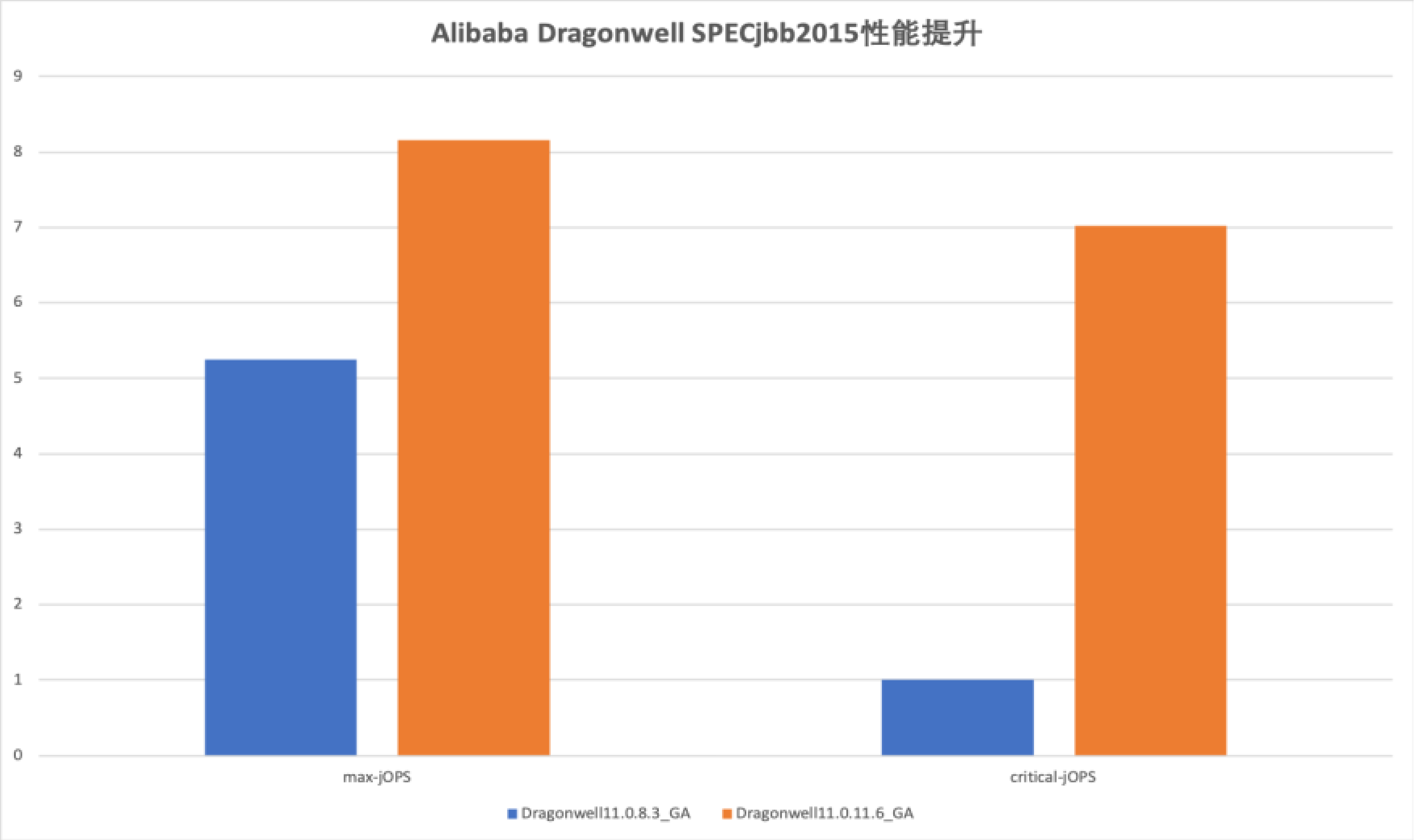
异构体系下的Java应用性能分析

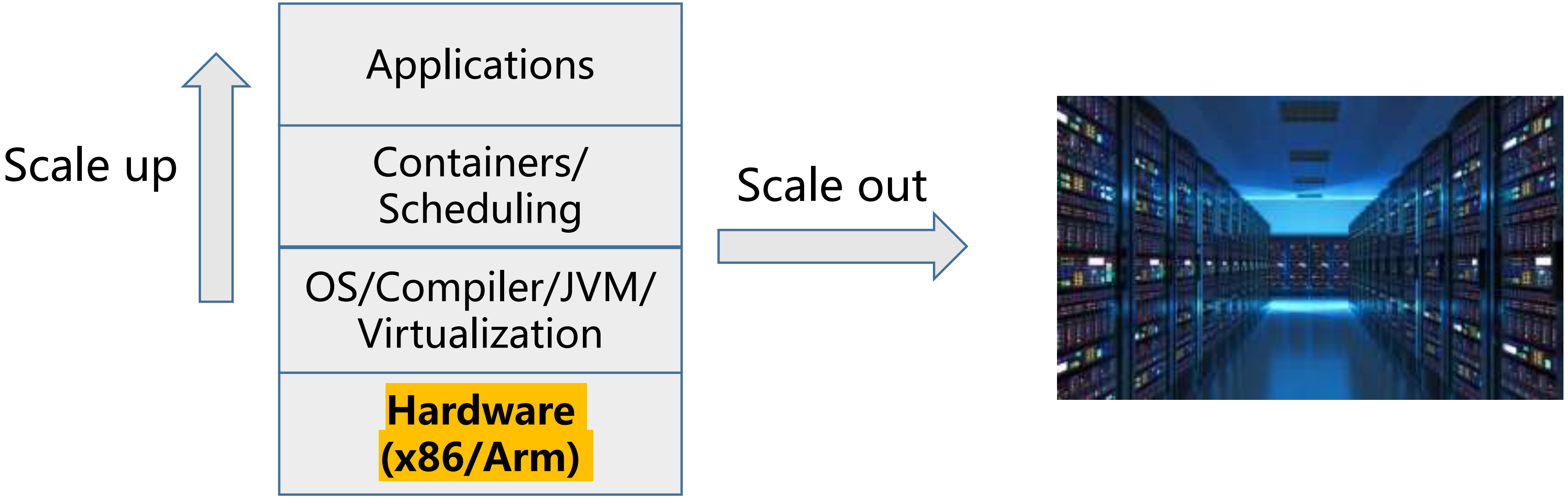
潘宇峰 技术专家
阿里集团-技术风险与效能部

超过50%！ Alibaba Dragonwell性能爆发式增长



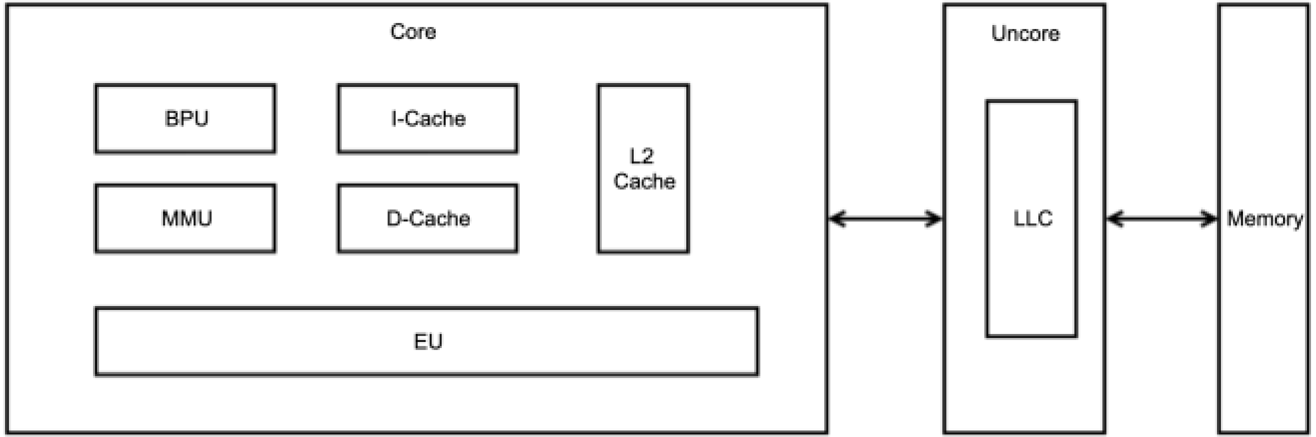
背景

- 新硬件平台 -> 全栈性能优化
- 异构体系下性能对比分析 -> 通用的性能分析方法
- 性能分析可复现 -> 分析方法流程化、自动化

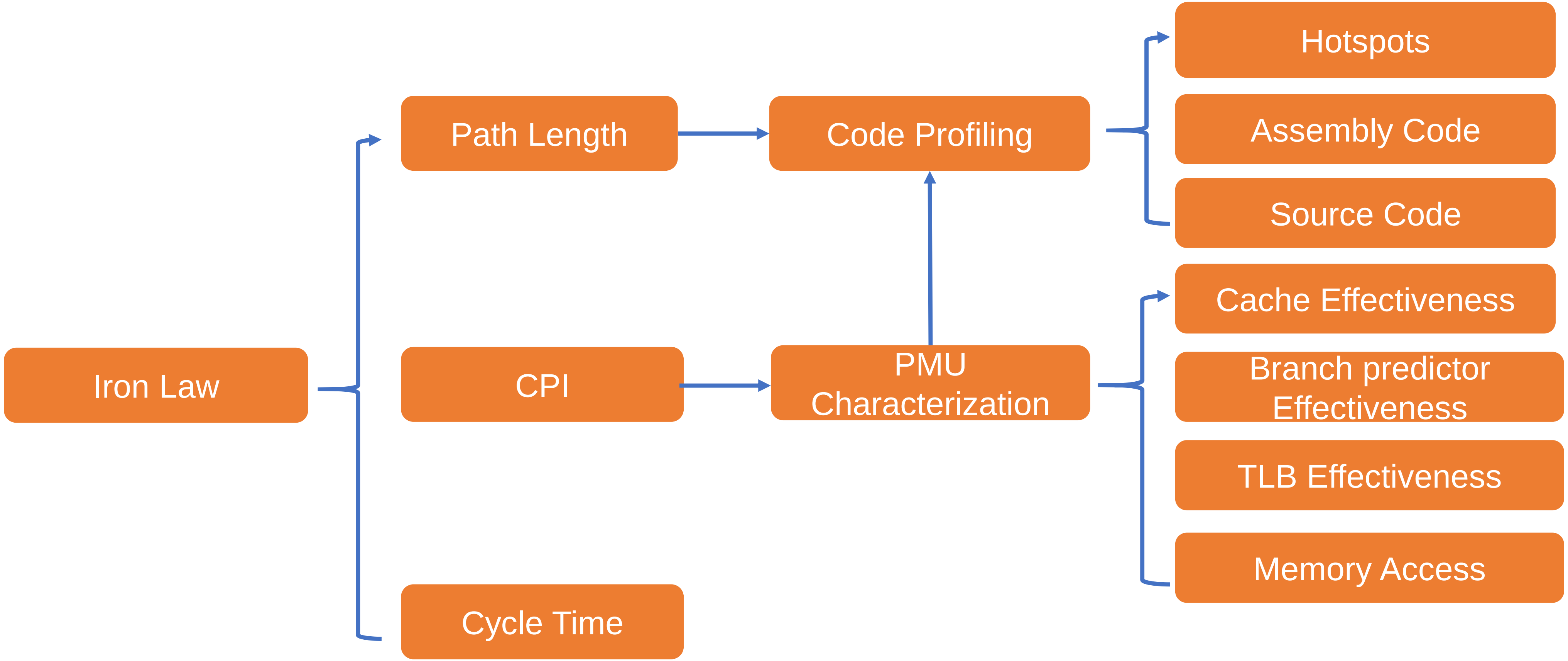


异构体系

- Architecture definition
- Micro-architecture definition
    - CPU configurations
        - Hyper-Threading
        - Turbo Boost
        - Base Frequency
        - #Cores, #Sockets
    - Cache
        - Hierarchy
        - Cache size
        - Cache line size
    - TLB
    - Memory
        - NUMA
        - #Channels
        - DIMM configurations



一套通用的异构体系下的性能分析方法



Iron Law

$$Processor\ Performance = \frac{Time}{Program}$$

$$processor\ Performance = \frac{Instructions}{Program} \times \frac{Cycles}{Instructions} \times \frac{Time}{Cycle}$$

$$processor\ Performance\ per\ Query = \frac{Instructions\ per\ second}{QPS} \times \frac{Cycles}{Instructions} \times \frac{Time}{Cycle}$$

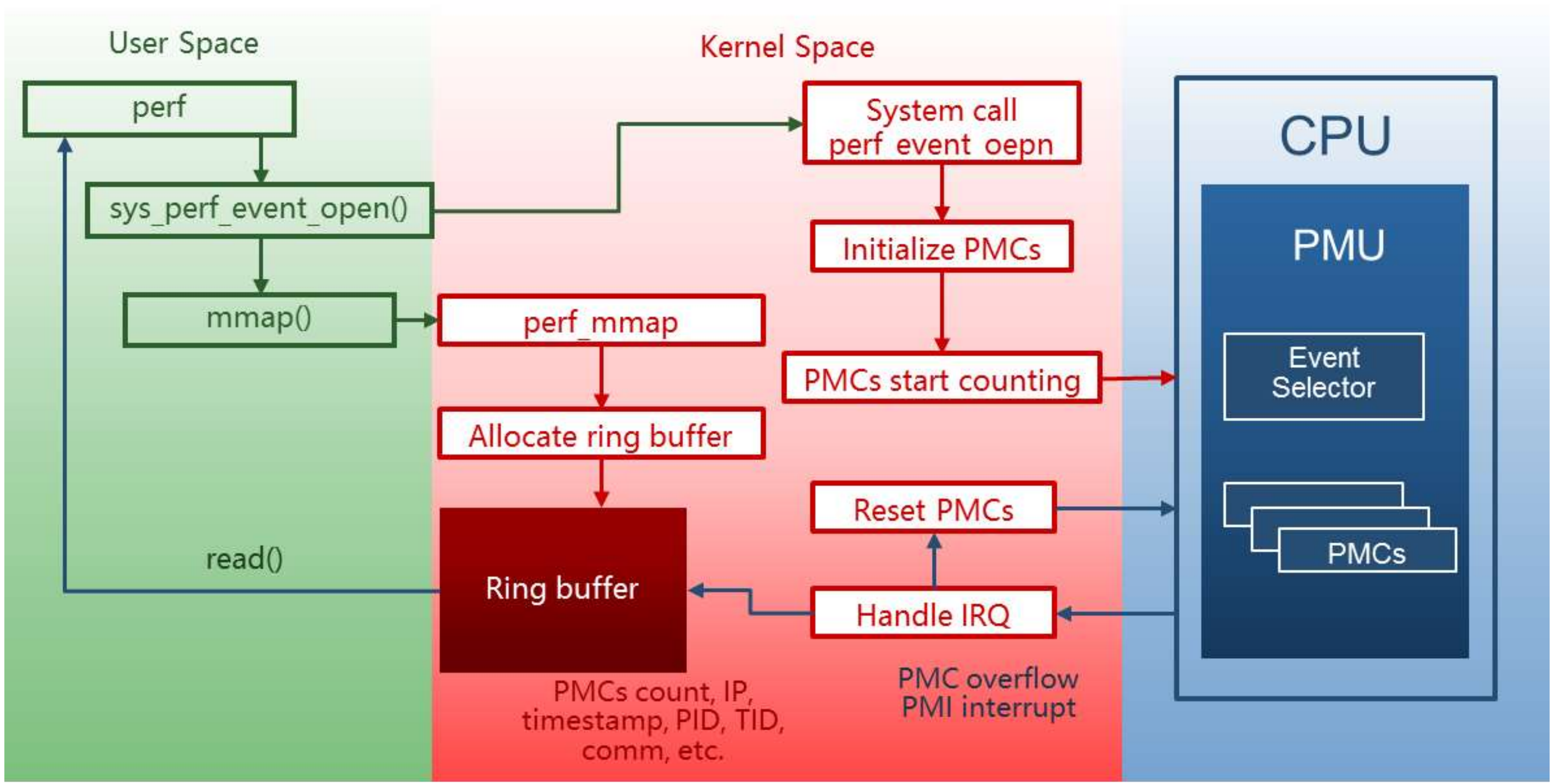
$$processor\ Performance\ per\ Query = Path\ Length \times \frac{Cycles}{Instructions} \times \frac{Time}{Cycle}$$

PMU Characterization

PMU metrics

- **Basic metrics**: CPI = Cycles/Instructions
- **Cache effectiveness**: L<n>_Cache_MPI = L<n>_Cache_misses/Instructions
- **Branch predictor effectiveness**: Branch mispredict ratio = Branch misses/Branches
- **TLB effectiveness**: TLB MPI = TLB misses/Instructions
- **Memory access**: Memory bandwidth; Memory access latency

PMU based code profiling



Perf jit support in Linux 4.6+

- Provides symbolization for jitted code
- Provides assembly and source levels profiles for jitted code
- Perf comes with a JVMTI agent for Java
- Support code movements(re-jitting)

```
$ java -agentpath:libperf-jvmti.so -jar specjbb2015.jar -m COMPOSITE

$ perf record -k 1 -e cycles -p $PID -F 99 -g -- sleep 60

$ perf inject -j -i perf.data -o perf.data.jitted

$ perf report --no-children -i perf.data.jitted
```

Code profiling analysis details

广泛使用的Flamegraph结果可能存在误导？

原因：
1.以相同weight处理不同的samples
2.以sample数量占比来代表CPU消耗占比

Code profiling analysis details

如何对比热点方法的结果？

- 对比实验中出现热点方法不同
- 热点方法按cycles percent排序不同
- 热点方法按cycles percent排序类似，但CPI或Path length不同
    - Leader sampling：perf record -k 1 -e {{cycles,instructions}}:S –F 99 –g -a – sleep 60

案例一：不同硬件平台下指令集支持的差异

- Workload

基于JMH(Java Microbenchmark Harness)框架开发的Java操作benchmark集合

- 实验环境

Arm V8.2 VS Intel Skylake

- 实验数据

	Score	Path Length	CPI	Cycle Time
Ratio(ARM/Intel)	14	0.032	2.33	0.96

1. Path length的大大降低提升了ARM上性能
2. Arm上的CPI是上升的，但是上升的比例对于性能的最终影响远不及Path Length的影响，CPI不能单独反映应用的性能。

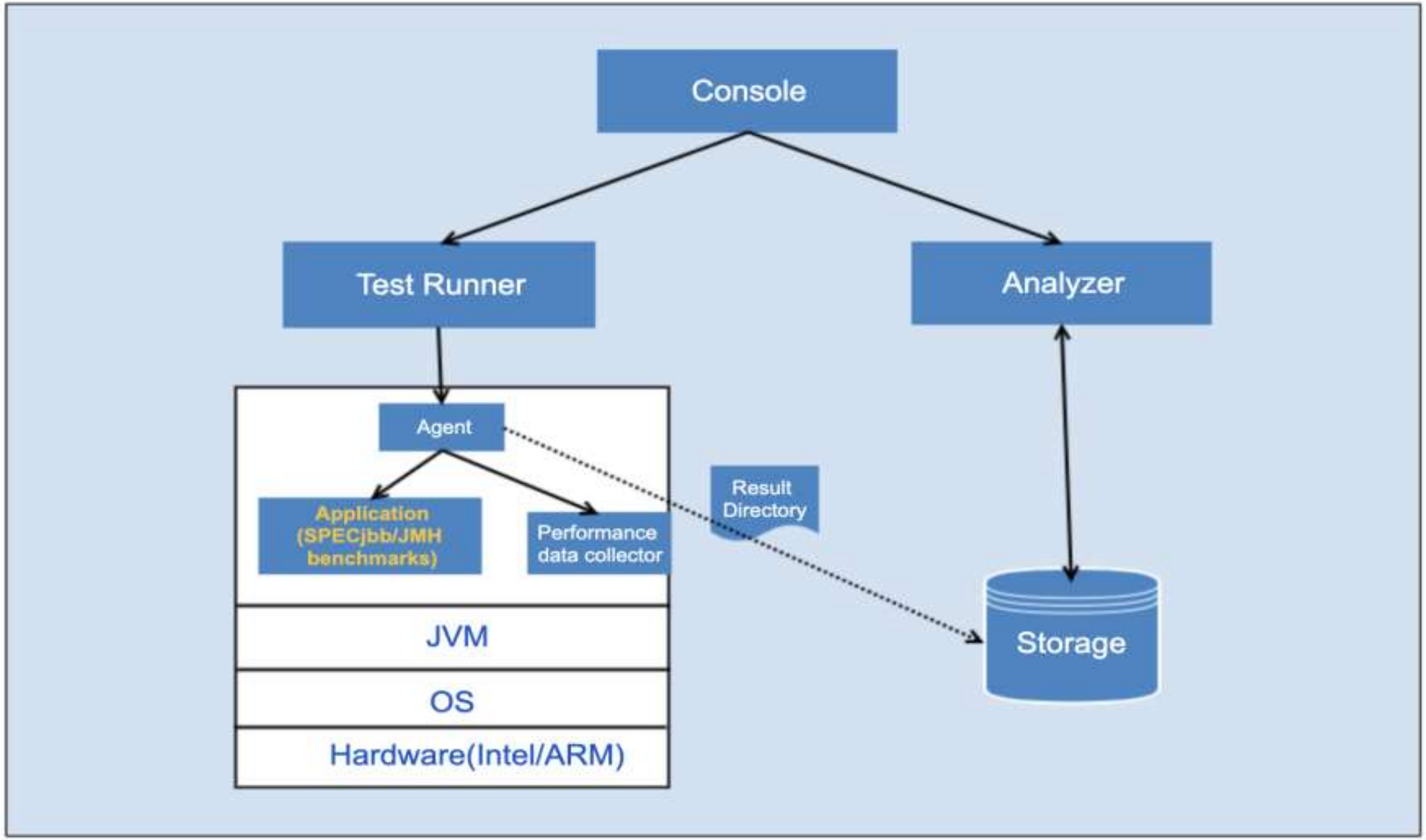
案例一：不同硬件平台下指令集支持的差异

- Hotspots对比

	Arm			Intel		
序号	method	cycles percent	CPI	method	cycles percent	CPI
1	c()	77.23	0.79	a()	89.31	0.33
2	b()	7.36	0.35			
3	a()	7.19	0.51			

- 在Arm上，method c和b的cycles percent 总和占到80%以上，而在Intel热点方法中没有发现这两个method，推测method c和b是在Arm平台上的特有实现
- J进行源码分析，定位到method c和b都是Arm平台上基于特有指令集实现的method，可以大大减少指令数量

自动性能测试和分析系统框架



Thanks！！！

Backup 案例二：不同版本JDK性能差异定位

- Workload
SPECjbb2015 running in PRESET mode, with two different JDK version: V1 and V2.

- 实验环境
  Arm Neoverse-N1

- 实验数据

	Score	Path Length	CPI	Cycle Time
Ratio(V1 / V2)	0.84	0.88	1.21	1

Backup 案例二：不同版本JDK性能差异定位

- Hotspots analysis

Method	DSO	Assembly Code	Cycles Percent	Instructions Percent	CPI
[illegible] GC Method	[illegible] r/libjv m.s[illegible]	Show	21.363	1.034	49.094
class org.spec.jbb.core.collections.HashMultiSet.update(class java.lang.Object., int)	/root/.debug/jit/java-jit-20201016.XXAfxMPX/jitted-110530-5514.so	Show	11.166	8.634	3.072
class org.spec.jbb.core.collections.HashMultiSet$EntrySet$1.next()	/root/.debug/jit/java-jit-20201016.XXAfxMPX/jitted-110530-2664.so	Show	5.401	3.615	3.548

One GC method with high CPI

- GC Analysis

	gcPause(s)
Ratio(V1/V2)	4.81