ALIBABA CLOUD
INTELLIGENCE GROUP

Observability beyond the three pillars

Continuous Profiling with Alibaba Dragonwell

Sanhong Li | Java Champion, JVM Architect | Alibaba Cloud

Content

01 Basics of Observability

02 Fundamentals of JFR

03 Continuous Profiling with Alibaba Dragonwell

04 Case Studies

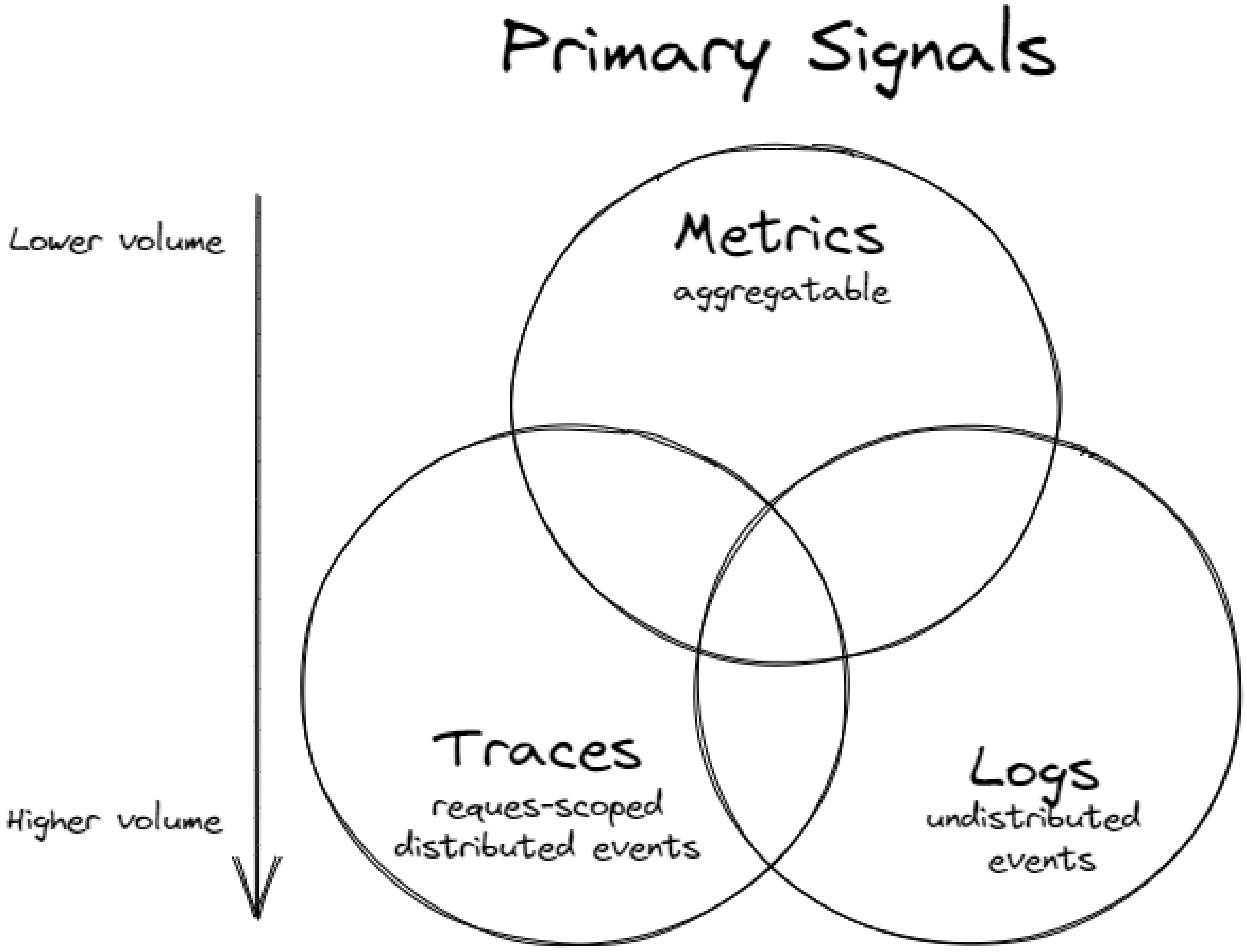
05 Wrap up

Observability to Software Systems

- “observability” was originally coined in 1960, used to describe mathematical control systems.
- Applied to modern software systems
  - **Understand** the inner workings of your application
  - **Understand** any system state your application may have gotten
  - **Understand** the inner workings and system state **solely** by observing and interrogating with external tools
  - **Understand** the internal state without shipping any new custom code to handle it

Observability Engineering, Charity Majors, Liz Fong-Jones, George Miranda 2022

(Traditional) Three Observability Pillars



Metrics

numeric representation of data measured over intervals of time(e.g via Java MXBean API)

Traces

representation of a series of causally related distributed events(e.g via JVMTi agent)

Logs

immutable, timestamped record of discrete events that happened over time.

(e.g gc log is a typical example)

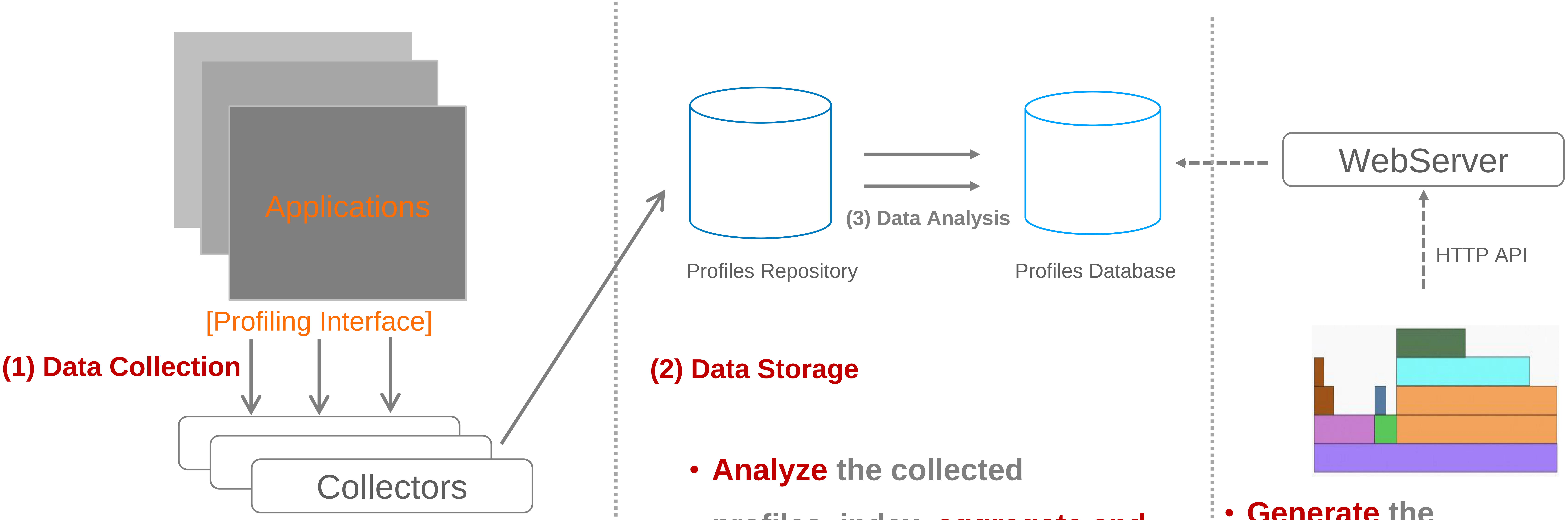
https://github.com/cncf/tag-observability/blob/main/whitepaper.md#observability-signals

Continuous Profiling: 4th aspect of Observability

- Google pioneered the continuous profiling concept in its own data centers
  - “Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers”, research paper published by Google in 2010
- Continuous profiling (constantly monitors an application's performance in real time)
  - Executing in a production environment(no need to develop accurate predictive load tests or benchmarks for the production)
  - Sampling(low overhead)

https://storage.googleapis.com/pub-tools-public-publication-data/pdf/36575.pdf

Continuous Profiling: Conceptual Architecture



- **Selects** a random sample of applications for profiling

- **Analyze** the collected profiles, index**, aggregate and store** them into K/V database for query

- **Generate** the performance dashboard
  - CPU
  - Memory
  - ……

Content

ALIBABA CLOUD INTELLIGENCE GROUP

01 Basics of Observability

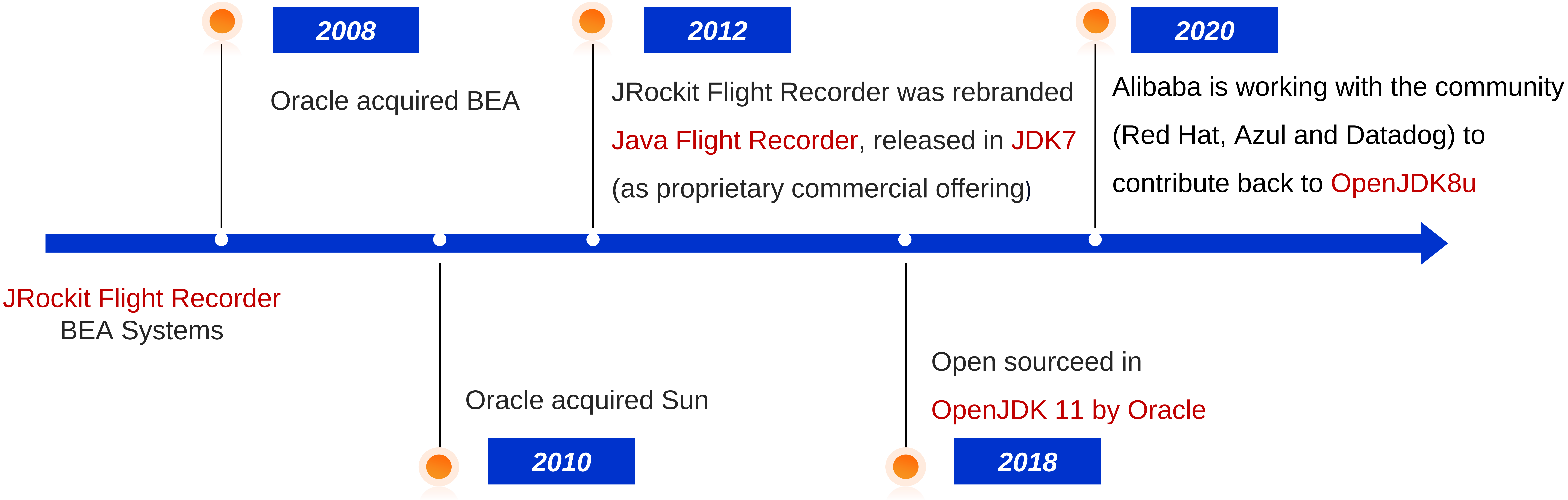
02 Fundamentals of JFR

03 Continuous Profiling with Alibaba Dragonwell

04 Case Studies

05 Wrap up

Java Flight Recorder(JFR) History



JRockit Flight Recorder
BEA Systems

2008

Oracle acquired BEA

2010

Oracle acquired Sun

2012

JRockit Flight Recorder was rebranded Java Flight Recorder, released in JDK7 (as proprietary commercial offering)

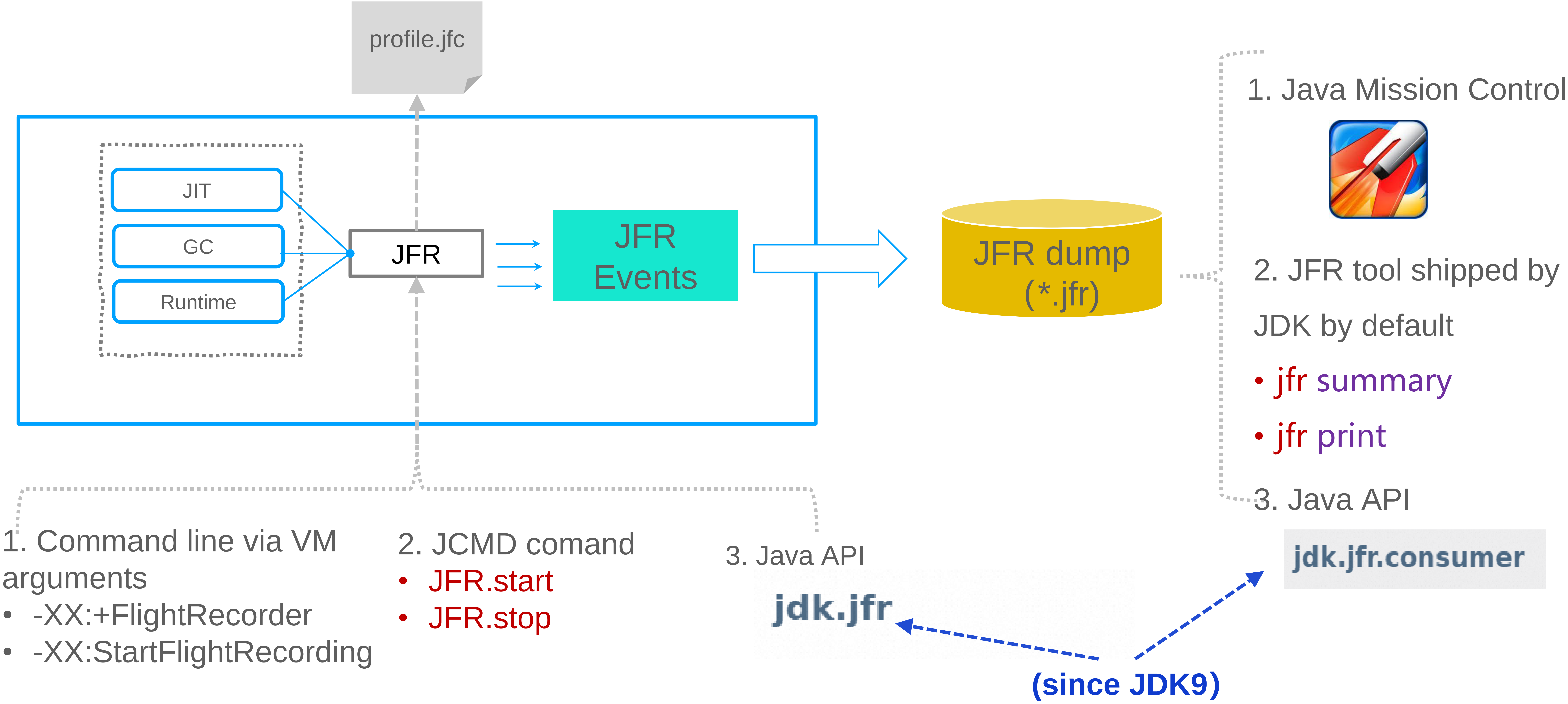
2018

Open sourceed in OpenJDK 11 by Oracle

2020

Alibaba is working with the community (Red Hat, Azul and Datadog) to contribute back to OpenJDK8u

JFR Workflow Overview



1. Command line via VM arguments
- -XX:+FlightRecorder
- -XX:StartFlightRecording

2. JCMD comand
- JFR.start
- JFR.stop

3. Java API

jdk.jfr

1. Java Mission Control

2. JFR tool shipped by JDK by default
- jfr summary
- jfr print

3. Java API

jdk.jfr.consumer

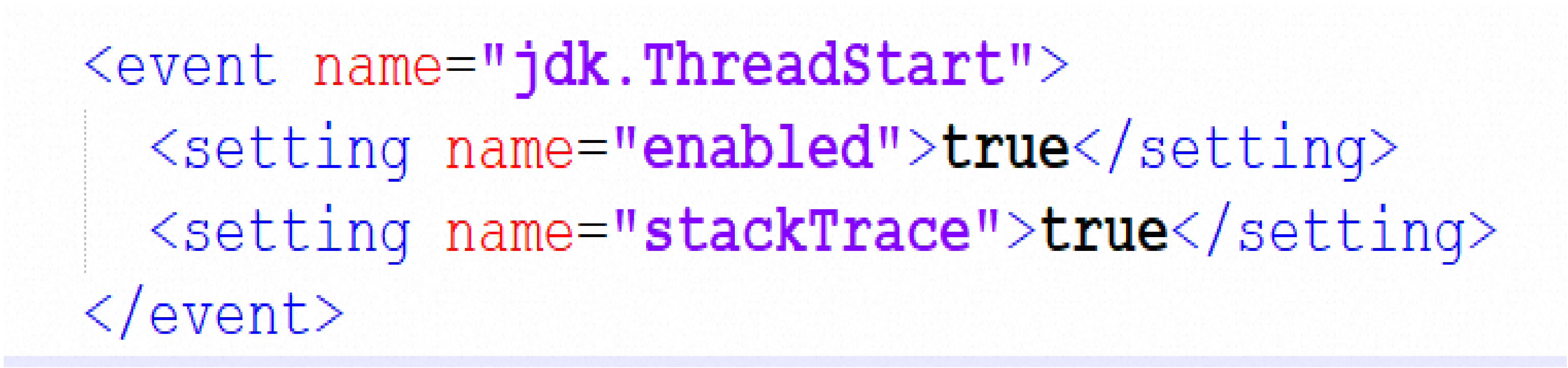
(since JDK9)

JFR Events

Examples from $JAVA_HOME/lib/jfr/default.jfc

Instant Event

- occur immediately



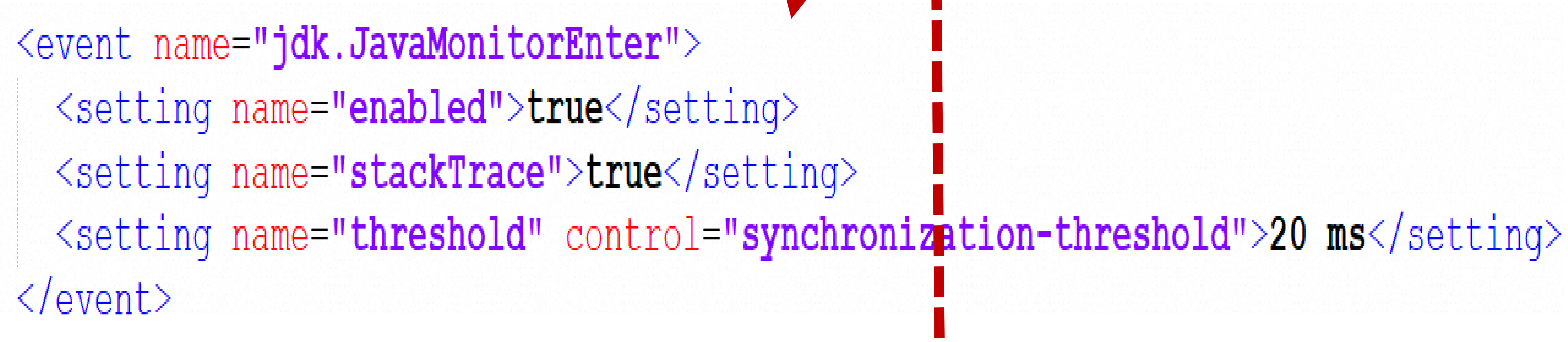
```
<event name="jdk.ThreadStart">
  <setting name="enabled">true</setting>
  <setting name="stackTrace">true</setting>
</event>
```

Duration Event

- have a start and end

Timed Event

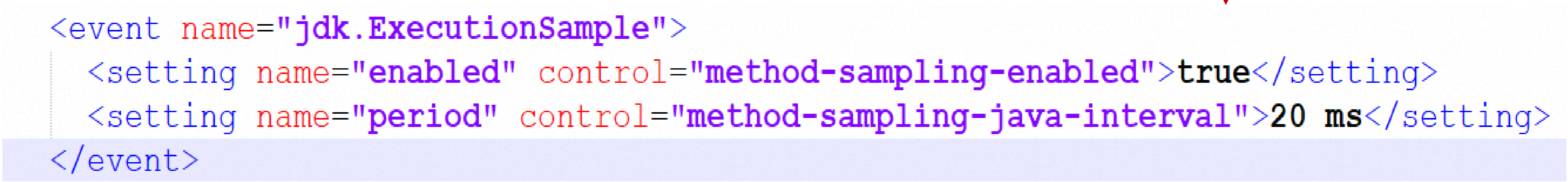
- Like Duration Event, but with threshold set



```
<event name="jdk.JavaMonitorEnter">
  <setting name="enabled">true</setting>
  <setting name="stackTrace">true</setting>
  <setting name="threshold" control="synchronization-threshold">20 ms</setting>
</event>
```

Sample Events

- logged at a regular interval



```
<event name="jdk.ExecutionSample">
  <setting name="enabled" control="method-sampling-enabled">true</setting>
  <setting name="period" control="method-sampling-java-interval">20 ms</setting>
</event>
```

Content

01 Basics of Observability

02 Fundamentals of JFR

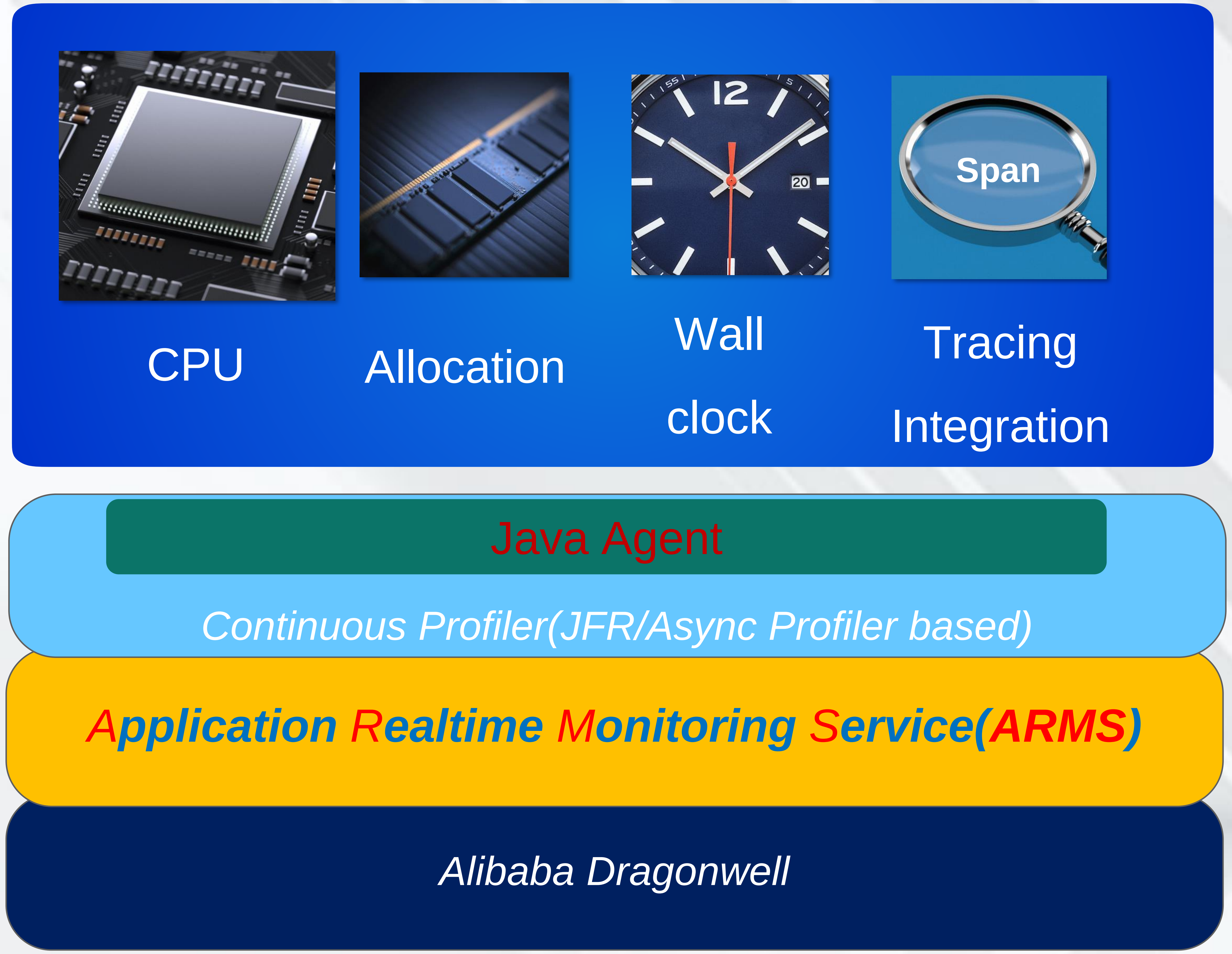
03 Continuous Profiling with Alibaba Dragonwell

04 Case Studies

05 Wrap up

ARMS Continuous Profiler - Introduction

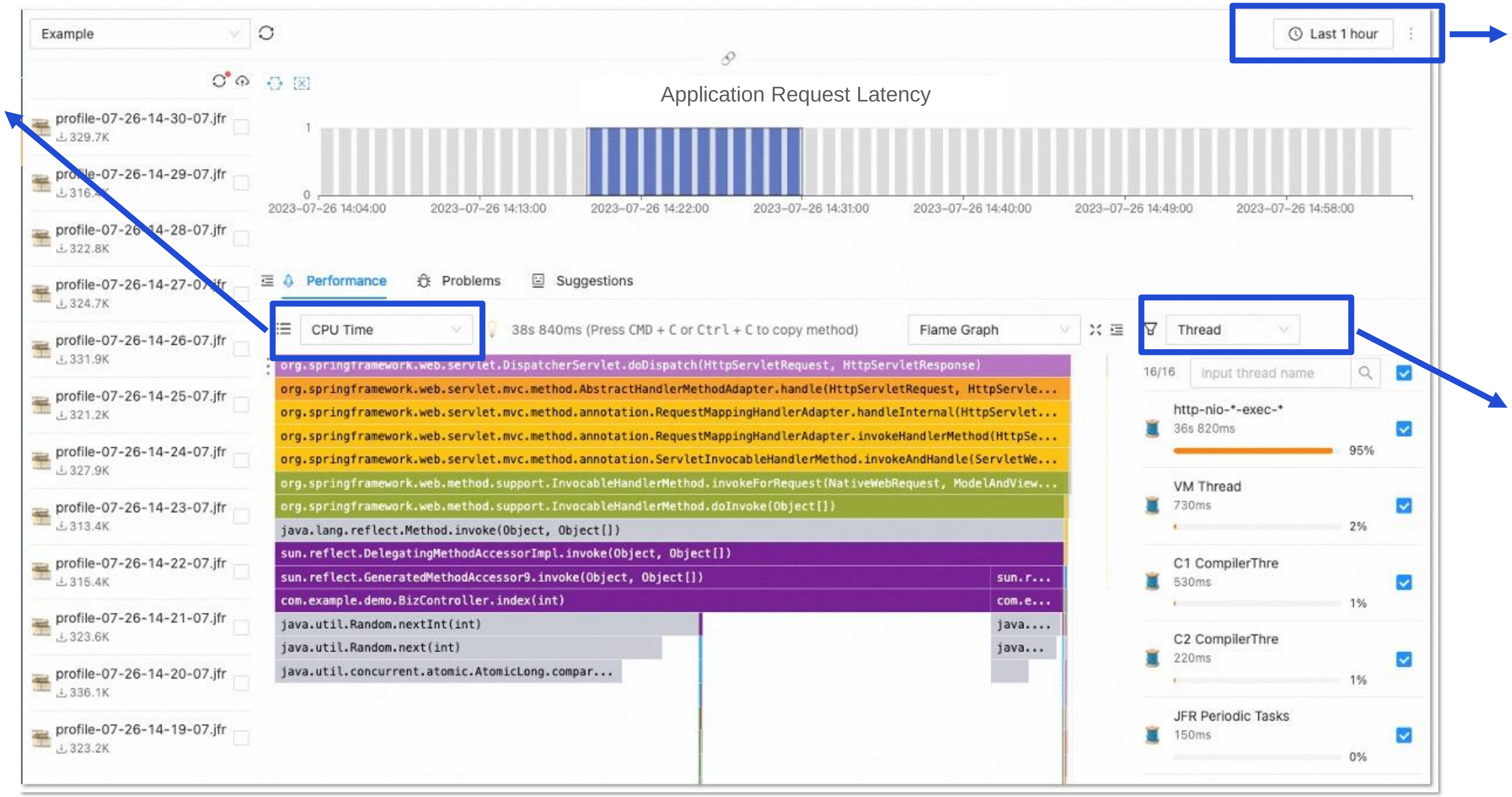
- ❑ Alibaba Dragonwell – downstream of OpenJDK, free LTS by Alibaba Cloud
- ❑ ARMS - Application Real-Time Monitoring Service (ARMS) is an application performance management (APM) service, written in Java, running on top of Alibaba Dragonwell
- ❑ Continuous Profiler, based on JFR and async-profiler technology, part of ARMS
  - ✓ Collecting profiling data via Java agent
  - ✓ Major features
    - ✓ CPU/Allocation/Wall clock profiling
    - ✓ Integration with Tracing


Span
CPU
Allocation
Wall clock
Tracing Integration
Java Agent
Continuous Profiler(JFR/Async Profiler based)
Application Realtime Monitoring Service(ARMS)
Alibaba Dragonwell

https://www.alibabacloud.com/product/arms

ARMS Continuous Profiler – Overview

- CPU
- Allocation
- Wall Clock
- Lock
- File IO
- Socket IO
- Direct Memory
- Exception
- Thread Activity



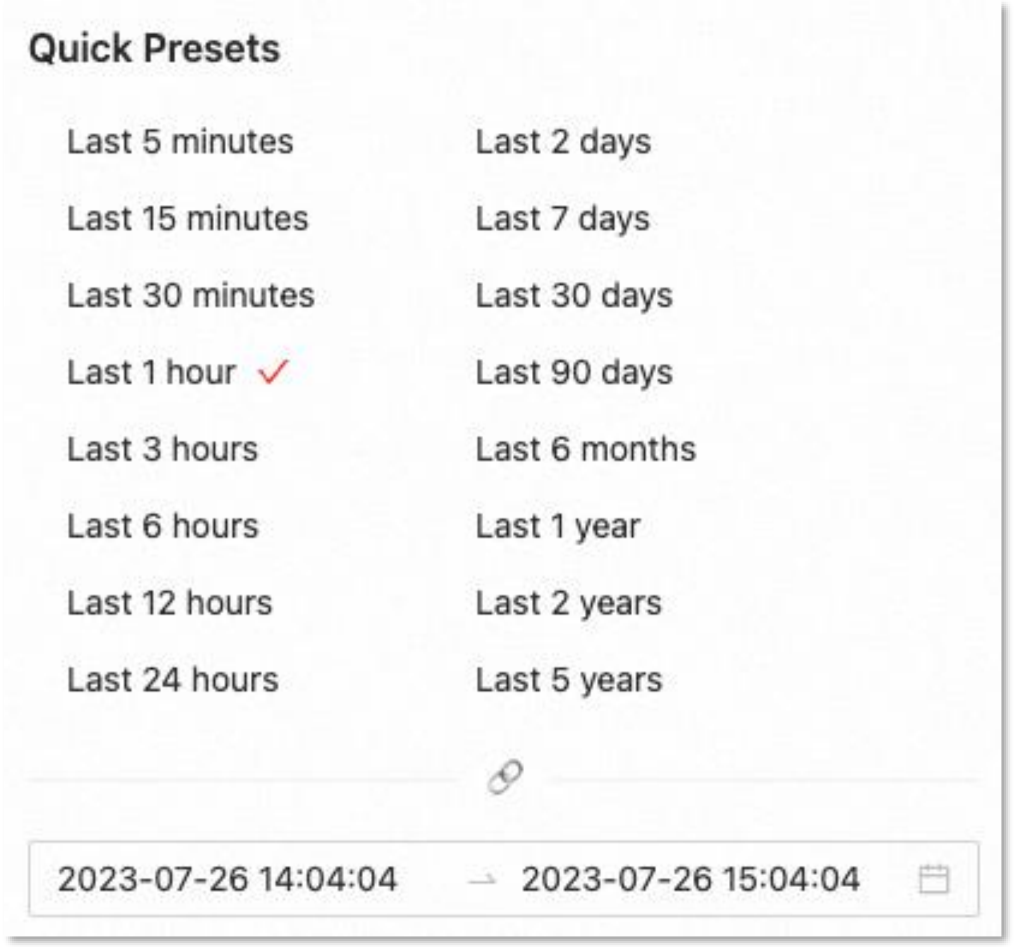
profile-07-26-14-30-07.jfr	329.7K
profile-07-26-14-29-07.jfr	316.4K
profile-07-26-14-28-07.jfr	322.8K
profile-07-26-14-27-07.jfr	324.7K
profile-07-26-14-26-07.jfr	331.9K
profile-07-26-14-25-07.jfr	321.2K
profile-07-26-14-24-07.jfr	327.9K
profile-07-26-14-23-07.jfr	313.4K
profile-07-26-14-22-07.jfr	315.4K
profile-07-26-14-21-07.jfr	323.6K
profile-07-26-14-20-07.jfr	336.1K
profile-07-26-14-19-07.jfr	323.2K

38s 840ms (Press CMD + C or Ctrl + C to copy method)

```
org.springframework.web.servlet.DispatcherServlet.doDispatch(HttpServletRequest, HttpServletResponse)
org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(HttpServletRequest, HttpServle...
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(HttpServlet...
org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(HttpSe...
org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletWe...
org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(NativeWebRequest, ModelAndView...
org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(Object[])
java.lang.reflect.Method.invoke(Object, Object[])
sun.reflect.DelegatingMethodAccessorImpl.invoke(Object, Object[])
sun.reflect.GeneratedMethodAccessor9.invoke(Object, Object[])                sun.r...
com.example.demo.BizController.index(int)                                    com.e...
java.util.Random.nextInt(int)                                                java....
java.util.Random.next(int)                                                   java...
java.util.concurrent.atomic.AtomicLong.compar...
```

16/16 input thread name ☑

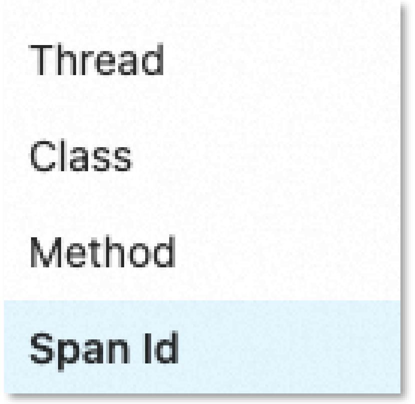
http-nio-*-exec-*	36s 820ms	95%	☑
VM Thread	730ms	2%	☑
C1 CompilerThre	530ms	1%	☑
C2 CompilerThre	220ms	1%	☑
JFR Periodic Tasks	150ms	0%	☑



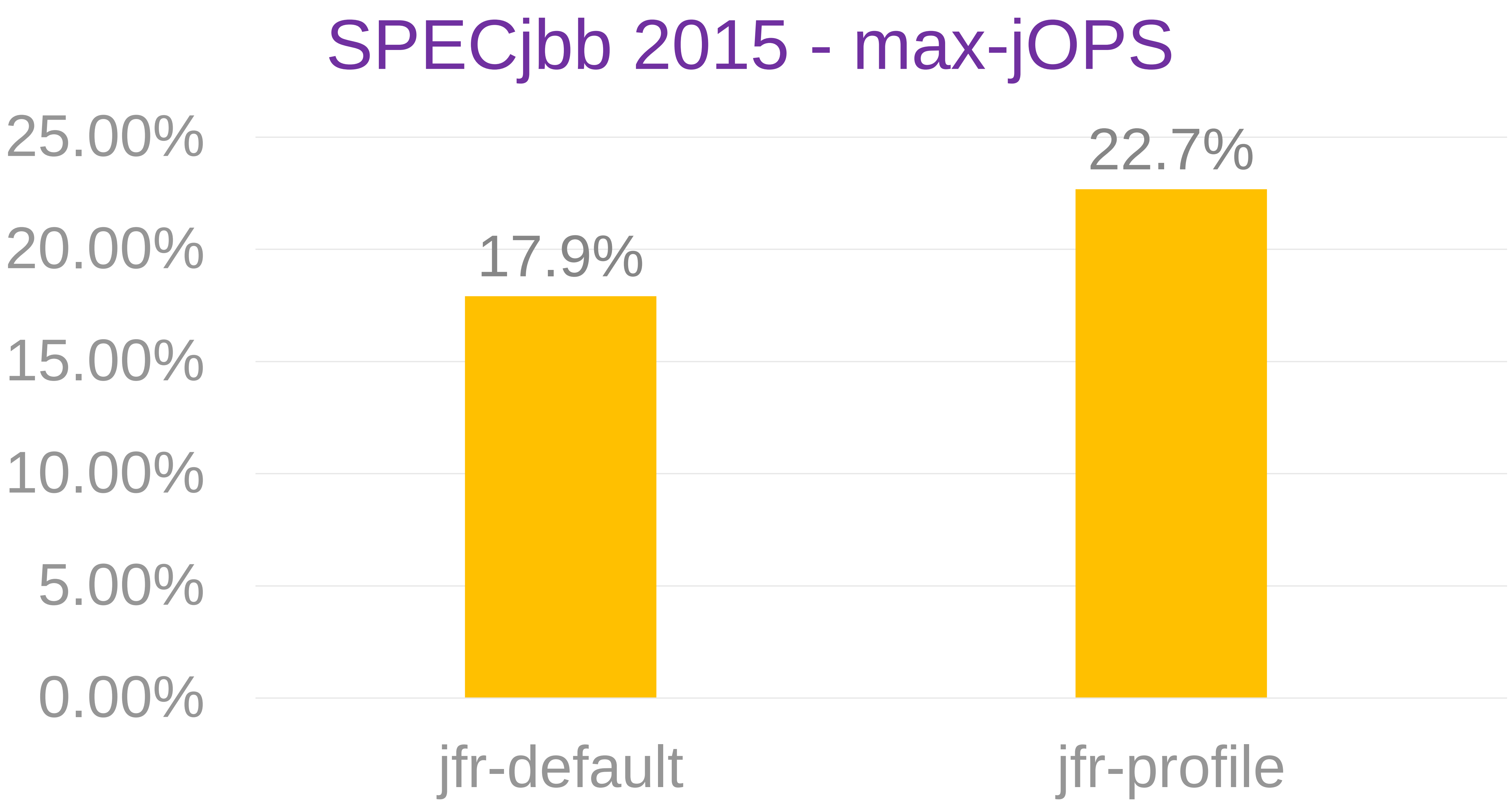
Quick Presets

- Last 5 minutes
- Last 15 minutes
- Last 30 minutes
- Last 1 hour ✓
- Last 3 hours
- Last 6 hours
- Last 12 hours
- Last 24 hours
- Last 2 days
- Last 7 days
- Last 30 days
- Last 90 days
- Last 6 months
- Last 1 year
- Last 2 years
- Last 5 years

2023-07-26 14:04:04 → 2023-07-26 15:04:04

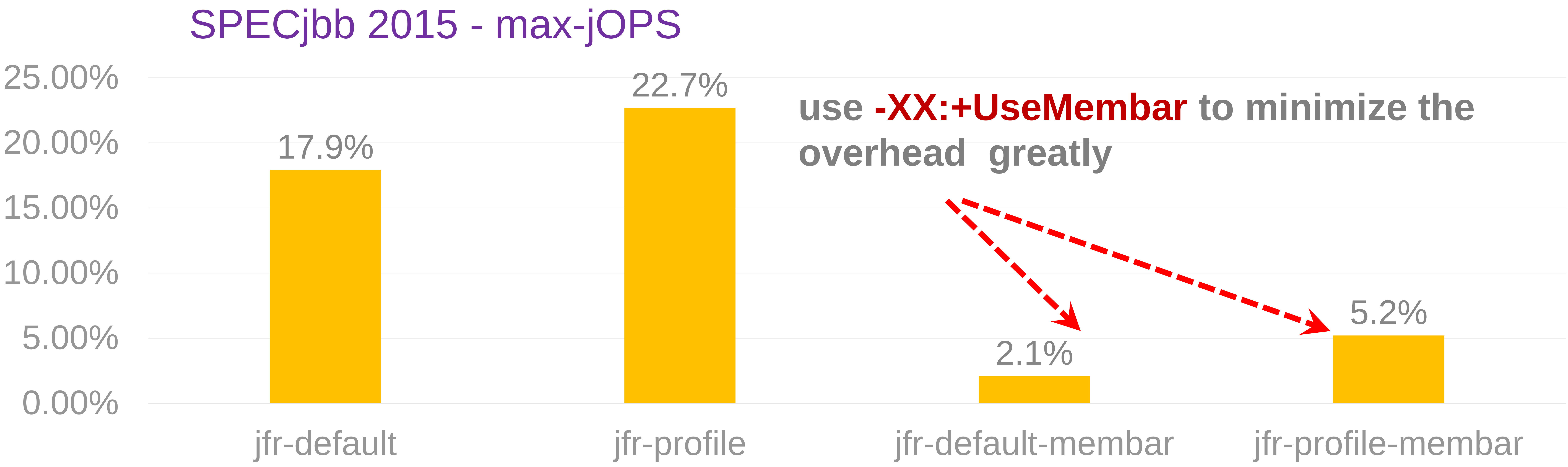


JFR Overhead Assessment



- OS: Linux version 3.10.0-1160.80.1.el7.x86_64
- CPU: vCore 24, x86_64, Intel(R) Xeon(R) Platinum 8369B CPU @ 2.70GHz
- Memory: 24G
- JDK: OpenJDK 1.8.0_362 64-Bit Server VM (build 25.362-b08, mixed mode)
- Flags: -XX:+UseConcMarkSweepGC -Xmx10g -Xmn5g -XX:MaxMetaspaceSize=512m -Xloggc:gc.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps

JFR Overhead Assessment



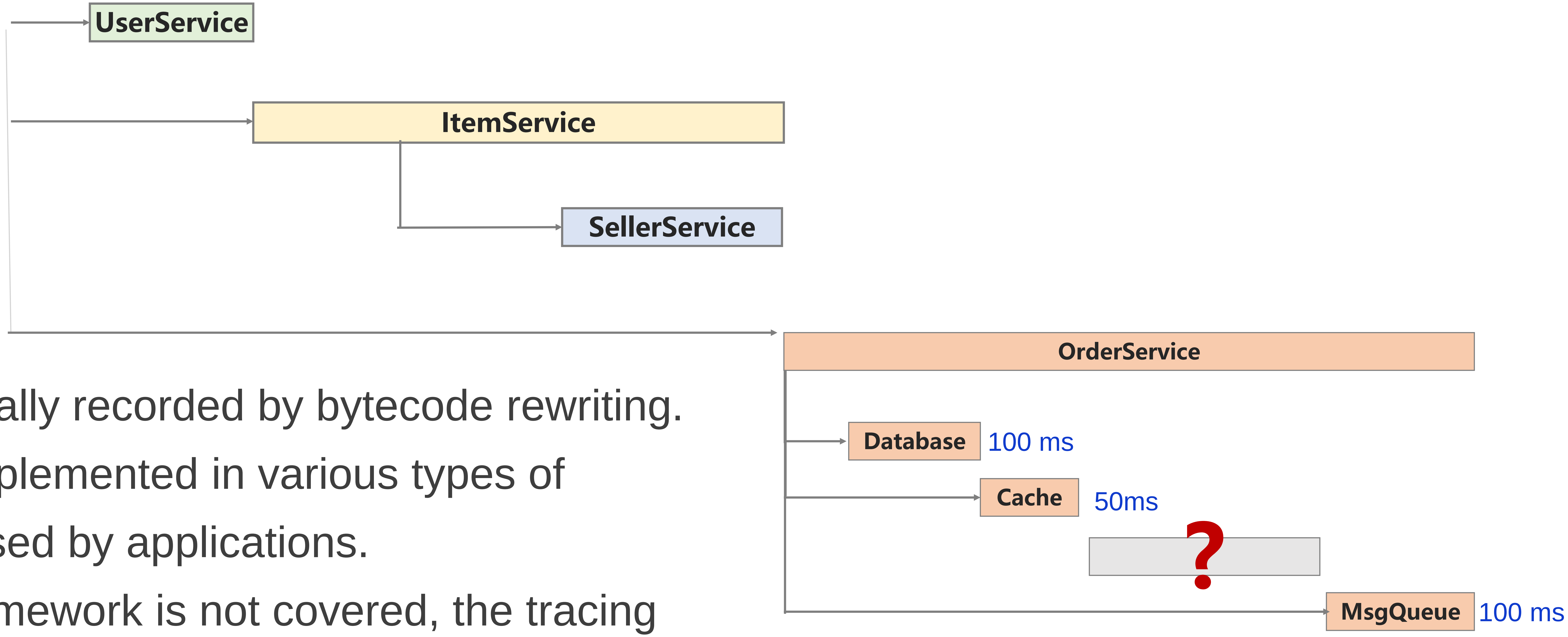
use **-XX:+UseMembar** to minimize the overhead greatly

- The cost was introduced by the call to os::serialize_thread_states (which uses mprotect to force a pseudo-memory barrier) by JfrThreadSampler thread. Notes: this sync mechanism has the performance issue in scalability!
- **-XX:+UseMembar** uses a direct memory fence operation, which is more cheap to get the state of java thread
- More detail info:
  - https://bugs.openjdk.org/browse/JDK-8187812
  - https://bugs.openjdk.org/browse/JDK-8276309

Contextualized JFR – Corelating with Tracing

ALIBABA CLOUD INTELLIGENCE GROUP

Tracing

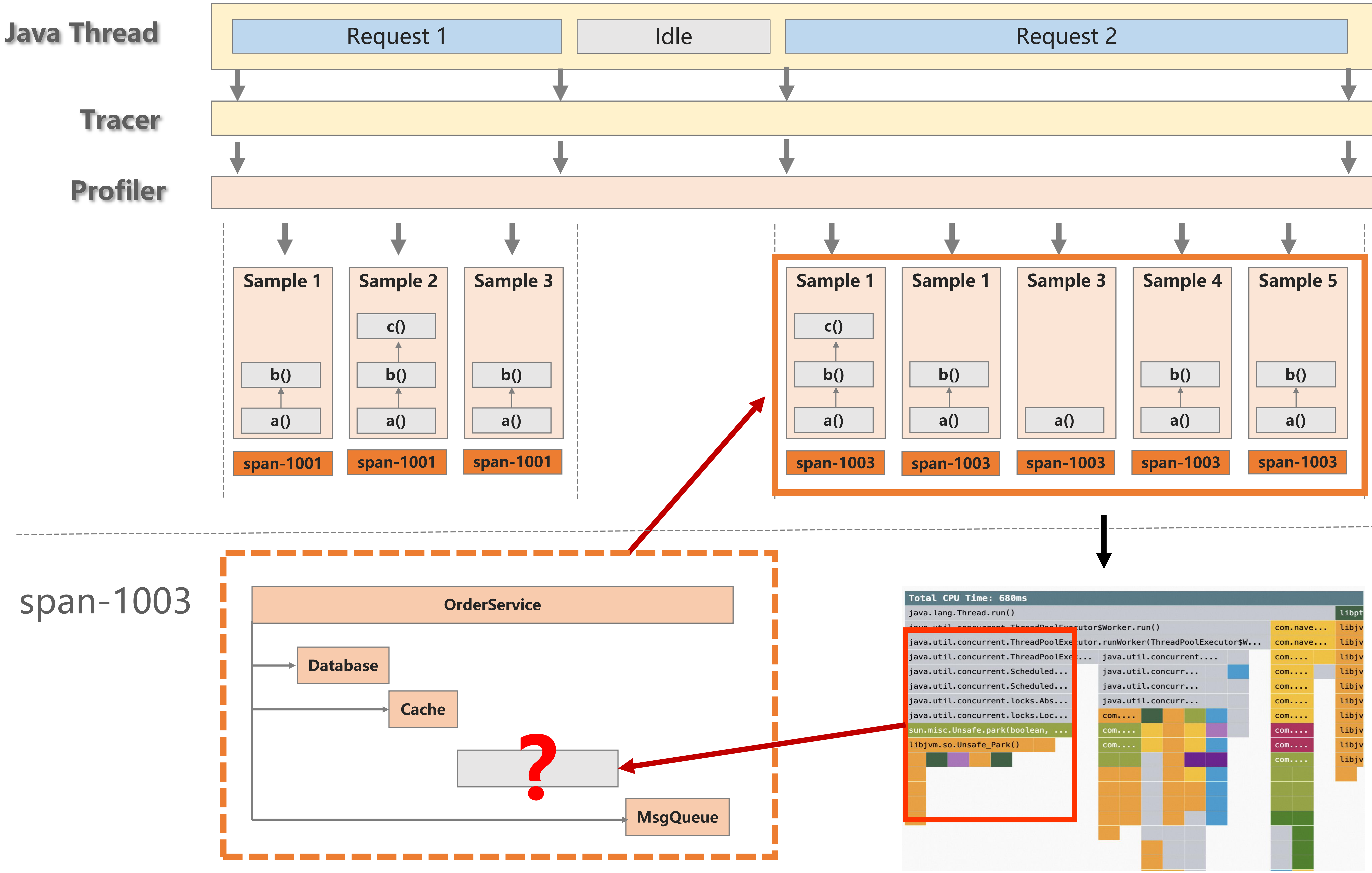


- Tracing is usually recorded by bytecode rewriting.
- Need to be implemented in various types of frameworks used by applications.
- If a certain framework is not covered, the tracing information will be missed, resulting in monitoring blind spots.

- The challenge is: even if we have collected JFR events at those blind spots, we don’t know the their relationship - JFR currently lacks the ability to associate context to JFR events.

Contextualized JFR – Corelating with Tracing(2)

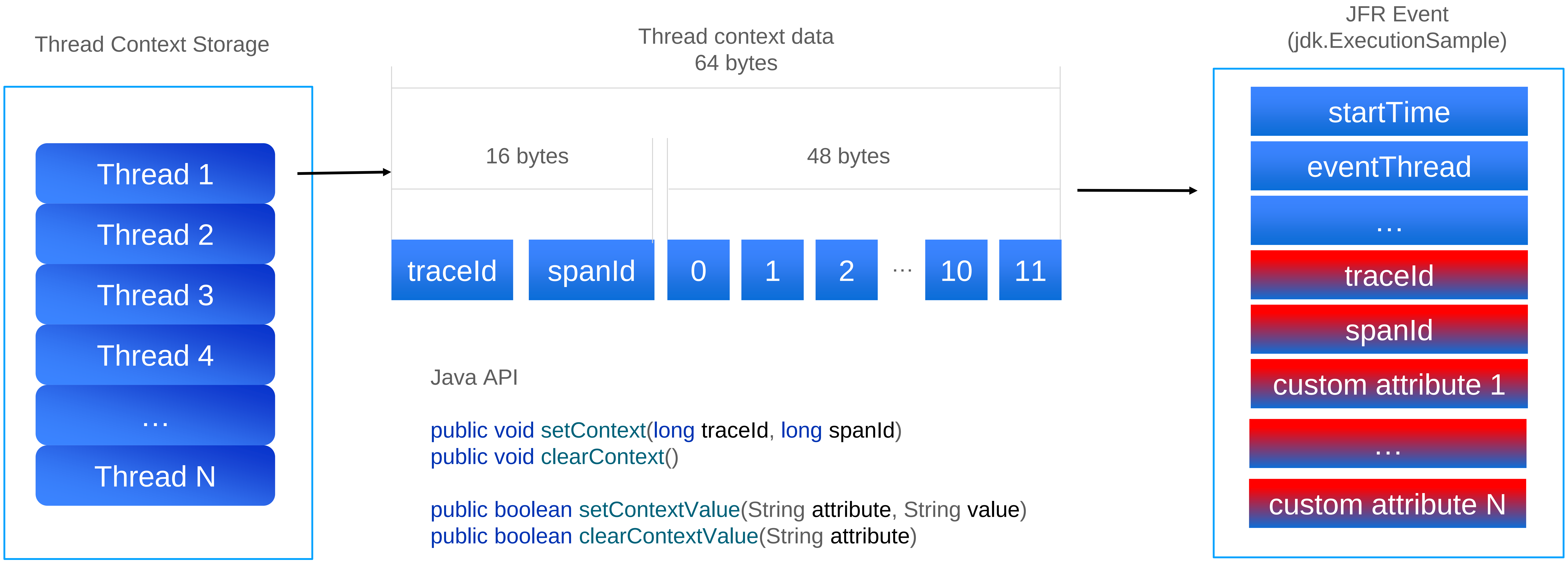
ALIBABA CLOUD
INTELLIGENCE GROUP



<span style="color:#c00000">Trace</span> is composed of one or more <span style="color:#c00000">spans</span>. Span in the trace represents one microservice in the execution path.

Contextualized JFR – Corelating with Tracing(3)

- An context implementation example(based on ideas of aync-profiler & Datadog implementation)



Java API

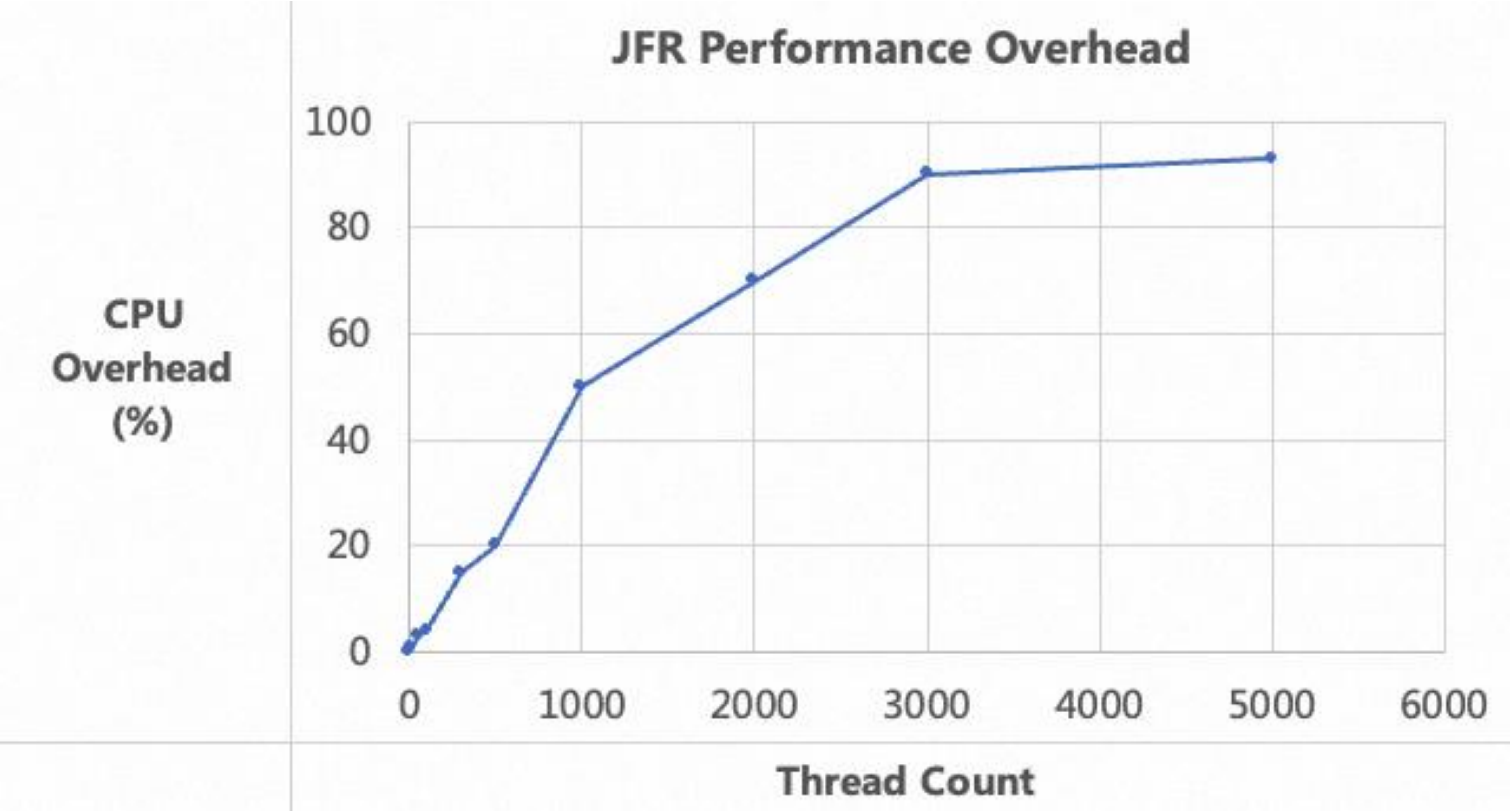
```
public void setContext(long traceId, long spanId)
public void clearContext()

public boolean setContextValue(String attribute, String value)
public boolean clearContextValue(String attribute)
```

More reference implementation info:

- https://github.com/async-profiler/async-profiler
- https://github.com/DataDog/java-profiler

Lessons Learned

#1 For JDK 8, if the number of Java threads is large (for example, more than 500), CPU overhead may be expensive and is positively related to the number of threads.



It is highly recommended to add **the -XX:+UseMembar** flag to avoid this problem.
It has been turned on by default since JDK 10: https://bugs.openjdk.org/browse/JDK-8187812

Lessons Learned(2)

#2 For JDK 8 and JDK 11, the amount of events for memory allocation may be large. It is not recommended to enable them for applications with fast memory allocation.

- https://bugs.openjdk.org/browse/JDK-8257602

#3 Before JDK 11.0.7, the OldObjectSample event may create unexpected amount of checkpoint data, cause the JFR file to be very large, and it is not recommended to enable it.

- https://bugs.openjdk.org/browse/JDK-8225797

#4 Before JDK 11.0.12, the OldObjectSample event is expensive, not suitable for production.

- https://bugs.openjdk.org/browse/JDK-8225797

Content

01 Basics of Observability

02 Fundamentals of JFR

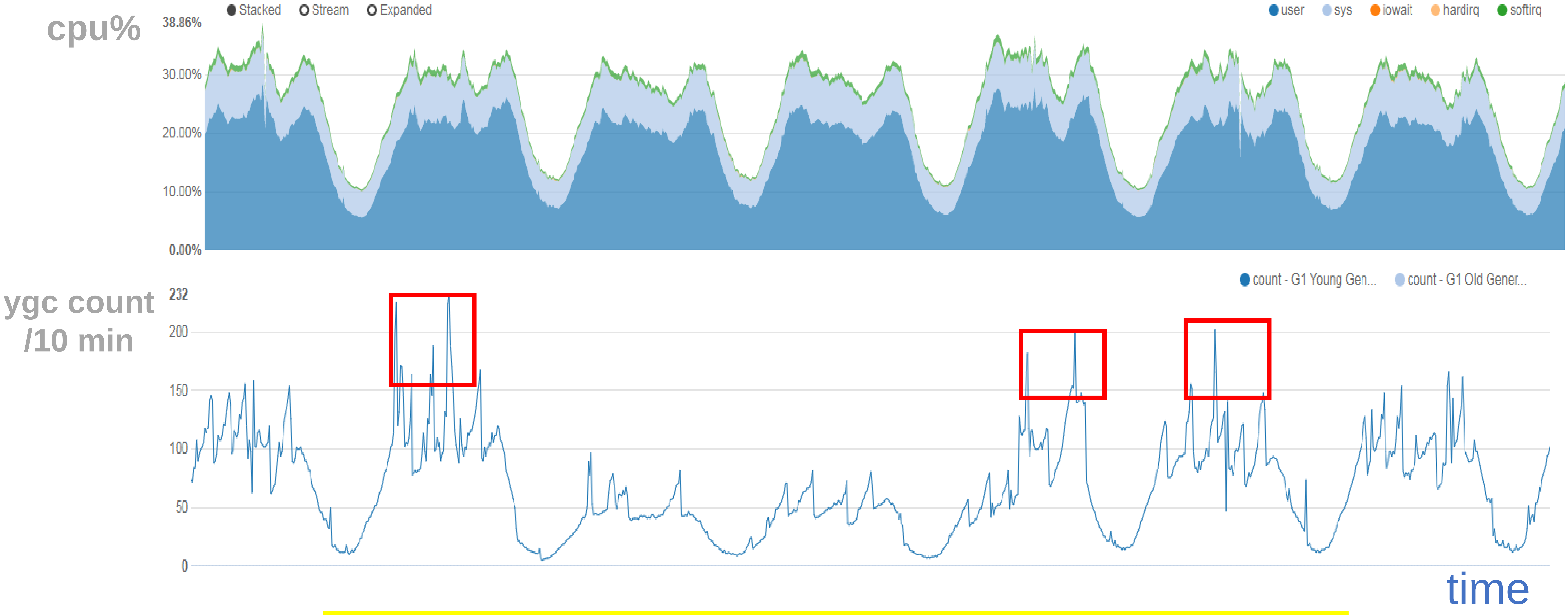
03 Continuous Profiling with Alibaba Dragonwell

04 Case Studies

05 Wrap up

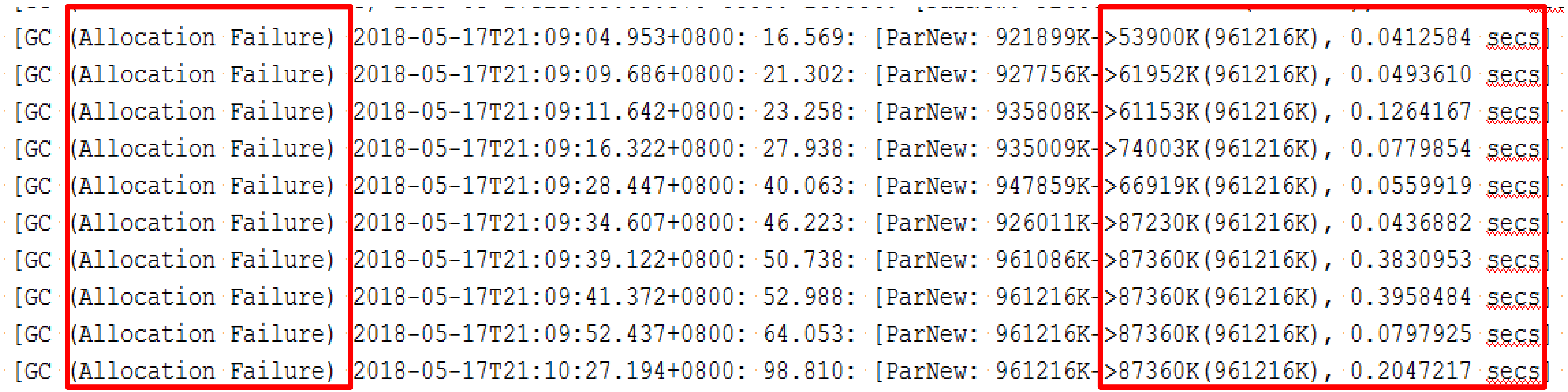
Case Study - Object Allocation

Metrics captured from real workloads



GC Spikes

Case Study - Object Allocation

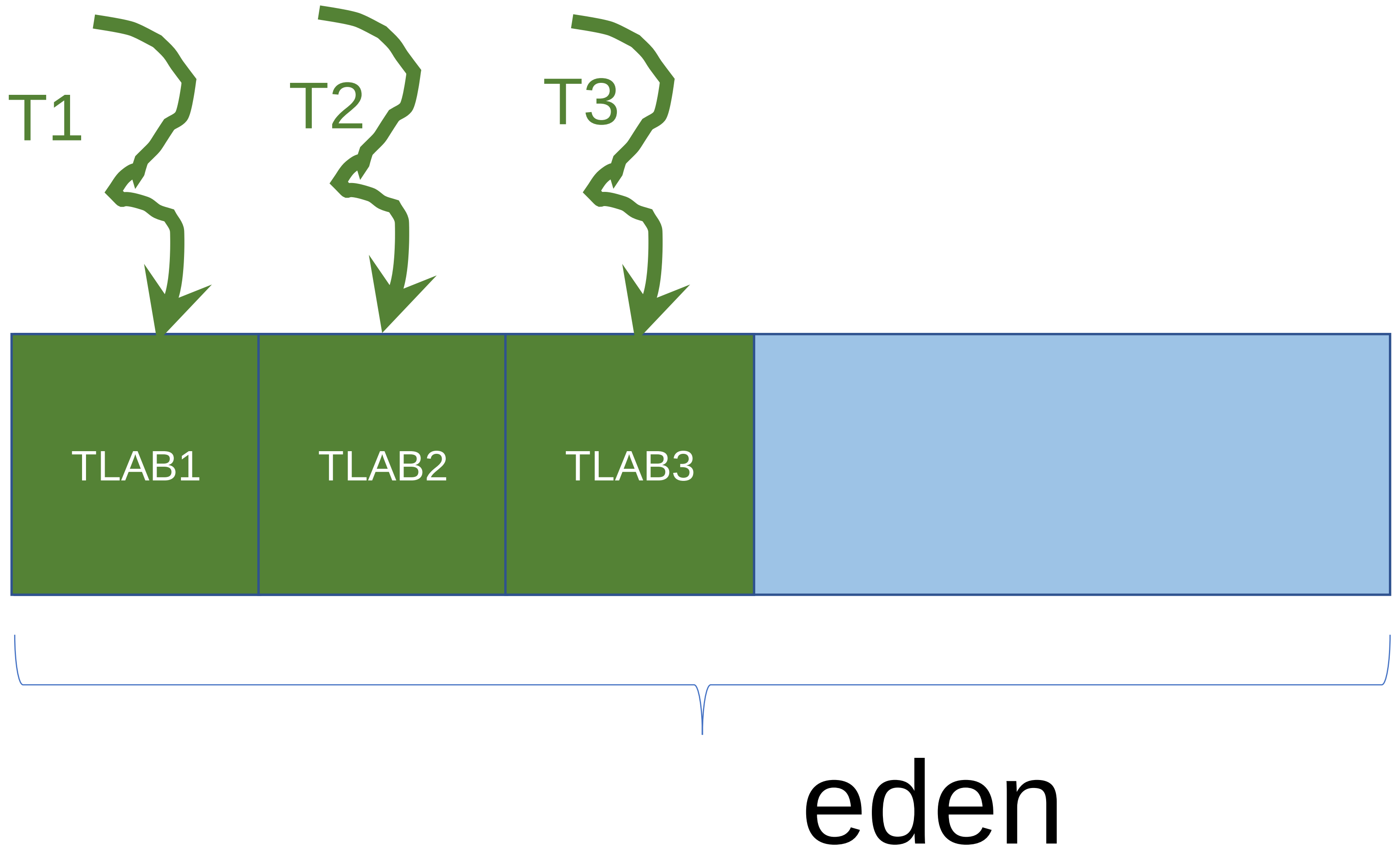


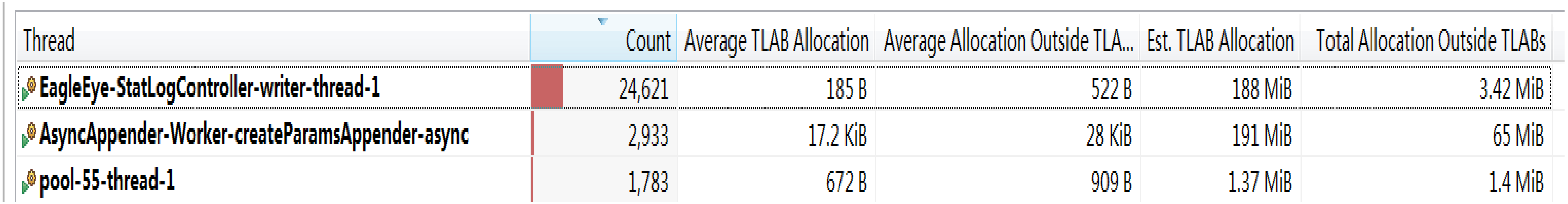
```
[GC (Allocation Failure) 2018-05-17T21:09:04.953+0800: 16.569: [ParNew: 921899K->53900K(961216K), 0.0412584 secs]
[GC (Allocation Failure) 2018-05-17T21:09:09.686+0800: 21.302: [ParNew: 927756K->61952K(961216K), 0.0493610 secs]
[GC (Allocation Failure) 2018-05-17T21:09:11.642+0800: 23.258: [ParNew: 935808K->61153K(961216K), 0.1264167 secs]
[GC (Allocation Failure) 2018-05-17T21:09:16.322+0800: 27.938: [ParNew: 935009K->74003K(961216K), 0.0779854 secs]
[GC (Allocation Failure) 2018-05-17T21:09:28.447+0800: 40.063: [ParNew: 947859K->66919K(961216K), 0.0559919 secs]
[GC (Allocation Failure) 2018-05-17T21:09:34.607+0800: 46.223: [ParNew: 926011K->87230K(961216K), 0.0436882 secs]
[GC (Allocation Failure) 2018-05-17T21:09:39.122+0800: 50.738: [ParNew: 961086K->87360K(961216K), 0.3830953 secs]
[GC (Allocation Failure) 2018-05-17T21:09:41.372+0800: 52.988: [ParNew: 961216K->87360K(961216K), 0.3958484 secs]
[GC (Allocation Failure) 2018-05-17T21:09:52.437+0800: 64.053: [ParNew: 961216K->87360K(961216K), 0.0797925 secs]
[GC (Allocation Failure) 2018-05-17T21:10:27.194+0800: 98.810: [ParNew: 961216K->87360K(961216K), 0.2047217 secs]
```

GC log cannot tell you what allocated the most objects

Case Study - Object Allocation

- Support TLAB allocation statistics by
  - EventObjectAllocation**Outside**TLAB
  - EventObjectAllocation**InNew**TLAB
- Used to find out where the allocation pressure is





Thread	Count	Average TLAB Allocation	Average Allocation Outside TLA...	Est. TLAB Allocation	Total Allocation Outside TLABs
EagleEye-StatLogController-writer-thread-1	24,621	185 B	522 B	188 MiB	3.42 MiB
AsyncAppender-Worker-createParamsAppender-async	2,933	17.2 KiB	28 KiB	191 MiB	65 MiB
pool-55-thread-1	1,783	672 B	909 B	1.37 MiB	1.4 MiB

Object Allocation JFR Demo

1. Run server: make object allocations on request arrival.

2. Use jcmd to enable object allocation event tracking.

3. Launch client: send requests to sever.

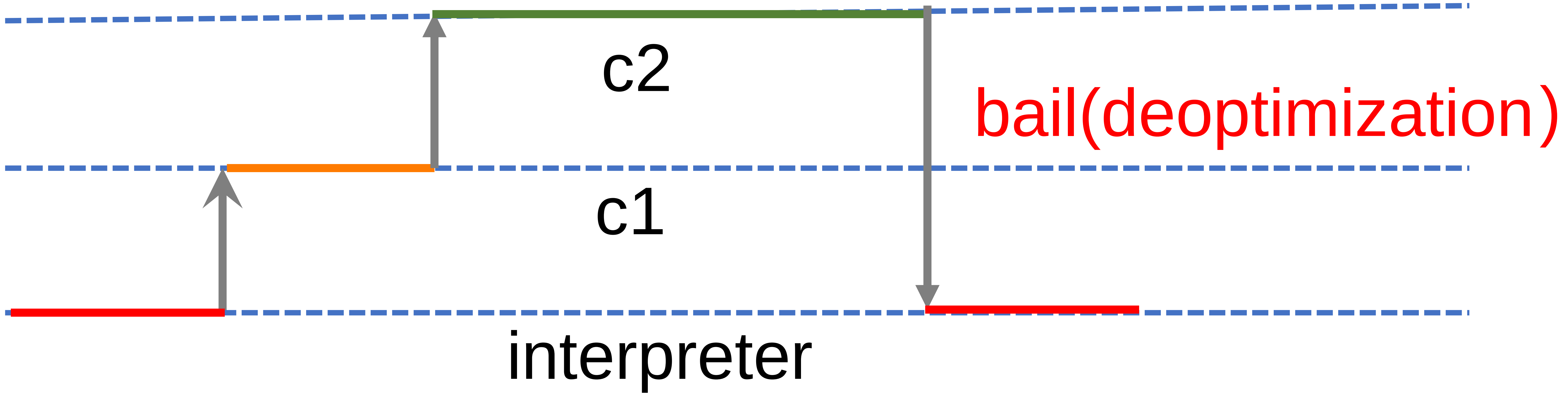
4. Use jmcx to generate folded stacks from JFR dump file.

5. Use flamegraph.pl to generate flame graph for object allocation.

Case Study II - Deoptimization

Basics concept of Just-in-Time compiler

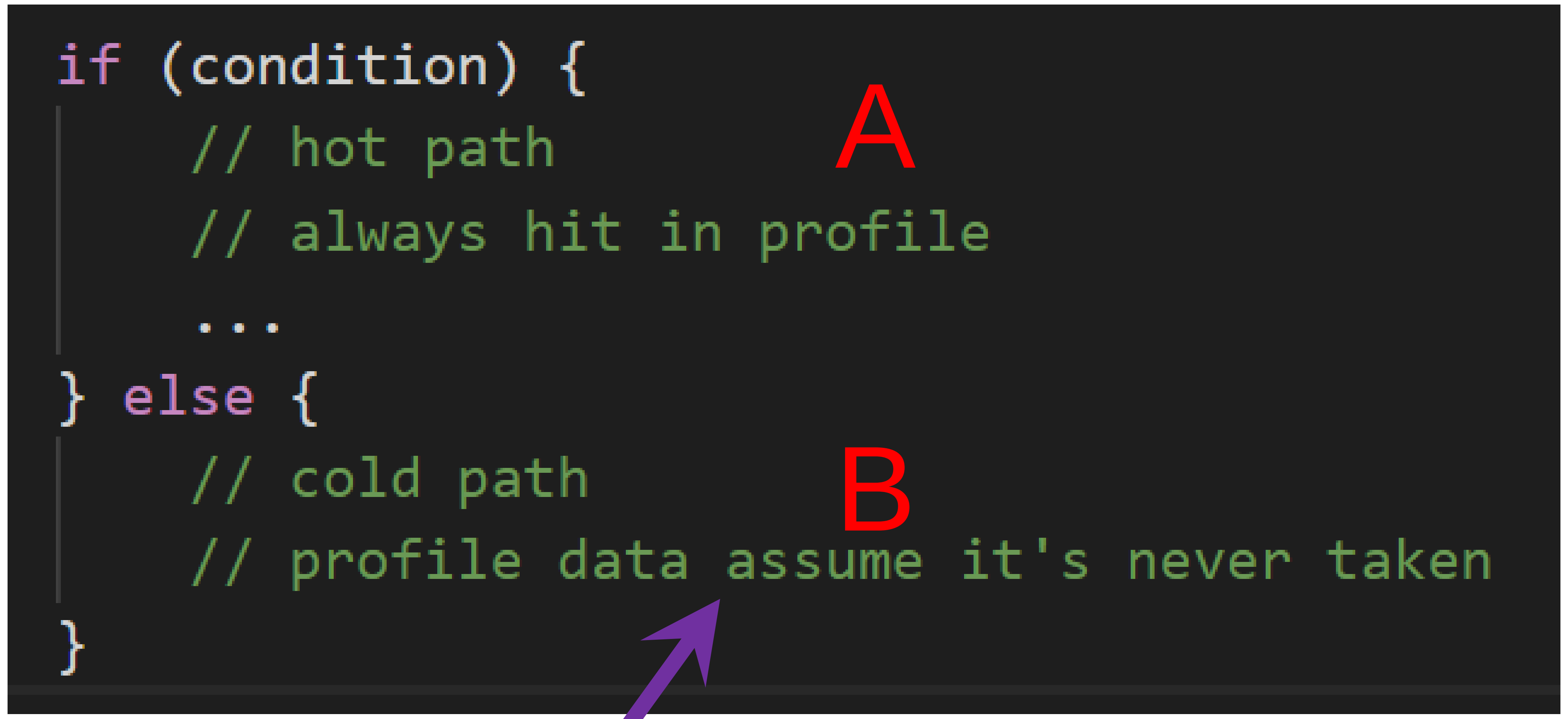
- Mix mode execution
- Profile Guided Optimization
  - Optimization decision are made dynamically
  - Bail to interpreter if the assumption is wrong



Deoptimization is very expensive if speculation is wrong:
fall back to interpreter and wait for re-compilation

Case Study II - Deoptimization

- Unstable if case



```
if (condition) {
    // hot path
    // always hit in profile
    ...
} else {
    // cold path
    // profile data assume it's never taken
}
```

trap to runtime system: uncommon branch is taken

Deoptimization JFR Demo

1. Run JMH benchmark: an example for Unstable-if deopt

   ✓Enable JFR setting via –jvmArgs(JMH parameter)

2. Use jmcx to generate folded stacks from JFR dump file.

3. Use flamegraph.pl to generate flame graph for object allocation.

ALIBABA CLOUD
INTELLIGENCE GROUP

Content

01 Basics of Observability

02 Fundamentals of JFR

03 Continuous Profiling with Alibaba Dragonwell

04 Case Studies

05 Wrap up

Round-up

1. The Definition of Observability and basics of Continuous Profiling(4th Pillar of Observability)

2. Basics of JFR(Observability tools for JVM applications)

   ✓ JFR Workflow(How to use it) and JFR Events(Understand what they are used for)

3. Alibaba Practice: ARMS Continuous Profiler

4. Two Case Studies : Object Allocations/Deoptimization

THANKS