大数据场景下语言虚拟机的性能分析与优化调研

贾兴国, 张正君, 余博识

Shanghai Jiao Tong University

{zhangzhengjunsjtu, 201608ybs, jiaxg1998}@sjtu.edu.cn

School of Software Engineering

版本：1.0

日期：2020 年 12 月 28 日

摘　要

随着大数据处理系统的快速发展，托管型语言（managed languages）由于其快速的开发周期以及丰富的社区资源得到了广泛的应用。例如，分布式数据处理框架 Spark、Hadoop、DryadLINQ、分布式文件系统 HDFS、分布式键值存储系统 HBase、Cassandra 等均使用开发效率较高的 Java 语言进行开发。托管型语言通过使用语言虚拟机实现对内存的管理，如使用 JVM（Java Virtual Machine）进行垃圾回收（garbage collection，GC），免去手动编写代码进行内存释放的需要，也降低了内存管理出现错误的可能。然而，开发的便利性也带来了一定的性能损耗。经过调研，研究者们从不同的角度发现了托管型语言在分布式大数据处理场景下出现的性能问题，如 GC 时间过长、节点间数据传输太慢、语言虚拟机启动耗时过长等，并且提出了相应的解决方案。本文总结了研究者们发现的问题及其解决方案，并且针对解决方案的局限性与适用性进行了分析，提出了相应的优化方案。

关键词: 语言虚拟机，分布式系统，大数据，垃圾回收

目录

List of Figures 4 研究背景 5

2 大数据场景下语言虚拟机的性能问题与优化 6
2.1 大数据场景下 GC 对性能的影响 . . . 6
2.1.1 问题概述 . . . 6
2.1.2 解决方案 . . . 6
2.2 JVM 启动耗时对延迟敏感型应用的影响 . . . 7
2.2.1 问题概述 . . . 7
2.2.2 解决方案 . . . 7
2.3 分布式环境下多个 JVM 的协调问题 . . . 8
2.3.1 问题概述 . . . 8
2.3.2 解决方案 . . . 8
2.4 大数据场景下节点数据交换的性能问题 . . . 8
2.4.1 问题概述 . . . 8
2.4.2 解决方案 . . . 9
2.5 资源解聚架构下的 GC 算法优化 . . . 10
2.5.1 问题概述 . . . 10
2.5.2 解决方案 . . . 10
2.6 相关工作小结 . . . 11

3 HotTub 研究思路分析 13
3.1 延迟敏感型应用中 Java 的性能问题 . . . 13
3.2 JVM 预热开销测试与分析 . . . 14
3.2.1 主要发现 . . . 14
3.2.2 JVM 预热开销的测量 . . . 14
3.2.3 JVM 预热开销分析 . . . 16
3.3 HotTub 设计与实现 . . . 19
3.3.1 总体设计 . . . 19
3.3.2 设计细节 . . . 21
3.3.3 实现细节 . . . 22
3.4 HotTub 性能测试 . . . 22
3.4.1 性能提升 . . . 23
3.4.2 工作负载变化的性能影响 . . . 24
3.4.3 引入的管理开销 . . . 24

3.5 HotTub 适用性与局限性分析 25

4 HotTub 局限性优化 **25**

5 总结 **26**

附录 **30**

A 附录 **30**

A.1 // 30

List of Figures

1 截获 `ret` 指令的执行模式切换 15
2 HDFS 请求预热耗时时长 16
3 HDFS 请求预热耗时占运行时间百分比 17
4 Sequential Read 1G 数据的运行时间分解 17
5 Datanode、client 数据包传送过程时间分解 18
6 Spark、Hive 请求运行时间分解 18
7 query 13 在 Spark 上执行的运行时间分解 19
8 HotTub 架构 20
9 HotTub client 运行过程 20
10 HotTub 性能分析 23

1 研究背景

随着摩尔定律的终结，单个机器所提供的计算资源已经无法满足海量数据的巨大算力需求。学术界和工业界将关注点从纵向扩展转向了横向扩展，使用多台配置较低的廉价机器代替了配置较高的单台机器，使用分布式系统向数据处理、机器学习、云计算等应用提供可横向扩展的计算资源。运行在分布式系统上的数据处理框架如 Spark[1]、Hadoop[2]、DryadLINQ[3]、Naiad[4] 等采用了分布式并行的数据处理模式，充分利用分布式集群提供的计算资源。也有分布式文件系统 HDFS[5]、分布式键值存储系统 HBase[6]、Cassandra[7] 等利用分布式系统较强的容错性、可扩展性，为数据存储提供了更优的解决方案。

运行在云平台上的分布式软件的快速发展使得托管型编程语言成为了开发人员和研究人员的关注重点。广泛使用的分布式处理框架 Hadoop、Spark、Cassandra 均使用 Java 进行开发，DryadLINQ、Naiad 等数据处理框架使用由微软开发的 C# 语言进行编写。还有更多的托管型语言被广泛使用，如 Go、Javascript、Python 等。托管型语言如 Java、C# 使用语言虚拟机进行自动内存回收，免去了手动编写代码释放内存的需要，还有面向对象的编程风格、丰富的社区资源，使得 Java 等语言有很高的开发效率。

语言虚拟机带来的开发便捷性建立在一定的性能损耗之上。例如为了运行 Java 程序，首先对 Java 源代码进行编译，生成 bytecode（字节码），再从 classpath 中加载、解析、验证字节码文件，以及运行该字节码所需的其他字节码文件，后在 JVM 中运行字节码。Java 语言运行环境（runtime）的加载以及字节码的解释执行相比于 Native 的编程语言（如 C、C++）产生了巨大的开销。同时，Java 程序在堆（heap）内存中为对象申请内存，无需手动编写代码释放堆内存，而是使用 JVM 自带的 GC（垃圾回收）功能进行对内存的自动释放。然而，进行 GC 的过程中，语言虚拟机需要占用应用的 CPU 资源，或需要应用暂停执行（Stop-the-world），这在延迟敏感的情境下对应用的性能表现有巨大的影响。在分布式数据处理的场景中，由于 Java 等管理型语言使用在堆上保存的对象（object）来存储、处理数据，GC 需要遍历海量的对象，其开销相对于单机、数据量较小的情景也被明显加大。分布式环境下请求的处理依赖于多个节点上的语言运行环境，语言虚拟机的性能优化需要依靠多个语言虚拟机之间的协调，这和单机情境下的语言虚拟机性能优化又大有不同。

本文针对大数据处理场景下的语言虚拟机性能优化进行调研，总结出了分布式大数据处理对语言虚拟机提出的新的挑战，以及研究者所提出的解决方案，并分析了特定解决方案的适用性与优化方案。本文第二章总结了语言虚拟机在大数据情境下表现出的性能问题，以及研究者对相关问题所提出的解决方案。第三章就 JVM 启动耗时过长问题的研究思路进行了分析，并总结了其局限性与适用范围。第四章提出了针对第三章中适用性问题的解决方案，并搭建原型、验证和测试了解决方案的有效性。在第五章中我们对本文的调研工作进行总结。

2 大数据场景下语言虚拟机的性能问题与优化

大数据处理为托管型语言带来了全新的使用场景，同时托管型语言为大数据处理框架带来了较高的开发效率。新的使用情景为托管型语言的语言虚拟机带来了新的挑战。本章对大数据环境下语言虚拟机表现出的性能问题做了总结分析，并对研究者的相应解决方案进行了总结。

2.1 大数据场景下 GC 对性能的影响

2.1.1 问题概述

在大数据场景下，应用需要良好的性能与扩展性，然而托管型语言的运行时环境存在臃肿、可扩展性差的问题。语言虚拟机的 GC 使得应用的主线程必须停止，需要遍历堆中保存的相互引用的复杂的对象，占用大量计算资源，无法进行有用的工作；内存管理模块也有巨大的额外内存开销，虽然所处理的数据量大小比堆内存的容量更小，也会产生 OOM（Out Of Memory，内存不足）的报错。但使用非管理型语言会增加应用编写者的负担，更容易出现错误，调试内存管理错误也是一件非常困难的事。当前大量的大数据处理框架已经使用管理型语言进行编写，向非管理型语言的迁移会带来巨大的工作量。故需要有一种方法优化语言虚拟机的 GC 问题与内存空间占用问题。

2.1.2 解决方案

Nguyen 等人提出了 Facade[8] 编译框架。他们认为将大量数据保存在堆内存上，让 JVM 进行管理是不明智的。通过限制堆内存上保存的对象数量，将数据面与控制面进行分离，仅在堆上保存控制面对象，而数据面中的数据保存在本地内存（native memory）中进行手动管理。这打破了面向对象编程语言的一条规则：对象应当包含数据以及操作数据的接口。对于每一个 Java 语言中的堆上保存的对象，Facade 编译系统生成一个相应的 facade，提供对象的操作接口而不保存对象中包含的数据，从而减轻垃圾收集器的负担，而程序员仅需标记出数据类（data class）。对于 facade，可以通过对象复用限制堆中对象的数量；而 native memory 中的数据在每个 iteration 结束后统一进行释放，无需访问数据的每个字段，相比于垃圾收集器有更优的性能。经测试，Facade 使得应用运行时间缩短了 48%，降低了 50% 的内存占用。

Gog 等人提出了 Broom[9] 内存管理系统，取代了 CLR[1]的 GC 系统。他们认为，大数据系统中存在多个 Actor，这些 Actor 独立地运行，它们之间传递消息，形成了高度结构化的数据流，例如 MapReduce[11] 中的 map 工作线程与 reduce 工作线程。每个 Actor

[1]CLR[10]，Common Language Runtime，是 Microsoft .Net 框架的语言虚拟机系统、运行时环境，运行.Net 程序。使用任何语言编写的.Net 程序均运行在 CLR 上，而 JVM 仅支持运行 Java 程序。

内的对象在 Actor 完成工作后不再会用到，仅有 Actor 之间传递的数据需要保存，无需 GC 扫描。为了针对这类高度结构化的应用对垃圾收集器进行优化，Broom 内存管理器使用了基于 region（分区）的内存管理策略[12]，对象在各个 region 中保存而不在堆中由 GC 管理，由程序员手动释放 region 中的对象。Broom 内存管理器减轻了 GC 的负担，减少了 59% 的运行时间，但使得程序员的负担增加。

2.2 JVM 启动耗时对延迟敏感型应用的影响

2.2.1 问题概述

社区中长期存在对 Java 性能的激烈讨论，尤其是 Java 是否适合编写延迟敏感型应用的问题[13-16]。一些程序员坚持使用 C++ 等非管理型语言编写键值存储系统，因为他们认为 Java 天生就是较慢的。Java 之所以适用于编写 Hadoop 框架是因为 Hadoop 大部分时间在进行 I/O 操作[17]。在学术界，对于大数据框架的性能优化主要集中在对 GC 性能的优化、调度策略的优化、数据交换开销的优化等，故我们缺少对 Java 各个方面性能的整体了解。经过实验测得，在 I/O 密集型任务中，JVM 的预热开销也是较大的，如在从 HDFS 中读取 1G 的文件的过程中，JVM 预热使用的时间占总运行时间的 33%b[18]，预热时间不随任务数据量的变化而变化。在大数据框架中，复杂的软件栈使得 Java 运行环境的加载更加缓慢，因为需要加载更多的 class，如 Spark 为了完成一次请求需要加载 19066 个 class。

2.2.2 解决方案

Lion 等人经过实验测得，JVM 的性能瓶颈在于 JVM 的启动预热（warm-up），并编写了 HotTub[18] 代替了 HotSpot[2]解决了 Java 虚拟机启动时间过长的问题。他们首先通过修改 Java 程序的调用过程测量出 class 加载与字节码解释执行的开销，观察出 JVM 预热对性能的巨大影响。同时，他们在 HotSpot 的基础上修改实现了 HotTub，在多次请求中复用同一个 JVM 实例，从而免去 JVM 的预热开销。

由于大数据应用的相似性，JVM 有较大的复用可能，同时频繁使用的字节码在多次复用后被 JIT 编译成机器码而无需解释执行。HotTub 可以让 Java 程序无修改地运行，免去了 JVM 的预热开销，使得 Spark 请求的性能提升到 1.8 倍。本文将在第三章对 Lion 等人的 HotTub 研究进行详细的分析，总结其使用范围与局限性，并在第四章中提出并测试相应的优化方案。

[2]HotSpot 是一个 Java 虚拟机的实现，由 Sun 公司开发，包含在 OpenJDK 中，是主流的 JVM。详见 https://stackoverflow.com/questions/16568253/difference-between-jvm-and-hotspot

2.3 分布式环境下多个 JVM 的协调问题

2.3.1 问题概述

在分布式大数据处理框架中，工作负载运行在多个节点上，即运行在多个相互独立的运行时系统（语言虚拟机）上。这会成为性能降低的原因之一，因为每个运行时系统仅拥有当前运行节点的运行状况，而非全局的分布式应用的运行状，无法做出对全局性能有利的决策。这是之前的工作无法解决的问题，之前的工作仅针对单个节点上的运行时系统进行性能优化。例如，某个节点在不合适的时间进行 GC，导致该节点成为整个请求处理流程中最慢的节点。虽然其余节点在较短时间内完成了自己的工作，但仍需等待最慢的节点完成 GC，才能将请求结果返回给用户。这将会对延时敏感型与面向吞吐量的应用都造成性能影响。

2.3.2 解决方案

Maas 等人设计的 Taurus[19] 系统通过在分布式处理系统中加入一个分布式决策机制，将每个节点上独立的运行时系统组织为一个 Holistic Runtime System（全局运行时系统，HRS），从而做出在分布式系统全局下正确的 GC、JIT 决策。HRS 将每个节点上支持分布式应用运行的 runtime 当做一个整体，看作一个分布式系统，从而做出合理的全局决策，而非每个节点上的 runtime 独立地做出决策。当在某个节点上运行的一个 Holistic Runtime System 实例启动时，该实例加入执行同一个 policy 的 runtime 的集合，或成为该集合的 leader，该集合称为一个 coordination group（协调组）。每个实例启动时会附带执行 monitor 线程，从 consensus layer（一致性层）获取本协调组的状态信息。在每个 epoch（阶段）结束时，协调组的 leader 执行该协调组的 policy，通过考虑所协调组中所有 runtime 的状态信息，制定下一个 epoch 中每个 runtime 需要执行的事件，同时，协调组中的每个 runtime 向 leader 发送自己的最新状态，从而开始下一个 epoch。在实验中，作者针对每个分布式应用所遇到的特定问题制定了相应的 policy，分别对批处理型工作负载和交互式工作负载进行测试。通过执行 STU（Stop The Universe）的 policy，批处理型工作负载 Spark PageRank 的执行时间降低了 21%；对于交互型工作负载 Apache Cassandra，read 请求的 99.99% 的尾延迟从 65.7 ms 降低到了 33.8 ms，99.9999% 尾延迟从 128.6 ms 降低到了 54.6 ms。

2.4 大数据场景下节点数据交换的性能问题

2.4.1 问题概述

在大数据系统中，节点间数据的传递是频繁发生的。和之前 GC 性能问题的根源相同，数据传递也因为需要处理海量的包含着数据的对象（object）而受到了性能影响。数

据发送方需要对于 Java 中的对象（object）进行 *serialize*（序列化）才可以在网络中传递，接收方需要将接收到的字节流进行 *deserialize*（反序列化）才能生成发送方想要发送的对象。序列化和反序列化严重依赖于 Java 中的反射机制，这是一个开销较大运行时操作，会严重影响应用的性能；同时程序员需要手写序列化和反序列化函数，较容易出现错误。在堆中使用 object 管理海量的数据不但会造成巨大的垃圾收集开销，同时因为 object 中存在的对象头、指向类的指针等产生巨大的额外内存开销。这要求我们进一步考虑 Java 等面向对象语言中 object 与数据处理的关系。

2.4.2 解决方案

Nguyen 等人提出的 Skyway[20] 将本地与远程的 JVM 进程中的堆进行连接，使得源节点的堆中的对象无需序列化即可传输到目的节点。作者观察到，在两个 JVM 之间传输对象只有一个方式，即首先将对象 serialize（序列化），将对象转化成一串二进制数据，在网络中传输这段二进制，才能够被另一端的 JVM deserialize（反序列化），重新组装成一个 object。如果在对象的跨节点传输中直接传输对象，则不存在序列化与反序列化的开销，程序员也无需编写序列化与反序列化对应的函数，减轻了开发负担，减小了出现错误的可能性。Skyway 可以在本地与远程 JVM 进程之间直接传送 object，源节点将对象直接写入 output buffer（放置于 native memory 中），通过网络将对象写入目的节点位于 JVM 堆中的 input buffer，对象无需反序列化即可使用。在对 Spark PageRank 和 TriangleCounting 程序的测试中，由于出现了大量的节点间数据交换，Skyway 节省了大量序列化和反序列化的时间，比默认 Java Serializer 快 36%，比 Kryo[21] 快 16%。

Navasca 等人认为，数据分析的任务经常使用的数据类型是不可变且被限制的，他们开发的 Gerenuk[22] 编译器将一个 SER（speculative execution region，预测执行区）编译为直接对来自于磁盘或网络的 native bytes 进行操作的代码，从而降低了 object 表示数据所产生的额外内存开销，同时减小了运行时开销、垃圾回收开销与序列化、反序列化开销。与 Skyway 不同，Gerenuk 更希望数据以 native bytes 存在，这样有多方面的优势，如 GC 负担减小、object 额外开销减小、序列化、反序列化开销减小。由于 SER 开始于一个反序列化点，终止于一个序列化点，Gerenuk 需要大数据框架开发者标记出这些起始点与终结点。通过将 object 编译为处于 native memory 中的数据结构、细粒度地将 Java bytecode 编译为访问 native memory 的指令，Gerenuk 获得了更低的 GC 开销与序列化、反序列化开销。Gerenuk 编译的 Spark、Hadoop 程序均获得了更低的内存占用、更低的 GC 开销、序列化、反序列化开销等。

2.5 资源解聚架构下的 GC 算法优化

2.5.1 问题概述

在大数据处理集群中，资源碎片化问题影响着集群资源使用率，如内存资源，单个节点不一定在全部的时间段都对内存有较高的利用率。当内存利用率较低时，单机上的空余内存无法被集群中其他节点所利用，即出现了资源碎片化的问题。资源解聚架构的出现解决了这一问题，与普通的单体服务器（monolithic servers）不同，资源解聚架构的数据中心中每个 server 仅负责对集群提供单一的一种资源，如内存服务器提供内存资源。这类架构使得某种资源可以被多个节点的进程使用，避免了资源碎片化问题；单个节点的崩溃不会影响整个系统，仅会使得某种资源的可用余量减小，提高了系统的容错能力，同时也提高了系统的可扩展性，可以方便地引入新的资源服务器扩充资源量。

对于内存资源，虽然 RDMA 大大降低了访问远程内存的吞吐量与延迟，但是访问远程节点的内存仍然是一个耗时操作，具有良好的内存访问局部性的（locality）应用程序才能够高性能地运行在资源解聚架构中。然而大数据场景下经常使用的管理型语言则不能表现出良好的内存访问局部性。GC 需要遍历堆中的所有 object 来查找不可达的对象，是一个典型的图计算型工作负载，不具有局部性。同时，类似 Java 的管理型编程语言使用指针连接各种对象，这也不具有局部性。例如 Spark 中的 RDD[3]，在资源解聚架构下，遍历一个 RDD 需要访问多个内存服务器。管理型语言的运行时需要对资源解聚架构进行适配优化。

2.5.2 解决方案

Chenxi Wang 等人开发的 Semeru[24] 是一个分布式的 JVM，可以支持 Java 程序无修改地运行在资源解聚架构的集群中，通过将 GC 的 object tracing、内存释放等工作卸载到内存服务器上，从而在 GC 时获得更好的并行性和局部性。为了实现一个资源解聚架构友好的 JVM，需要解决三个问题：

一是应当提供怎样的内存抽象。由于需要将 GC 卸载到内存服务器上，则需要在内存服务器上运行一个对象管理进程（如 JVM），那么同一个 object 在 CPU 服务器上的内存虚拟地址则与内存服务器上的不同。Semeru 提供了一个统一 Java 堆（Universal Java heap，UJH）对于 CPU 服务器上的 main 进程可以看到一个连续的统一的虚拟地址空间，存放着它操作的对象，而该 main 进程的辅助进程运行在各自的内存服务器上，可以看到并管理统一虚拟地址空间的一部分。

二是应当将什么工作移交到内存服务器上进行。GC 中仅有跟踪对象（tracing objects）这一工作可以与应用程序无错误地并行运行，而移动、释放对象则需要应用程序停止

[3]RDD，Resilient Distributed Datasets[23]，是 Spark 的核心数据结构，是一组可以被并行操作的可容错的分布式数据集，保存在内存中。

才可以保证正确性。Semeru 不断地跟踪其负责的对象，不论 CPU 服务器是否需要进行 GC，也能够利用到各类硬件加速器。当需要对象的清理与释放时，内存服务器向 CPU 服务器发送请求，停止 CPU 服务器上的应用程序（stop-the-world），但这也为内存服务器将对象重新布局从而增强内存访问局部性提供了机会。

三是如何高效地进行 swap。目前的 swap 系统尚未对语言运行时进行适配，如在 Infiniswap[25] 上运行 Spark 会报错。Semeru 修改了 NVMe-oF 的实现从而实现了高效的远程内存访问，同时添加了 system call 使得语言运行时能够高效地与 swap 系统进行交互。

Semeru 使得 Spark 和 Flink 在资源解聚系统上获得了较大的性能提升，端到端性能在 CPU 服务器 cache 大小是 50%、25% 的 heap 大小的情况下分别提升了 2.1 倍、3.7 倍，应用性能提升 1.9 倍、3.3 倍，GC 性能提升 4.2 倍、5.6 倍。

2.6 相关工作小结

本章对语言虚拟机在大数据情景下遇到的问题做了总结，并针对每个问题分类、总结了相关研究工作的基本思想与原理。

语言虚拟机在大数据环境下需要大量的对象对数据进行管理，这会造成内存开销过大、GC 耗时过长的问题。Facade 将 Java 程序中的数据面和控制面分开，从而将对象从 heap（堆）中移动到 native memory 中，达到了优化目的，但有极大的开发难度，无法使用在真实生产环境下。

Broom 则关注了 object 的明确的生命周期，使用 region-based 的内存管理策略，为用户提供手动分配、释放数据对象的接口，但仅仅适用于 Naiad 开发者，且没有提供自动化支持，开发负担较大。Yak[26] 也关注到了大数据环境下数据对象的生命周期，通过使得 GC 算法适应于数据对象的生命周期，减小了 GC 的开销，但没有解决 object 造成的内存膨胀问题与序列化、反序列化问题。

Skyway 和 Gerenuk 则关注了大数据环境下节点间数据交换的问题，Skyway 直接在节点之间传输 object，省略了数据的 native bytes 格式，但没有解决 object 额外内存开销过大的问题；Gerenuk 则将 Java 字节码编译为直接操作 native bytes 的语句，使得数据不以 object 形式存在，从根本上解决了之前研究成果遇到的问题。

针对语言虚拟机的大量优化仅关注了单机上的语言虚拟机的优化，没有关注全局的运行情况对语言虚拟机的影响。Taurus 使用 Holistic Runtime System 对分布式系统中每个节点上的 heap 进行管理，通过监控每个节点的运行情况，在合适的时间进行 GC，使得垃圾回收不影响分布式应用的性能。

Semeru 则关注内存解聚架构下的语言虚拟机 GC 性能问题。内存解聚架构由于其较好的可扩展性、容错性而越来越受到关注，然而这类架构需要应用有较好的内存访问局部性，管理型语言的 GC 却是一个图计算类型的工作负载，不具有局部性。Semeru 将

表 1: 相关研究工作对比

对比项	Facade	Broom	Taurus	HotTub	Skyway	Gerenuk	Yak	Semeru
发表会议	ASPLOS	HotOS	ASPLOS	OSDI	ASPLOS	SOSP	OSDI	OSDI
发表年份	2015	2015	2016	2016	2018	2019	2016	2020
减轻 GC 负担	✓	✓	✗	✗	✗	✓	✓	✗
减少 S/D[1]操作	✗	✗	✗	✗	✓	✓	✗	✗
控制/数据面分离	✓	✓	✗	✗	✓	✓	✓	✗
关注对象 lifetime	✗	✓	✗	✗	✗	✗	✓	✗
无需手动修改应用	✗	✗	✓	✓	✗	✓	✗	✓
缩短 JVM 预热时间	✗	✗	✗	✓	✗	✗	✗	✗
运行时系统间协调	✗	✗	✓	✗	✗	✗	✗	✗
资源解聚架构 GC	✗	✗	✗	✗	✗	✗	✗	✓
基于 JVM	✓	✗	✓	✓	✓	✓	✓	✓

GC 卸载到内存服务器上，提高了 GC 的内存访问局部性，使得管理型语言更适用于资源解聚架构。

HotTub 则关注 JVM 启动预热的开销问题。大数据框架中的延迟敏感型任务会因为管理型语言固有的性能问题而受到影响。研究者认为，语言虚拟机的预热加载过程是管理型语言性能问题的根源所在，HotTub 通过在内存中保留前次请求的 JVM 进程数据，从而免去了类加载、字节码解释执行的开销，但也加大了内存开销。

本章总结的工作较大部分集中在如何降低 Java 使用对象保存、管理数据的开销上，也有关注分布式系统下各个节点上语言运行时的协调问题，以及语言运行时的预热问题，较大程度上解决了语言运行时在大数据场景下表现出的性能问题。各类研究的对比见表 1。

[1]指 Serialization 序列化/Deserialization 反序列化

3 HotTub 研究思路分析

管理型语言相比于非管理型语言存在着语言运行时带来的额外开销。如 Java 语言，虽然无需程序员手动释放内存，JVM 进行自动 GC，降低了应用程序的开发复杂度以及出错的可能，但需要垃圾回收。语言虚拟机的性能优化研究有较高的热度，尤其是大数据时代的到来，使得管理型语言的应用范围更加广阔。然而，已有的研究成果大部分集中于优化垃圾回收、shuffling 的优化、以及调度优化等，但尚未对 Java 程序运行的整个过程中可能造成性能下降的部分有过研究。Lion 等人通过对 Java 工作负载的运行时间分解得出，Java 虚拟机（Java Virtual Machine，JVM）的预热占用了 Java 应用程序的大量运行时间，且 JVM 预热时间对延迟敏感型应用的响应时间有很大的影响。Lion 等人实现了 HotTub[18]，在多次请求中复用同一个长时间运行的 JVM，从而消除了 JVM 的预热开销，提高了交互式应用的性能。本章对 HotTub 的研究思路进行分析，并对其设计与实现进行介绍、测试了其带来的性能提升。

3.1 延迟敏感型应用中 Java 的性能问题

近年来，latency-sensitive（延迟敏感型）的交互型大数据应用受到广泛关注，例如 Hive[27]，是建立在 Hadoop 上的数据仓储系统，向用户提供类似于 SQL 的数据访问接口，掩盖了底层的分布式架构的复杂性，学习成本较低，且使得传统的 SQL 编写的应用更容易移植到分布式系统上，获得分布式系统所带来的好处。HiveQL 是 Hive 提供的数据库请求语言，当用户使用 HiveQL 向 Hive 发出 query（请求），Hive 将用户请求转化为 MapReduce、Apache Tez[28] 或 Spark 的 jobs。类似的 SQL query 引擎还有 Impala[29]、Spark SQL[30] 等。相比于批处理型任务，SQL query 需要更低的请求时延，故需要较高的性能。

然而，开发者们认为 Java 语言天生运行速度慢。例如，Hypertable[31] 的开发者使用 C++ 是因为 Java 在延迟敏感的环境下表现不佳，而 Hadoop 框架使用 Java 语言编写是因为其大部分工作是 I/O 处理。Hypertable 是一个内存密集型的程序，频繁调用 malloc，且需要大量的内存，内存不足会导致内存中的数据溢出到磁盘上，造成较大的性能影响。而 Java 的内存管理性能比 C++ 差 2 到 3 倍，这使得 malloc 的性能无法保证，且 Java 对象管理需要额外的内存开销，与 Hypertable 中的数据争夺内存空间。

也有开发者认为，只要运行环境的 CPU、内存资源充足，并且将运行时间较长的任务改写为并行度较高的、较短的任务，那么 JVM 造成的开销就可忽略不计。如果系统中有充足的内存，则 GC 的负担将会很小，即使需要 GC，也可以使用空闲的 CPU 资源而不抢占应用程序的 CPU 资源。JVM 的性能也与应用程序的特性有关，如果应用程序有一小部分频繁执行的代码，则会被 JIT（Just In Time）编译器编译为机器码而无需 JVM 解释执行。前面所述的各类性能问题需要通过对 Java 程序的运行进行全局的统计

测试，才可以找出真正的性能瓶颈，得知为什么 Java 会对延迟敏感型的大数据应用造成性能影响。

3.2 JVM 预热开销测试与分析

3.2.1 主要发现

是什么导致开发者们普遍认为 Java 比 C/C++ 更慢？经过在 JVM 中加入计时代码，记录应用各个部分的执行时间，研究者发现 JVM 的预热占据了整个执行时间的较大部分。JVM 的预热指 class loading[32]（类的加载）[4]，以及 bytecode interpretation[33]（字节码的解释执行）。

I/O 密集型应用也有较大的 JVM 预热开销 JVM 的预热开销在 I/O 密集型的工作负载中依然十分显著，从 HDFS 中读取 1G 的数据需要花费 33% 的时间进行 JVM 预热，同时 Java 字节码的解释执行也有巨大的开销，例如作为 HDFS 读取的性能瓶颈的 CRC checksum 计算，当 JIT 将其代码编译成机器码后有了 230 倍的性能提升，而只有频繁被执行的代码段才会被 JIT 系统编译为机器码。

JVM 的预热开销不会随着任务的运行时长变化 Spark 请求处理需要的 JVM 预热需要 21 秒，无论工作量的大小如何变化。这意味着，通过提高程序并行度从而优化程序性能（即将运行时间长的工作负载切分成可以并行运行的运行时间较短的工作负载）的方法因为 JVM 预热的开销较大而不再可行，甚至需要向另一个方向发展，即延长程序的运行时间，从而减小 JVM 预热时间占程序运行时间的比例。

复杂的软件栈进一步加大了 JVM 预热开销 虽然大数据框架提高了开发效率，简化了开发过程，但这建立在复杂的软件栈之上。例如，Spark 处理请求需要加载 19066 个 class，而 Hive 软件栈较为简单，仅需加载相比于 Spark 三分之一的 class。同时，由于 Spark 处理请求依赖的 class 较多，Spark 更容易调用一些很少调用的函数，这些函数只能够解释执行，这进一步增加了 JVM 的预热开销。但由于大数据处理框架的 homogeneity（同质性），这些类和函数会在不同的请求之间复用，这为我们提供了优化的可能。

3.2.2 JVM 预热开销的测量

为了更加准确地测量 JVM 在主流的工作负载下的预热开销，研究者在 JVM 代码中添加计时代码，从而测量单个线程中 class loading 以及 bytecode interpretation 的耗时。其中 class loading 已有工具可以统计一段时间内某个 JVM 进程的 class loading 情况，包括

[4]JVM 从 classpath 中按需加载应用程序所需的 class，从磁盘上读取.class 文件进入内存，并进行初始化。

加载、卸载的 class 数目以及内存大小、加载卸载 class 所用的时间，只需将 class loading 的状况改写为 thread local 进行统计即可。

而测量 bytecode interpretation 的耗时较为困难。JVM 代码有三种执行的模式：JVM 解释执行，JIT 编译执行，以及执行 C/C++ 语言的 native 代码。为了测量 JVM 解释执行 bytecode 的耗时，需要对执行模式转换进行标记，我们需要关注的模式转换有从 JVM 解释执行转换到 JIT 编译执行，以及从 JIT 编译执行转换到 JVM 解释执行。而这两种执行模式的转换发生在 `call` 指令和 `ret` 指令执行之时。对于 `call` 指令，如 bytecode 调用 JIT 编译好的代码，或 JIT 编译好的代码调用 bytecode 代码，问题比较简单，由于这样的调用必须首先经过一个 adapter。对于一个 Java 中的函数，都有一个 `Method` 结构体与之对应，其中包含了解释执行的字节码的入口点，以及 JIT 编译的机器码的入口点，还有 `i2c`、`c2i` 两个 adapter 分别供 bytecode 调用 JIT 编译好的代码、JIT 编译好的代码调用 bytecode 代码。adapter 的存在是由于两种执行模式的 calling convention 不同，即函数间传递参数的约定不同，它提供了标记执行模式转换的机会。而对于 `ret` 指令，则没有这样的标记机会。`ret` 指令仅仅将栈顶的返回地址弹出，并执行返回地址处的代码，没有标记的机会。

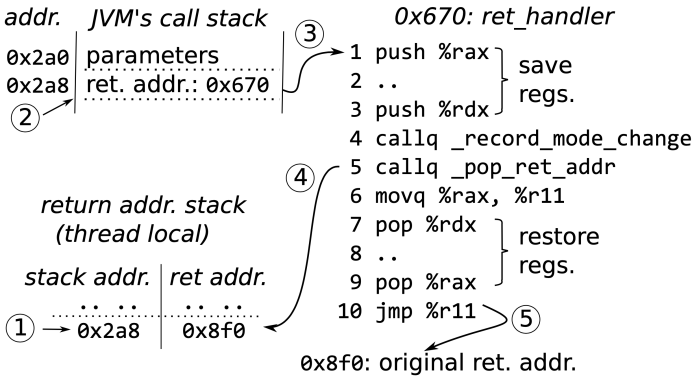


图 1: 截获 `ret` 指令的执行模式切换

故需要对 `ret` 指令进行特殊处理。如图1，其主要思想是将 `ret` 指令保存在栈上的返回地址修改为自定义的函数，通过 5 个步骤实现。(1) 每当引发执行模式切换的 `call` 指令执行时，进行标记，同时将原有的返回地址保存在一个 thread-local 的栈上；(2) 将原有的返回地址（`0x8f0`）替换为 `ret_handler` 的地址（`0x670:ret_handler`），使得 `ret` 指令执行后返回到 `ret_handler` 函数；(3) `ret_handler` 函数保存所有的寄存器，并调用 `_record_mode_change` 记录一次执行模式转换；(4) 调用 `_pop_ret_addr` 得到原有的返回地址（`0x8f0`），保存在`%r11` 寄存器中；(5) 恢复所有的寄存器，并且跳转到原有的返回地址，完成原有 `ret` 指令的功能。`ret_handler` 的实现使用了 15 行汇编代码。

但是，修改返回地址在一些 corner cases 下会产生错误，例如 GC 使用返回地址寻找 caller。在 GC 开始时，复原全部的返回地址，从而使得 GC 正常工作。Java 语言的异常追踪也依赖于返回地址，故在抛出一个异常之前也需要复原全部的返回地址。JVM

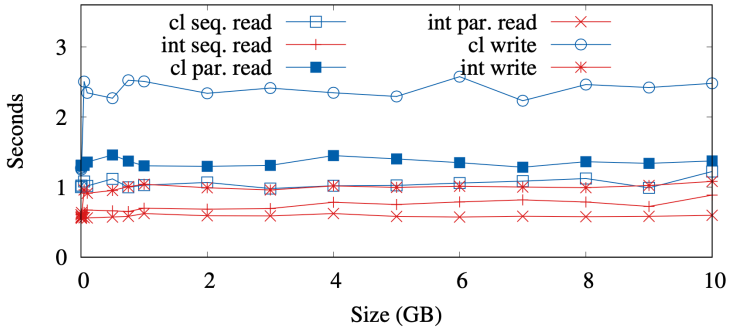


图 2: HDFS 请求预热耗时时长

的标记仅造成了微小的性能开销，当 class loading 和 bytecode interpretation 的注释全部开启时，HDFS 的工作负载仅受到了 3.3% 的性能影响。

3.2.3 JVM 预热开销分析

借助上一章中的 JVM 标记，本节对广泛使用的延迟敏感型大数据框架 Hive[27]、Spark SQL[30] 以及分布式文件系统 HDFS[5] 进行 JVM 预热开销分析。Hive 和 Spark SQL 是广泛使用的并行数据处理框架，运行在分布式文件系统 HDFS 上，它们将用户请求分解成为多个并行运行的较短的 jobs 进行处理。在 Spark 中，所有的短 jobs 运行在同一个 JVM 进程（*executor*）中，每个 JVM 线程运行一个 job；而在 Hive 中，每个 job 运行在一个独立的 JVM 进程中。本文使用 BigBench[34] 对 Spark SQL 和 Hive 进行测试，它包含了 30 个结构化的、半结构化的、非结构化的数据请求，均来自于真实环境，被 Cloudera、Horton-works 等公司用于其优化方案。本节测试运行在由 10 个服务器组成的室内集群中，它们通过 10Gbps 的网络相互连接。服务组件在测试前经过数周的预热和上千次的试运行。

对于 Spark，实验测试了 100, 300, 500, 700, 1K, 2K, 3K 共 7 个数据量的请求，Hive 则测试了 100, 300, 500, 700, 1K 共 5 个数据量的请求，其单位是 1G（如 100 的测试量代表请求 100G 的数据）。每个请求运行 10 次，且只取请求时间最短的一次的测试结果，这是为了避免同一个服务器上运行的其他线程（如 GC、JIT 线程，还有其他操作系统中的线程）对请求性能的影响，最真实地反应请求的时延。同时，由于 BigBench 测试的方面比较全面，包含了各种真实环境下可能出现的业务请求，其中包含了一些运行时间较长的请求，不属于延迟敏感型请求，作者只测试了 BigBench 中完成时间最短的 10 个请求（编号为 1, 9, 11, 12, 13, 14, 15, 17, 22, 24），作者关注的重点是 latency sensitive 的请求。JVM 标记仅测量了每个线程中的 JVM 预热耗时，却没有测量整个进程中 JVM 预热所占用的时间，作者使用 Ousterhout 等人提出的估算方法估算出了 JVM 预热在整个并行程序中所占的运行时间。

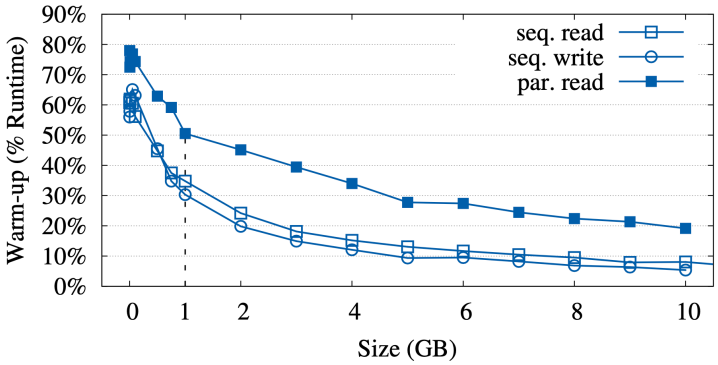


图 3: HDFS 请求预热耗时占运行时间百分比

HFDS 读写测试 由于尚未有现成的 HDFS 测试工具，作者自己编写了 sequential read，16 个线程的 parallel read，以及 sequential write 共 3 个客户端对 HDFS 性能进行分解测试。如图2所示是三个工作负载在不同的请求数据量大小下的 JVM 预热时间，其中 int 代表 bytecode interpretation，而 cl 代表 class loading。可见，JVM 预热耗时不随着数据量的增大而增大，sequential write 有更大的 JVM 预热开销，因为其运行了一条更加复杂的控制路径，使用了更多的 class。图3展示了预热时间占总运行时间的百分比，可见运行时间较短的、并行度较高的任务更容易受到 JVM 预热开销的影响，当数据量小于 1G 时，预热时间占 sequential read 的总运行时间的 33% 以上，占 parallel read 运行时间的 48% 以上。实际上，根据 Cloudera 发布的数据，真实场景下 Hadoop 的请求数据量小于 1G，因为 Hadoop 将用户的请求并行化为多个较小的请求。多数用户从 HDFS 中读取的数据量不超过 1MB，其中 60% 的时间消耗在了 JVM 预热上。

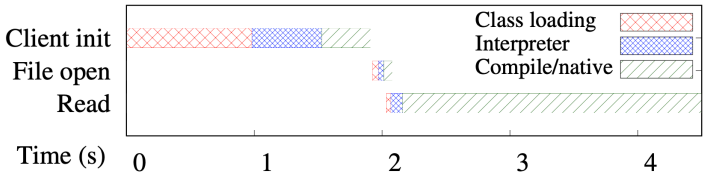


图 4: Sequential Read 1G 数据的运行时间分解

图4进一步展示了一个 sequential read 请求的整个运行过程中运行时间分解情况。可见，client 端初始化时进行了较长时间的 JVM 预热，才开始从 HDFS 的 datanode 打开文件、读取数据。图5进一步展示了 client 端和 datanode 端处理网络包的时间分解情况。当 datanode 收到了客户端的读请求之后，立刻返回了 13Bytes 的 ack 信息，继续调用 sendfile 系统调用发送 64KB 大小的数据包。如图可见，第 1 个 sendfile 花费了非常长的时间，因为需要从硬盘读取数据，而其余的数据包发送要明显快于第一个数据包。然而，客户端却由于 JVM 预热而无法及时相应 datanode 发送来的数据包，当客户端完成 JVM 预热并处理 ack 消息之后，datanode 已经发送了 13 个数据包；再加载了处理第 1 个数据包所需的 class、JIT 编译了相关代码后，完成了第 1 个数据包的处理，并进行了 CRC

checksum 的计算，花费了 26ms，此时 datanode 已经向客户端发送了 103 个数据包。由此可见，I/O 甚至不在 I/O 密集型应用的 critical path 上，影响 I/O 密集型应用性能的主要原因在于 class loading 和 bytecode interpretation。由图5也可以看到，解释执行的 CRC checksum 计算代码比 JIT 编译过的代码性能有巨大的差距，解释执行需要 $65ms$ 而执行编译过的代码仅需 $65\mu s$。这进一步说明了 JVM 预热开销的影响之大。

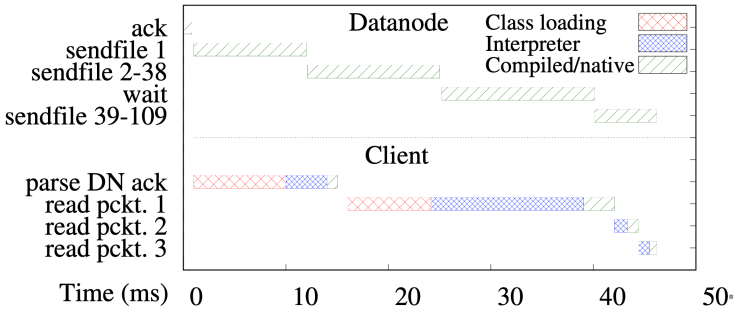


图 5: Datanode、client 数据包传送过程时间分解

Spark、Hive 请求测试 图6展示了 Spark、Hive 处理请求的运行时间分解。Spark、Hive 请求分别平均使用了 21.0s 和 12.6 秒进行了 JVM 预热，和 HDFS 测试相同，JVM 预热的耗时不随着请求数据量的变化而明显变化。由于 Spark 依赖的软件栈较为复杂（包括来自 Hadoop、Scala、derby 等 library 的 class），共计 19066 个 class，而 Hive 仅需加载 5855 个 class，故 Spark 的客户端在 class loading 上花费了 6.3 秒而 Hive 客户端仅花费了 3.7 秒。加载的 class 数量大也会导致解释执行耗时长，Spark 客户端调用了 242291 个函数，其中 91% 从未被 JIT 编译器编译，而 Hive 客户端仅调用了 113944 个函数，其中的 96% 从未被 JIT 编译器编译。

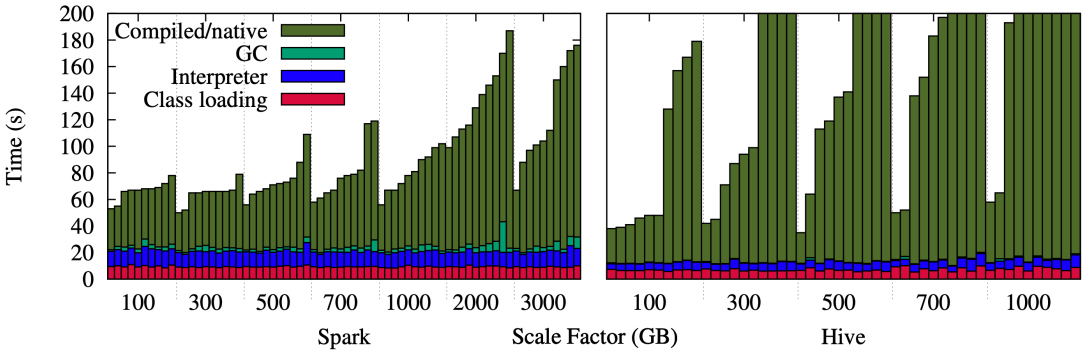


图 6: Spark、Hive 请求运行时间分解

图7展示了 Spark 的客户端与 executor（见第3.2.3节）在进行 query 13 测试时的执行时间分解。由于 JVM 预热时间不随请求数据量的变化而显著变化，故此时间分解图有较好的代表性。整个请求花费了 68s 完成，其中 12.4s 花费在 client 的 JVM 预热，

而 12.2s 花费在 executor 的 JVM 预热。由于 client 的 JVM 预热时间较长，executor 的 JVM 预热时间被覆盖。但在每个 iteration 开始之时，executor 同样经历了较长的 bytecode interpretation 时间。

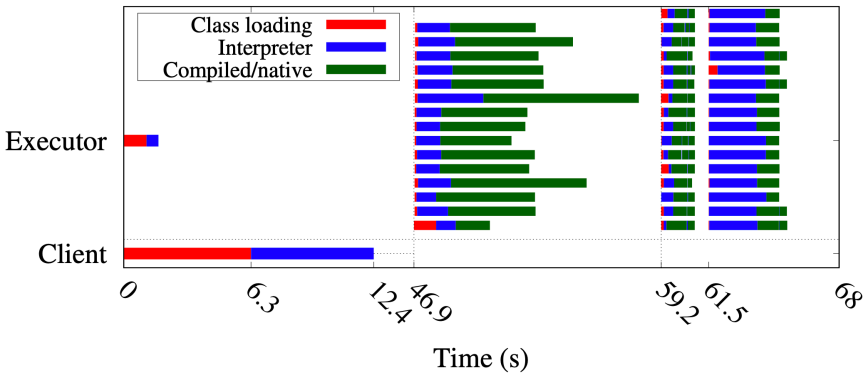


图 7: query 13 在 Spark 上执行的运行时间分解

Hive 和 Spark 并行化客户请求的方式有所不同，Hive 将用户的请求并行地执行在多个 JVM 进程中，而 Spark 仅有一个 executor 进程执行并行化的客户请求。每个 JVM 进程运行在 container 中，作为一个计算线程。然而，Hive 和 Tez 在处理同一个用户请求的过程中会复用用一个 JVM 的 container 进程，这在一定程度上减小了 JVM 预热开销对性能的影响，可以通过图6看到，Hive 遇到的 JVM 预热开销要小于 Spark。

测试收获总结 经过测试，研究者得知 JVM 预热耗时不随着请求数据量的变化而明显变化，同时运行时间短、并行度高的应用的运行时间会大部分地被 JVM 预热占据。事实上，大数据处理框架更倾向于进行运行时间短的请求，提升程序的并行度，充分利用分布式系统中充足的计算资源。复杂的分布式框架加重了 JVM 预热开销大的问题，因为需要加载更多的 class、执行更多的不同的函数。

3.3 HotTub 设计与实现

3.3.1 总体设计

HotTub 的设计目标是，使得应用程序能够重用 JVM 进程，即重用热数据，从而减小 JVM 的预热开销。有两种设计策略：一是显式地在 JVM 进程之间拷贝热数据，二是在正确地重置 JVM 进程的状态后重新使用同一个 JVM 进程。研究者首先使用第一种设计策略进行开发，通过将 class meta-data 与 JIT 编译好的机器码保存在磁盘上等待下一个 JVM 进程重用。虽然成功地在不同的 JVM 进程之间共享了已经加载到内存重的 class，但研究者们最终放弃了这个设计策略，由于维护两个不同地址空间中的指针的一致性太过复杂。举例而言，JIT 编译生成的代码是不可重定位（relocatable）的，而一个

JIT 编译过的函数可能跳转到另一个 JIT 编译过的函数，产生错误。为了解决这个问题，我们必须在同一个 JVM 的地址空间中进行已加载 class、JIT 编译代码的共享，这不具有灵活性；或修改所有指针的值，这是不现实的。

图 8: HotTub 架构

于是，研究者采用了第二个设计策略，即在多次用户请求中重用同一个 JVM 进程，同时利用 JVM 自带的 GC 功能清除垃圾数据。图8是 HotTub 的架构。HotTub 无需用户修改代码，和未修改的 JVM 相同，只需运行 `java` 命令即可启动 java 应用。而与未修改的 JVM 不同，执行 HotTub 的 `java` 命令会 `fork` 出一个 HotTub 的 *client*，与 JVM pool 中已经预热了的 JVM 进行通信，称之为 *server*，并将执行 java 程序的任务移交给 server。server 运行完毕，并将返回值发送给 client，使得 client 正常退出。server 完成一次 client 的请求之后进行 GC，并重置 JVM 的状态，以供下次 client 请求使用。

```
1  struct sockaddr_un add; // create unix sock.
2  char* sum = md5(classpath);
3  while (true) {
4    for (int i = 0; i < POOL_SIZE; i++) {
5     strcpy(add.sun_path,strcat(sum,itoc(i)));
6     if (connect(fd, add, sizeof(add))==0)
7       return reuse_server_and_wait(fd);
8     if (server_busy(i))
9       continue;
10    /* No JVM/server created. */
11    if (fork() == 0) // spawn new jvm in child
12      exec("/HotTub/java", args);
13    /* else, parent, go back to find server */
14   }
15 }
```

图 9: HotTub client 运行过程

如图9是 client 的具体执行过程。为了使得进程找到最合适的 JVM 进程，client 对 classpath 以及 classpath 上的所有包含 class 的文件计算 checksum，从而使得使用相同 class、执行类似的代码的 java 程序重用同一个 JVM server，从而最大程度地减小 class loading 以及 bytecode interpretation 的开销。同时，client 在 checksum 尾部加上一个 `0` 到 `POOL_SIZE` 之间的数字（图9中第 5 行），作为一个 server 的地址（ID），client 使用该 ID 与 server 进行通信。client 会尝试与 `0` 到 `POOL_SIZE` 之间的 server 进行通信，如果一个 server 正在忙于处理某个客户端的请求，则跳过（图9中第 9 行）。如果没有找到合适的 server，则通过 `fork` 创建一个新的 JVM 进程。这样的 `fork` 设计具有良好的容错

性，同时也为应用找到最合适的 server 提供了可能：最合适的 server 不但会降低 JVM 预热开销，同时提升了 memory cache、TLB 的命中率，由于无需切换或重新创建进程，操作系统无需冲刷 memory cache 和 TLB。

3.3.2 设计细节

HotTub 的设计困难在于，要保证应用在 HotTub 上执行的结果与在未修改的 JVM 上运行得到的结果相同，即保证一致性。首先，栈上的数据不会造成不一致性，因为 Java 程序退出时会将栈中的数据全部弹出。其次，HotTub 的 server 在处理完一个 client 的请求之后即运行 GC，堆上的数据也不会造成不一致性的问题。在 JVM server 进程地址空间中残留的数据有（1）已加载的 class 信息，（2）JIT 编译好的机器码，（3）静态变量，（4）文件描述符。这三类需要保证一致性的数据中，前三者保存在 JVM 的方法区，需要在 server 请求运行完毕后重置；文件描述符需要 client 传输给 server，才能使得 server 正常执行。

已加载的 class 信息 HotTub 必须保证 HotTub server 中保留的 class 信息与正常启动的 JVM 进程相同，这些信息可能在运行时被改变。在保证 class 信息一致性的同时，也保证了 JIT 已编译机器码的一致性，由于 JIT 已编译机器码来自于 class 的字节码（bytecode）。事实上，在 client 选取 server 之时，已经对 classpath 中所有的 class 计算了 checksum，这使得从 classpath 加载的 class 有了一致性。而用户代码自动生成的 class 将不会保留到下一次 server 请求的执行，由于每次 server 运行必定会产生一个全新的 class loader。

静态变量 在 server 请求执行后，静态变量残留的值依然存在。HotTub 必须重置这些静态变量从而保证一致性。调用 class 的 <clinit> 初始化方法即可重置这些静态变量。然而，需要保证每个 class 的 <clinit> 函数的调用顺序正确。事实上，有初始化时间依赖的静态变量是 bad programming practice，即 class A 的静态变量初始化依赖于 class B 的静态变量初始化。这会导致程序难以调试，理解也十分困难。事实上，在测试阶段，尚未有大数据框架出现了这类坏的编程风格，故我们无需关注这一 corner case。

文件描述符 HotTub server 在处理完一次 client 请求后即关闭所有由应用程序打开的的文件描述符，包括 stdin，stdout，stderr，以供下一个请求的处理。剩下没有关闭的文件描述符均是由 JVM 打开，大多是 JAR 文件。当新的 client 请求到来时，client 将所有的文件描述符通过 Unix domain socket 发送给 server，这样 server 即可继承 client 所有已经打开的文件描述符。但会存在 server 占用了 client 已经打开的文件描述符的情况，故在 client 挑选 server 时，会检查是否存在文件描述符的冲突。

信号处理与退出 HotTub 需要处理 `SIGTERM` 和 `SIGINT` 信号，从而使得应用程序按照预期的行为退出。如果应用程序注册了 `SIGTERM` 和 `SIGINT` 的信号处理函数，则将信号转发给应用。如果应用没有注册相应的信号处理函数，则将非守护的 Java 线程的栈解开，再 kill 掉此线程。JVM 守护线程不会在 JVM 退出时退出，我们同样需要将其 kill，以保证一致性。但是，如果进程调用了 native library 中的 `_exit` 函数，则无法保留这个 server 供未来的 client 请求使用。

3.3.3 实现细节

HotTub client 的实现独立于 JVM，共 800 行 C 代码；server 的实现基于 OpenJDK 的 HotSpot JVM，添加了 800 行 C/C++ 代码，server 和 client 在不同的进程中。由于 Unix domain socket 可以方便地在进程之间发送文件描述符，我们使用 Unix domain socket 进行 client 和 server 之间的信息传递，且这样可以免去在 client、server 进程之间传递文件数据的必要，大大提升了 client 和 server 的交互性能。

进程管理 对于 OpenJDK 中 HotSpot 的修改，作者没有使用新创建的线程接收 client 的请求，而是修改 Java 的 main thread，使之更像一个 server：在 Java main thread 的逻辑结束运行后，使用 main thread 完成一系列的重置操作，包括：（1）kill 其他的 Java 线程（2）重置所有静态变量为默认值（3）对 heap 进行 GC（4）可选择卸载所有加载的 native library。当一个 server 接收到 client 的连接后，server 重新初始化所有静态变量，正确地设置所有文件描述符，并正确地设置所有 Java 属性值、环境变量等，最终调用 `main()` 函数。

静态变量初始化 对于 JVM 中所有的 class，都存在相应的 `java.lang.Class` 对象，可以通过该对象访问该 class 的 field 和 method。然而，对于一个 Enum 类型的 class，其 `java.lang.Class` 对象保存了 Enum 名字符串和对应的 object 的映射，而在每次静态变量初始化后，都会心产生一个对应于 Enum 名字符串的 object，故在静态变量初始化之后应当更新 Enum 的 `java.lang.Class` 对象中保存的映射关系。其次，启用 static final reference inline 的 JIT 编译过的代码会直接访问 inline 的 object 的地址。在 HotTub 中，static final 的 object 将被重新初始化，其地址也会改变。为了方便起见，我们将 static final reference inline 特性关闭。

3.4 HotTub 性能测试

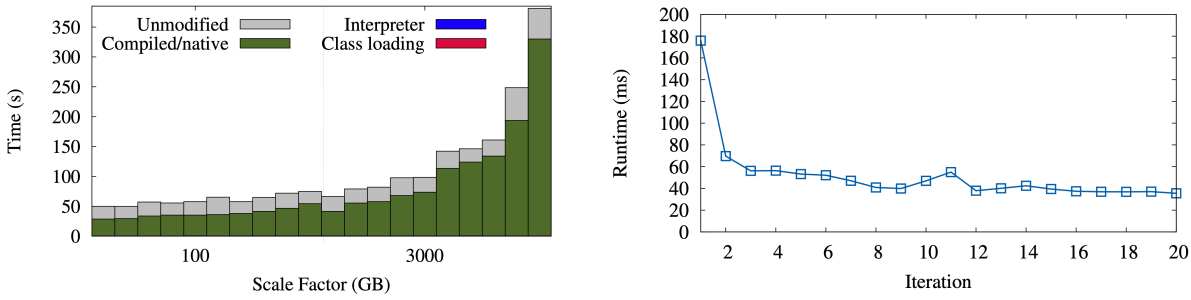
本节对 HotTub 各方面的性能进行了测试。首先，通过在 HotTub 和未修改的 JVM 上不断重复执行相同的 workload，测试 HotTub 是否可以消除 JVM 预热开销。其次，由于相同的 workload 最大化了 warm data 的复用，HotTub 会表现出最好的性能，还需要测

试不同的 workload 下 HotTub 的性能降低程度。最后，由于 HotTub 的 server 和 client 需要通信，server 处理完一个 client 的请求之后需要重置 JVM 的状态，新建 sever 时需要调用 `fork` 等系统调用，且在内存中驻留的 server 进程保存了 warm data，占据了额外的内存，HotTub 相比于未修改的 JVM 有额外的开销。我们称之为管理开销（management overhead）。如果管理开销过大，则 HotTub 不具有实用性，我们还需测试 HotTub 的管理开销。

3.4.1 性能提升

首先我们测试 HotTub 利用 warm data 减小 JVM 预热开销的能力。对于未修改的 JVM，我们运行五次相同的 workload；对于 HotTub，首先运行此 workload 一次，完成 warm data 的加载，再运行此 workload 五次。通过比较在未修改的 JVM 上运行五次相同的 workload，和在 HotTub 预热后运行五次相同的 workload，可以观察 HotTub 是否达到了减小预热时间的目的。

首先，我们关闭 JVM 中的时间标记，对真实的大数据场景下的高频事件进行测试。重新进行3.2.3中的实验，HotTub 显著提升了延迟敏感性应用的性能。对于 Spark SQL，每次请求 100GB 数据耗费的时间减小了 29 秒。对于 HDFS，读取 1MB 数据的性能被提升了 30.08 倍。在 200 次对比测试中，所有的在 HotTub 上执行的任务均比未修改的 JVM 上执行的任务耗时更短。



(a) HotTub 上 Spark 请求执行时间分解测试　　(b) HotTub 上的 HDFS read 请求测试

图 10: HotTub 性能分析

再打开 JVM 中的时间标记，对 JVM 中各部分的执行时间进行观察。如图10（a），在所有的测试中，HotTub 上运行的应用需要的 JVM 预热时间总是小于 1% 的总执行时间，图中的预热时间太小已经无法观察到。如图10（b），可以观察到在 HotTub 上初次执行的应用仍然有较长的延时，这是由于尚未有 warm data 以供使用。同时可以观察到，经过了 12 次的重复执行，HDFS read 达到了最高性能，这进一步说明较短的任务甚至在没有达到最高执行性能时已经完成执行，HotTub 保存 warm data 对于短任务的性能有极高的提升（30.08 倍，如前文所述）。在微观性能上，HotTub 减小了 memory footprint，也减小了 cache miss，同时减小了进程切换导致的 TLB flush，由于篇幅原因，这些微观

表 2: Spark query 的不同对 HotTub 性能优化的影响程度

特点 \ 算法	q11	q14	q15	q09	q01
q11	1.78	1.67	1.51	1.49	1.55
q14	1.64	1.65	1.47	1.49	1.50
q15	1.72	1.67	1.62	1.54	1.62
q09	1.57	1.59	1.55	1.53	1.53
q01	1.76	1.74	1.65	1.54	1.74

指标的测试不做介绍。

3.4.2 工作负载变化的性能影响

在真实场景下，JVM 很少会执行同一个 query 多次，而是不断更换处理的工作负载的种类。这在一定程度上削尖了 HotTub 的优势，由于不同的工作负载将用到不同的 class 和 JIT compiled code。为了测试这种工作负载的变化对 HotTub 的性能影响，我们首先运行 training query 多次，使得 HotTub server 中充满 training query 的 warm data，再执行一次不同于 training query 的 testing query，通过对比此情况下 testing query 的性能与在未修改的 JVM 上运行的性能，即可得知工作负载变化对 HotTub 的性能影响。如表2所示，由于 query 之间的相似性（使用相同的 class 和 code），不同 query 之间获得的性能提升有最低的 1.47 倍（q14->q15），但相同 query 获得的性能提升最高有 1.78 倍（q11->q11）。

3.4.3 引入的管理开销

管理开销可以分为两部分，一部分在 critical path 上，直接影响 client 的性能，如 client 与 server 建立连接，server 的 class 初始化；另一部分不在 critical path 上，仅影响 server 在处理完一次 client 请求之后多久才可以被重新使用，如 server 重置所有静态变量。经过测试，当 JVM pool 中没有待使用的 server 时，client 需要 $81ms$ 与 server 建立连接；如果有可重用的 server，则连接耗时仅为 $300\mu s$。server 中 class 初始化耗时对于 Hive 而言是 $350ms$，对于 Spark executor 是 $400ms$，对于 Spark 客户端是 $720ms$。每个未使用的已预热过的 JVM server 占用约 1GB 的内存。

3.5 HotTub 适用性与局限性分析

4 HotTub 局限性优化

// TODO

5 总结

// TODO

参考文献

[1] APACHE. Apache spark - unified analytics engine for big data[M]. Website: http://spark.apache.org, 2020.

[2] APACHE. Apache hadoop[M]. Website: http://hadoop.apache.org, 2020.

[3] YU Y, ISARD M, FETTERLY D, et al. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language[C/OL]//DRAVES R, VAN RENESSE R. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. USENIX Association, 2008: 1-14. http://www.usenix.org/events/osdi08/tech/full_papers/yu_y/yu_y.pdf.

[4] MURRAY D G, MCSHERRY F, ISAACS R, et al. Naiad: a timely dataflow system[C/OL]//KAMINSKY M, DAHLIN M. ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013. ACM, 2013: 439-455. https://doi.org/10.1145/2517349.2522738.

[5] APACHE. Hadoop distributed file system (hdfs)[M]. Website: http://hadoop.apache.org/docs/stable/hdfs_design.html, 2020.

[6] APACHE. Apache hbase[M]. Website: http://hbase.apache.org, 2020.

[7] APACHE. Cassandra[M]. Website: http://cassandra.apache.org, 2020.

[8] NGUYEN K, WANG K, BU Y, et al. FACADE: A compiler and runtime for (almost) object-bounded big data applications[C/OL]//ÖZTURK Ö, EBCIOGLU K, DWARKADAS S. Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015. ACM, 2015: 675-690. https://doi.org/10.1145/2694344.2694345.

[9] GOG I, GICEVA J, SCHWARZKOPF M, et al. Broom: Sweeping out garbage collection from big data systems[C/OL]//CANDEA G. 15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015. USENIX Association, 2015. https://www.usenix.org/conference/hotos15/workshop-program/presentation/gog.

[10] CAVALIERI S, SCROPPO M S, GALVAGNO L. A framework based on CLR virtual machine to deploy IEC 61131-3 programs[C/OL]//14th IEEE International Conference on Industrial Informatics, INDIN 2016, Poitiers, France, July 19-21, 2016. IEEE, 2016: 126-131. https://doi.org/10.1109/INDIN.2016.7819146.

[11] DEAN J, GHEMAWAT S. Mapreduce: simplified data processing on large clusters[J/OL]. Commun. ACM, 2008, 51(1):107-113. http://doi.acm.org/10.1145/1327452.1327492.

[12] ELSMAN M, HALLENBERG N. On the effects of integrating region-based memory management and generational garbage collection in ML[C/OL]//KOMENDANTSKAYA E, LIU Y A. Lecture Notes in Computer Science: volume 12007 Practical Aspects of Declarative Languages - 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20-21, 2020, Proceedings. Springer, 2020: 95-112. https://doi.org/10.1007/978-3-030-39197-3_7.

[13] QUORA. In what cases is java faster than c[M]. Website: https://www.quora.com/In-what-cases-is-Java-faster-if-at-all-than-C, 2015.

[14] QUORA. In what cases is java slower than c by a big margin[M]. Website: https://www.quora.com/In-what-cases-is-Java-slower-than-C-by-a-big-margin, 2015.

[15] STACKOVERFLOW. Why do people still say java is slow[M]. Website: http://programmers.stackexchange.com/questions/368/why-do-people-still-say-java-is-slow, 2010.

[16] JAYNENE D. Performance comparison - c++/java/python/ruby/jython/jruby/groovy[M]. Website: http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/, 2008.

[17] HYPERTABLE. Why we chose cpp over java[M]. Website: https://code.google.com/p/hypertable/wiki/WhyWeChoseCppOverJava, 2014.

[18] LION D, CHIU A, SUN H, et al. Don't get caught in the cold, warm-up your JVM: understand and eliminate JVM warm-up overhead in data-parallel systems[C/OL]//KEETON K, ROSCOE T. 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. USENIX Association, 2016: 383-400. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lion.

[19] MAAS M, ASANOVIC K, HARRIS T, et al. Taurus: A holistic language runtime system for coordinating distributed managed-language applications[C/OL]//CONTE T, ZHOU Y. Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016. ACM, 2016: 457-471. https://doi.org/10.1145/2872362.2872386.

[20] NGUYEN K, FANG L, NAVASCA C, et al. Skyway: Connecting managed heaps in distributed big data systems[C/OL]//SHEN X, TUCK J, BIANCHINI R, et al. Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018. ACM, 2018: 56-69. https://doi.org/10.1145/3173162.3173200.

[21] ESOTERICSOFTWARE. Kyro[M]. Website: https://github.com/EsotericSoftware/kryo, 2020.

[22] NAVASCA C, CAI C, NGUYEN K, et al. Gerenuk: thin computation over big native data using speculative program transformation[C/OL]//BRECHT T, WILLIAMSON C. Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. ACM, 2019: 538-553. https://doi.org/10.1145/3341301.3359643.

[23] ZAHARIA M, CHOWDHURY M, DAS T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C/OL]//GRIBBLE S D, KATABI D. Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012. USENIX Association, 2012: 15-28. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[24] WANG C, MA H, LIU S, et al. Semeru: A memory-disaggregated managed runtime[C/OL]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020: 261-280. https://www.usenix.org/conference/osdi20/presentation/wang.

[25] GU J, LEE Y, ZHANG Y, et al. Efficient memory disaggregation with infiniswap[C/OL]//AKELLA A, HOWELL J. 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017. USENIX Association, 2017: 649-667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[26] NGUYEN K, FANG L, XU G H, et al. Yak: A high-performance big-data-friendly garbage collector[C/OL]//KEETON K, ROSCOE T. 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. USENIX Association, 2016: 349-365. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen.

[27] APACHE. Apache hive tm[M]. Website: https://hive.apache.org/, 2014.

[28] APACHE. Apache tez[M]. Website: http://tez.apache.org/, 2014.

[29] APACHE. Apache impala[M]. Website: https://impala.apache.org/, 2020.

[30] ARMBRUST M, XIN R S, LIAN C, et al. Spark SQL: relational data processing in spark[C/OL]//SELLIS T K, DAVIDSON S B, IVES Z G. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015. ACM, 2015: 1383-1394.

https://doi.org/10.1145/2723372.2742797.

[31] RIOS G, JUDD D. Load balancing for hypertable[C/OL]//AAAI Workshops: WS-11-08 AI for Data Center Management and Cloud Computing, Papers from the 2011 AAAI Workshop, San Francisco, California, USA, August 7, 2011. AAAI, 2011. http://www.aaai.org/ocs/index.php/WS/AAAIW11/paper/view/3802.

[32] STACKOVERFLOW. Java verbose class loading[M]. Website: https://stackoverflow.com/questions/10230279/java-verbose-class-loading, 2012.

[33] STACKOVERFLOW. Why is java bytecode interpreted?[M]. Website: https://stackoverflow.com/questions/5625512/why-is-java-bytecode-interpreted, 2011.

[34] GHAZAL A, RABL T, HU M, et al. Bigbench: towards an industry standard benchmark for big data analytics [C/OL]//ROSS K A, SRIVASTAVA D, PAPADIAS D. Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013. ACM, 2013: 1197-1208. https://doi.org/10.1145/2463676.2463712.

A 附录

A.1 //