Alibaba Cloud | Worldwide Cloud Services Partner

How to Statically Compile a Serverless Application to Survive the Biggest Bursty E-Commerce Data Traffic in the World

Ziyi Lin, Yifei Zhang, Wei Kuai, Sanhong Li

Alibaba Group

Alibaba Cloud | Worldwide Cloud Services Partner

Tianxiao Gu

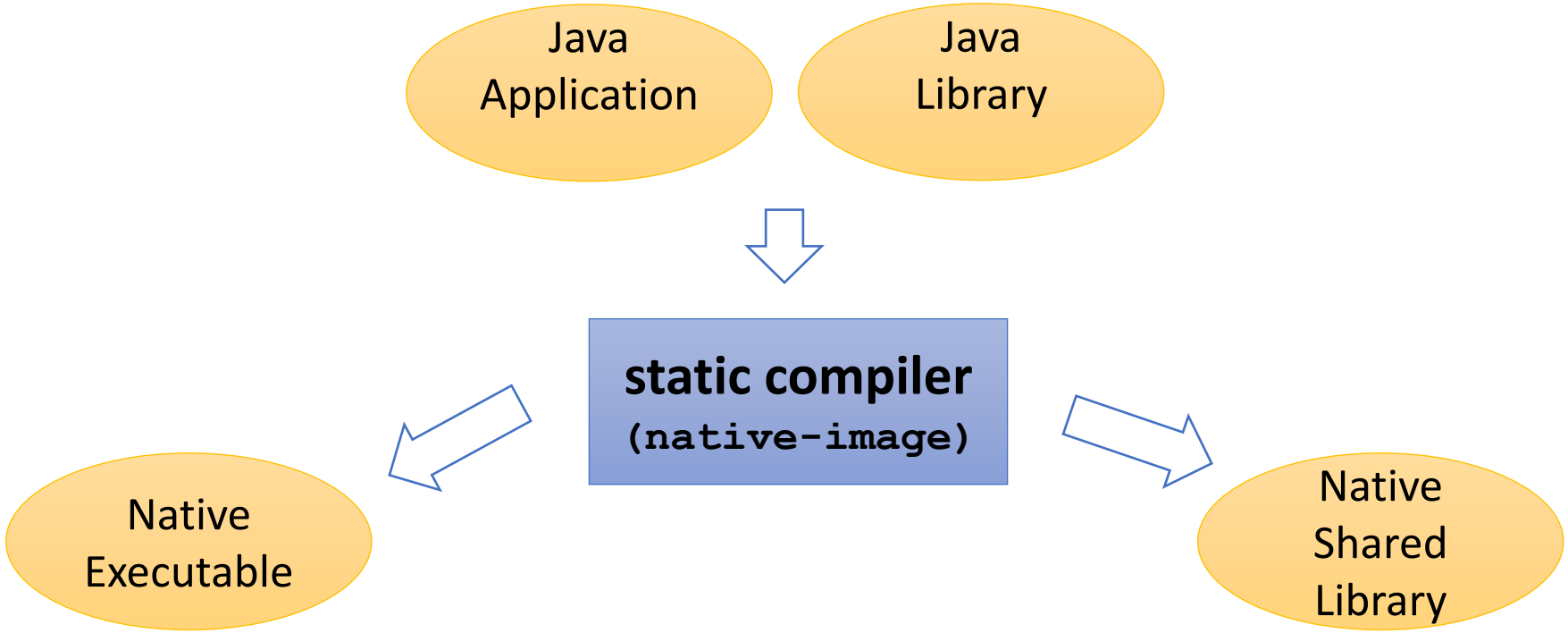
tianxiao.gu@alibaba-inc.com

Alibaba Group, Sunnyvale, CA

Motivations

- Cold startup problem
  - Serverless services require fast scaling up for bursty requests
  - Java applications start up slowly in JVM
- Security problem
  - Java programs are distributed in bytecode and bytecode obfuscation is weak
  - JVM features such as dynamic class loading can be abused by malicious code
- High cost of developing and maintaining projects for multiple programming languages
  - For example, RocketMQ client SDK has various language implementations

Static Compilation of Java Applications



- Benefits:
  - Start up faster
  - More secured

- Benefits:
  - Used as foreign language
  - More secured

Native Image Generator and Substrate VM

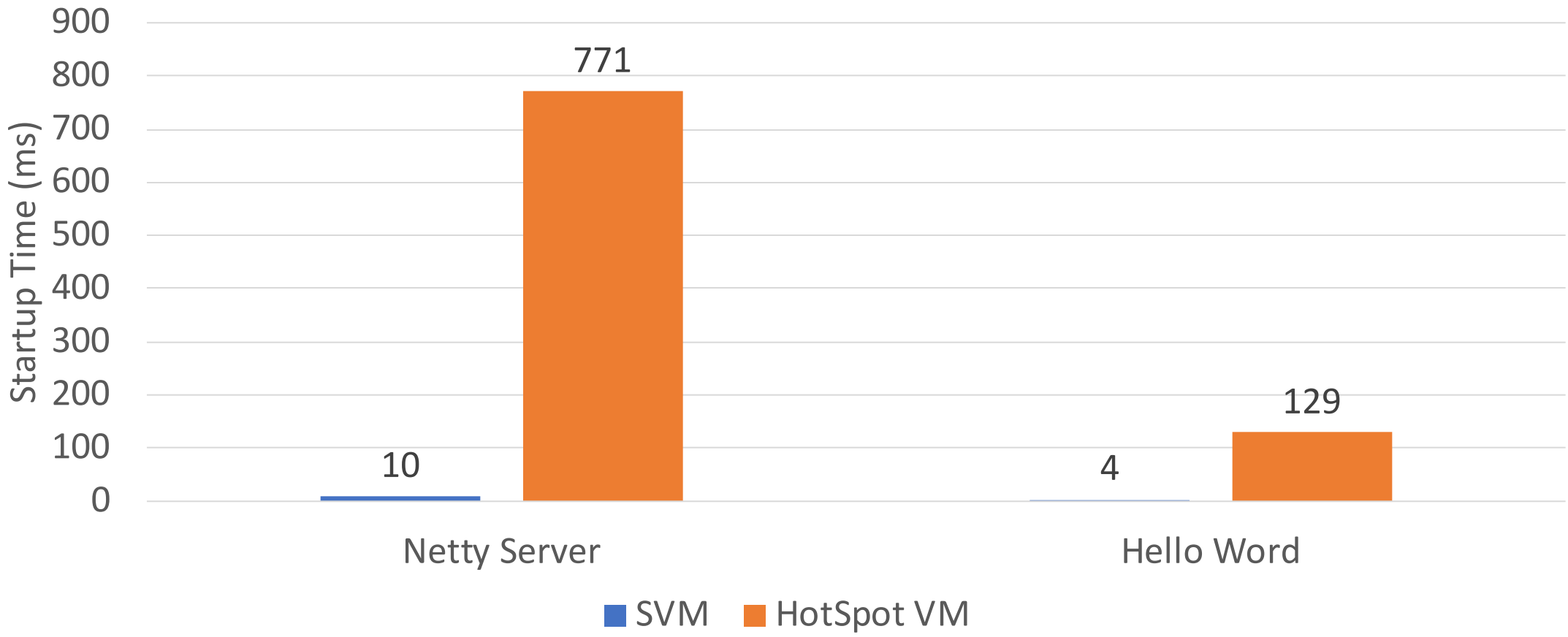
GraalVM Native Image allows you to ahead-of-time compile Java code to a standalone executable, called a native image. This executable includes the application classes, classes from its dependencies, runtime library classes from JDK and statically linked native code from JDK. It does not run on the Java VM, but includes **necessary components like memory management and thread scheduling from a different virtual machine, called "Substrate VM".** Substrate VM is the name for the runtime components (like the deoptimizer, garbage collector, thread scheduling etc.). The resulting program has **faster startup time and lower runtime memory overhead** compared to a Java VM.

```
$ native-image -cp ..  Main
$ ls
main
```

Use SVM to refer to the static compilation technique in this talk

https://www.graalvm.org/docs/reference-manual/native-image/

Simple Demo



Looks perfect for the cold startup problem!

Concerns in Practice

Scalability	• Can SVM scale to large real-world applications?
Adaptation Difficulty	• How much effort do we need to run a JVM-based application in SVM?
Stability	• Will the SVM-based application run as stable as the JVM-based one?
Performance	• What startup time can we gain after static compilation? Is it worth the effort? • Is it possible to gain fast startup without sacrificing too much peak performance?

Evaluation

Hardware

- Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz
- 512 GB memory
- 1 TB SSD

Software

- AliOS 7.2 (Paladin)
- OpenJDK 1.8.0-222
- GraalVM **CE** 19.2.0

Evaluation (cont.)

Subjects

- Three ***Sofa-boot Applications*** named ***Sofa-Small, Sofa-Mid, and Sofa-Large*** in this talk
  - Built on the **Sofa-boot framework**, Tomcat, Netty, Hessian and many other Ant Financial's middleware
- Big enough for SVM: 33 MB, 56 MB and 123 MB, respectively
  - Each application is packed into a fat JAR, including all libraries

Steps

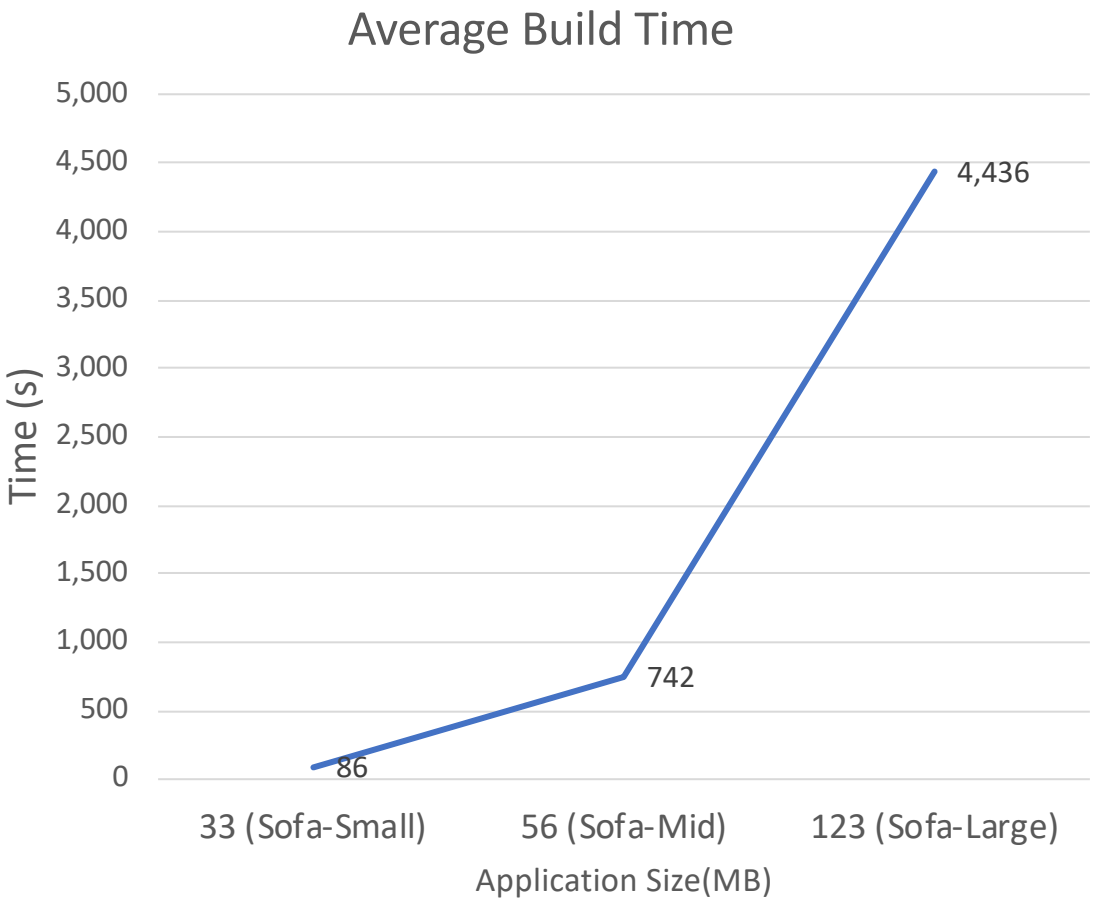
- Statically compile the application using SVM
- Run and diagnose the binary in testing environment before deploying in production environment

Brief Results of Sofa-Large

- Costed 25 man-months
- Deployed in Alibaba Group's production environment
- Used in Double 11 (Single's Day) online shopping festival, 2019

Scalability

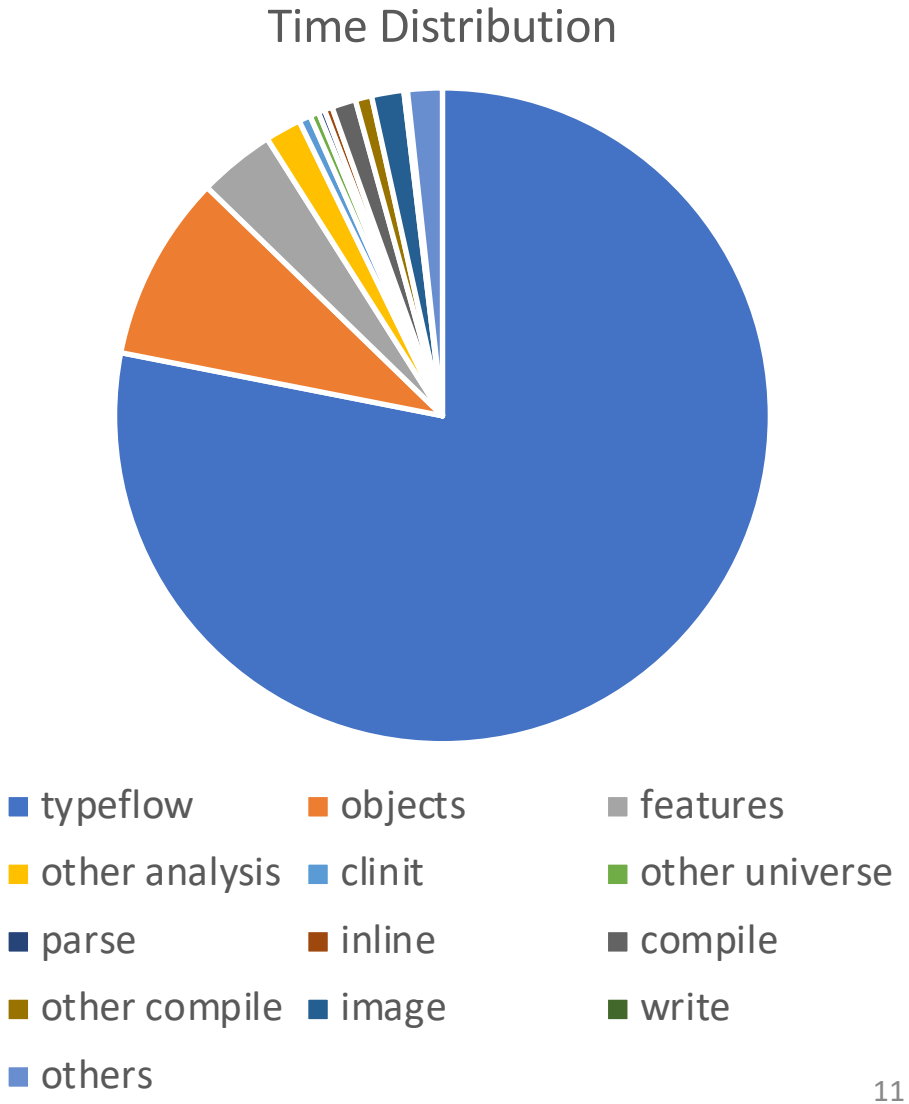
- Build time increases significantly
- Why do we care about build time?
  - Quickly verify results and debug ***at develop time***
  - Promptly response to system failures and user actions ***in production***
- Current build time is unacceptable for large applications in Alibaba



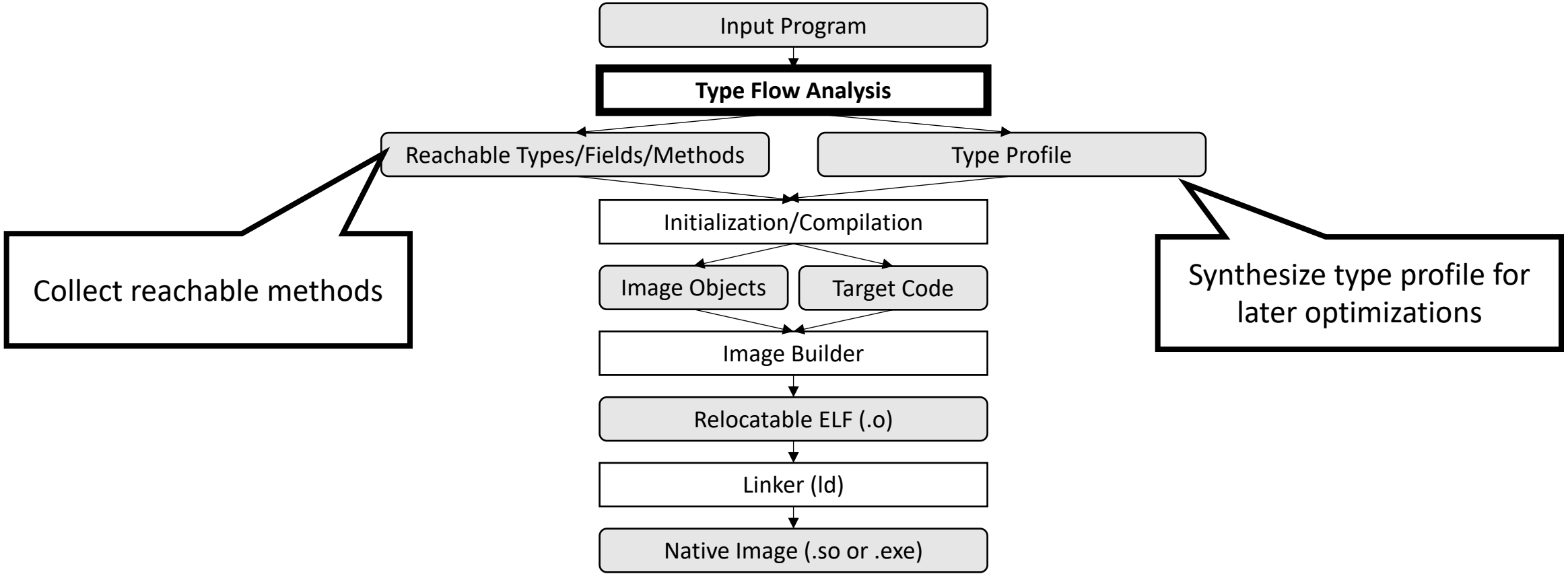
Type Flow Analysis is the Most Expensive

- Sofa-Large: a 123 MB fat JAR
- 120 GB memory, 3,724 seconds

```
...
[sofabootapplication:80081]    (typeflow): 2,907,607.83 ms
[sofabootapplication:80081]     (objects):   341,685.73 ms
[sofabootapplication:80081]    (features):   140,063.05 ms
[sofabootapplication:80081]      analysis: 3,455,448.55 ms
[sofabootapplication:80081]      (clinit):    21,791.25 ms
[sofabootapplication:80081]      universe:    38,116.97 ms
[sofabootapplication:80081]       (parse):    11,899.39 ms
[sofabootapplication:80081]      (inline):    14,506.99 ms
[sofabootapplication:80081]     (compile):    44,150.99 ms
[sofabootapplication:80081]       compile:   100,575.20 ms
[sofabootapplication:80081]         image:    59,238.25 ms
[sofabootapplication:80081]         write:     6,632.67 ms
[sofabootapplication:80081]       [total]: 3,723,600.94 ms
```

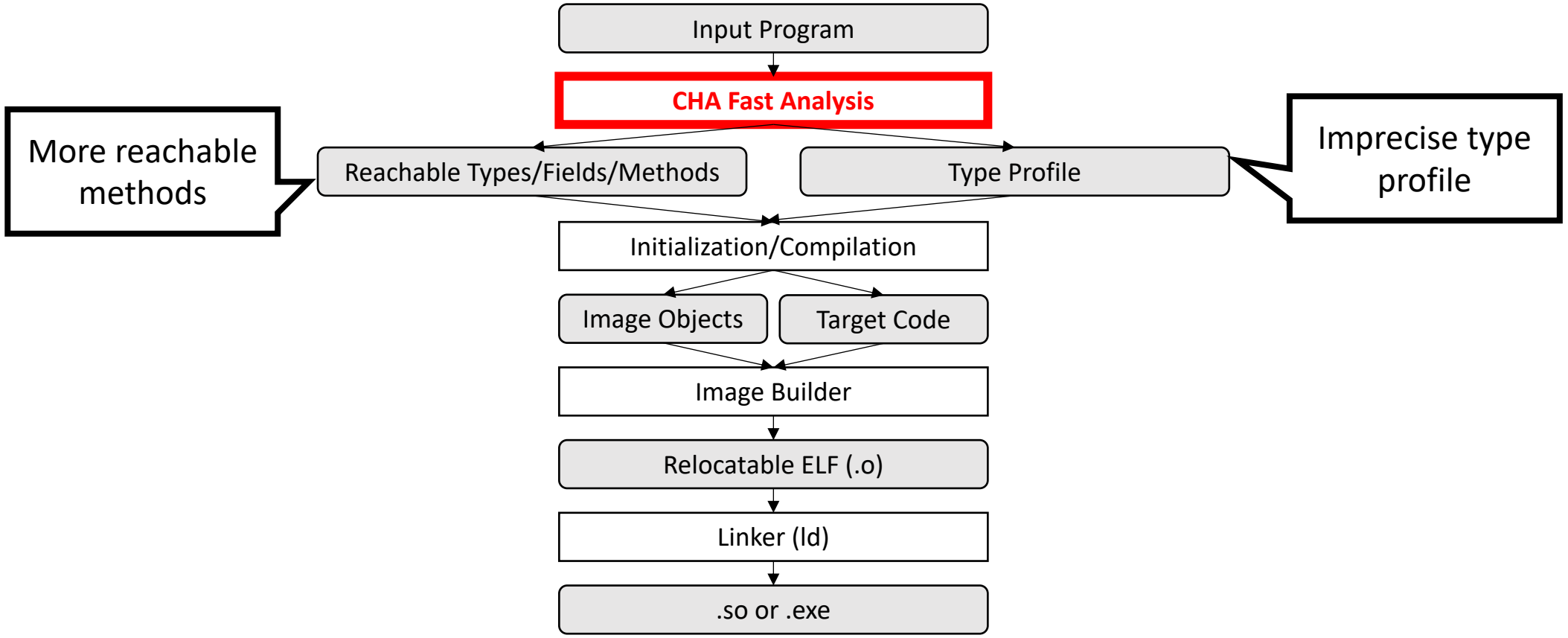


Native Image Generator — Overview



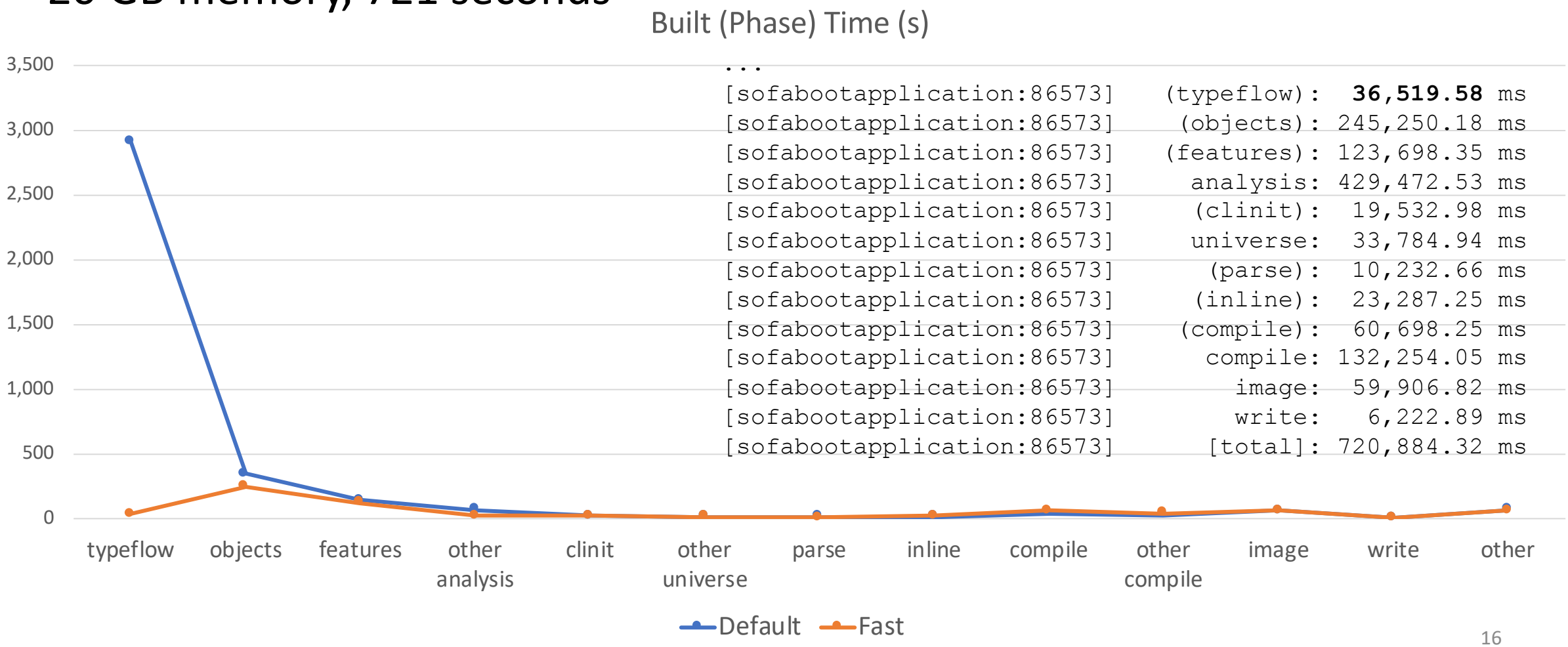
Fast Analysis

- Use Class Hierarchy Analysis (CHA) to determine the type profile



Fast Analysis: Results

- 20 GB memory, 721 seconds

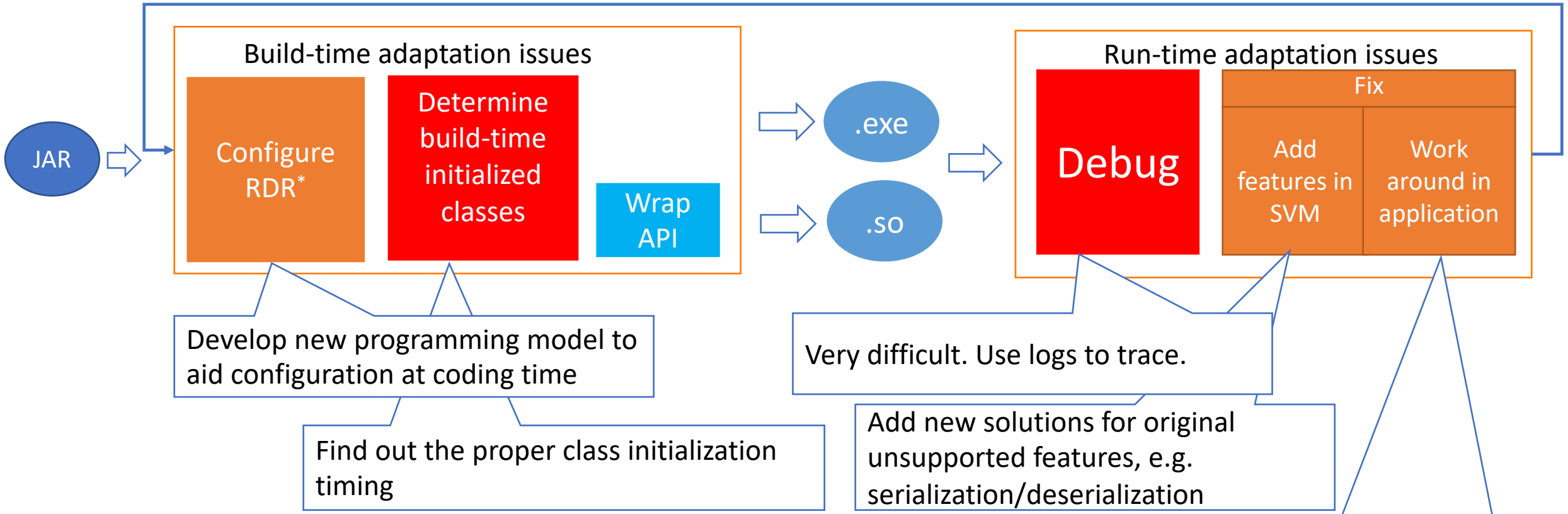



```
...
[sofabootapplication:86573]    (typeflow):  36,519.58 ms
[sofabootapplication:86573]     (objects): 245,250.18 ms
[sofabootapplication:86573]    (features): 123,698.35 ms
[sofabootapplication:86573]      analysis: 429,472.53 ms
[sofabootapplication:86573]      (clinit):  19,532.98 ms
[sofabootapplication:86573]      universe:  33,784.94 ms
[sofabootapplication:86573]       (parse):  10,232.66 ms
[sofabootapplication:86573]      (inline):  23,287.25 ms
[sofabootapplication:86573]     (compile):  60,698.25 ms
[sofabootapplication:86573]       compile: 132,254.05 ms
[sofabootapplication:86573]         image:  59,906.82 ms
[sofabootapplication:86573]         write:   6,222.89 ms
[sofabootapplication:86573]       [total]: 720,884.32 ms
```

Fast Analysis: Summary

- Pros:
  - Fast for debugging large applications
- Cons:
  - Not fast for small programs because more methods are included
  - Unavoidable runtime performance degradation due to imprecise analysis results
- The correctness is still under validation
  - So far so good
  - Still only deployed in testing environment

Adaptation Difficulty



*RDR: Reflection/Dynamic proxy/Resources

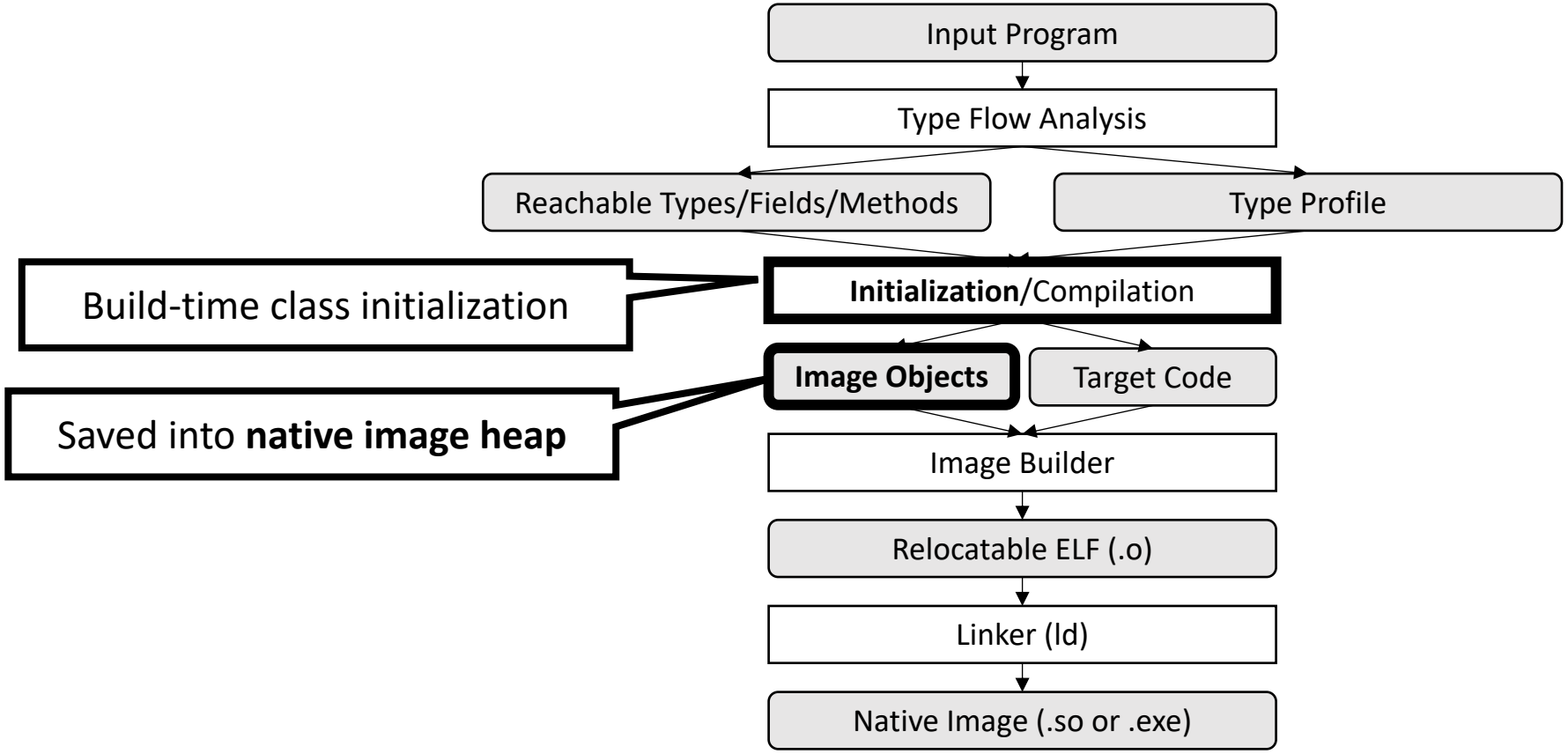
Configuring RDR

- native-image-agent* can run the program and generate configurations
  - convenient but has the coverage problem
- Define new annotations in Alibaba Dragonwell JDK to aid coding time configurations of RDR
  - Annotations must be added where a reflection call is made
  - Every reflection target must be specified in annotations
  - The **`javac`** compiler issues compilation errors for missing annotations
    - Help find missing configurations at coding time

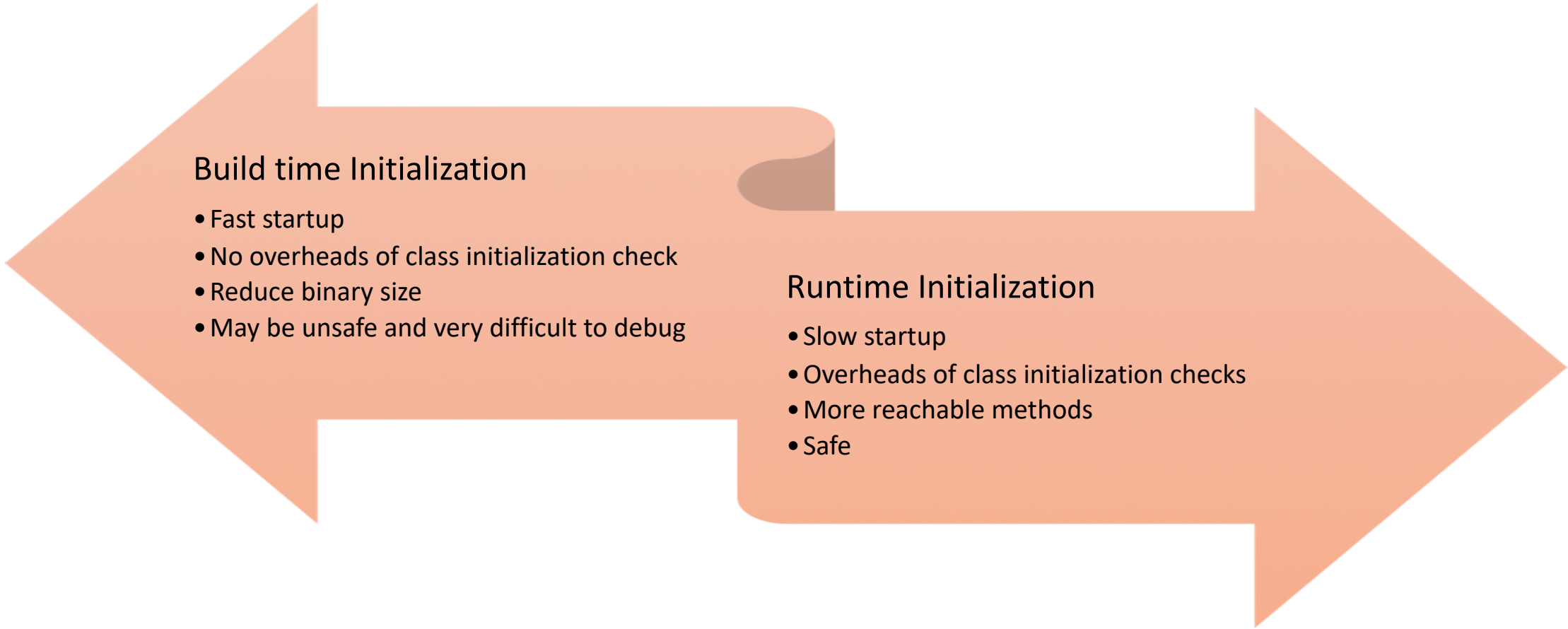
```
@ContainReflection(method="subString",paramTypes={"int"})
Method m = String.class.getDeclaredMethod(name, int.class);
```

* https://github.com/oracle/graal/blob/master/substratevm/CONFIGURE.md

Build-Time Class Initialization



Build-Time Class Initialization (cont.)



Build-Time Class Initialization (cont.)

- Manual configuration
  - Specify a **whitelist** of build-time initialized classes
  - Specify a **blacklist** of build-time initialized classes
    - A **whitelist** of run-time initialized classes
- More classes are transitively initialized at build-time
  - All super classes
  - All accessed classes during executing the class initializer at build-time

A 100% CPU Occupied Issue

sun.nio.ch. EPollArrayWrapper.epollWait(long pollAddress, int numfds, long timeout, int epfd)

0

io.netty.util.concurrent.ScheduledFutureTask.delayNanos:

```
public long delayNanos(long currentTimeNanos) {
    return Math.max(0, deadlineNanos() - (currentTimeNanos - START_TIME));
}
```

An incorrect value can result 0

```
final class ScheduledFutureTask<V> extends PromiseTask<V> implements ScheduledFuture<V>, PriorityQueueNode {
    private static final AtomicLong nextTaskId = new AtomicLong();
    private static final long START_TIME = System.nanoTime();
```

Should be initialized at runtime
but is mistakenly initialized at build time

```
public final class GlobalEventExecutor extends AbstractScheduledEventExecutor implements OrderedEventExecutor {

kQueue(): void tatic final InternalLogger logger = InternalLoggerFactory.getInstance(GlobalEventExecutor.class);

    private static final long SCHEDULE_QUIET_PERIOD_INTERVAL = TimeUnit.SECONDS.toNanos( duration: 1);

    public static final GlobalEventExecutor INSTANCE = new GlobalEventExecutor();

    final BlockingQueue<Runnable> taskQueue = new LinkedBlockingQueue<~>();
    final ScheduledFutureTask<Void> quietPeriodTask = new ScheduledFutureTask<~>(
```

Transitively initialized at build time

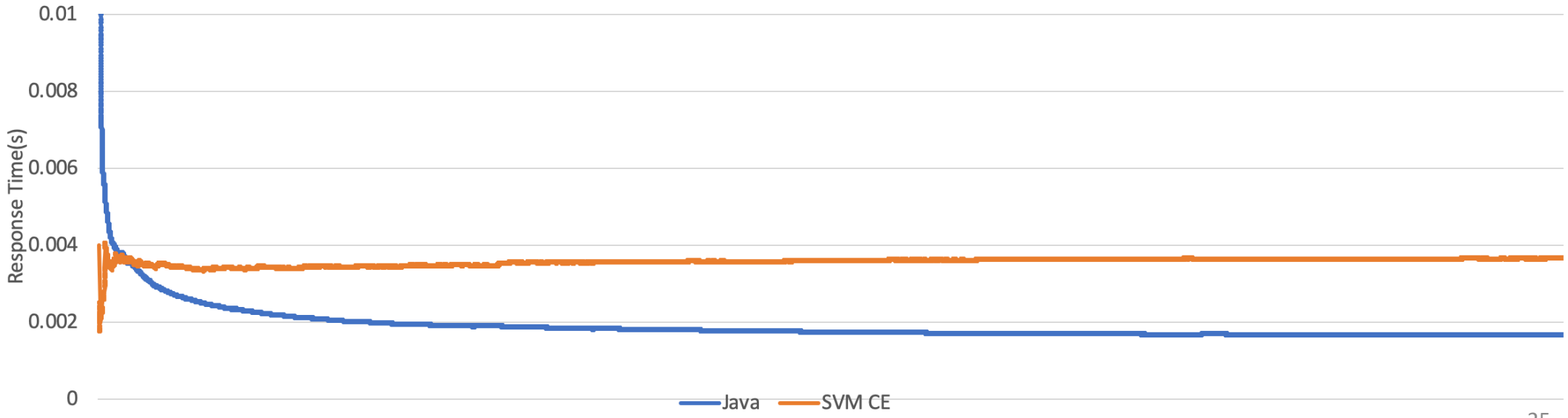
io.netty.util.concurrent.GlobalEventExecutor was configured to be initialized at build time in **spring-boot-graal-feature**(https://github.com/aclement/spring-boot-graal-feature) which is now moved to **spring-graal-native** (https://github.com/spring-projects-experimental/spring-graal-native)

Trace Transitive Initializations

- SVM has provided **-H:+TraceClassInitialization**
  - Allows to trace the class initialization via bytecode instrumentation
  - Unable to instrument all classes, e.g., some dynamically generated classes
- We further modified Hotspot VM to trace all initializations
  - Observe the process of class initializations
    - i.e., the creation and initialization of ***InstanceKlass*** instances by the HotSpot VM
  - Print the stack trace of class initializations
  - Locate the root class of the chain of class initializations

Runtime Performance

- Peak performance of SVM is only half of Hotspot VM on Sofa-Small
- Improve performance by two approaches:
  - Improve GC
  - Reduce safepoint checks



GC Problems

- We observed that
  - The full GC occurred more frequently than the HotSpot VM
  - Some GC operations were unexpected long
    - Releasing unaligned chunks
    - Full native image heap scanning

	HotSpot VM	SVM
#GC	794	3,617
#Full GC	0	367

Data were collected by sending 60,000 curl requests to drive Sofa-Small

GC Improvements

- Add multi-survivor to reduce the frequency of full GC.
- Asynchronously release unaligned chunks
- Card table based incremental scanning of the native image heap

Results of GC Improvements

	Original SVM GC	Improved SVM GC
#Full GC	29	1
Average Pause (s)	0.798	0.112

Data were collected by running Sofa-Mid

Reduce Safepoint Checks

- SVM adds more safepoint checks than HotSpot VM
- Try to add safepoint checks using the same strategy as JIT in the HotSpot VM
- About 15% performance improvement

```
public void demo() {
    for (int i = 0; i < 1000; i++) {
        sum = A[i] + B[i];
        Safepoint
    }
    Safepoint
    …
}
```

Summary

- Can SVM scale to large real-world applications?
  - Not really.
  - For Sofa-Large, SVM compiles 327,372 methods from 65,834 types. The input fat jar is 123 MB, and the output binary executable is 500 MB. It takes on average 4,436 seconds on a server with Intel(R) Xeon(R) Platinum 8163 CPU @ 2.50GHz with 120 GB memory to complete the compilation. It consumes too much time and resources for compilation.
  - Fast analysis can significantly reduce compiling costs, but its correctness and performance have not been evaluated thoroughly

Summary (cont.)

- How much effort do we need to run a JVM-based application in SVM?
  - 25 man-months for Sofa-Large
- Will the SVM-based application run as stable as the JVM-based one?
  - Yes. Sofa-Large has survived the traffic of double 11 online shopping festival

Summary (cont.)

- What startup time can we gain after static compilation? Is it worth the effort?
  - The startup time of Sofa-Large has been reduced from 60 seconds to 3 seconds.
- Is it possible to gain fast startup without sacrificing too much peak performance?
  - Yes. With improvements on GC and safepoint checks, the peak performance of Sofa-Small is nearly the same as Hotspot VM.

Future Work

- Ensure stability
  - Issues caused by unsupported JVM features such as RDR, build-time class initializations and dynamic class loading
- Reduce output binary size
  - The output of Sofa-Large is 500 MB.
    - Larger than the total size of the JAR of Sofa-Large and the JDK
- Reduce adaption efforts
  - More automated tools aiding adaptation

Alibaba Cloud | Worldwide Cloud Services Partner

Thanks
Q&A

Ziyi Lin, Yifei Zhang, Wei Kuai, Sanhong Li

cengfeng.lzy@alibaba-inc.com, lingyue.zyf@alibaba-inc.com,

kuaiwei.kw@alibaba-inc.com, sanhong.lsh@alibaba-inc.com