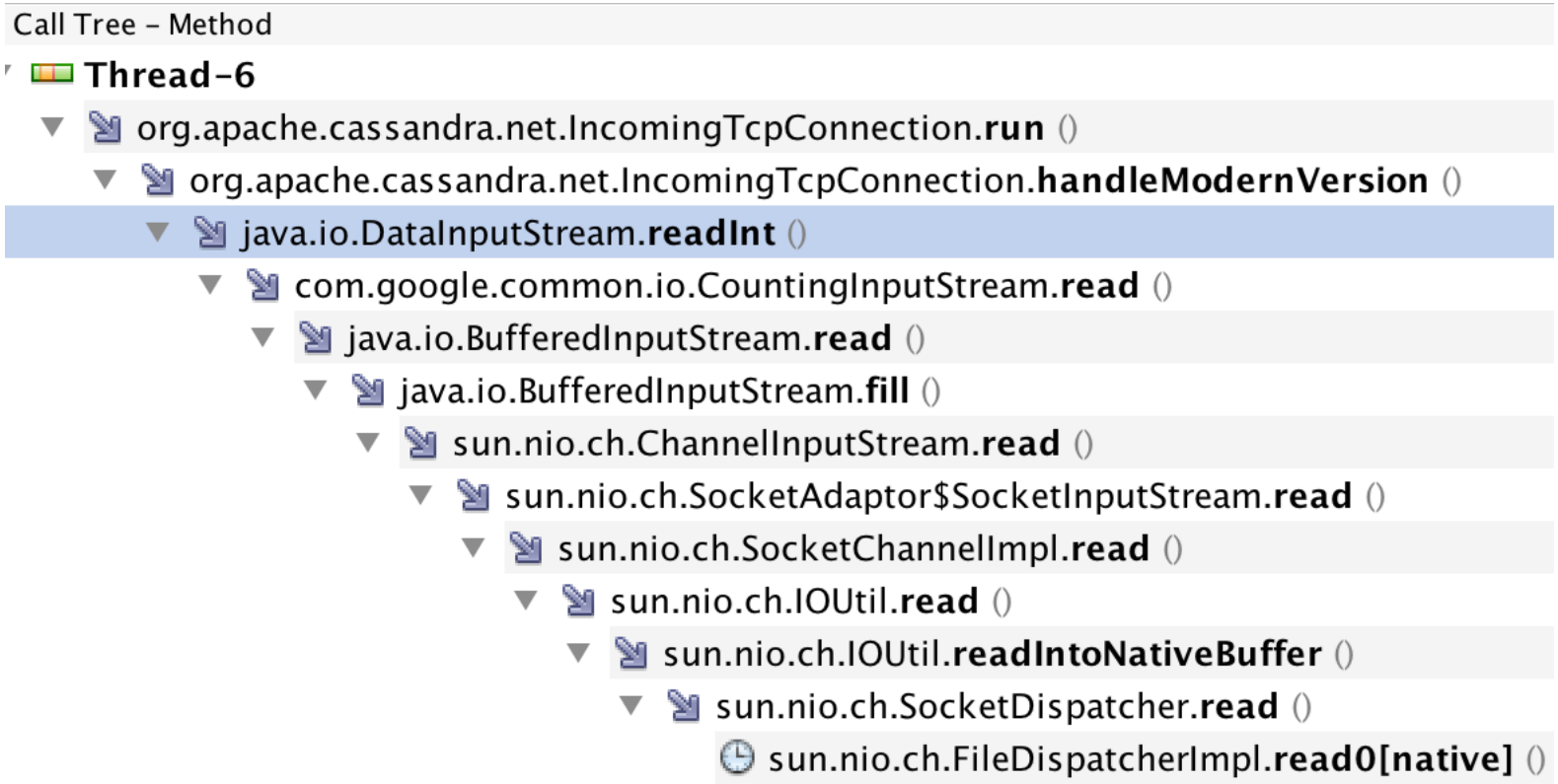
Flame Graph


ip_local_out
ip_queue_xmit
tcp_transmit_skb
tcp_write_xmit
__tcp_push_pending_fra..
tcp_sendmsg
inet_sendmsg
sock_aio_write
do_sync_write
vfs_write
sys_write
system_call_fastpath
[unknown]
Lsun/nio/ch/FileDispatcherImpl:...
Lsun/nio/ch/SocketChannelImpl:.wr..
Lio/netty/channel/nio/AbstractNioBy..
Lio/netty/channel/DefaultChannelPipe..
Lio/netty/channel/AbstractChannelHand..
Lio/netty/channel/ChannelOutboundHand..
Lio/netty/channel/AbstractChannelHand..
Lio/netty/channel/ChannelDuplexHandle..
Lio/netty/channel/AbstractChannelHand..
Lorg/vertx/java/core/net/impl/VertxHan..
Lio/netty/channel/AbstractChannelHandl..
Lio/netty/handler/codec/ByteToMessageD..
Lio/netty/channel/AbstractChannelHandl..
Lio/net..
Lio..
Lor..
Lorg..
Lorg/..
Lorg/mozilla..
Lorg/m..
Lorg/moz..
Lorg/mozilla/javascript..
Lorg/mozilla/javascript/..
Lorg/mozilla/javascript/gen/file__root_vert_x_2_1_5_sys_mods_io_ve..
Lorg/mozilla/javascript/gen/file__root_vert_x_2_1_5_sys_mods_io_ver..
Lorg/mozilla/javascript/gen/file__root_vert_x_2_1_5_sys_mods_io_vertx_lang_js_1_1_0..
Lorg/vertx/java/core/http/impl/ServerConnection:.handleRequest
Lorg/vertx/java/core/http/impl/DefaultHttpServer$ServerHandler:.doMessageReceived
Lorg/vertx/java/core/net/impl/VertxHandler:.channelRead
Lio/netty/channel/AbstractChannelHandlerContext:.fireChannelRead
Lio/netty/handler/codec/ByteToMessageDecoder:.channelRead
Lio/netty/channel/AbstractChannelHandlerContext:.fireChannelRead
Lio/netty/channel/nio/AbstractNioByteChannel$NioByteUnsafe:.read
Lio/netty/channel/nio/NioEventLoop:.processSelectedKeysOptimized
Lio/netty/channel/nio/NioEventLoop:.processSelectedKeys
Interpreter
Interpreter
Interpreter
call_stub
JavaCalls::call_helper
JavaCalls::call_virtual
JavaCalls::call_virtual
thread_entry
JavaThread::thread_main_inner
JavaThread::run
java_start
start_thread
java-339

Flat profile

Hot Spots – Method	Self Time [%]
sun.nio.ch.FileDispatcherImpl.**read0[native]** ()	
java.net.SocketInputStream.**socketRead0[native]** ()	
one.nio.net.NativeSelector.**epollWait[native]** ()	
one.nio.net.NativeSocket.**accept0[native]** ()	
sun.nio.ch.ServerSocketChannelImpl.**accept0[native]** ()	
sun.management.ThreadImpl.**dumpThreads0[native]** ()	
sun.misc.Unsafe.**unpark[native]** ()	
sun.reflect.Reflection.**getCallerClass[native]** ()	
sun.nio.ch.NativeThread.**current[native]** ()	
sun.nio.ch.FileDispatcherImpl.**write0[native]** ()	
sun.misc.Unsafe.**park[native]** ()	
org.apache.cassandra.locator.DynamicEndpointSnitch.**getDatacenter** ()	
org.apache.cassandra.locator.AbstractNetworkTopologySnitch.**compareEndpoints** ()	
java.lang.Object.**hashCode[native]** ()	
java.lang.Object.**notifyAll[native]** ()	
java.util.concurrent.ConcurrentHashMap.**get** ()	
java.security.AccessController.**doPrivileged[native]** ()	
java.lang.String.**intern[native]** ()	

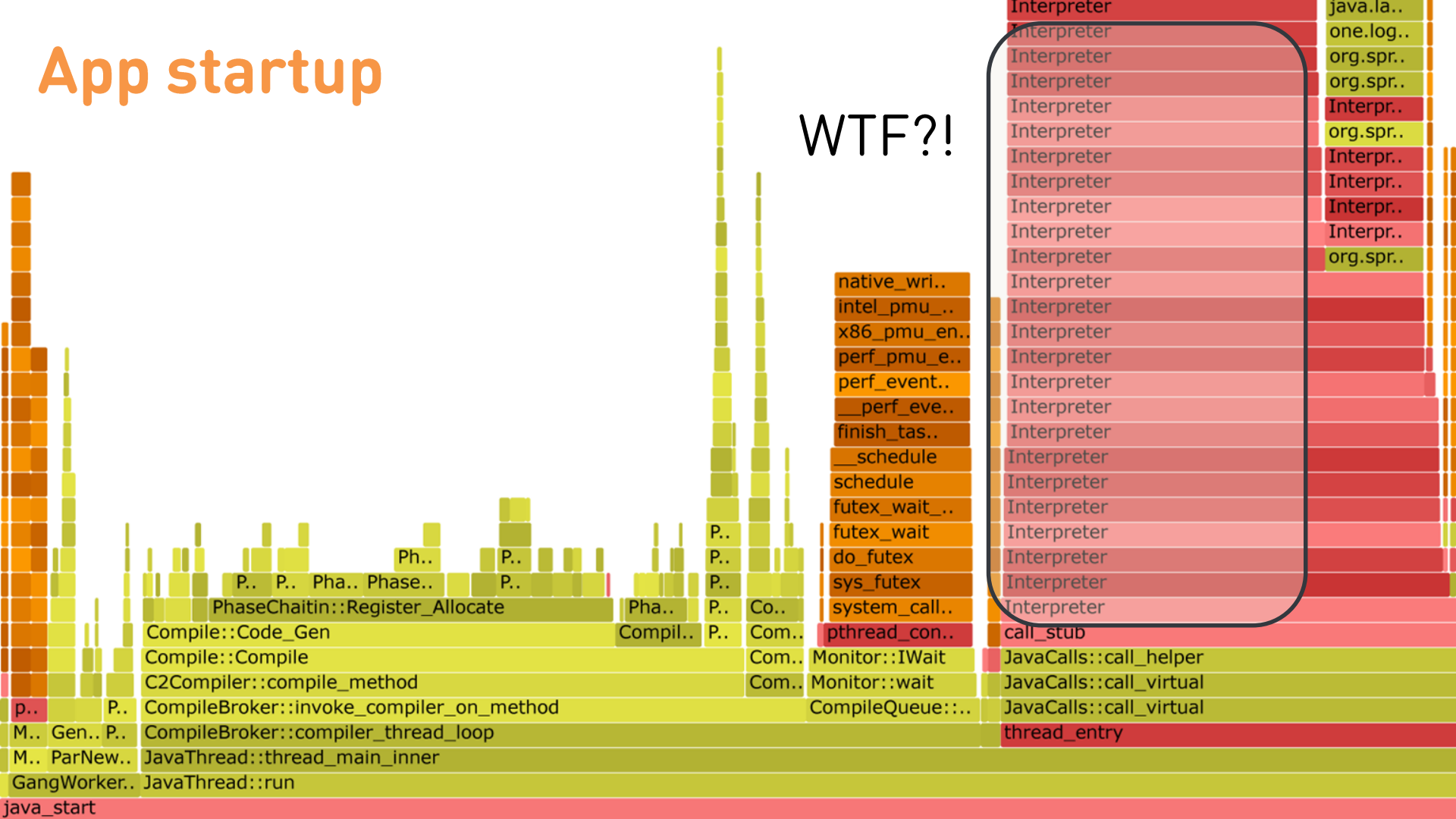
Tree view


Call Tree - Method
Thread-6
org.apache.cassandra.net.IncomingTcpConnection.run ()
org.apache.cassandra.net.IncomingTcpConnection.handleModernVersion ()
java.io.DataInputStream.readInt ()
com.google.common.io.CountingInputStream.read ()
java.io.BufferedInputStream.read ()
java.io.BufferedInputStream.fill ()
sun.nio.ch.ChannelInputStream.read ()
sun.nio.ch.SocketAdaptor$SocketInputStream.read ()
sun.nio.ch.SocketChannelImpl.read ()
sun.nio.ch.IOUtil.read ()
sun.nio.ch.IOUtil.readIntoNativeBuffer ()
sun.nio.ch.SocketDispatcher.read ()
sun.nio.ch.FileDispatcherImpl.read0[native] ()

Demo

App startup

WTF?!



Perf disadvantages

- No interpreted frames
- perf-map-agent for symbols
- -XX:+PreserveFramePointer
- JDK ⩾ 8u60
- /proc/sys/kernel/perf_event_paranoid
- /proc/sys/kernel/perf_event_max_stack = 127
- «Big Data»

Mixed approach

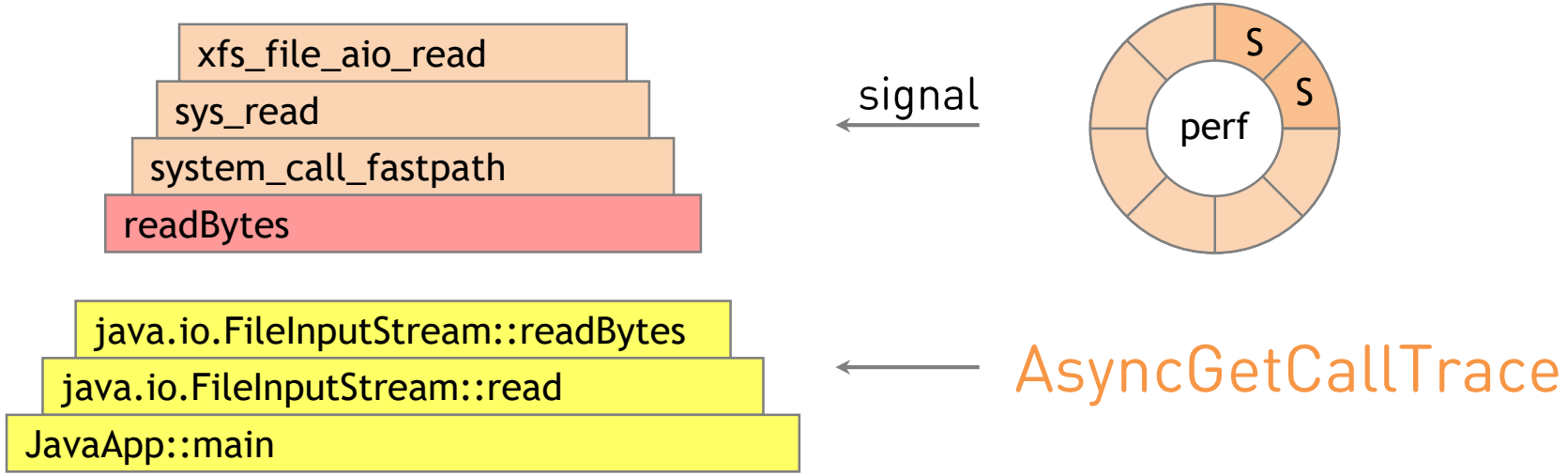
perf_event_open

- Kernel + native stacks
- HW counters

- Entire Java
- Fast and precise

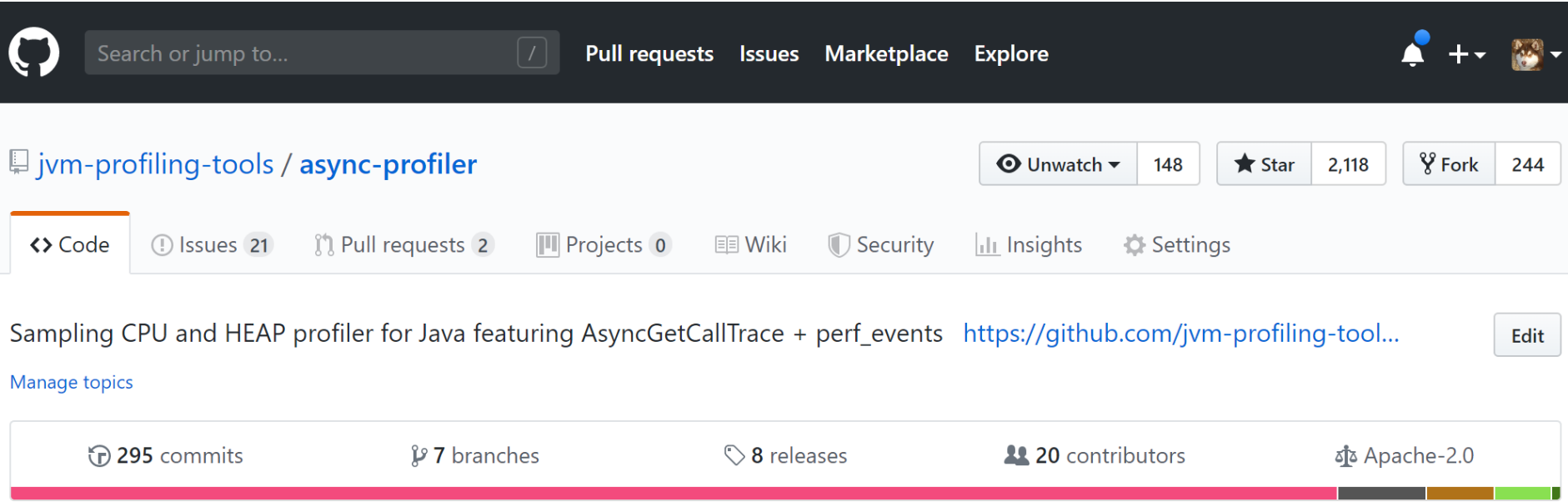
AsyncGetCallTrace

How it works in async-profiler



Introducing async-profiler

github.com/jvm-profiling-tools/async-profiler



github.com/jvm-profiling-tools/async-profiler

Download

Latest release (1.6):

- Linux x64 (glibc): async-profiler-1.6-linux-x64.tar.gz
- Linux x64 (musl): async-profiler-1.6-linux-x64-musl.tar.gz
- Linux ARM: async-profiler-1.6-linux-arm.tar.gz
- macOS x64: async-profiler-1.6-macos-x64.tar.gz

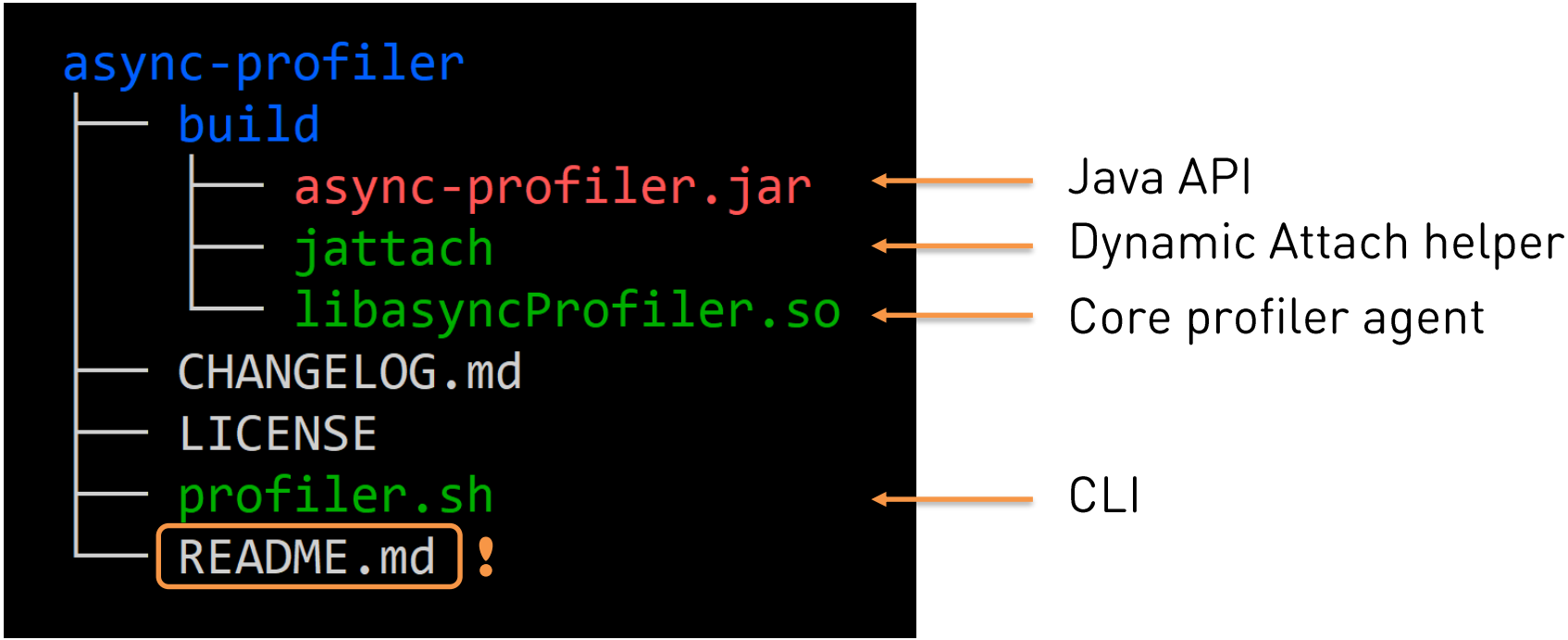
Previous releases

Supported platforms

- **Linux** / x64 / x86 / ARM / AArch64
- **macOS** / x64

Note: macOS profiling is limited to user space code only.

Package contents



Demo

Case study: file reading

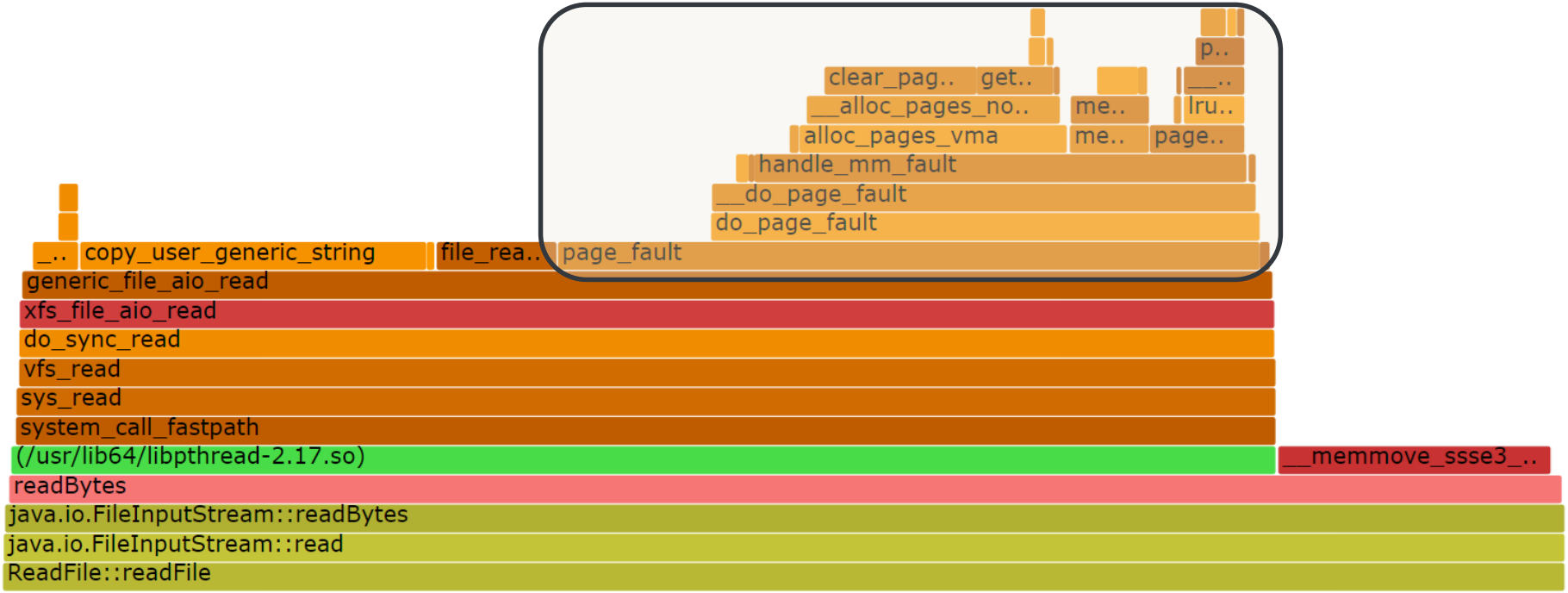
```
byte[] buf = new byte[bufSize];

try (FileInputStream in = new FileInputStream(fileName)) {
    int bytesRead;
    while ((bytesRead = in.read(buf)) > 0) {
        ...
    }
}
```

Buffer size?

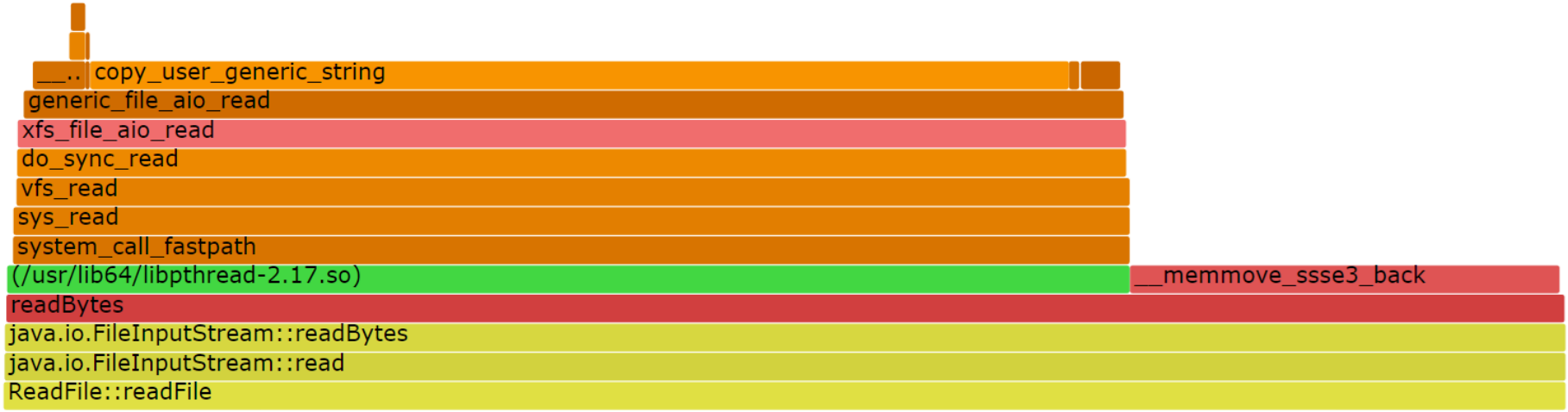
- ❑ 64 K
- ❑ 1 M
- ❑ 4 M
- ❑ 16 M
- ❑ 32 M

Down to Linux Kernel



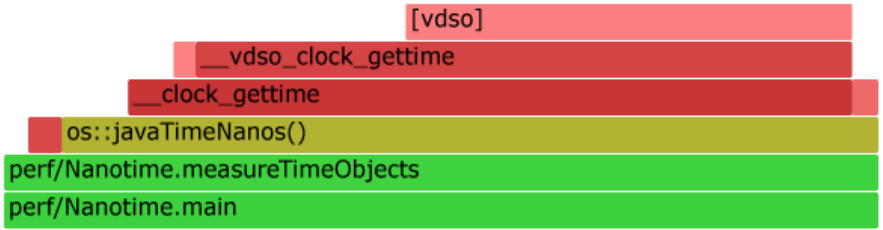
Down to Linux Kernel

Buffer size: 32M => 31M

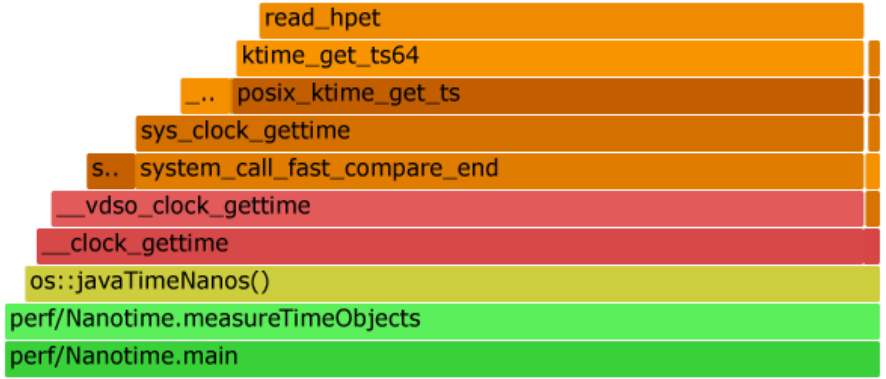


Case study: nanoTime

System.nanoTime() latency: 40 ns → 10 000 ns



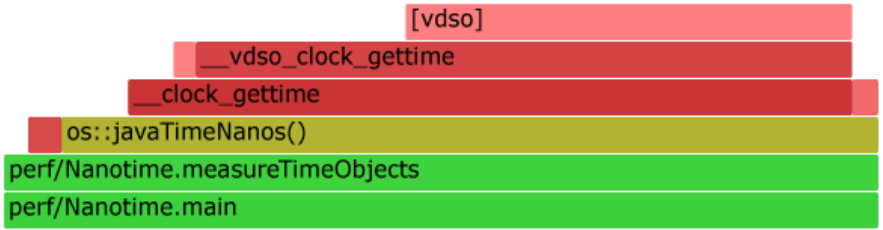
"good" nanoTime



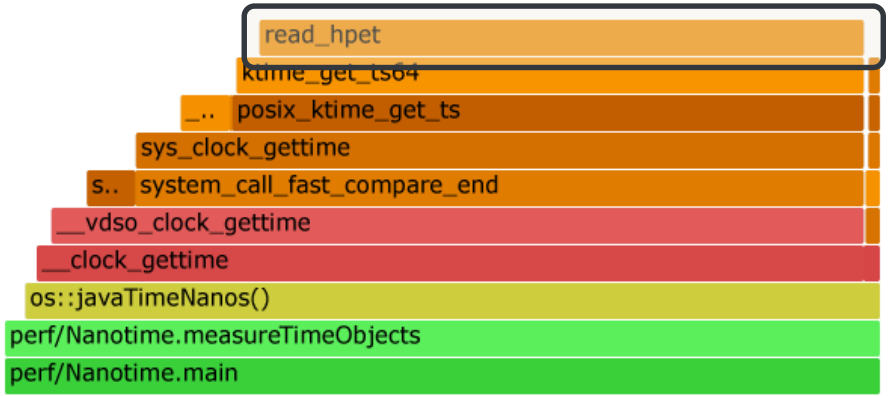
"bad" nanoTime

Case study: nanoTime

System.nanoTime() latency: 40 ns → 10 000 ns



"good" nanoTime



"bad" nanoTime

```
[ 0.065868] TSC synchronization [CPU#0 -> CPU#1]:
[ 0.065870] Measured 679995254538 cycles TSC warp between CPUs, turning off TSC clock.
[ 0.065874] tsc: Marking TSC unstable due to check_tsc_sync_source failed
```

Replying to @AndreiPangin

I'm happy with ✸Profiler ... no need for another tool

9:37 PM - 18 May 2018




	JFR	perf	async-profiler
Java stack	Yes	No interpreted	Yes
Native stack	No	Yes	Yes
Kernel stack	No	Yes	Yes
JDK support	8*, 11+	8u60+	6+
OS support	All	Linux only	Linux, macOS*
Permanent overhead	0	2-5%	0
System-wide profiling	No	Possible	No

Wall clock profiling

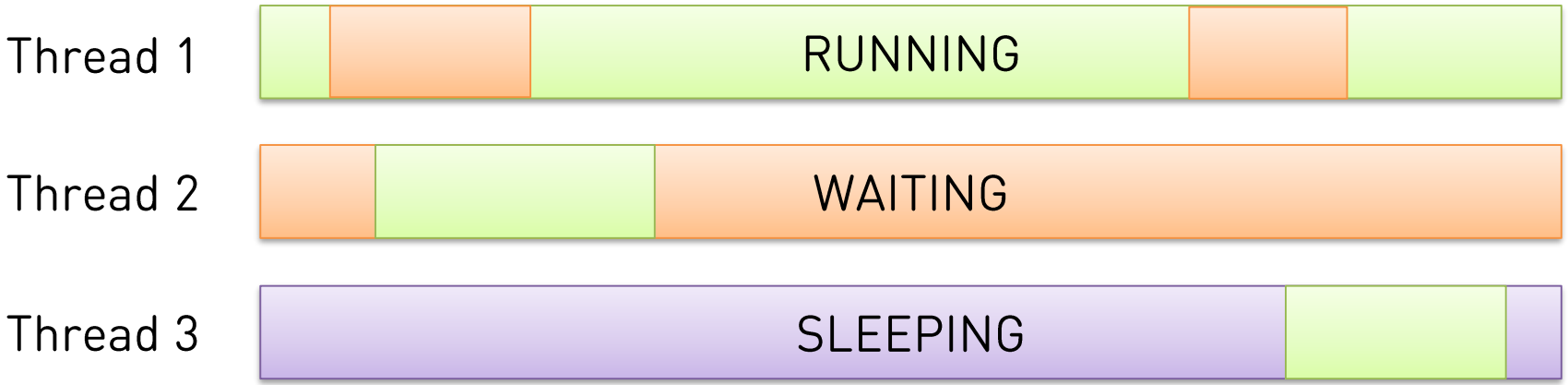
Waste time doing nothing

- Thread.sleep
- Object.wait / Condition.await
- Wait to acquire a lock / semaphore / etc.
- Wait for I/O
  - Socket read
  - DB query
  - Disk I/O

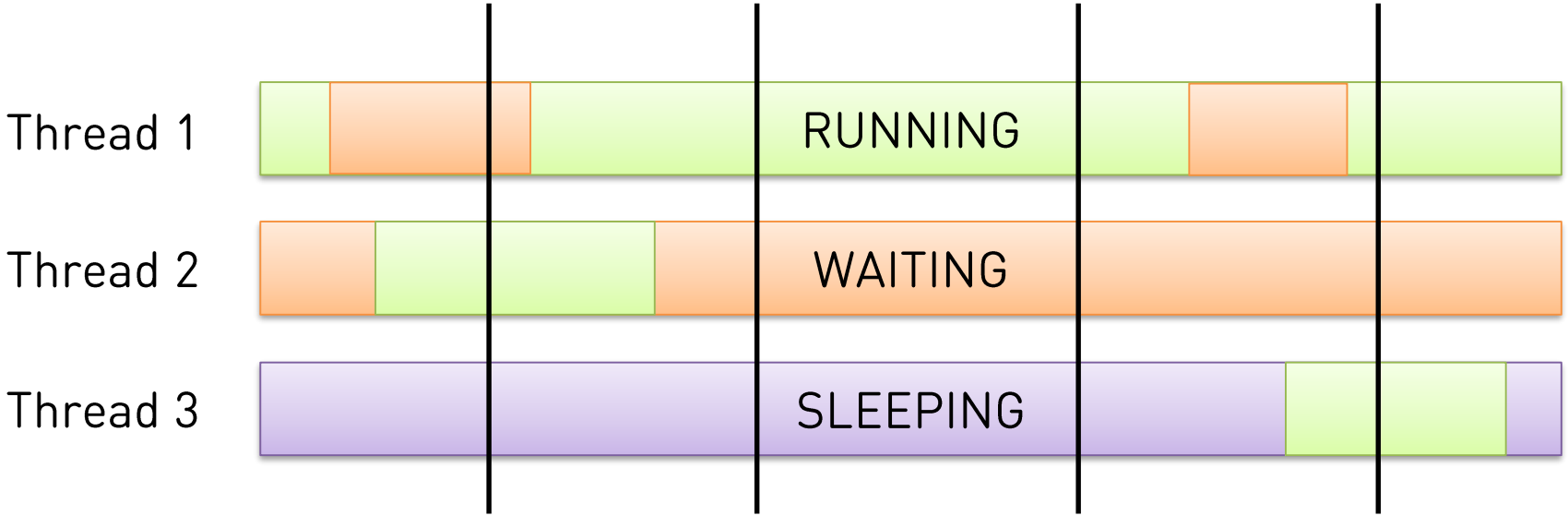
Waste time doing nothing

- Thread.sleep
- Object.wait / Condition.await
- Wait to acquire a lock / semaphore / etc.
- Wait for I/O
  - Socket read
  - DB query
  - Disk I/O

Wall clock profiling



Wall clock profiling



Wall clock mode in async-profiler

CLI option	Agent option
-e wall	event=wall
-t	threads
-i 1ms	interval=1ms

Lock contention


```
synchronized (obj) {
```

Profiler option
-e lock
-o svg=total

Lock contention

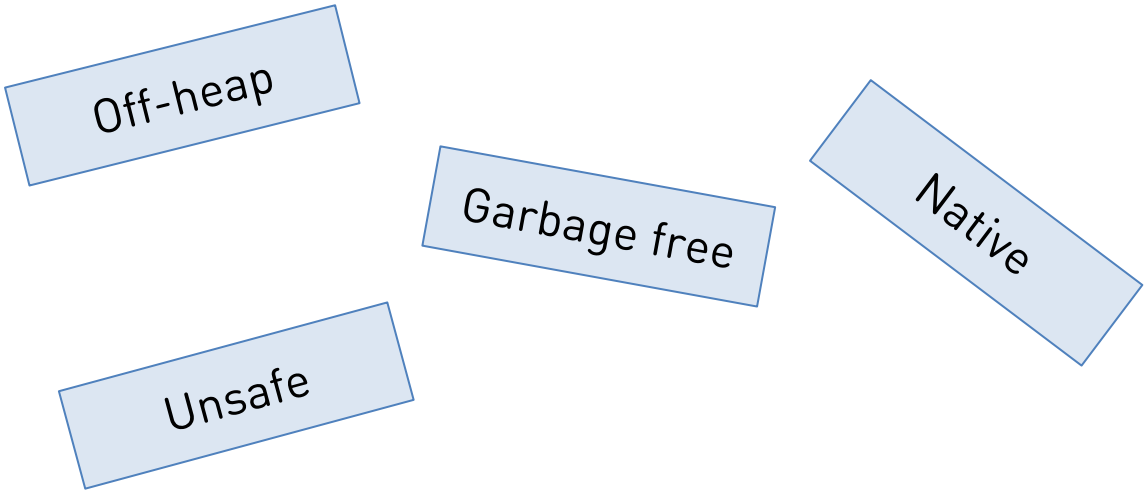
`ReentrantLock.lock()`

Profiler option
-e lock
-o svg=total

Demo

Allocation profiling

«new is the root of all evil»



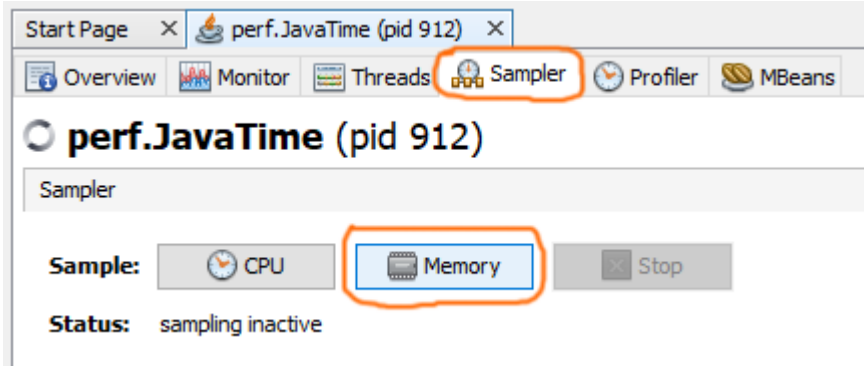
We want full-featured Java!

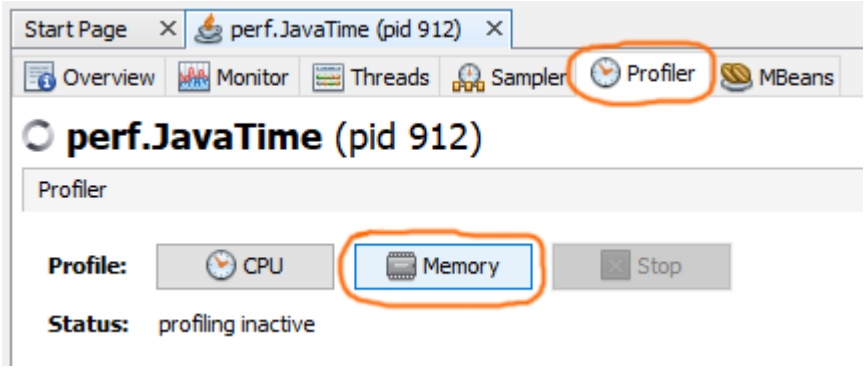
How many objects created?

```
LocalDateTime deadline =
        LocalDateTime.now().plusSeconds(10);

while (LocalDateTime.now().isBefore(deadline)) {
    iterations++;
}
```

Profiling with VisualVM





Sampler

- Walk through heap
- Collect class histogram

Class Name	Bytes [%]	Bytes	Instances
java.util.concurrent.**ConcurrentHashMap**		15,441,856 (20.7%)	241,279 (12.2%)
sun.util.calendar.**ZoneInfo**		13,510,112 (18.1%)	241,252 (12.2%)
java.time.zone.**ZoneRules**		9,649,880 (12.9%)	241,247 (12.2%)
java.time.**LocalTime**		5,790,672 (7.7%)	241,278 (12.2%)
java.time.**LocalDateTime**		5,790,120 (7.7%)	241,255 (12.2%)
java.time.**LocalDate**		5,790,120 (7.7%)	241,255 (12.2%)
java.time.**ZoneOffset[]**		5,789,928 (7.7%)	241,247 (12.2%)
java.time.**ZoneRegion**		5,789,928 (7.7%)	241,247 (12.2%)

Profiler

- Bytecode instrumentation

```
→ Tracer.recordAllocation(ArrayList.class, 24);
  List<String> list = new ArrayList<>();
```

- getStackTrace() each Nth sample

DTrace / SystemTap

- -XX:+DTraceAllocProbes
- Slow path allocations

```
$ stap -e '
    probe hotspot.object_alloc { log(probestr) }
'
```

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/dtrace.html
https://epickrram.blogspot.ru/2017/09/heap-allocation-flamegraphs.html

Other tools

- Aprof
  https://code.devexperts.com/display/AProf

- Allocation Instrumenter
  https://github.com/google/allocation-instrumenter

Overhead

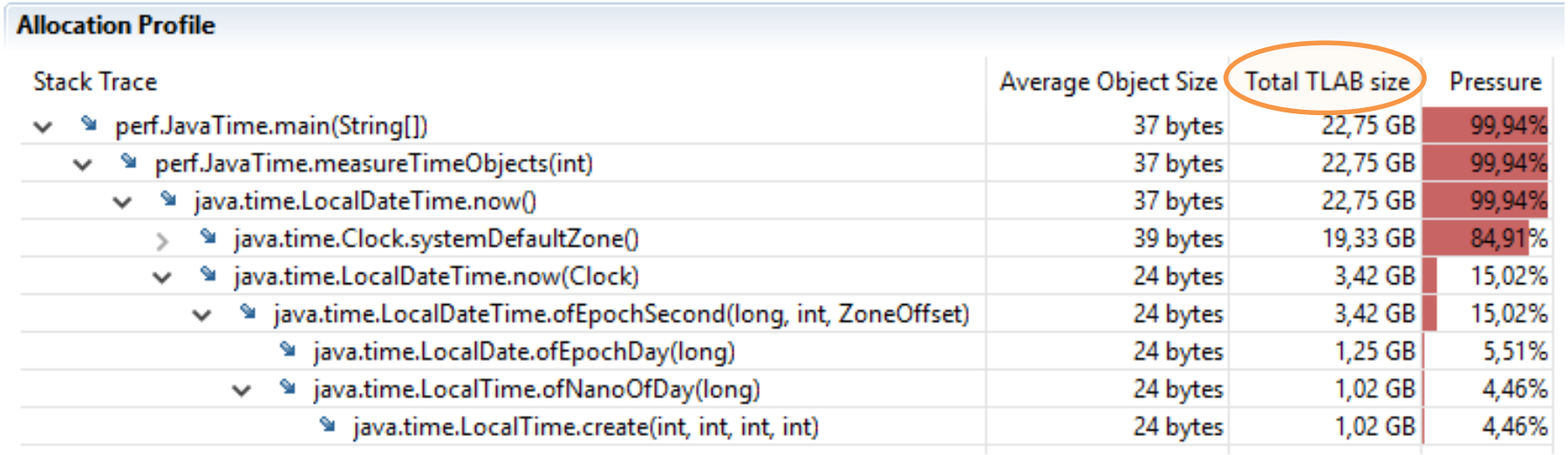
Profiler		x10^6 ops	Slowdown
No profiling		33,2	
VisualVM Sampler	1G	28,2	-15%
	2G	25,6	-23%
	4G	18,1	-45%
VisualVM Profiler		6,8	-80%
JProfiler		2,8	-92%
DTraceAllocProbes		13,3	-60%
Aprof		18,9	-43%
Allocation Instrumenter		22,8	-31%

JMC / Flight Recorder

Allocation Profile

Stack Trace	Average Object Size	Total TLAB size	Pressure
perf.JavaTime.main(String[])	37 bytes	22,75 GB	99,94%
perf.JavaTime.measureTimeObjects(int)	37 bytes	22,75 GB	99,94%
java.time.LocalDateTime.now()	37 bytes	22,75 GB	99,94%
java.time.Clock.systemDefaultZone()	39 bytes	19,33 GB	84,91%
java.time.LocalDateTime.now(Clock)	24 bytes	3,42 GB	15,02%
java.time.LocalDateTime.ofEpochSecond(long, int, ZoneOffset)	24 bytes	3,42 GB	15,02%
java.time.LocalDate.ofEpochDay(long)	24 bytes	1,25 GB	5,51%
java.time.LocalTime.ofNanoOfDay(long)	24 bytes	1,02 GB	4,46%
java.time.LocalTime.create(int, int, int, int)	24 bytes	1,02 GB	4,46%

JMC / Flight Recorder

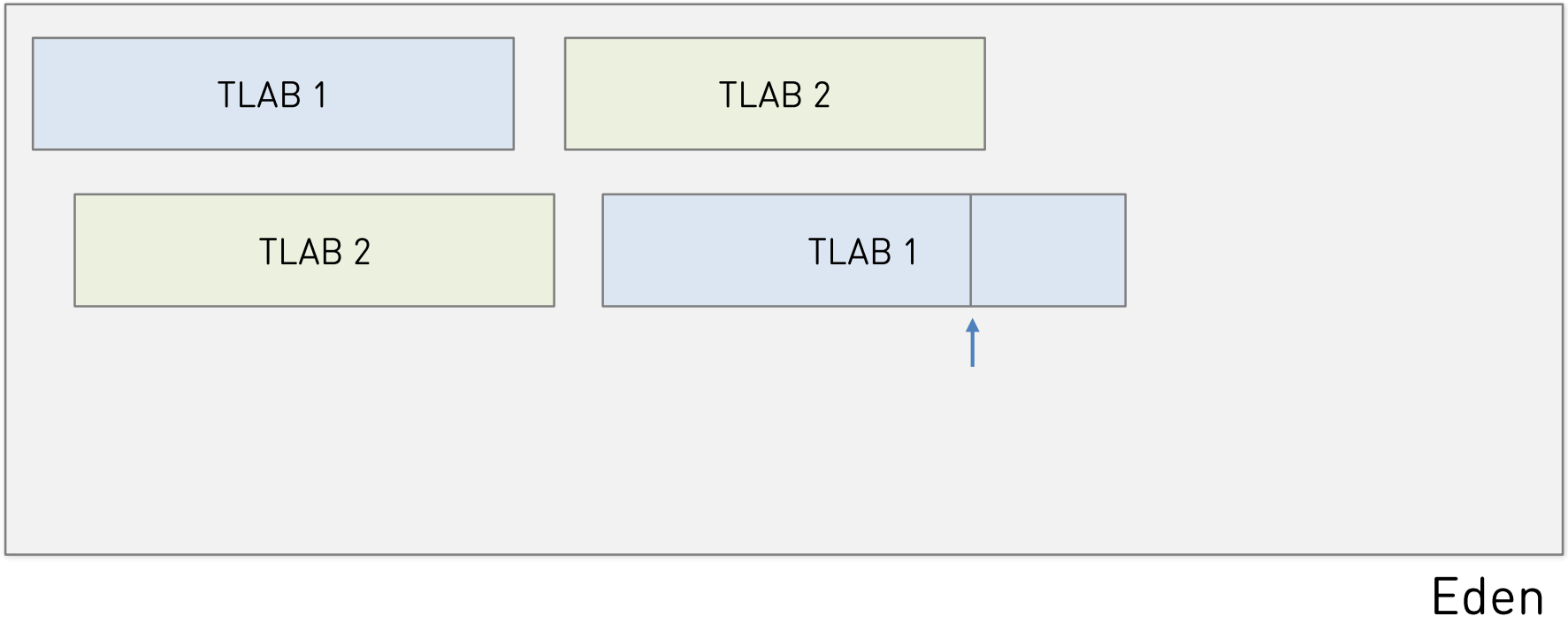


Allocation Profile

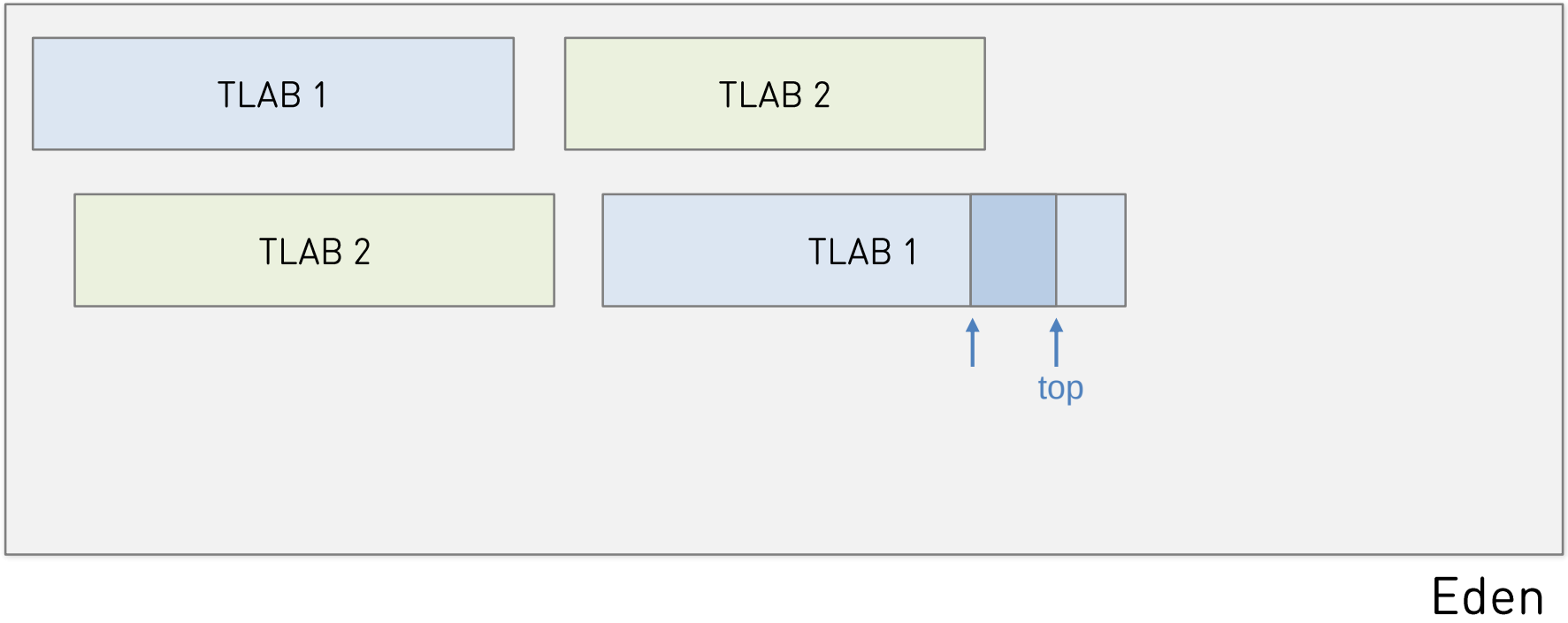
Stack Trace	Average Object Size	Total TLAB size	Pressure
perf.JavaTime.main(String[])	37 bytes	22,75 GB	99,94%
perf.JavaTime.measureTimeObjects(int)	37 bytes	22,75 GB	99,94%
java.time.LocalDateTime.now()	37 bytes	22,75 GB	99,94%
java.time.Clock.systemDefaultZone()	39 bytes	19,33 GB	84,91%
java.time.LocalDateTime.now(Clock)	24 bytes	3,42 GB	15,02%
java.time.LocalDateTime.ofEpochSecond(long, int, ZoneOffset)	24 bytes	3,42 GB	15,02%
java.time.LocalDate.ofEpochDay(long)	24 bytes	1,25 GB	5,51%
java.time.LocalTime.ofNanoOfDay(long)	24 bytes	1,02 GB	4,46%
java.time.LocalTime.create(int, int, int, int)	24 bytes	1,02 GB	4,46%

- 😃 Open source since JDK 11
- 😃 Overhead < 5%

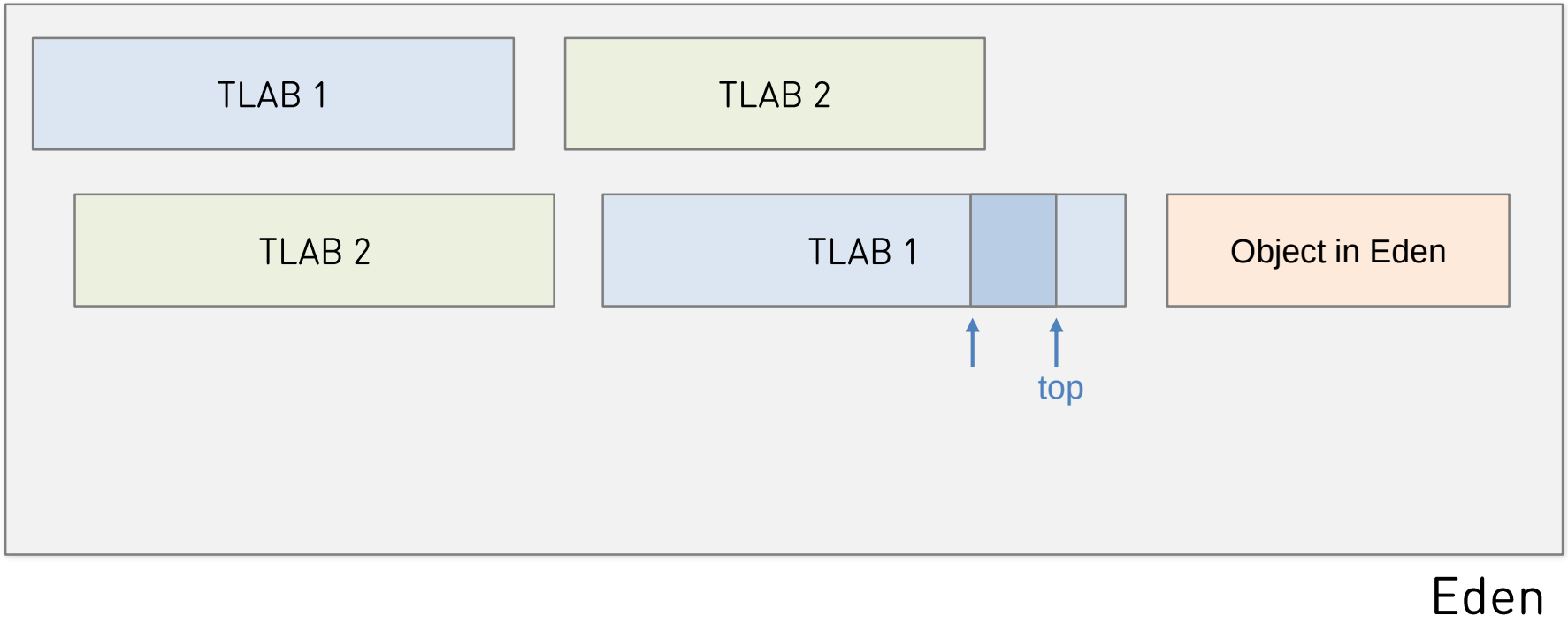
Thread Local Allocation Buffer



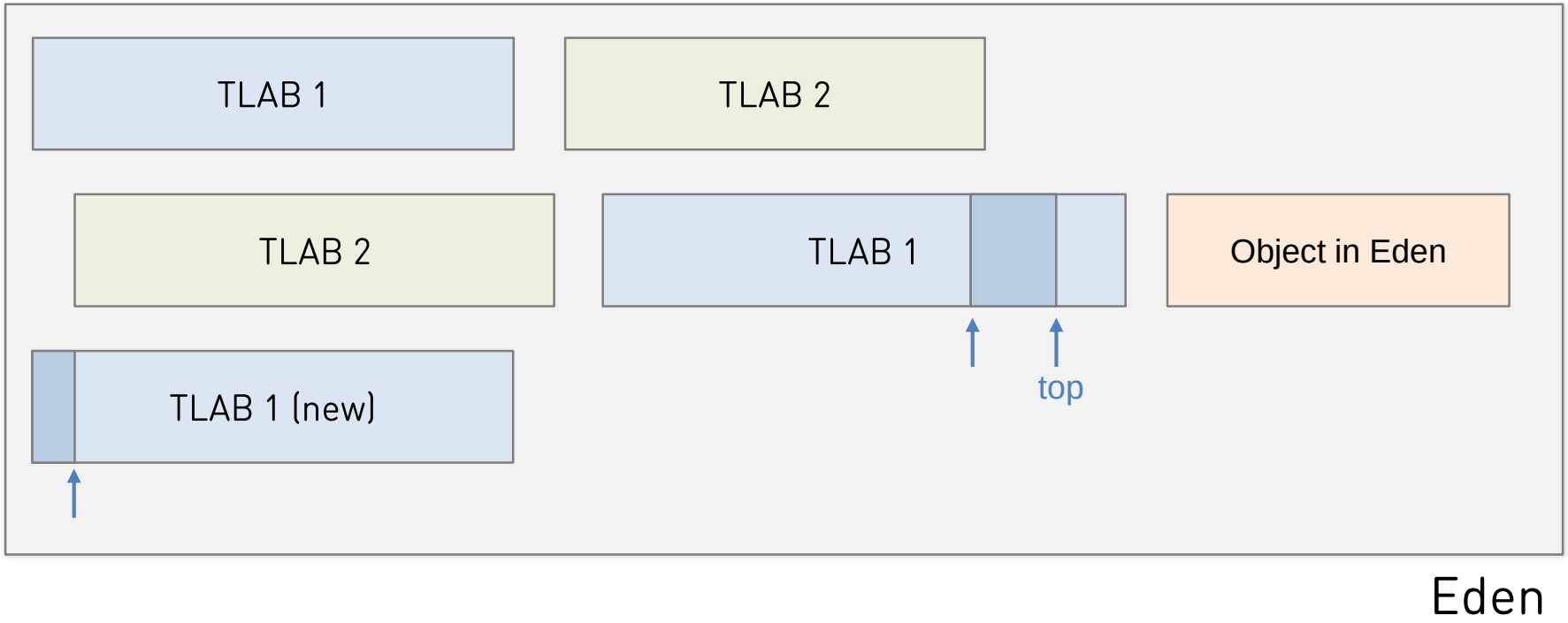
Thread Local Allocation Buffer



Thread Local Allocation Buffer



Thread Local Allocation Buffer



Allocation path

- Fast: inlined
  - Inside TLAB
- Slow: call to VM Runtime
  - Outside TLAB
  - Allocation of new TLAB

Profile

Allocation sampler in async-profiler

1. Intercept slow allocations in JVM runtime
2. Record allocated object + stack trace

   ...
3. PROFIT!

Allocation sampler in async-profiler

1. Intercept slow allocations in JVM runtime
2. Record allocated object + stack trace

   ...
3. PROFIT!

😀 Works with OpenJDK 7u40+

😀 Does not rely on commercial features

Appeared in JDK 11

- JEP 331: Low-Overhead Heap Profiling

```
SampledObjectAlloc(jvmtiEnv *jvmti_env,
                   JNIEnv* jni_env,
                   jthread thread,
                   jobject object,
                   jclass object_klass,
                   jlong size)

SetHeapSamplingInterval(jint sampling_interval)
```

Demo

Contribute to async-profiler

- Try it out
- Submit issues
- Update documentation
- Send pull requests

Thank you

@AndreiPangin
https://pangin.pro