Java企业应用
-性能优化原则, 方法与策略

阿里巴巴　李三红

自我介绍

- 2014年 加入蚂蚁金服， 目前在阿里基础架构事业群， 基础软件部门：
  - 开发基于OpenJDK阿里定制版本：**AJDK**
  - 开发性能故障诊断工具： ZProfiler， ZDebugger， PIPA

- 联系方式

  mail: sanhong.lsh@alibaba-inc.com

  微博: sanhong_li

AJDK

Performance Analysis

Outline

1. Performance Basics and Methodology
2. Fundamentals of Performance Tuning
    - Profiling Driven Optimization
    - JVM Tuning
        - GC
        - JIT
3. Optimization Strategy for JavaEE
4. Recap

Recall Little‘s law

$$L = \lambda * W$$

In queueing theory, the long-term average number of customers in a stable system, L， is equal to the long-term average arrival rate, λ, multiplied by the average time a customer spends in the system, W

source: https://en.wikipedia.org/wiki/Little%27s_law

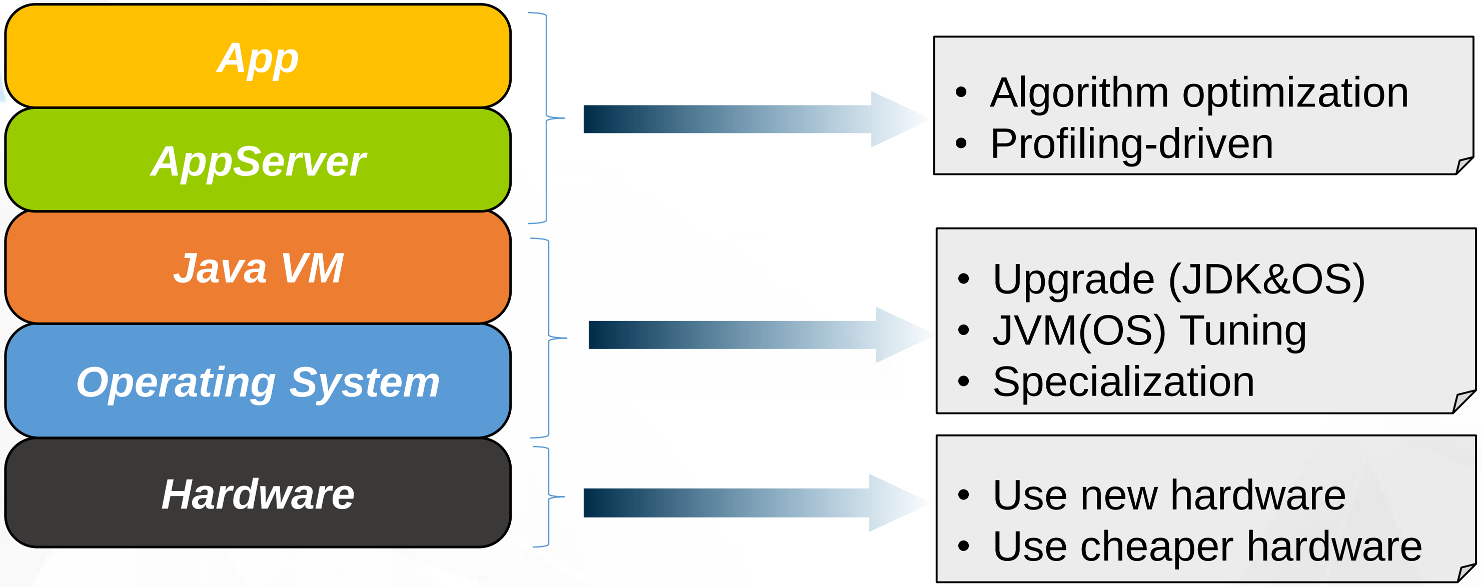
Throughput and RT

$$\mathit{MeanNumberInSystem} = \mathit{MeanThroughput} * \mathit{MeanResponseTime}$$

- Throughput and RT are related
  - ✓ Decreasing RT ”almost always” improves Throughput
  - ✓ Throughput improving doesn’t necessarily mean RT decreasing
- Performance tuning and cost saving
  - ✓ More higher throughput/lower RT **but without** adding new hardware

source: https://en.wikipedia.org/wiki/Little%27s_law

Approaches to Performance



App

AppServer

Java VM

Operating System

Hardware

- Algorithm optimization
- Profiling-driven

- Upgrade (JDK&OS)
- JVM(OS) Tuning
- Specialization

- Use new hardware
- Use cheaper hardware

Java’s view

Approaches:

a) Outside in approach(performance baseline)

b) Layered approach(“Bottom up” or “Top down”)

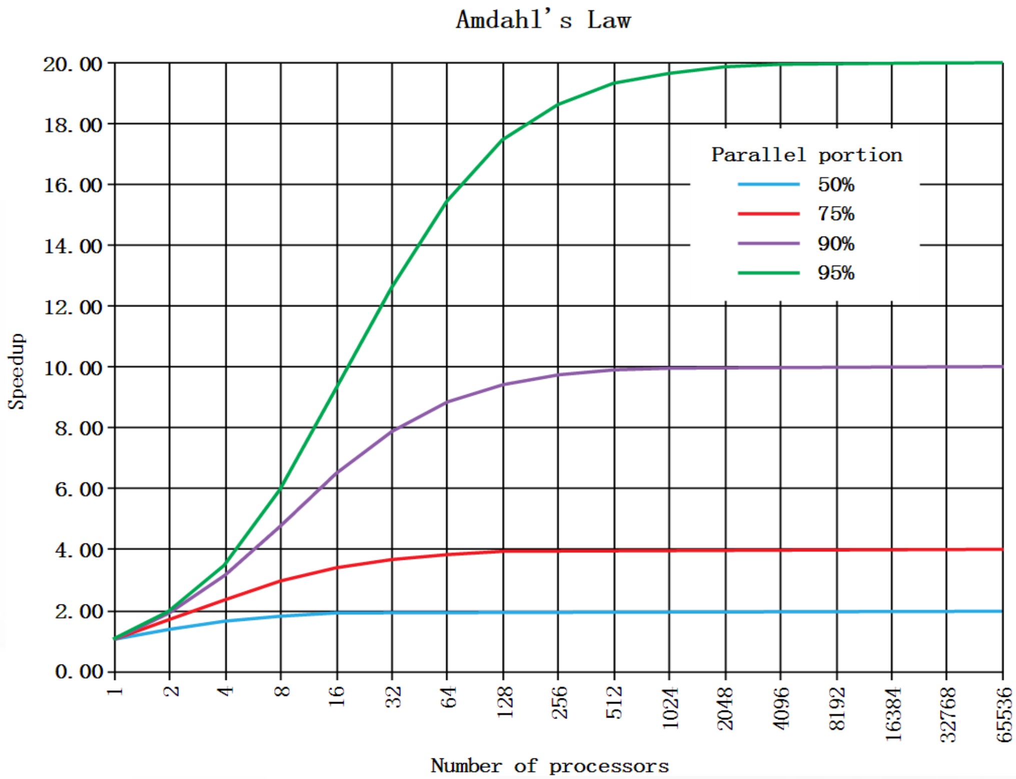
c) A hybrid of both a), b)

Amdahl's Scaling Law

$$\text{Speedup} = \frac{1}{F + \frac{1-F}{N}}$$

F: is fraction of work that is serial

N: is number of threads



source: https://en.wikipedia.org/wiki/Amdahl%27s_law

- Reduce the amount of serial work performed

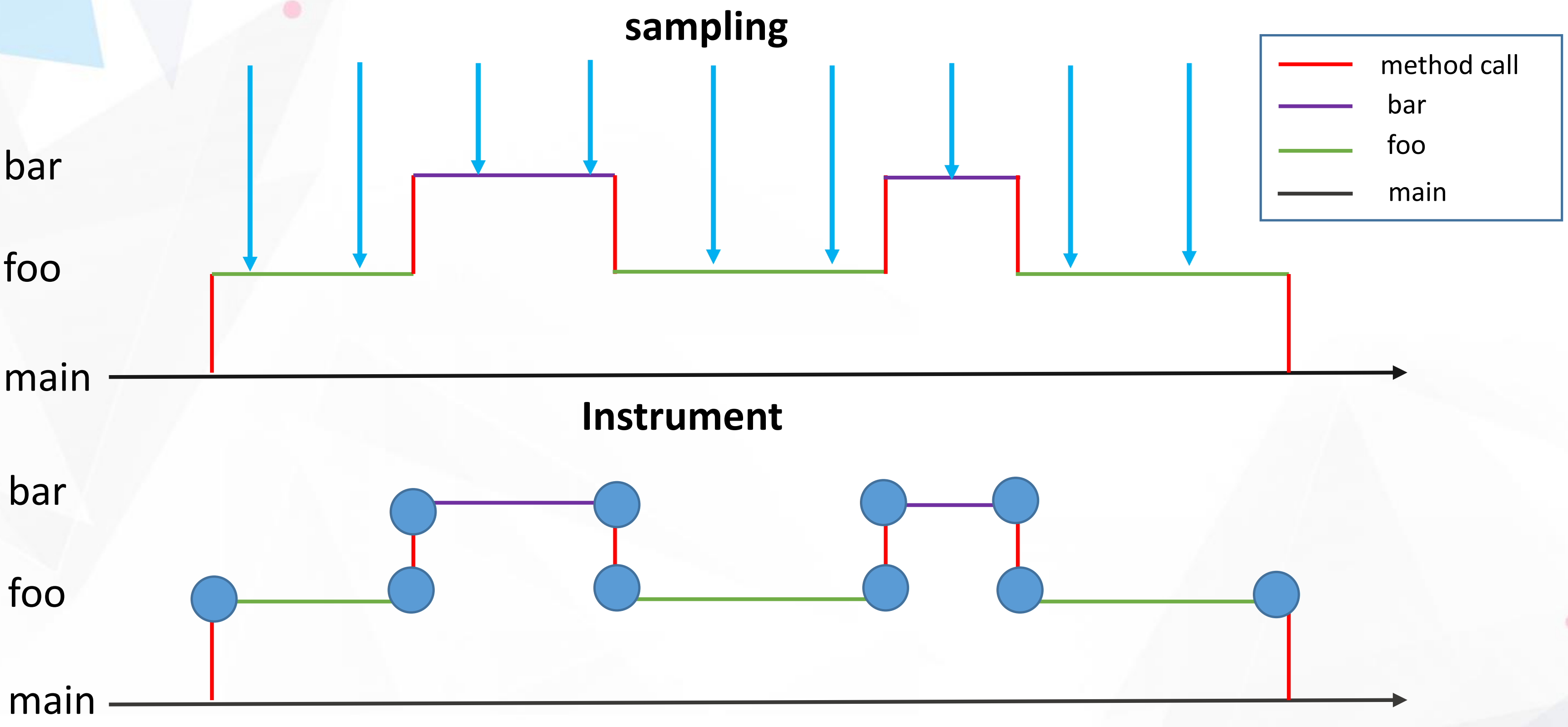
Costs Reduce Scaling

1. Potential contributors to F:
   - Synchronization(synchronized&j.u.c.Lock)
     - data structures need to be thread safe
     - communication overhead between threads
   - Infamous “stop the world” (aka STW) in JVM

- JConsole (MXBean)
- Java Mission Control
- JProfiler
- HealthCenter&jucProfiler

2. Cost incurred when the N gets increased
   - Thread context switch

Profiling: Sampling vs Instrument



Available Technology:
BCI, JVMTi, javax.management, System.currentTimeMillis()

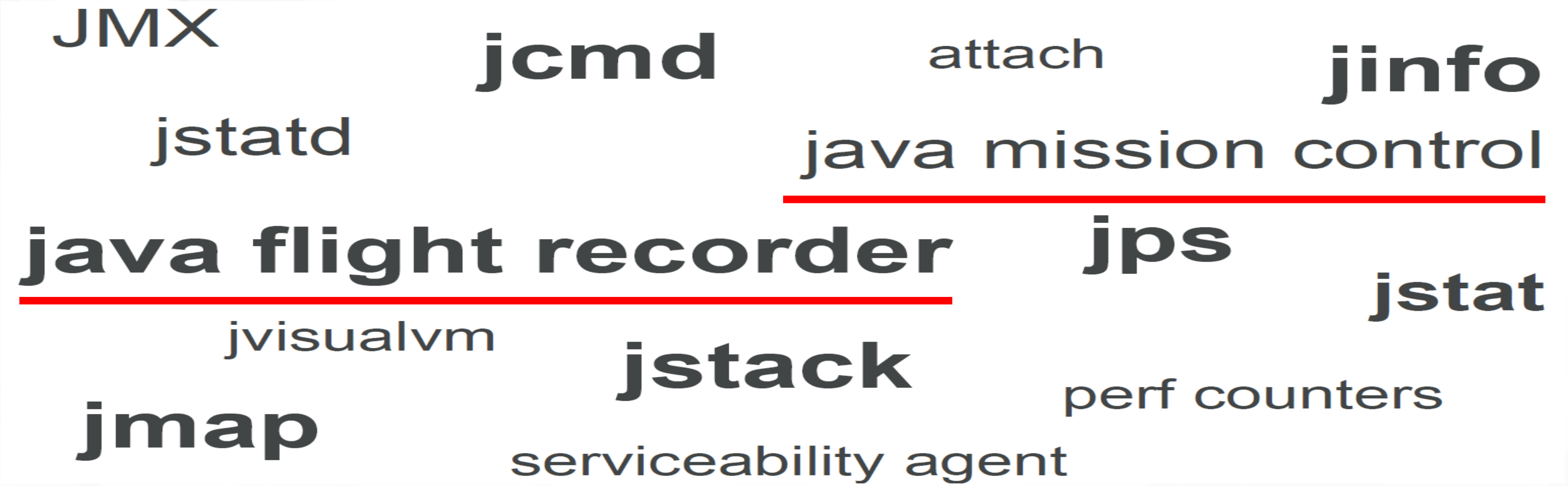
Sampling vs Instrument

- Sampling
  - ✓ Lower overhead (determined by sampling interval)
  - ✓ Discover unknown code
  - ✓ Non intrusive
  - ✓ No execution path
  - ✓ Periodicity Bias
- Instrument
  - ✓ Wall time (estimate IO time)
  - ✓ Full execution path
  - ✓ Configuration on what methods to instrument
  - ✓ Generally more data to be collected

Safepoint Bias

- Stack trace sampling happens only when the given thread at a *safepoint*
  - ✓ The hot loop may not get profiled anymore
- Use following tools instead
  - ✓ Java Mission Control
  - ✓ Honest Profiler(githup)
  - ✓ ZProfiler(alipay internal profiling tool)

Tools for Diagnostics



JMX

jcmd

attach

jinfo

jstatd

java mission control

java flight recorder

jps

jstat

jvisualvm

jstack

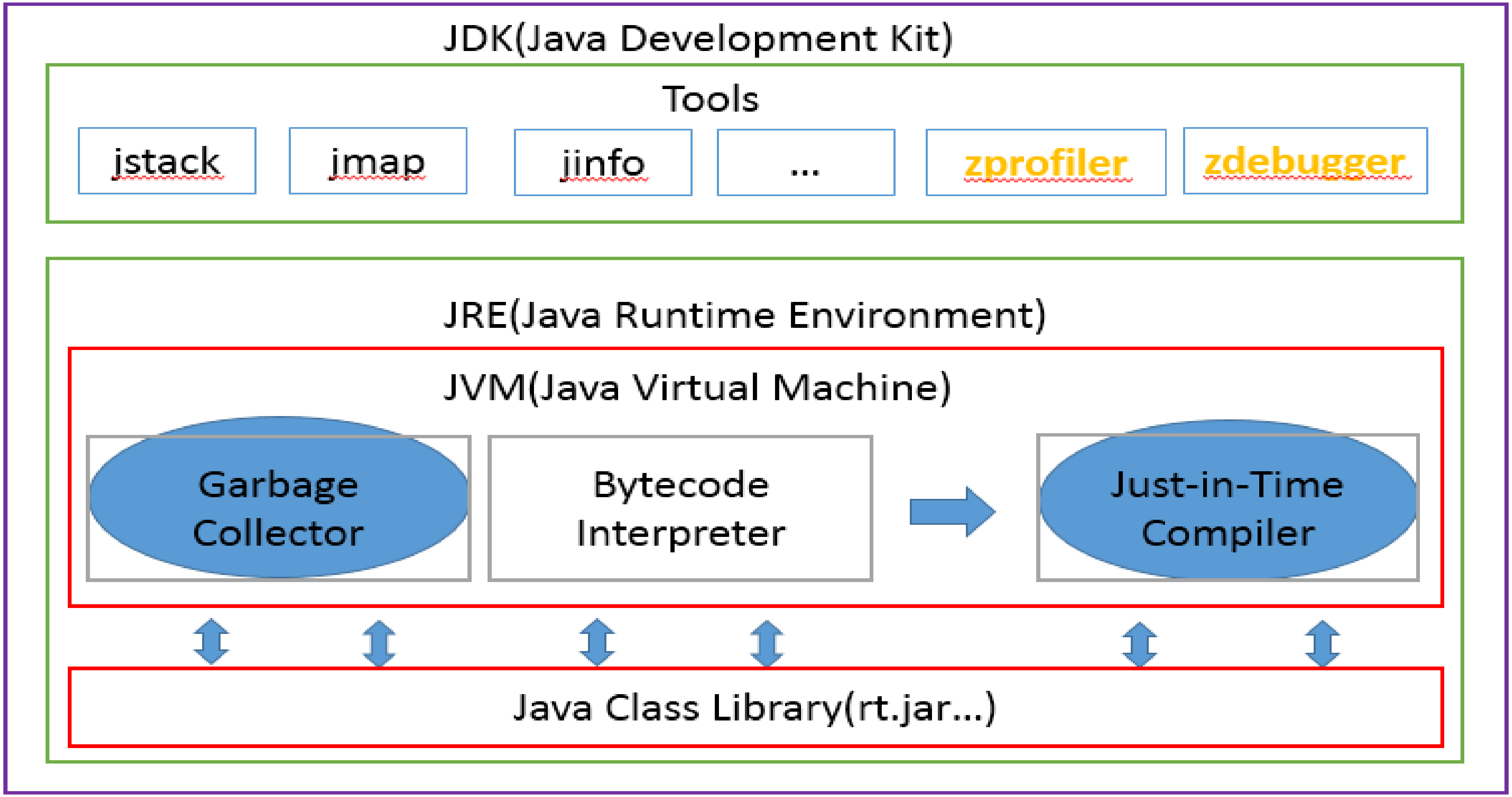
perf counters

jmap

serviceability agent

- Most of them could be found in JAVA_HOEM/bin
- Good reference: Troubleshooting Guide for JavaSE 6 with HotSpotVM

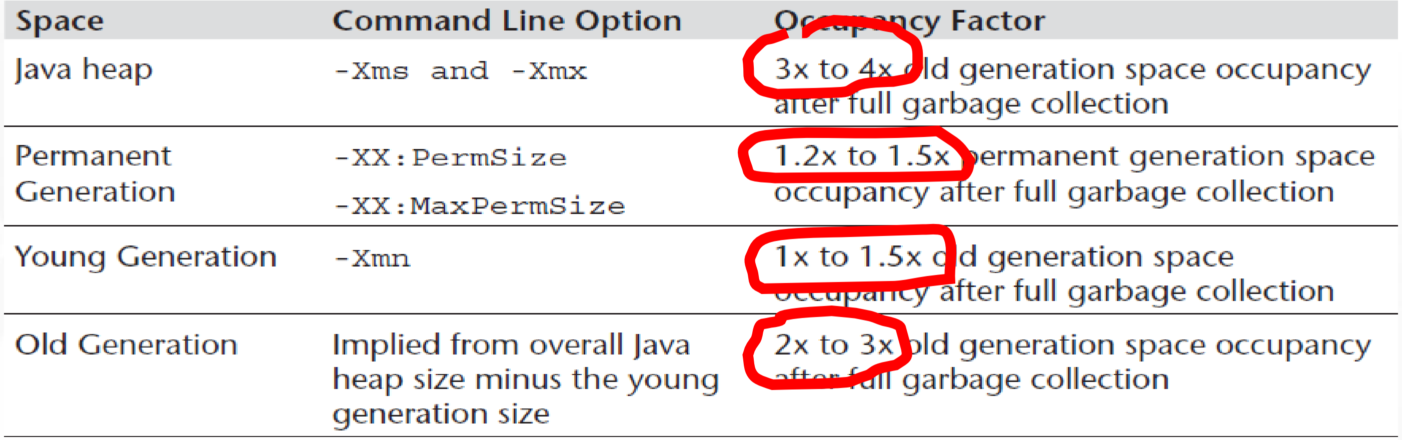
Basics of JVM Tuning



JDK(Java Development Kit)
Tools
jstack
jmap
jinfo
...
zprofiler
zdebugger
JRE(Java Runtime Environment)
JVM(Java Virtual Machine)
Garbage Collector
Bytecode Interpreter
Just-in-Time Compiler
Java Class Library(rt.jar...)

Guild for GC Tuning

- Select the right GC algorithm
  - parallel old ,CMS and G1 collector
  - Rule of thumb: GC overhead is ideally < 10%
- Choose the right heap size



Space	Command Line Option	Occupancy Factor
Java heap	-Xms and -Xmx	3x to 4x old generation space occupancy after full garbage collection
Permanent Generation	-XX:PermSize -XX:MaxPermSize	1.2x to 1.5x permanent generation space occupancy after full garbage collection
Young Generation	-Xmn	1x to 1.5x old generation space occupancy after full garbage collection
Old Generation	Implied from overall Java heap size minus the young generation size	2x to 3x old generation space occupancy after full garbage collection

source: Charlie Hunt, Binu John Java^{TM} performance

- Configure the appropriate GG parameters

JIT and common optimization

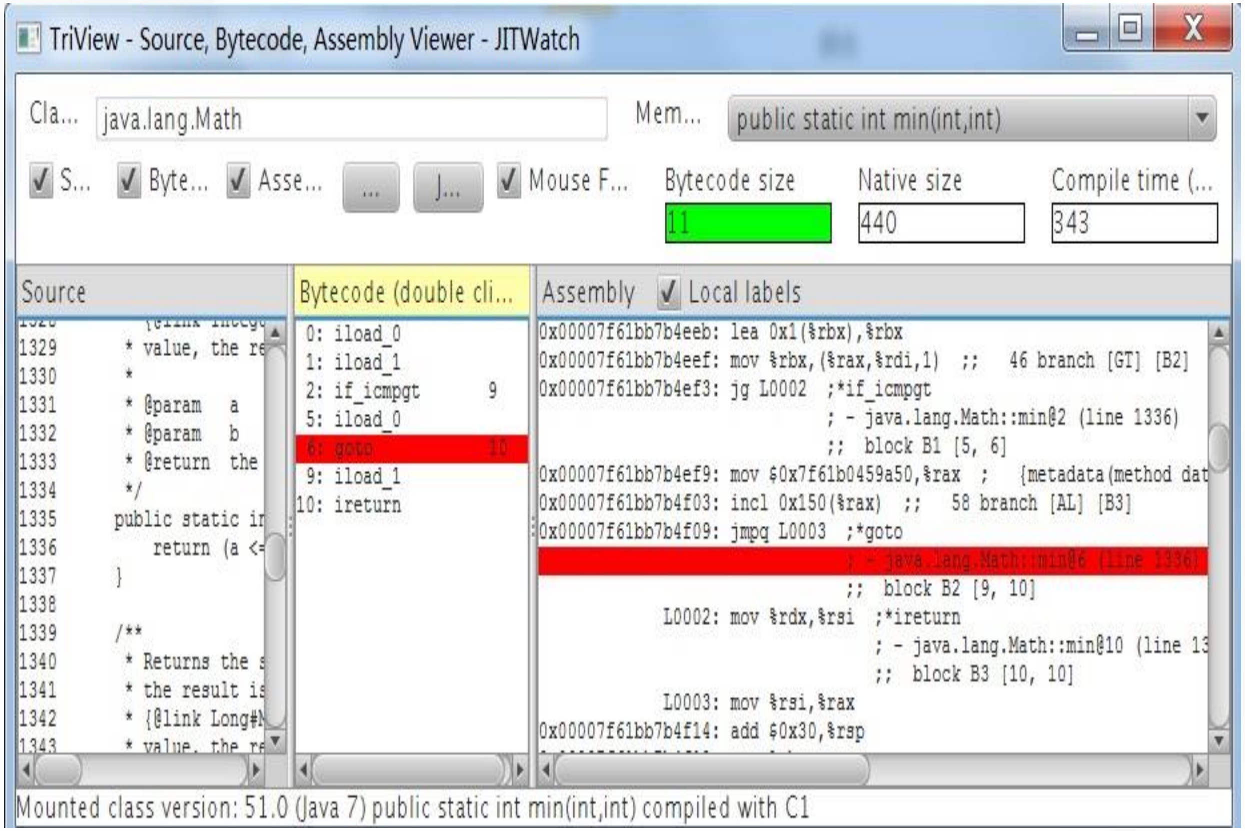
- Important concepts
  - Profiler guided optimization(PGO)
  - Optimization decisions are made dynamically
  - Mix mode execution
- Some common optimization
  - Inlining
  - Intrinsic
  - Monomorphic dispatch
  - …

❑ **Liskov substitution principle**

Subtypes MUST be substitutable for their base types

JIT Profiling with JITWatch

- JITWatch: a **graphical visualization and analysis tool** for understanding the JIT



TriView - Source, Bytecode, Assembly Viewer - JITWatch

Cla... java.lang.Math

Mem... public static int min(int,int)

☑ S... ☑ Byte... ☑ Asse... ... J... ☑ Mouse F...

Bytecode size: 11

Native size: 440

Compile time (...: 343

Source

```
1329        * value, the re
1330        *
1331        * @param   a
1332        * @param   b
1333        * @return  the
1334        */
1335       public static in
1336           return (a <=
1337       }
1338
1339       /**
1340        * Returns the s
1341        * the result is
1342        * {@link Long#M
1343        * value, the re
```

Bytecode (double cli...

```
 0: iload_0
 1: iload_1
 2: if_icmpgt        9
 5: iload_0
 6: goto            10
 9: iload_1
10: ireturn
```

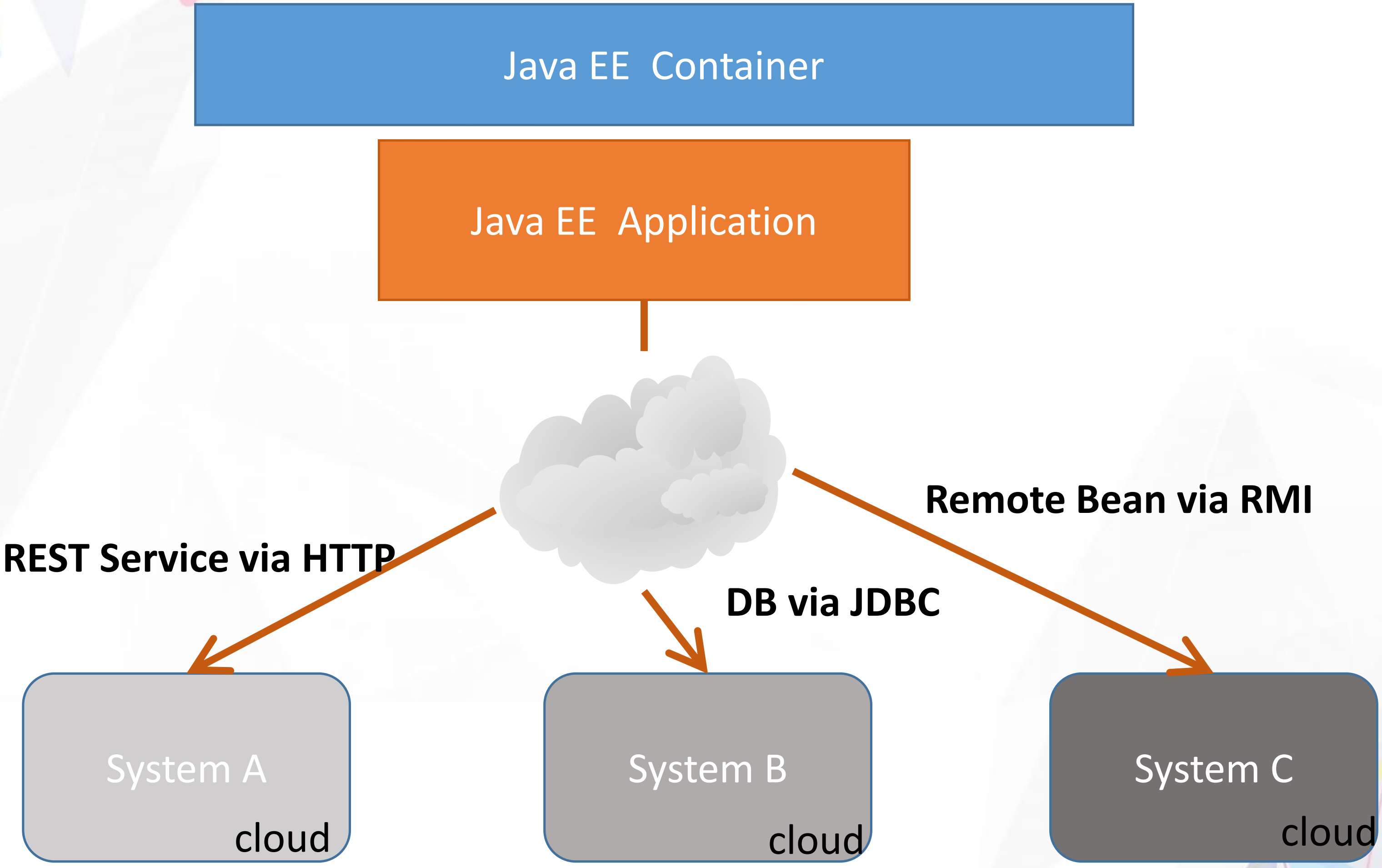
Assembly ☑ Local labels

```
0x00007f61bb7b4eeb: lea 0x1(%rbx),%rbx
0x00007f61bb7b4eef: mov %rbx,(%rax,%rdi,1)  ;;   46 branch [GT] [B2]
0x00007f61bb7b4ef3: jg L0002  ;*if_icmpgt
                              ; - java.lang.Math::min@2 (line 1336)
                              ;;  block B1 [5, 6]
0x00007f61bb7b4ef9: mov $0x7f61b0459a50,%rax  ;   {metadata(method dat
0x00007f61bb7b4f03: incl 0x150(%rax)  ;;   58 branch [AL] [B3]
0x00007f61bb7b4f09: jmpq L0003  ;*goto
                                ; - java.lang.Math::min@6 (line 1336)
                                ;;  block B2 [9, 10]
             L0002: mov %rdx,%rsi  ;*ireturn
                                   ; - java.lang.Math::min@10 (line 13
                                   ;;  block B3 [10, 10]
             L0003: mov %rsi,%rax
0x00007f61bb7b4f14: add $0x30,%rsp
```

Mounted class version: 51.0 (Java 7) public static int min(int,int) compiled with C1

Enabled by:
-XX:+UnlockDiagnosticVMOptions
-XX:+TraceClassLoading
-XX:+LogCompilation
-XX:+PrintAssembly

Typical distributed JEE architecture



Java EE Container

Java EE Application

REST Service via HTTP

DB via JDBC

Remote Bean via RMI

System A

cloud

System B

cloud

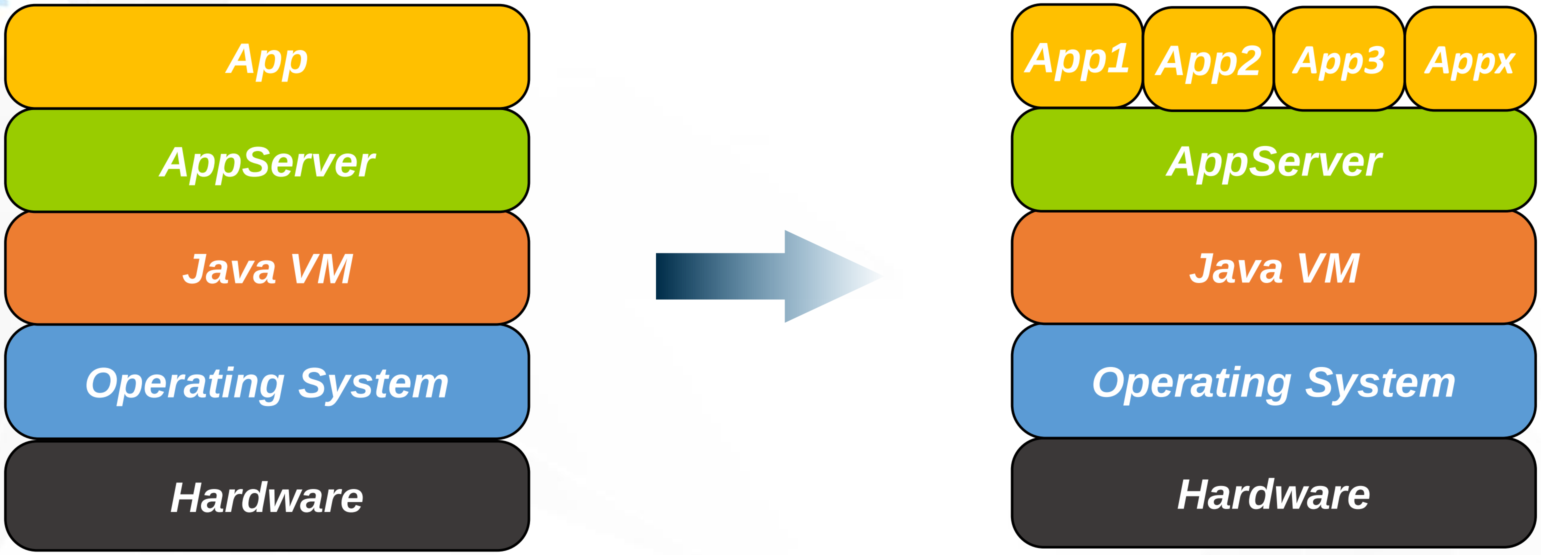
System C

cloud

The problem…

- Add communication cost
  - ✓ RPC
  - ✓ serialization/deserialization
- Can **not** shift resources towards demand
- Can **not** share the underlying Java artifacts(such as JIT)

Multitenancy for JavaEE



- ❑ Run multiple Java EE applications (**as tenants**) into same Java EE container

High Density Cloud for JavaEE

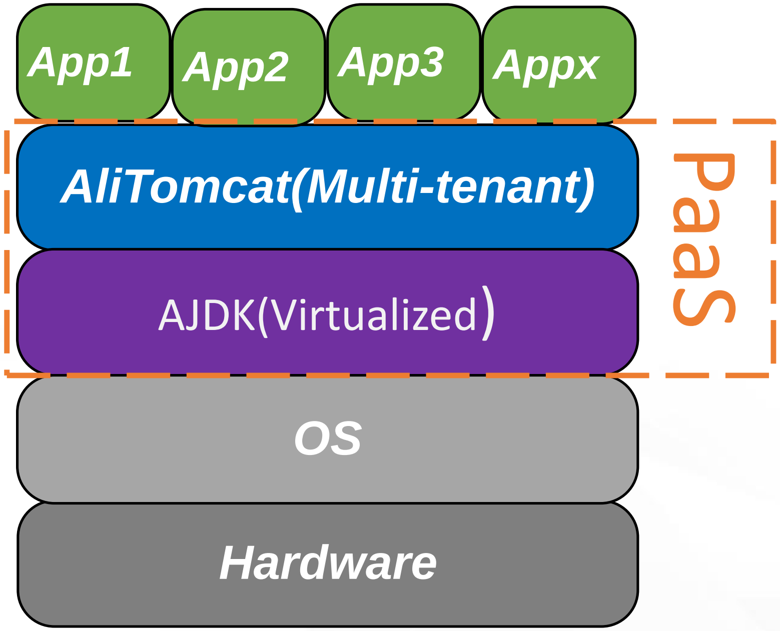
The JavaEE applications developed separately can be deployed seamlessly into the same container.

- devOps
  - ✓ Orchestrate JavaEE application at scale
- Infrastructure
  - ✓ ‘Multi-tenant’ JavaEE container
  - ✓ ‘Virtualized’ JVM

source: https://www.dreamstime.com

Tomcat/JDK extended for PaaS

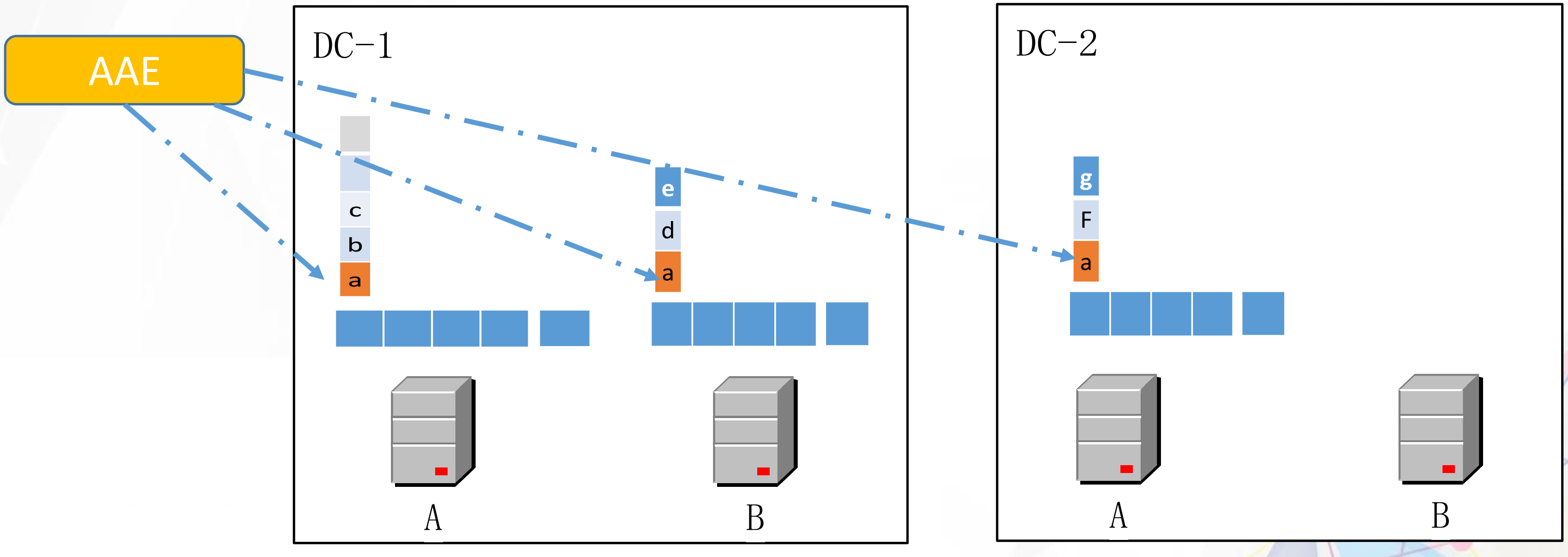
- **AliTomcat:** run multiple apps side-by-side safely
- AJDK allows for collocation of multiple JEE apps(as tenant) in a single instance of JVM:
  - **Isolate** application from one another.
  - **Share** metadata aggressively and transparently, such as:
    - ✓ bytecodes of methods
    - ✓ GC
    - ✓ JIT



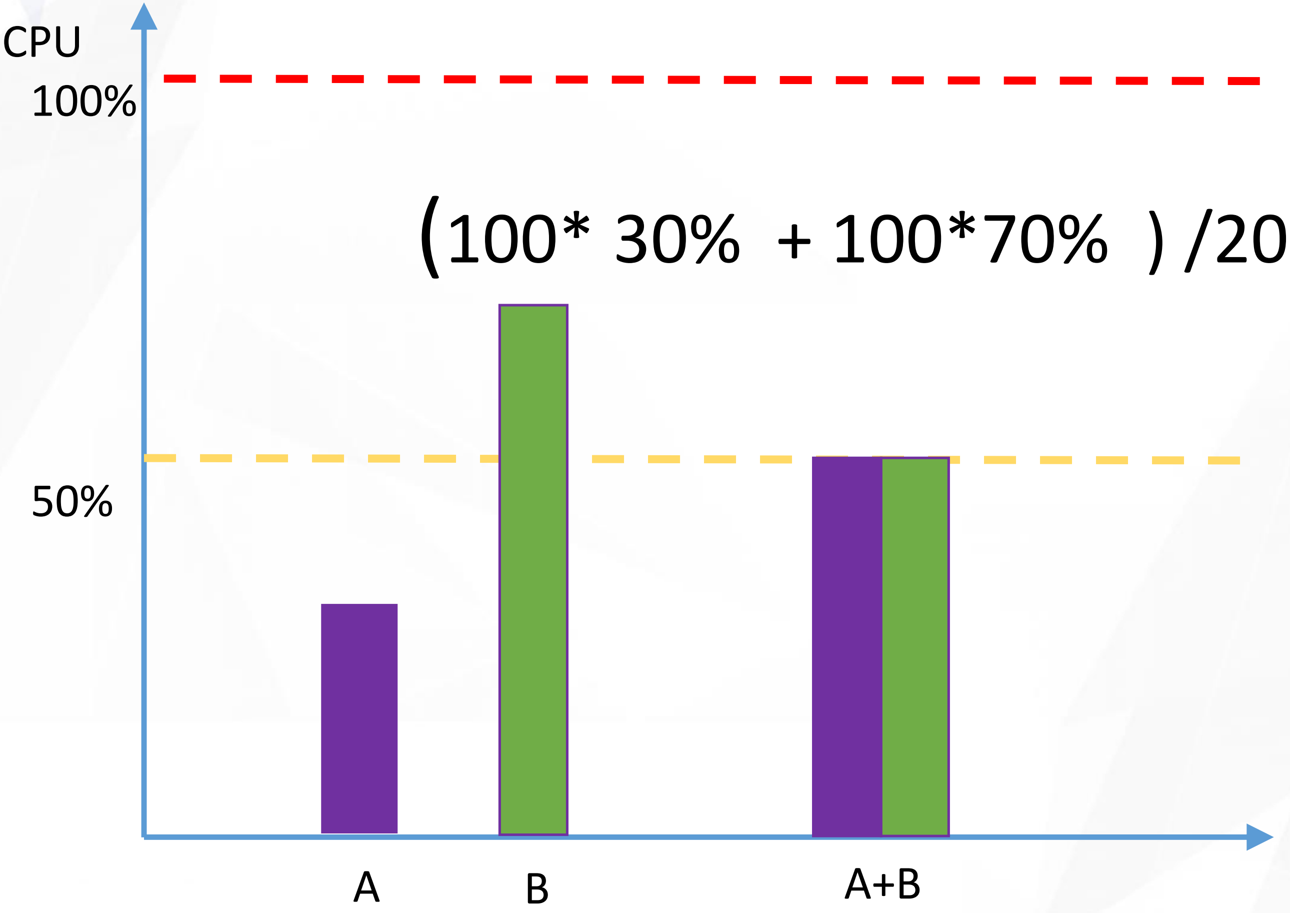
AJDK：**Alibaba/Alipay JDK, based on OpenJDK**

AAE: Alibaba Application Engine

- Scaling tenant application with AAE
  - spread application evenly across hosts
  - but **pack** applications on the single JVM as mush as possible, based on its resource capacity:
    - CPU usage
    - Memory(monitoring GC)



Benefits

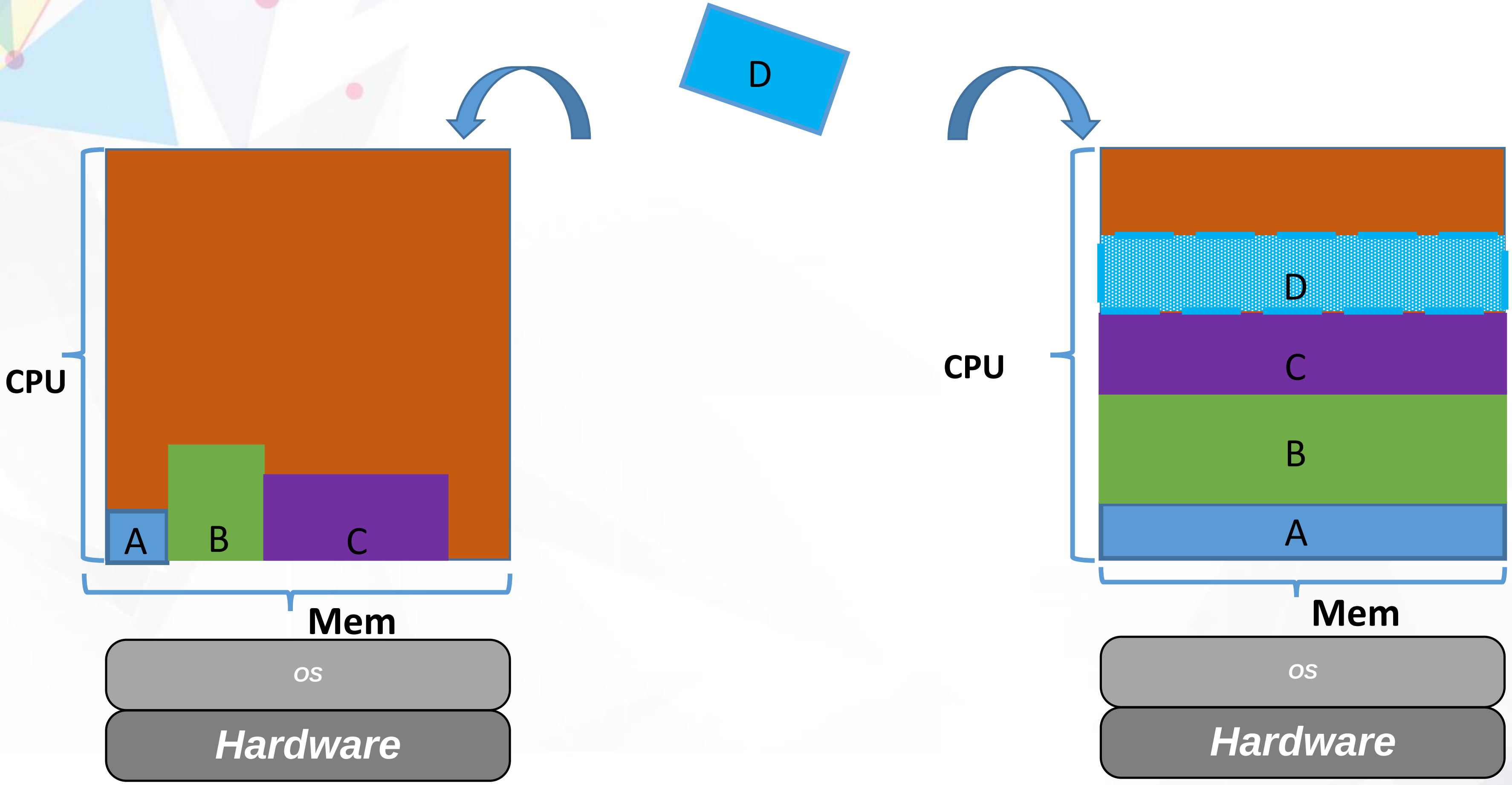


(100* 30% + 100*70%) /200 = **50%**

Benefits(Cont.)

- Eliminate the unnecessary RPC
- Minimize the cost caused by object serialization/de-serialization
- Share underlying Java artifacts as much as possible
  - GC
  - JIT
  - Heap

Compared with Docker



CPU overcommit via docker

'shared memory' via Multitenancy

Summary

- **What we covered:**
  - Performance basics& methodology
  - Performance tuning
    - Profiling
    - Tuning from JVM perspective
  - Multitenancy for JavaEE

THANKS

reach to me:

mail: sanhong.lsh@alibaba-inc.com

weibo: sanhong_li