Alibaba Cloud | Worldwide Cloud Services Partner

Alibaba Dragonwell:
Towards a Java Runtime for Cloud Computing

Sanhong Li, Chief JVM Architect

Agenda

- Introduction: Java at Alibaba
- Alibaba Dragonwell: Optimizing OpenJDK for Our Needs
  - ElasticHeap
  - JWarmUp
  - JFR Extensions

Alibaba Infrastructure

Database / Storage / Middleware / Computing Platform

Resource Scheduling / Cluster Management / Container

System Software (OS / JVM / Virtualization)

Alibaba lives on JVM

Web & Application Server

RPC

In-Memory Database

Big Data

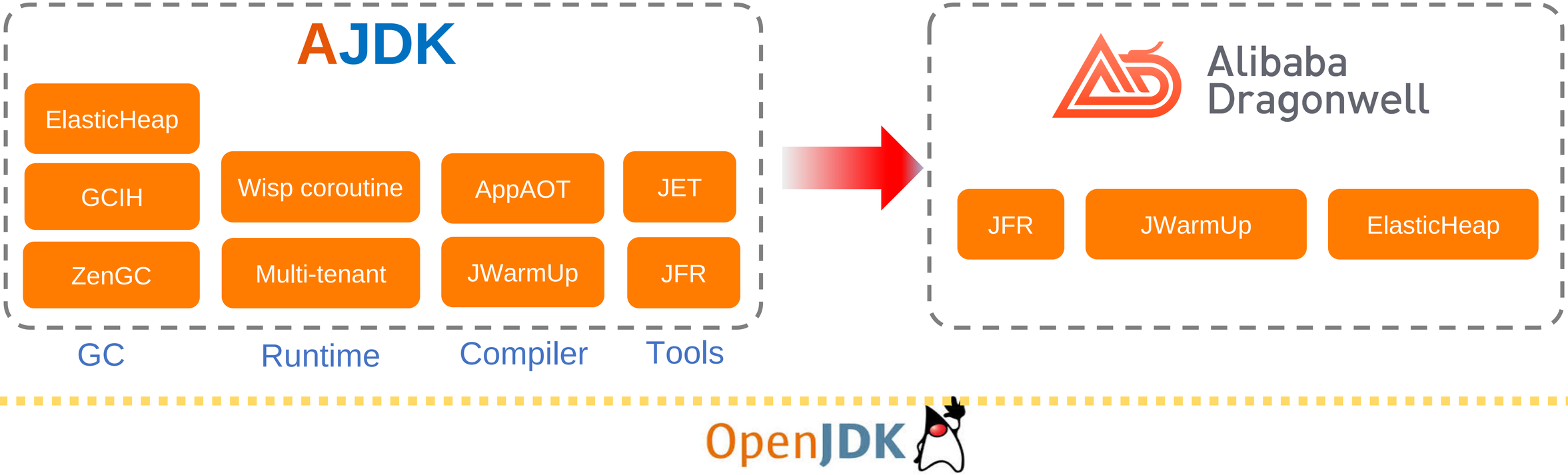
Usage in Alibaba (approx.)

10,000 developers

100,000 applications

1,000,000 JVM instances

OpenJDK, AJDK and Dragonwell



Plan to ‘ship’ ALL of them to Alibaba Dragonwell and talk to OpenJDK community for contribution

Alibaba Dragonwell

- A customized downstream of OpenJDK with **free LTS**
- https://github.com/alibaba/dragonwell8
  - GA in June 2019
  - Default JDK distribution in Alibaba Cloud
  - Released quarterly
  - Dragonwell11 will be available at end of 2019

Alibaba Dragonwell

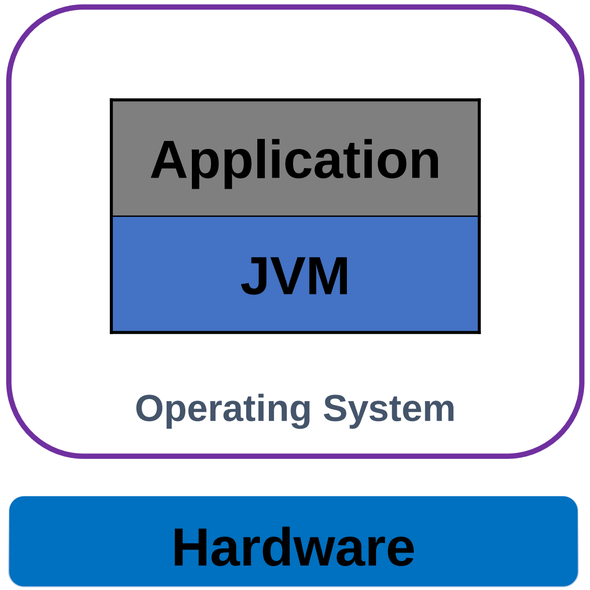
Agenda

- Introduction: Java at Alibaba
- Alibaba Dragonwell: Optimizing OpenJDK for Our Needs
  - ElasticHeap
  - JWarmUp
  - JFR Extensions

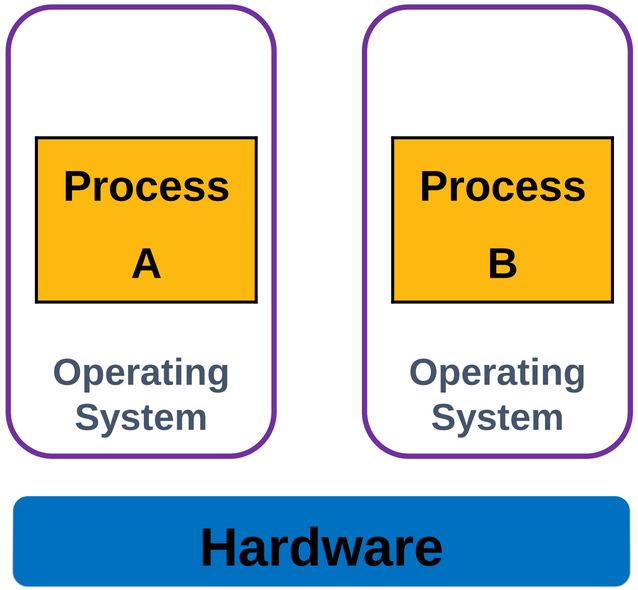
Characteristics of Cloud

- **Consolidation**(shared system with varied workloads)
  - CPU, memory, storage
- **Isolation**
  - Virtualization for security

JVM vs Hypervisor(Similar to Container)



VS



JVM's view

- JVM is managing resources on behalf of user

Hypervisor's view

- Hypervisor is managing resources used by each Guest

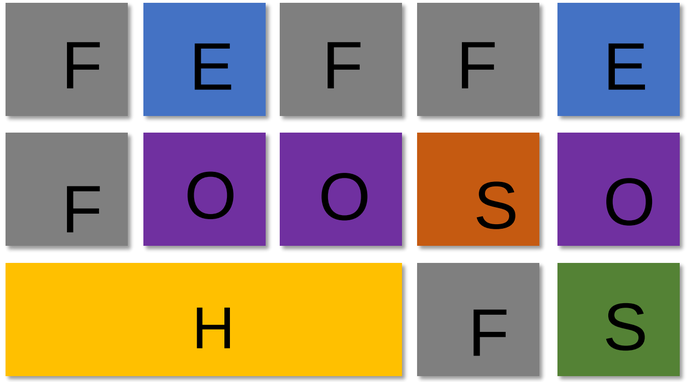
Dynamic Memory Scalability

- Cloud offering model is “pay-as-you-use”
- JVM heap sizing strategy is “static”
  - -Xmx must be configured statically at launch time
- Inability to return “unused” memory to OS
- **Vertical Scalability**: scale up/down memory used by JVM on demand

ElasticHeap: Scale memory utilization according to application’s needs

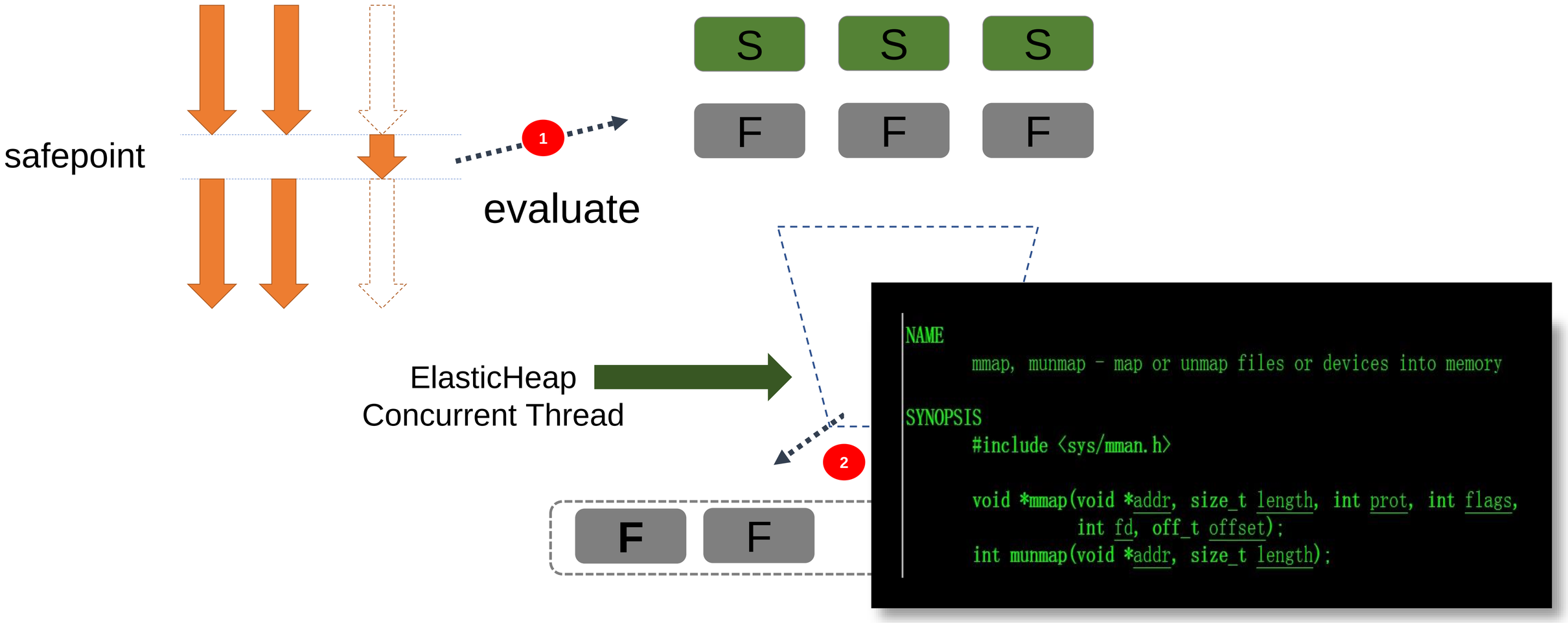
G1 Basic Concepts

- G1, introduced in Java 6, fully supported from Java7 u4, made as default in Java 9
  - **Generational**
  - **Region-based**



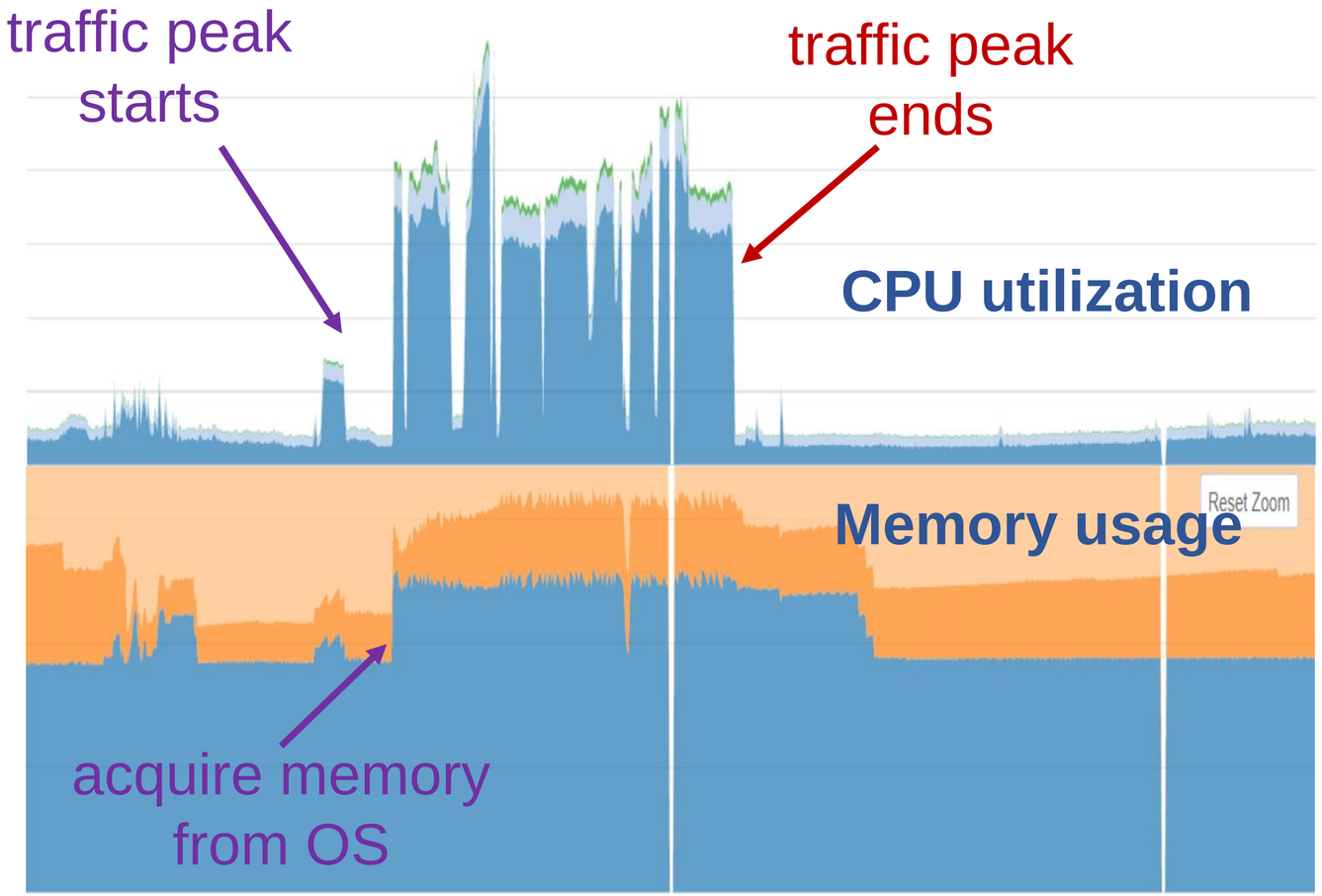
E: Eden(young)
S: Survivor(young)
O: Old(old)
H: Humongous
F: Free

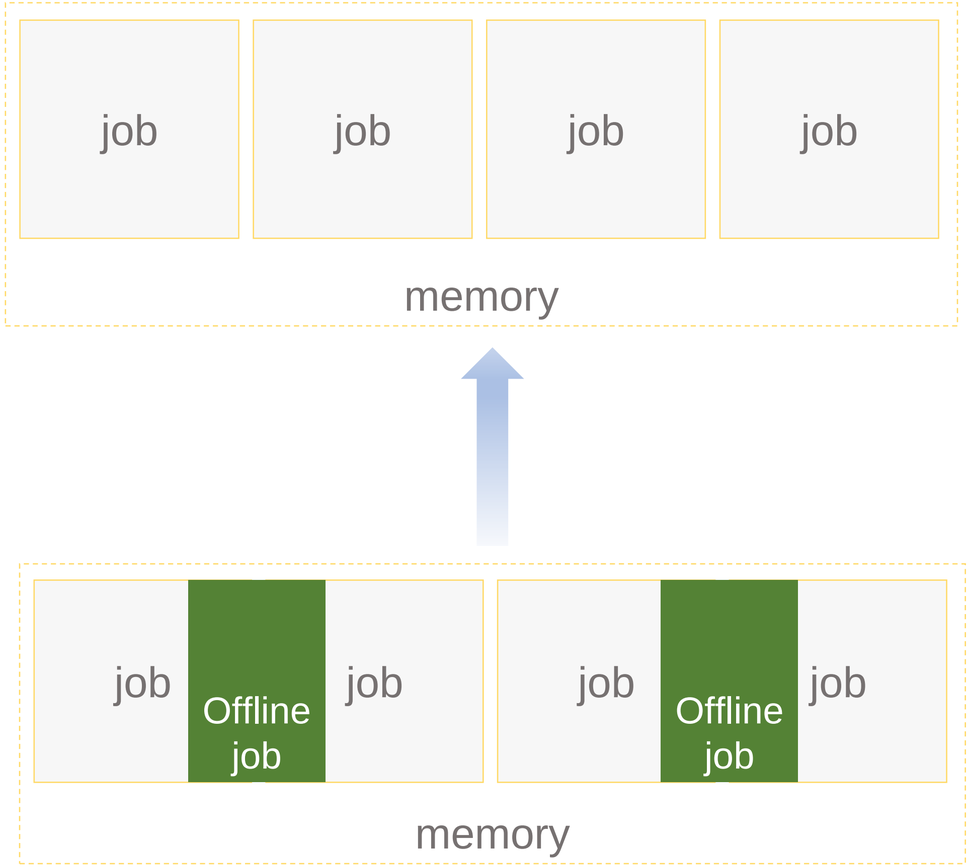
‘Under the hood view’ of ElasticHeap



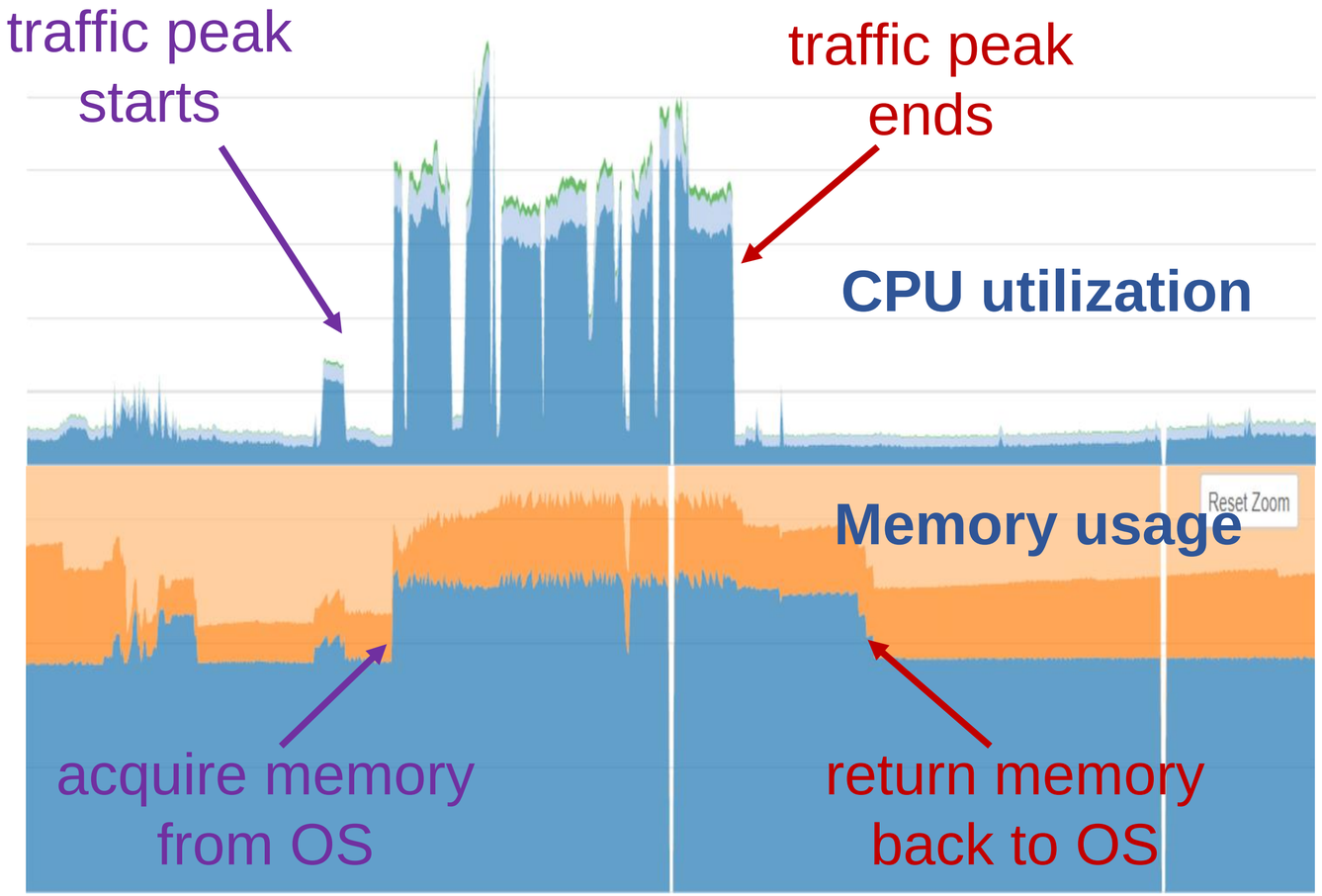
Offload map/unmap operations from VM thread in STW to concurrent thread

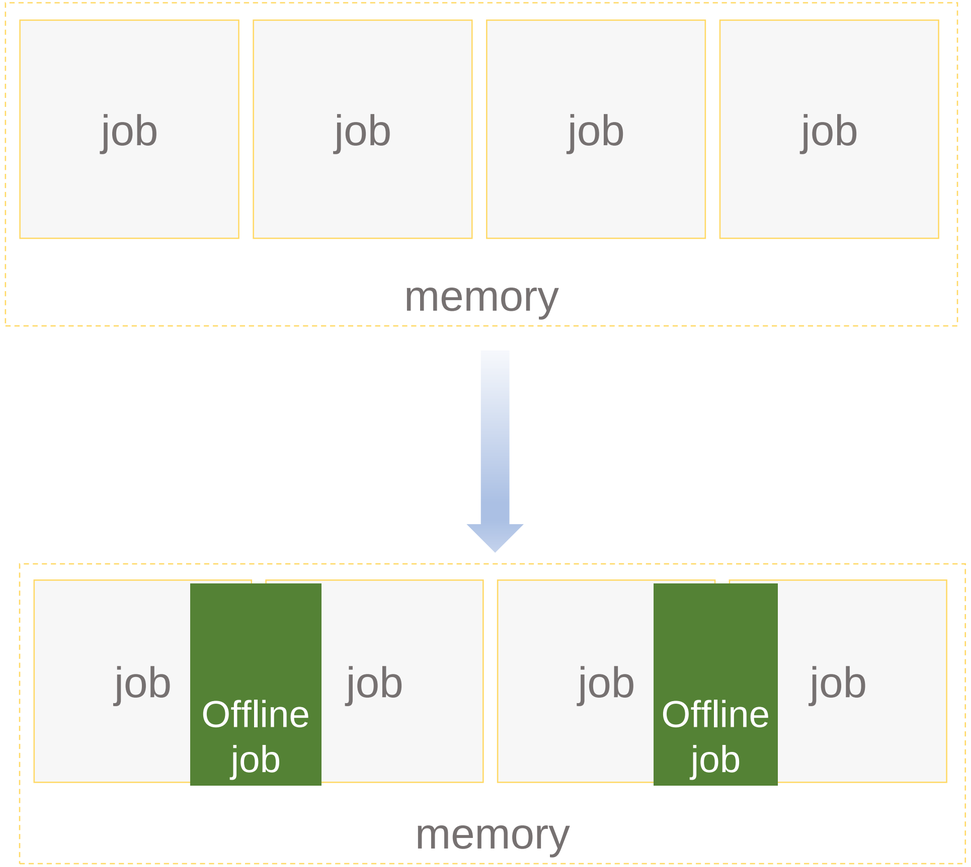
User Story of ElasticHeap





User Story of ElasticHeap





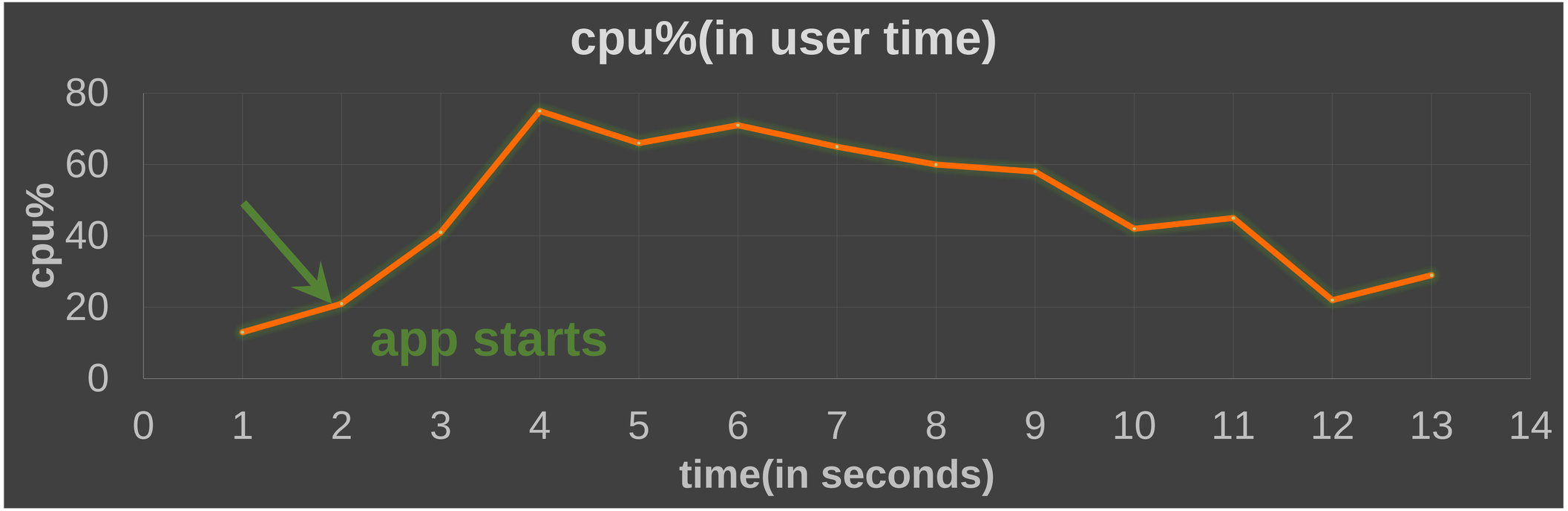
Agenda

- Introduction: Java at Alibaba
- Alibaba Dragonwell: Optimizing OpenJDK for Our Needs
  - ElasticHeap
  - JWarmUp
  - JFR Extensions

Warmup Overhead in Java

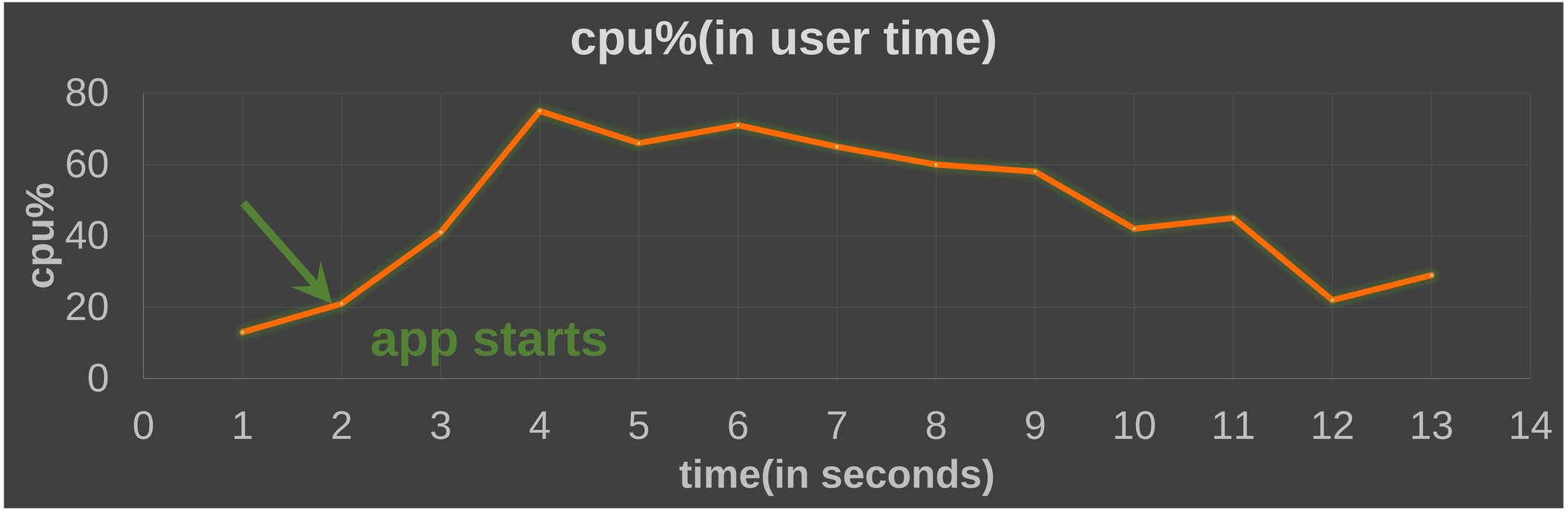
- ☑ Class loading
- ☑ Interpreter
- ☑ Profiling & Method Compilation

Warmup Overhead in Real Case



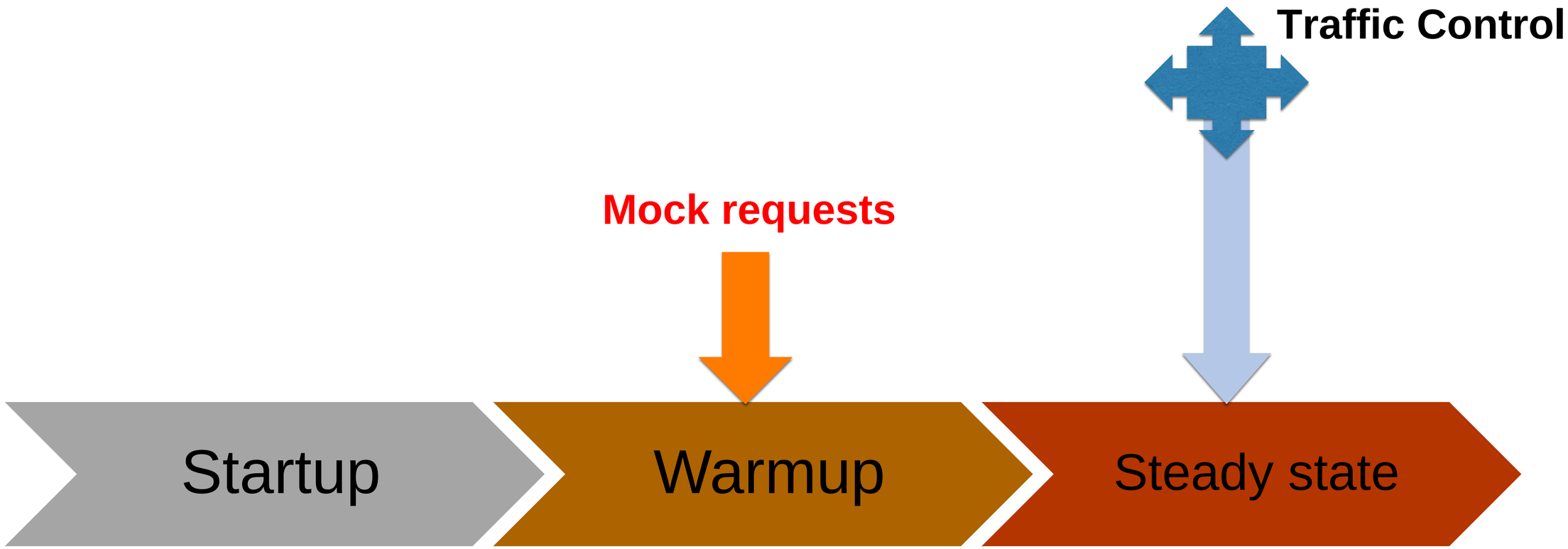
- Observation #1 High CPU consumption
  - Compiler threads consume much more CPU

Warmup Overhead in Real Case



- Observation #1 High CPU consumption
  - Compiler threads consume much more CPU
- Observation #2 Longer response time(RT)
  - Most methods are executed in interpreter

Warmup by Mock Requests



RISKY: may NOT produce desired optimization

Warmup Techniques in JVM

- Ahead-of-Time(AOT) compilation
  - Experimental feature in JDK9

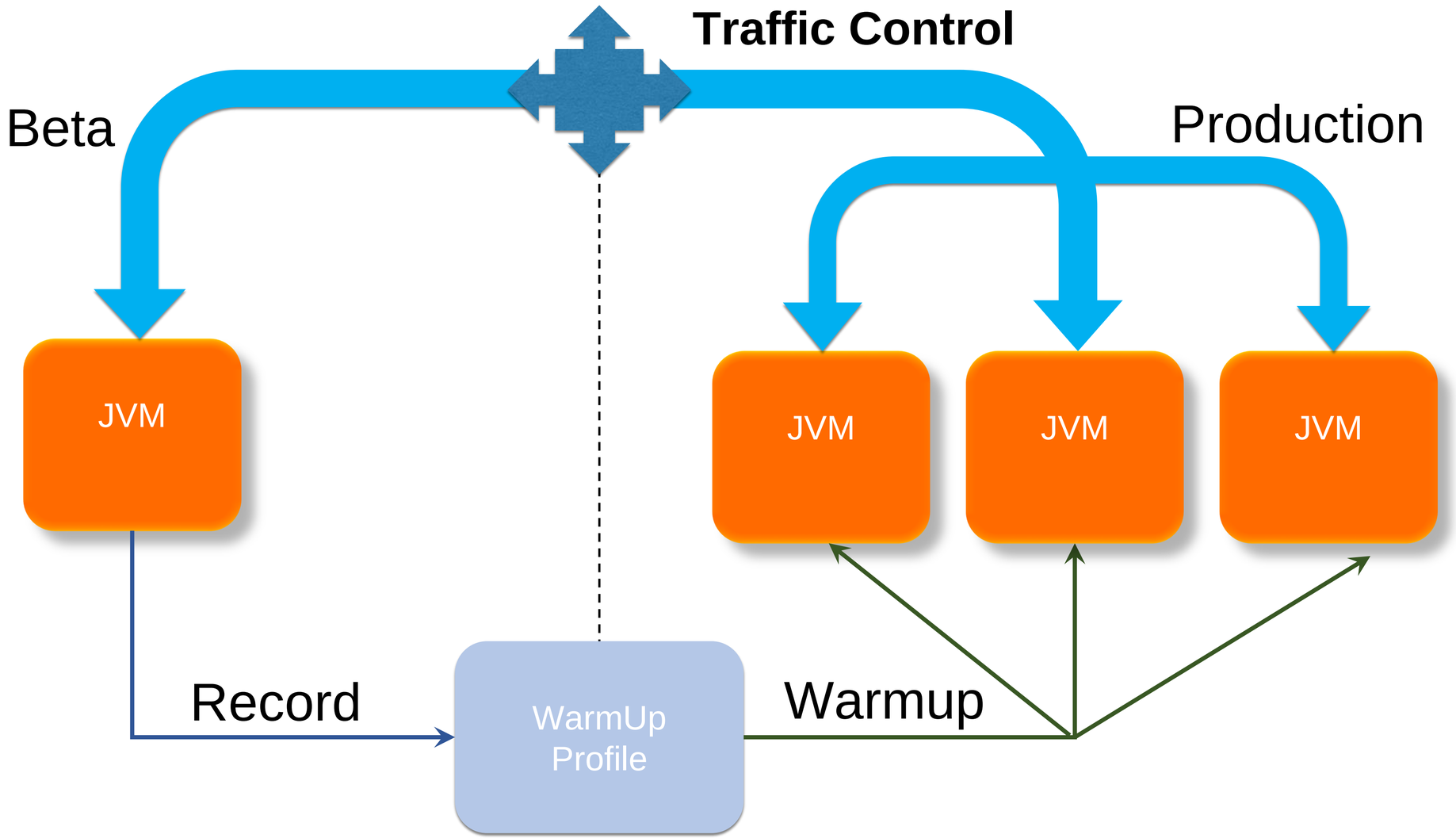
Warmup Techniques in JVM

- Ahead-of-Time(AOT) compilation
  - Experimental feature in JDK9
- Cache compilation
  - “Dynamic AOT” in OpenJ9
  - “Compile Stashing” in Zing

Warmup Techniques in JVM

- Ahead-of-Time(AOT) compilation
  - Experimental feature in JDK9
- Cache compilation
  - “Dynamic AOT” in OpenJ9
  - “Compile Stashing” in Zing
- ‘Trace-and-Replay’ compilation
  - ReadyNow in Zing
  - JWarmUp in Alibaba Dragonwell

JWarmUp Overview



WarmUp Profile

- WarmUp profile mainly contains
  - ✓ Class info
  - ✓ Method info
- Stored in binary format as file on disk
- Recorded once in previous run, distributed and used for all subsequent runs.
- Doesn’t support merge

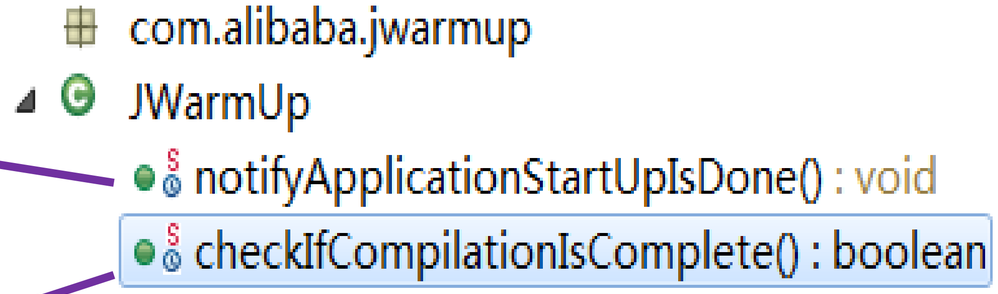
```
-------------------------------------------------------
|       section name        |       size        |
-------------------------------------------------------
|                           |                   |  <-- header
|       version number      |        4          |    <--  version number
|       magic number        |        4          |    <--  0xBABA
|       file size           |        4          |    <--  file size
|       crc32 of total file |        4          |    <--  crc32
|       appid               |        4          |    <--  appid (not use yet)
|       record count        |        4          |    <--  number of records
|       record time         |        8          |    <--  UTC time
|                           |                   |
|                           |                   |  <-- init order section
|       record size         |        4          |    <--  size of this section
|       record count        |        4          |    <--  record count
|       class name          |      string       |    <--  class name
|       class loader name   |      string       |    <--  class loader name
|       class path          |      string       |    <--  class file path
|       class name          |      string       |    <--  class name
|       class loader name   |      string       |    <--  class loader name
|       class path          |      string       |    <--  class file path
```

```
|       method name        |      string      |    <--  method name
|       method signatue    |      string      |    <--  method signature
|       init order         |        4         |    <--  first invoke init order
|       method size        |        4         |    <--  method bytecode size
|       method hash        |        4         |    <--  method hash value
|       bci                |        4         |    <--  if bci > -1, then is OSR compilation
|       class name         |      string      |    <--  class name
```

JWarmUp API

Gives DevOps new control over JIT compilation

- Trigger warmup compilation after application startup is done.
- Let user requests come in after warmup compilation is done



github: https://github.com/alibaba/dragonwell8_jdk/blob/master/src/share/classes/com/alibaba/jwarmup/JWarmUp.java

JWarmup Compilation Process

4 Requests come in

checkIfCompilationIsComplete() : boolean

3 Check warmup is done

notifyApplicationStartUpIsDone() : void

2 Trigger WarmUp compilation eagerly

Startup → Warmup → Full Speed run

1
- Parse warmup profile
- compile methods if condition is satisfied.

- Eagerly load/initialize necessary classes in safe way
- Submit for compilation

github wiki guide: https://github.com/alibaba/dragonwell8/wiki/Alibaba-Dragonwell8-User-Guide

Pitfalls in Class Initialization

Normally, **Bar.<clinit>** is triggered by **Bar.test()** in **Foo().test()**

```
2 public class Foo {
3     public static int count = 0;
4     public void test() {
5         Foo.count++;
6         Bar.test();
7     }
```

```
2 public class Bar {
3     public static int count = 0;
4     static {
5         Bar.count = Foo.count;
6         Bar.count ++;
7     }
8     static void test() {}
```

new Foo().test(); **1→ Foo.count** **2 → Bar.count**

Pitfalls in Class Initialization

Initializing Bar eagerly is WRONG!

```
2 public class Foo {
3     public static int count = 0;
4     public void test() {
5         Foo.count++;
6         Bar.test();
7     }
```

```
2 public class Bar {
3     public static int count = 0;
4     static {
5         Bar.count = Foo.count;
6         Bar.count ++;
7     }
8     static void test() {}
```

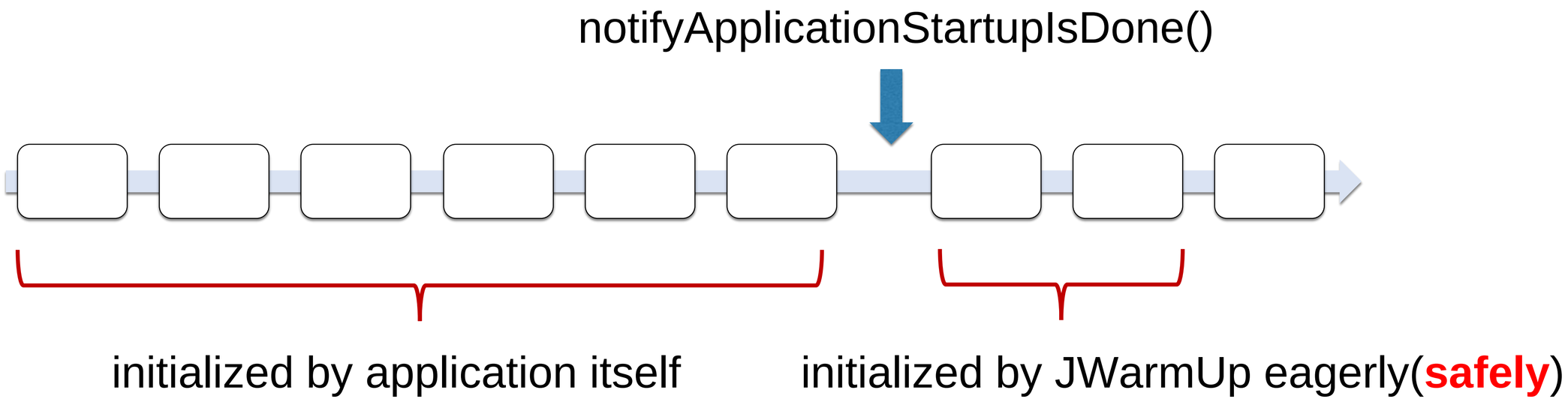
Bar.<clinit>

1 → Bar.count ✗

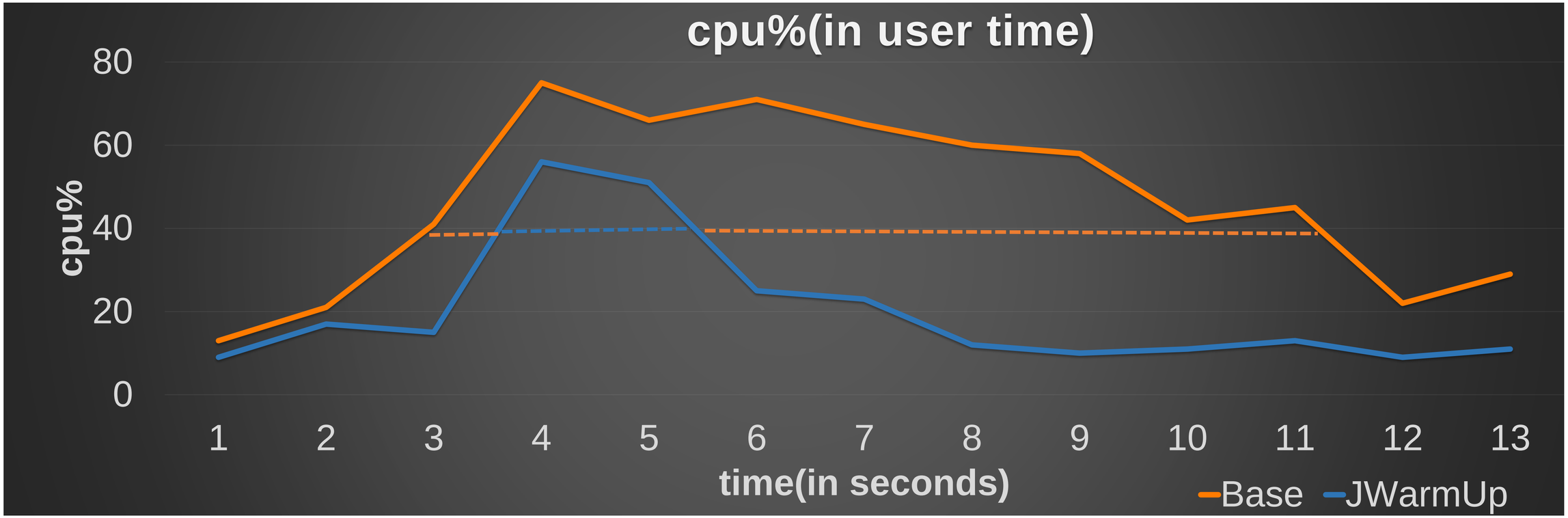
NOT safe to initialize class EAGERLY!

notifyApplicationStartupIsDone() API

- Hint by application owner
  - The initialization of application is done(assumption: almost of all classes have been initialized by application itself)
- Try best to do ‘SAFE’ initialization for remaining classes EAGERLY after calling API



Optimized Results by JWarmUp



- ❑ Peak CPU usage has been reduced from **75% to 56%**
- ❑ Time range in peak has been reduced from **8s to 2s**
- ❑ 99.9th response time has reduced from **100ms to 80ms**

JIT and JWarmUp

JIT

- **Reactive** compilation, user CAN NOT directly invoke

JWarmUp

- **Proactive** compilation, user can trigger it explicitly

JIT and JWarmUp

JIT

- **Reactive** compilation, user CAN NOT directly invoke
- Need runtime information as inputs, **dynamically collected in runtime.**

JWarmUp

- **Proactive** compilation, user can trigger it explicitly
- Need runtime information as inputs, **some of them are from log in previous run**

JIT and JWarmUp

JIT

- **Reactive** compilation, user CAN NOT directly invoke
- Need runtime information as inputs, **dynamically collected in runtime.**
  - Resolution status for external references
    - **All of classes are resolved by runtime lazily**

JWarmUp

- **Proactive** compilation, user can trigger it explicitly
- Need runtime information as inputs, **some of them are from log in previous run**
  - Resolution status for external references
    - **Some of classes are resolved eagerly(safely)**

JIT and JWarmUp

JIT

- **Reactive** compilation, user CAN NOT directly invoke
- Need runtime information as inputs, **dynamically collected in runtime.**
  - Resolution status for external references
    - **All of classes are resolved by runtime lazily**
  - Runtime profiles
    - Block profiling
    - Edge profiling
    - Value profiling
    - Method profiling

JWarmUp

- **Proactive** compilation, user can trigger it explicitly
- Need runtime information as inputs, **some of them are from log in previous run**
  - Resolution status for external references
    - **Some of classes are resolved eagerly(safely)**
  - Runtime profiles
    - **Only method profiling recorded in log in current implementation, used to proactively compile methods.**

ReplayCompiles and JWarmUp

- Only existed in debug version
- Used in production

ReplayCompiles and JWarmUp

- Only existed in debug version
- Find root cause of crashed java process in compiled method
- Used in production
- Eliminate the warmup overhead at application startup.

ReplayCompiles and JWarmUp

- Only exists in debug version
- Find root cause of crashed java process in compiled method
- Only repeats the compiling process for last compilation task

- Used in production
- Eliminate the warmup overhead at application startup.
- Repeats the compiling process for all recorded methods(concept is similar)

```
ciMethodData java/lang/Object <init> ()V 2 1421 orig 304 240 16 136 32 101 43 0 0 0 0 0
ciMethod java/lang/ClassValue$Entry <init> (Ljava/lang/ClassValue$Version;)V 0 0 1 0 -1
ciMethod java/lang/ClassValue$Version <init> (Ljava/lang/ClassValue;)V 1 1 1 0 -1
ciMethodData java/lang/ClassValue$Version <init> (Ljava/lang/ClassValue;)V 1 0 orig 304
compile java/lang/ClassValue$Version <init> (Ljava/lang/ClassValue;)V -1 3
```

Agenda

- Introduction: Java at Alibaba
- Alibaba Dragonwell: Optimizing OpenJDK for Our Needs
  - ElasticHeap
  - JWarmUp
  - JFR Extensions
    - ✓ Motivations
      - ✓ Excessive GC pauses
      - ✓ Excessive de-optimizations
    - ✓ Implementation

Java Flight Recorder (JFR)

- Open Source in OpenJDK 11
- Backport into AlibabaJDK8 for internal use
- Included in Alibaba Dragonwell
- Working with the community to contribute back to OpenJDK8u → **8u-jfr-incubator**

Re: Proposal for back-porting JFR to OpenJDK8u

guangyu.zhu guangyu.zhu at aliyun.com
Tue Jan 29 11:41:35 UTC 2019

- Previous message: [8u] Request for approval for CR 8215318 - Amend the Standard Algorithm Names specification to clarify that names can be defined in later versions
- Next message: Proposal for back-porting JFR to OpenJDK8u
- **Messages sorted by:** [date] [thread] [subject] [author]

Hi there,

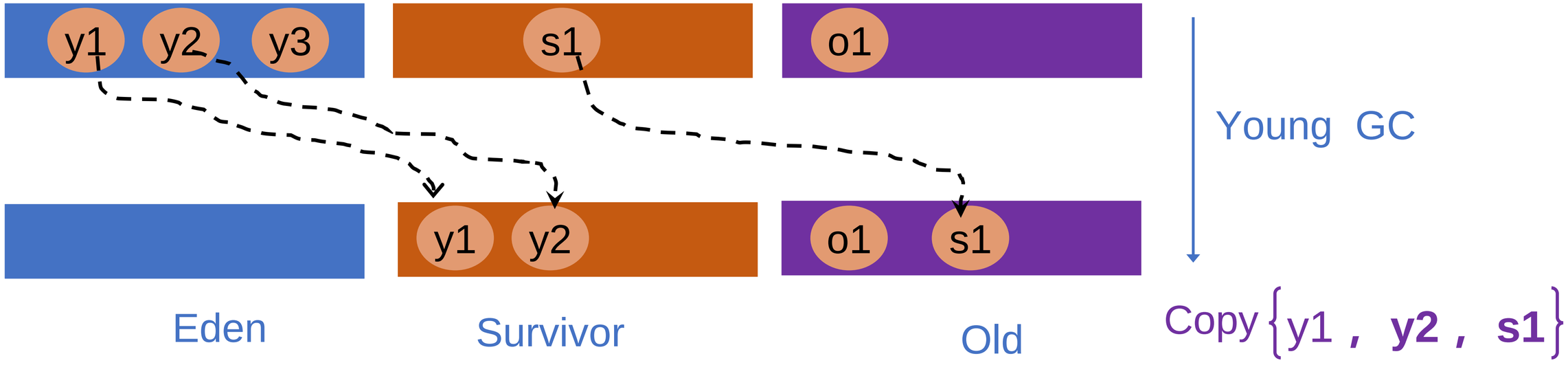
JFR backport patch has been uploaded to cr.openjdk. Please have a review for the patch.

Webrev:
http://cr.openjdk.java.net/~luchsh/hs_jfr_cr/
http://cr.openjdk.java.net/~luchsh/jdk_jfr_cr/

GC Performance Basics

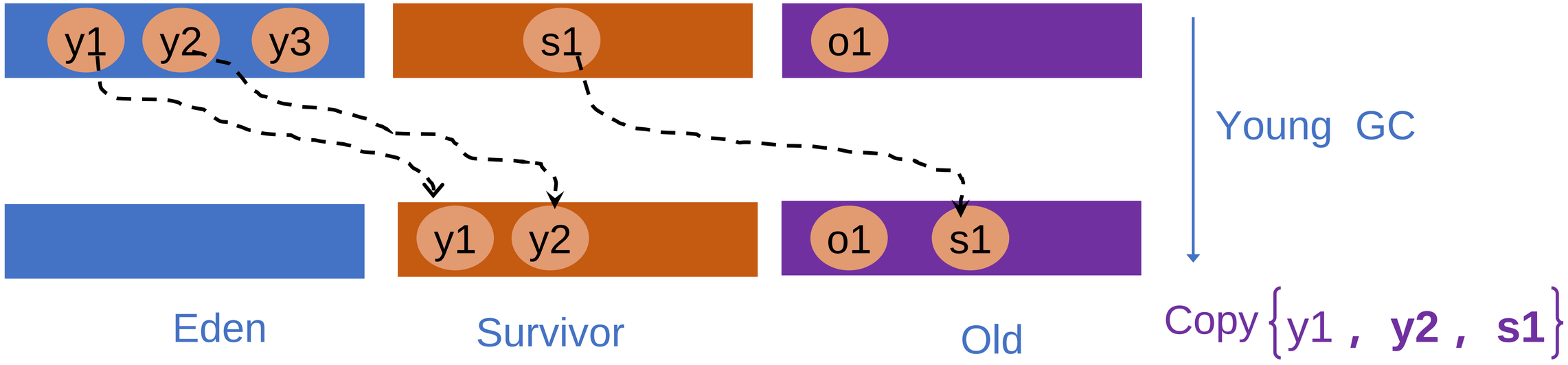
- Frequency
  - How frequent are the GC pauses?
- Duration
  - How long are those GC pauses (Stop-the-World) can you tolerate?

Young GC (YGC) in G1



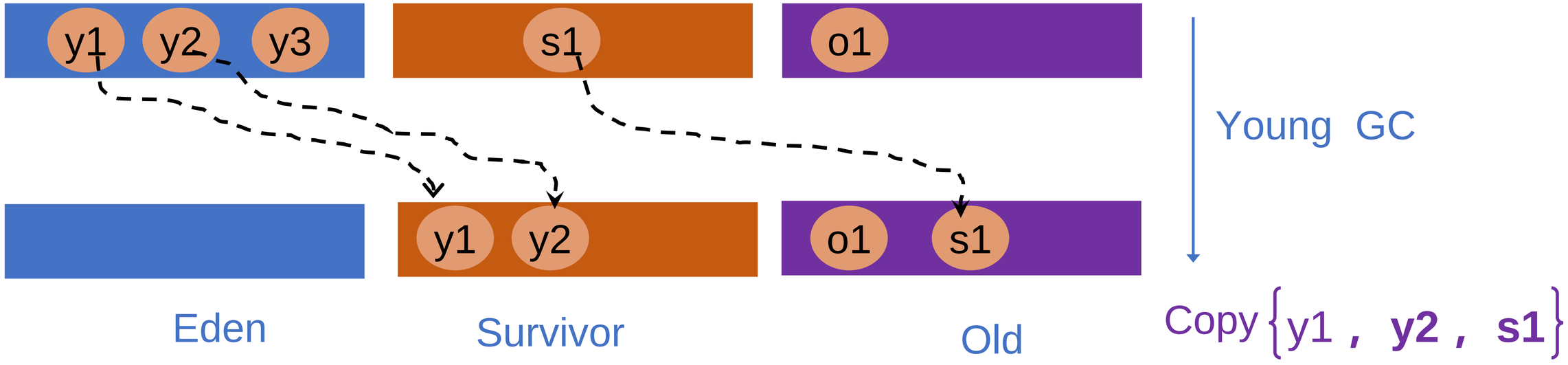
- G1 uses Mark - Copy algorithm to do the collection

Young GC (YGC) in G1



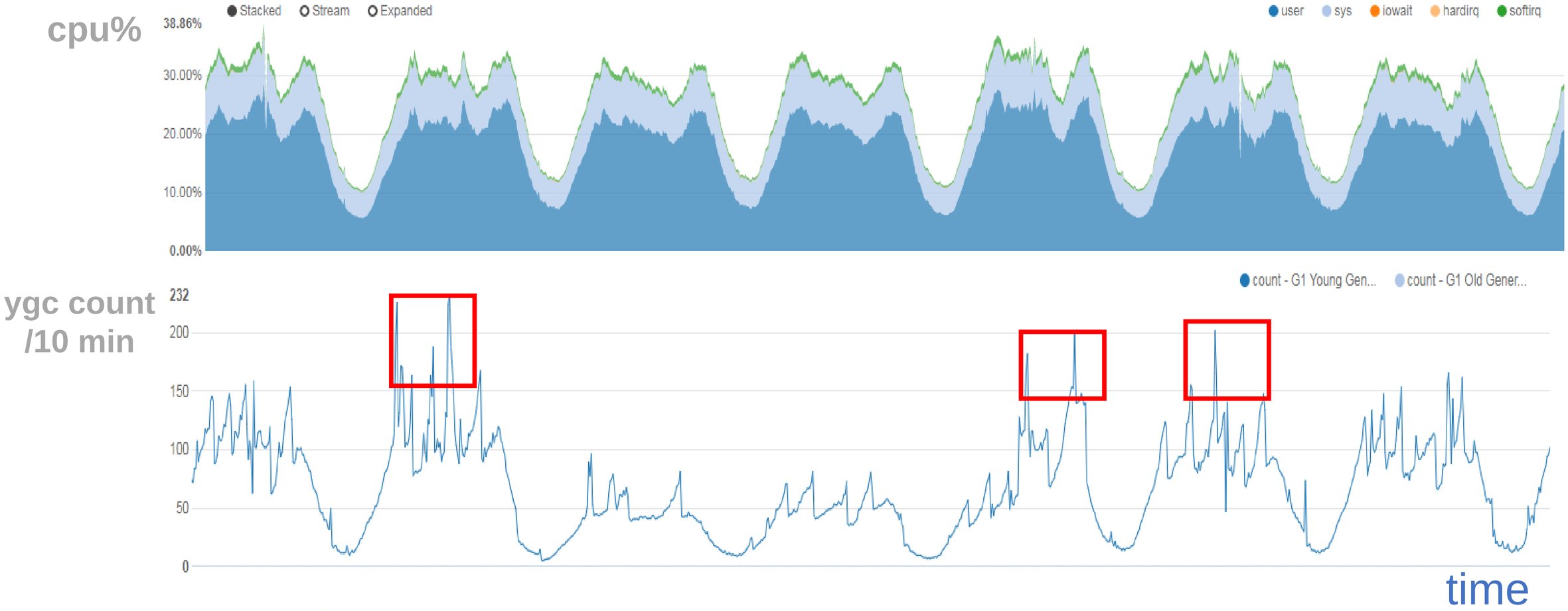
- G1 uses Mark – Copy algorithm to do the collection
- Rule #1: Frequency of a YGC event is dominated by
  - Application object allocation rate
  - Size of eden space

Young GC (YGC) in G1



- G1 uses Mark – Copy algorithm to do the collection [**Cheney, 1970**]
- Rule #1: Frequency of a YGC event is dominated by
  - Application object allocation rate
  - Size of eden space
- Rule #2: Pause time of YGC is dominated by
  - size of live objects, not dead.

GC Challenge: Why Pause Happened Frequently?



GC Spikes

Questions in YGC

- What allocated the most objects?

```
[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]
```

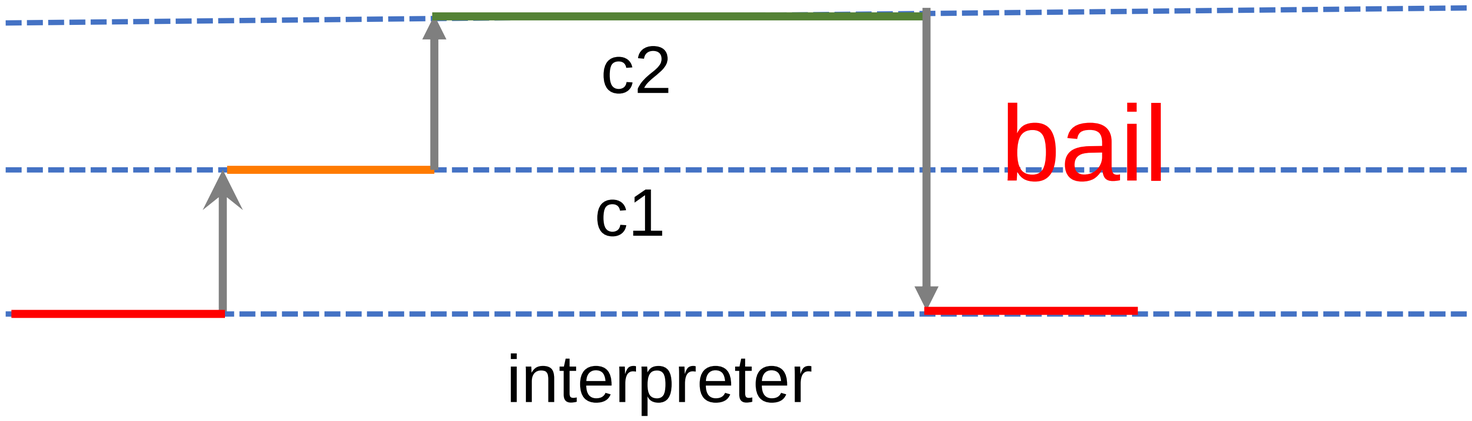
Deciphering the GC log files is simply daunting

Agenda

- Introduction: Java at Alibaba
- Alibaba Dragonwell: Optimizing OpenJDK for Our Needs
  - ElasticHeap
  - JWarmUp
  - JFR Extensions
    - ✓ Motivations
      - ✓ Excessive GC pauses
      - ✓ Excessive de-optimizations
    - ✓ Implementation

JIT Basic Concepts

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



Top Reasons for Deoptimization

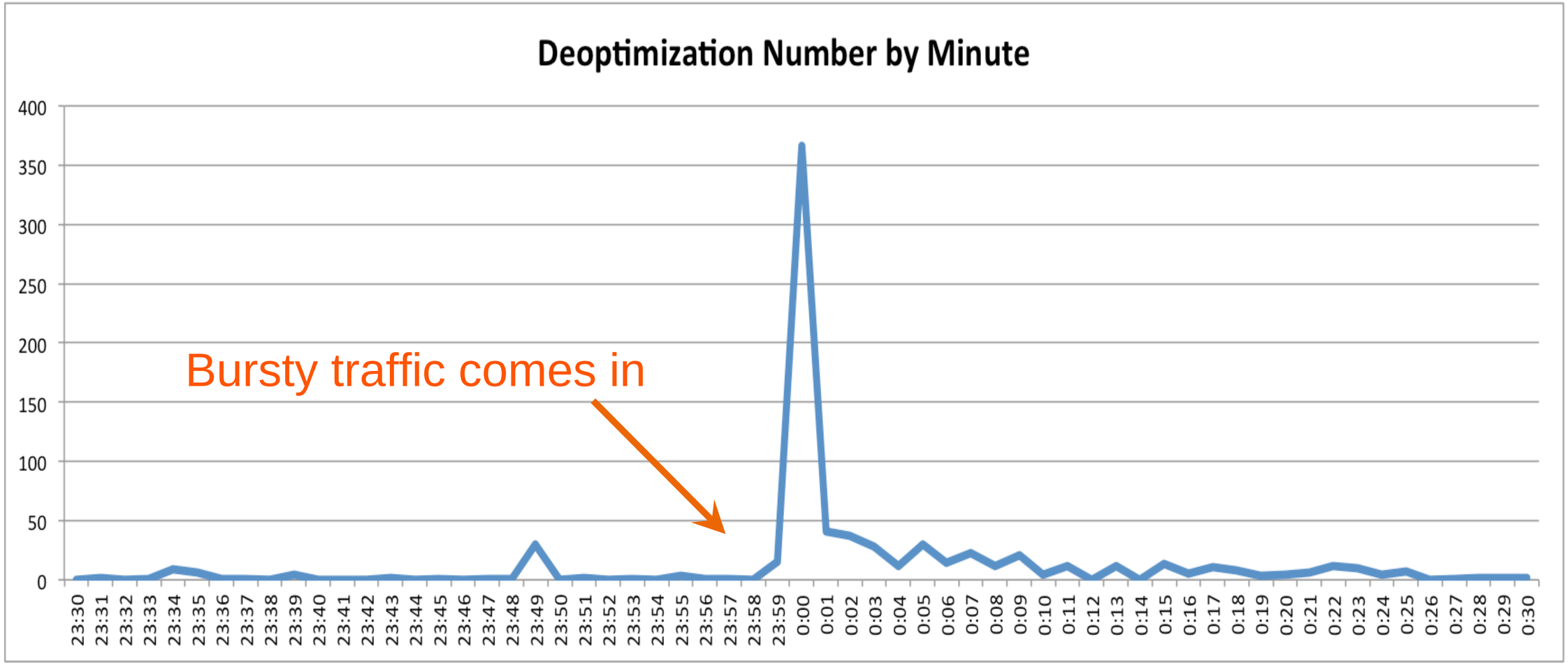
- Unstable if
- Null check
- Class check
- Bimorphic

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

} else {
    de-optimize
}
```

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

JIT Challenge: Why Deoptimization Happened Frequently?



Questions in JIT Performance

- How we can avoid JIT deoptimization?

```
31662 1074       3       sun.nio.cs.StreamEncoder::ensureOpen (18 bytes)
31663 1073       3       java.lang.StringBuilder::append (10 bytes)
31663 1075       3       java.io.OutputStreamWriter::flushBuffer (8 bytes)
31663 1076       1       java.util.Formatter::access$000 (5 bytes)
31791 1077       4       java.lang.StringBuilder::<init> (7 bytes)
31792  125       3       java.lang.StringBuilder::<init> (7 bytes)   made not entrant
31803 1078       3       com.sun.org.apache.xerces.internal.dom.ParentNode::<init> (21 bytes)
31804  930   s   4       spec.jbb.Order::processLines (240 bytes)   made not entrant
31804  590   s   4       spec.jbb.Orderline::validateAndProcess (63 bytes)   made not entrant
31805 1079   s   3       spec.jbb.Orderline::validateAndProcess (63 bytes)
31805  591       4       spec.jbb.Orderline::process (181 bytes)   made not entrant
31806 1080       3       spec.jbb.Orderline::process (181 bytes)
31808 1083   s   4       spec.jbb.Stock::getData (14 bytes)
31809 1084   s   4       spec.jbb.Stock::incrementYTD (14 bytes)
```

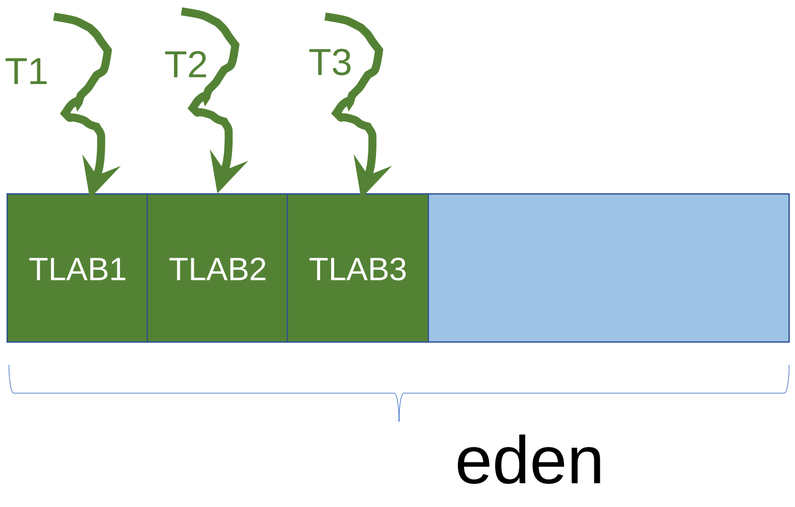
(-XX:+PrintCompilation example)

Agenda

- Introduction: Java at Alibaba
- Alibaba Dragonwell: Optimizing OpenJDK for Our Needs
  - ElasticHeap
  - JWarmUp
  - JFR Extensions
    - ✓ Motivations
      - ✓ Excessive GC pauses
      - ✓ Excessive de-optimizations
    - ✓ Implementation

The JFR State of Art

- Support TLAB allocation statistics by
  - EventObjectAllocation**Outside**TLAB
  - EventObjectAllocation**InNew**TLAB
- Very useful to check if the occurrence of allocations outside of the TLAB is significant



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

Notes: all events only occurred in slow path(not in compiled code)!

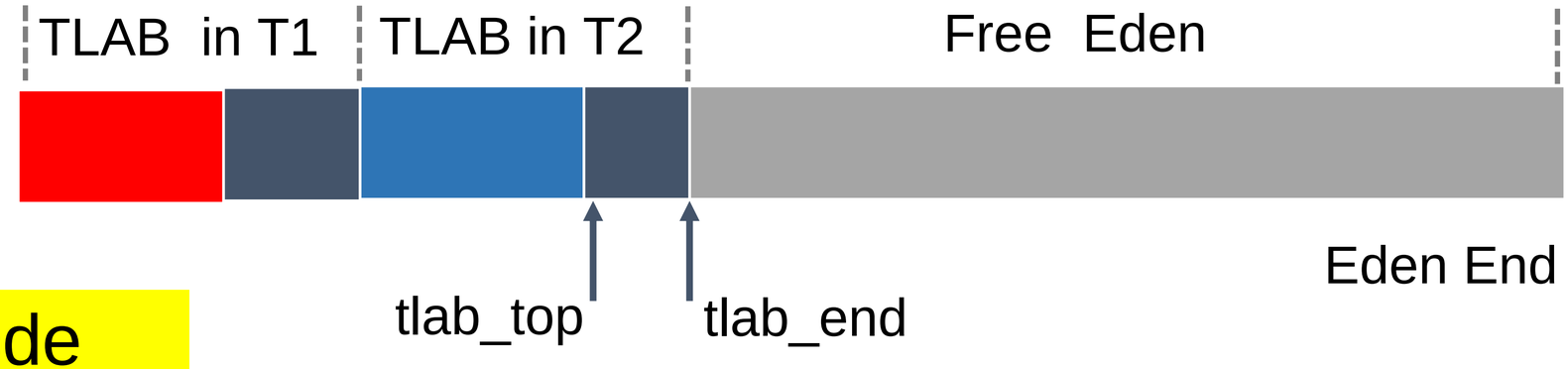
JFR Options

- -XX:+EnableJFR // Enable JFR feature.
- -XX:FlightRecorderOptions // Options for flight recorder
  - sampleobjectallocations // true or false
  - objectallocationssamplinginterval=2048

    // sampling interval, measured by allocation count

```
JFR_OPTS="-XX:+EnableJFR -XX:FlightRecorderOptions=sampleobjectallocations=true,objectallocationssamplinginterval=10"
```

Sample Object Allocation



pseudo code

```
current = tlab_top;
end = current + klass_size;
if (end > tlab_end) {
    goto slow_path;
}
tlab_end = end;
init_obj(obj, end);
jfr_sample_fast_alloc
slow_path:
// synchronize with other threads to get new
tlab
jfr_sample_slow_alloc
```

Do the object allocation sampling both in fast and slow path!

Sampling Implementation in Assembly

```
--- object initialization ---
0x00007faedfa1fd2c: mov $0x7faee47c9180,%r10      ; enable_sample_flag_address = ObjectProfiler::enabled_flag_address();
0x00007faedfa1fd36: mov (%r10),%r11d              ; enable_sample_flag = *enable_sample_flag_address;
0x00007faedfa1fd39: cmp $0x1,%r11d.               ; if enable_sample_flag is true?
0x00007faedfa1fd3d: je 0x00007faedfa1fdc8         ; if true, jump to sample object allocation.
                ......
0x00007faedfa1fdc8: mov $0x1,%r10d                ; alloc_count = 1;
0x00007faedfa1fdce: add 0x160(%r15),%r10          ; alloc_count = alloc_count + thread->trace_data()->alloc_count();
0x00007faedfa1fdd5: mov %r10,0x160(%r15)          ; thread->trace_data()->set_alloc_count(alloc_count);
0x00007faedfa1fddc: mov 0x168(%r15),%r11          ; next_alloc_count_for_sample = thread->trace_data()-> next_alloc_count_for_sample()
0x00007faedfa1fde3: cmp %r10,%r11.                ; check if we should sample current allocation.
0x00007faedfa1fde6: jne 0x00007faedfa1fd43        ; If not, just skip it.
                ......
0x00007faedfa1fdf7: mov $0x7faee40ce1d0,%r10
0x00007faedfa1fe01: callq *%r10                   ; Otherwise,jump to runtime call to fire JFR event.
```

#1: Check if sampling flag is enabled

`sampleobjectallocations`

#2: Check if the current allocation is sampling target

`objectallocationssamplinginterval`

#3: Do the sampling and fire JFR event

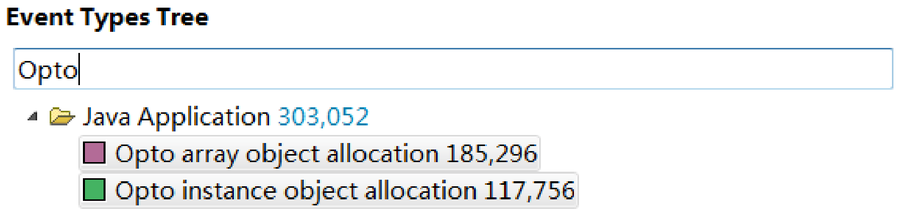
Object Allocation Events Extension

src/share/vm/trace/traceevents.xml

```
<!-- Allocation events -->
    <event id="        OptoInstanceObjectAllocation        " path="java/opto_instance_object_alloc" label="Opto instance object allocation"
description="Allocation by Opto jitted method" has_thread="true" has_stacktrace="true" is_instant="true">
      <value type="CLASS" field="objectClass" label="Object Class" description="Class of allocated instance objec  t"/>
      <value type="ADDRESS" field="address" label="Opto Instance Object Allocation Address" description="Address
  of allocated instance object"/>
    </event>
```

Opto(prefix): only generated in c2 compiled code

```
    <event id="        OptoArrayObjectAllocation        " path="java/opto_array_object_alloc" label="Opto array object allocation  " description="Array
Allocation by Opto jitted method" has_thread="true" has_stacktrace="true" is_instant="true  ">
      <value type="CLASS" field="objectClass" label="Object Class" description="Class of allocated array object"/  >
      <value type="ADDRESS" field="address" label="Opto Array Object Allocation Address" description="Address of
  allocated instance object"/>
      <value type="BYTES64" field="allocationSize" label="Object Size" description="The Array Object Size" />
    </event>
```



Start Time	Event Thread	Event Type	Opto Instance O...	Object Class
8/23/19 12:12:08 ...	Attach Listener	Opto array objec...	0xFFAAF018	char[]
8/23/19 12:12:08 ...	Attach Listener	Opto array objec...	0xFFAAF060	char[]
8/23/19 12:12:08 ...	Attach Listener	Opto array objec...	0xFFAAF130	char[]
8/23/19 12:12:08 ...	Attach Listener	Opto array objec...	0xFFAAF160	char[]
8/23/19 12:12:08 ...	Attach Listener	Opto array objec...	0xFFAAF200	char[]

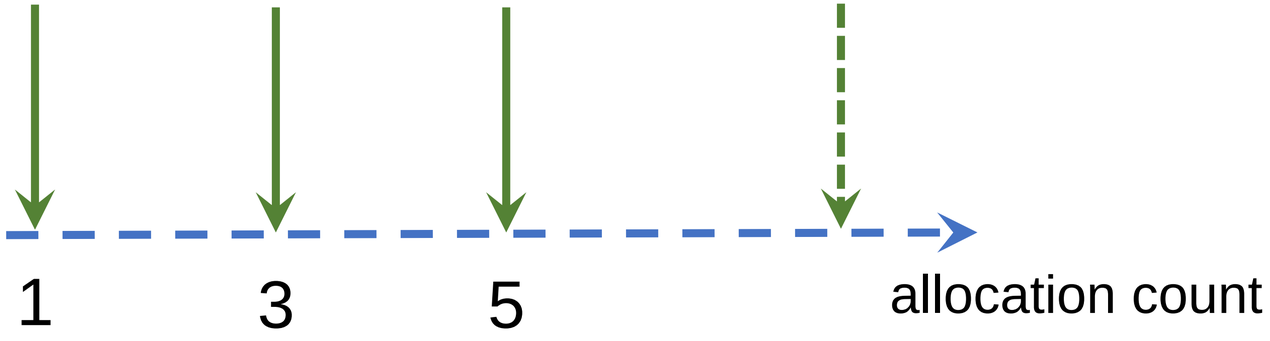
Event Footprint Optimization

- For array objects， size cannot be determined statically
  - array_length is a variant
  - record array size for every sample
- For instance object, object size can be determined from class
  - instanceSize = ((InstanceKlass*) klass)->size_helper() * HeapWordSize
  - Record object size in **class_constants** of JFR binary.

No need to record object size for every instance object event

Sample Biased

```
for (int i = 0; i < 1000; i ++) {
  instance = new Object();
  array = new int[1_000_000];
}
```

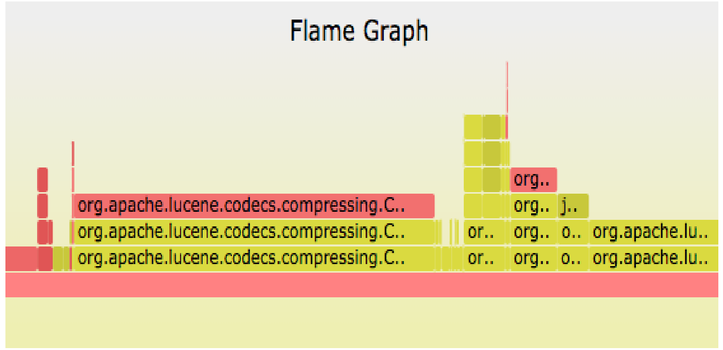


The above array allocation event will be missed if we take ‘2’ as the sampling interval

- Periodicity bias issue
- Consider introducing statistical distributions, e.g., Poisson, in future work

JMC Extension(JMCX)

- JMCX: command line tool to parse JFR event results
- Use the API provided by jmc-core library.
- Usage: jmcx flamegraph



Use memory flame graph to identify the most frequent code-paths accurately

Sampling Overhead

Sampling Overhead

- When ‘sampleobjectallocations’ is false.
  - ✓ Sampling code will not even be generated (at compile time)
  - ✓ Nothing impact on real workload.

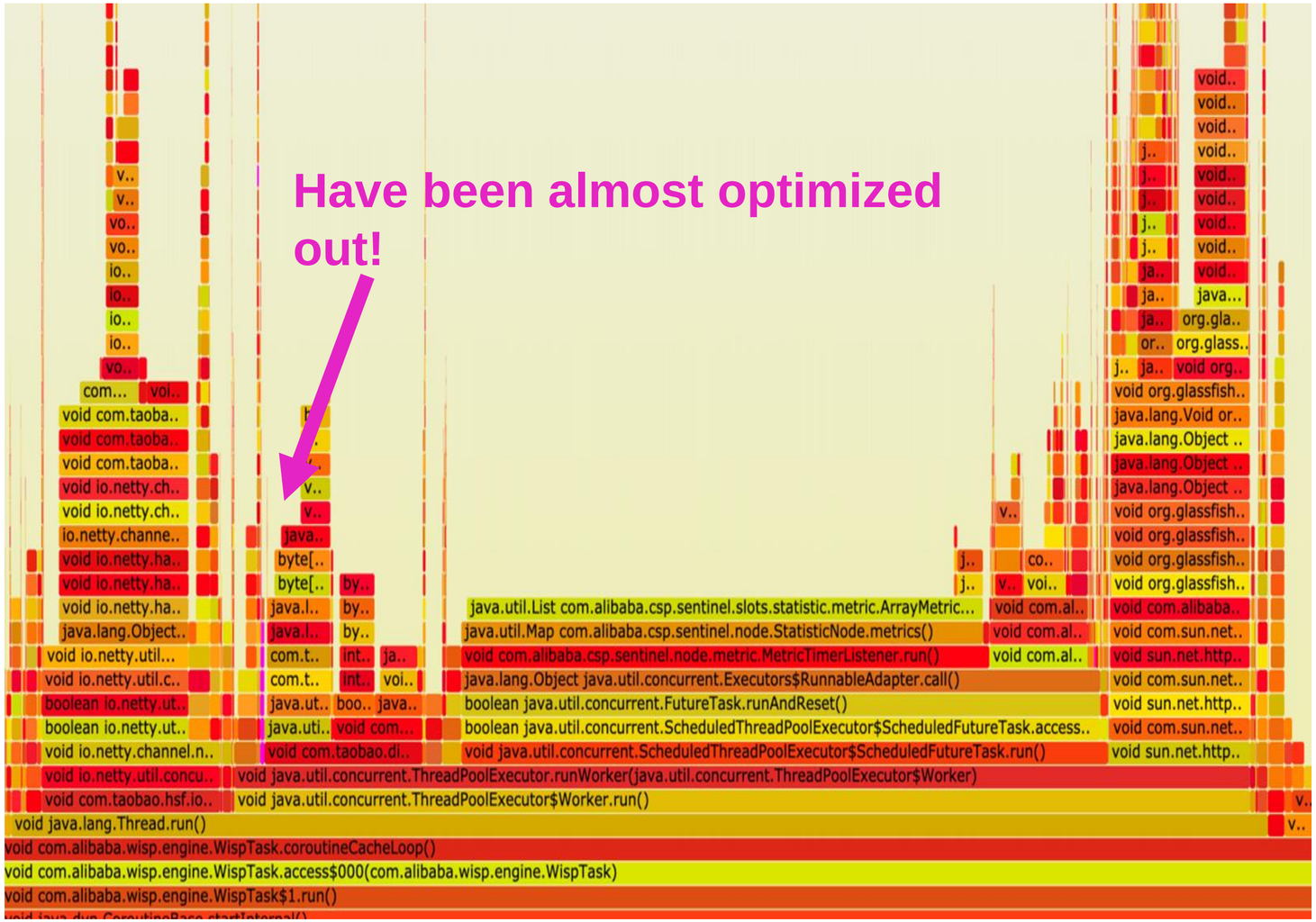
Sampling Overhead

- When ‘sampleobjectallocations’ is false.
  - ✓ Sampling code will not even be generated (at compile time)
  - ✓ Nothing impact on real workload.
- Otherwise
  - ✓ The overhead is dictated by how often it samples object allocation event
    - ✓ ‘objectallocationssamplinginterval’

Case Study: e-commerce Application

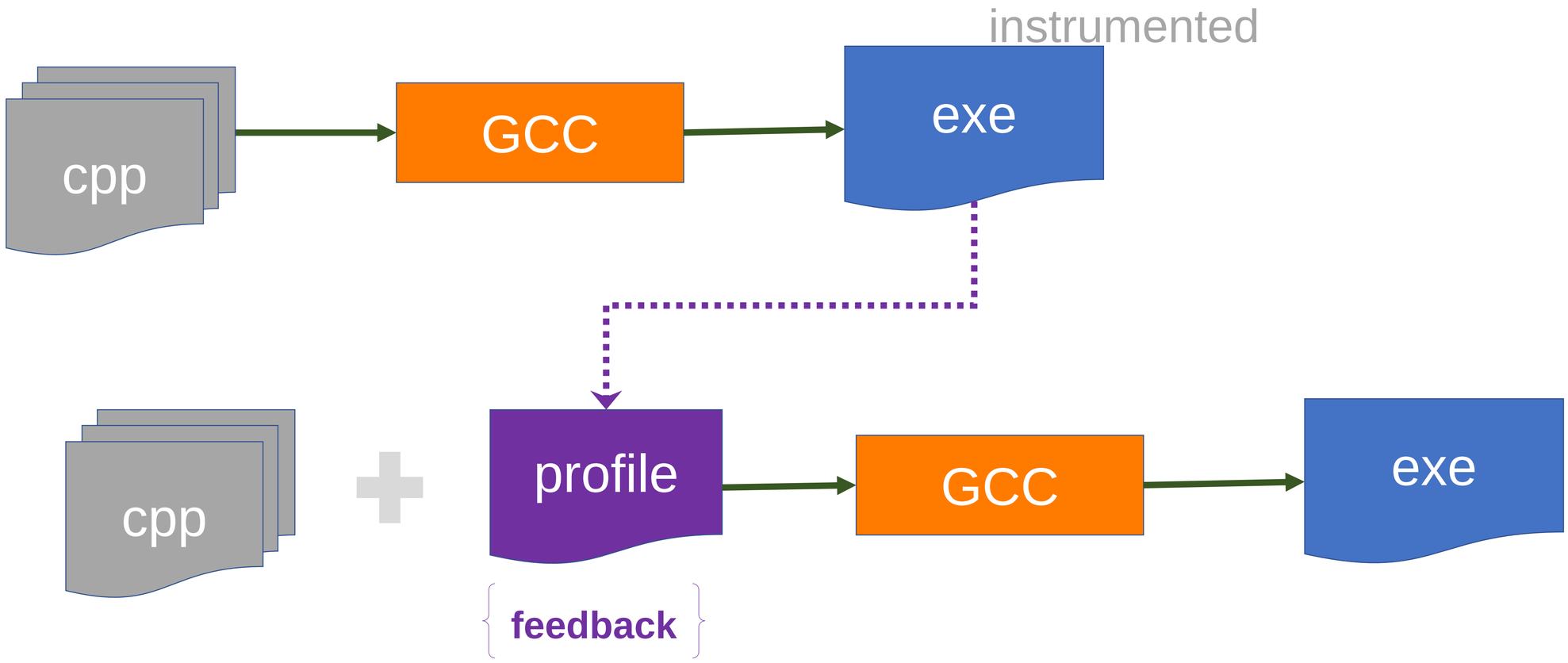
Optimization opportunity:
Most business objects are allocated here

Reduce Object Allocation by Profiling Feedback

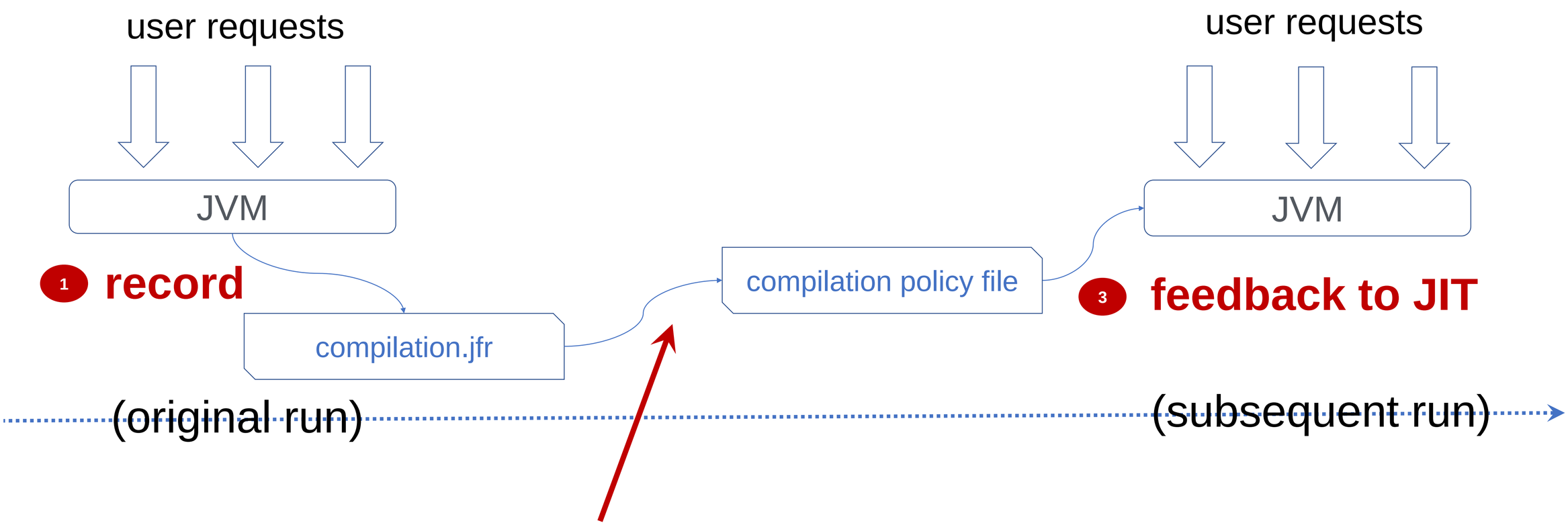


The YGC frequency has been reduced by **~30%**

Traditional ‘Feedback Direct Optimization’ (FDO)



FDO in AlibabaJDK



2 **jmcx createcompilationpolicy** : create compilation policy file from JFR events

Deoptimization Event Extension

src/share/vm/trace/traceevents.xml

```
  <event id="Deoptimization" path="vm/compiler/deoptimization"
label="Deoptimization“ has_thread="false" is_instant="true">
      <value type="STRING" field="className" label="Class Name"/>
      <value type="STRING" field="classloaderName" label="Classloader Name"/>
      <value type="STRING" field="filePath" label="File Path"/>
      <value type="STRING" field="reason" label="Reason"/>
      <value type="STRING" field="methodName" label="Method Name"/>
      <value type="STRING" field="signature" label="Signature"/>
  </event>
```

Compilation Policy File

```
sun/misc/Launcher$AppClassLoader {
  org/spec/jbb/core/generator/CustomerGenerator NULL {
    unstable_if getLocalCustomerIDsForSM (Ljava/lang/String;)Ljava/util/List;
    unstable_if generate (IJ)Ljava/util/List;
    unstable_if getCustomersForHQ (Ljava/lang/String;Lorg/spec/jbb/sm/CustomerSelection$CustomerType;)Ljava/util/List;
  }
  org/spec/jbb/core/collections/HashMultiSet NULL {
    unstable_if count (Ljava/lang/Object;)I
    null_check count (Ljava/lang/Object;)I
    unstable_if update (Ljava/lang/Object;I)V
  }
```

classloader entry

class entry

method name &signature

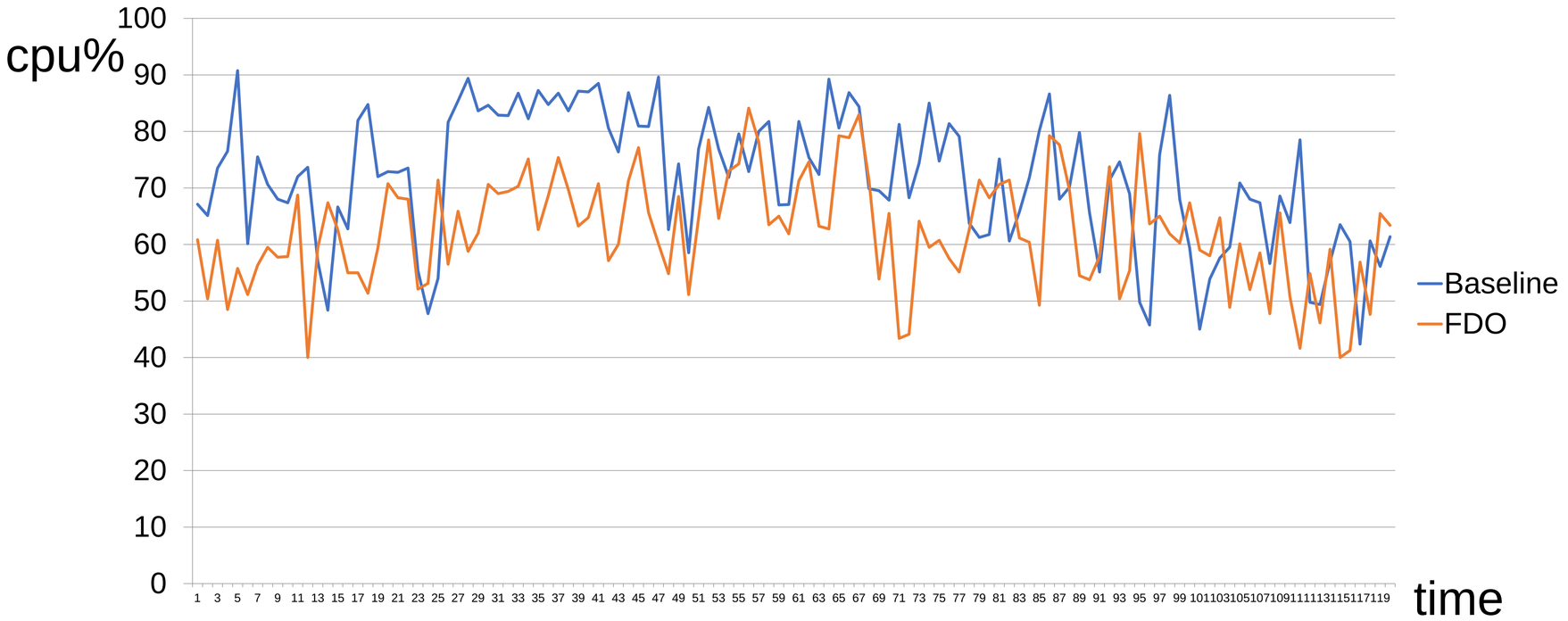
class entry

method item

deoptimization reason

Usage: -XX:+UseFeedbakDirectedOpt -XX:FDOPolicyFile=‘policy file name’

FDO Performance in Production



~10% cpu saving at peak time

- Baseline: normal JIT compilation
- FDO: disable **speculative optimization** based on feedback in previous run

Round-up

- Features of AlibabaJDK covered in this talk
  - ElasticHeap
  - JWarmUp
  - JFR extensions
- JFR extension will come to the next release of Dragonwell (stay tuned!)

Alibaba Cloud | Worldwide Cloud Services Partner

WWW.ALIBABA CLOUD.COM