



“大学生创新训练计划”项目结题报告

中文题目 基于SDN的云中大数据网络优化的研究

英 文 Research of bigdata network optimization

题 目 based on SDN in cloud computing

学 院 信息学院

指导教师 张文逸

小组成员:

姓 名 孟思尧 学 号 PB12210076

姓 名 王笑霄 学 号 PB12210100

姓 名 张广帅 学 号 PB12210080

2016 年 5 月 15 日

摘要

由于当前数据中心用并行计算的框架处理大规模的数据，根据调查发现网络是限制大数据处理性能的一大瓶颈。针对网络问题，我们在软件定义网络平台上针对开源的Apache Hadoop MapReduce 平台实现了一个可行的调度控制系统，并将Coflow 的概念引入了大数据平台，建立了控制架构，实现了两个Coflow的调度算法。仿真结果显示我们设计的最小化Coflow完成时间的调度算法要比先到达先服务的调度算法节省10倍左右的Coflow完成时间。进一步我们尝试通过一系列工业优化方法克服SDN 中高带宽TCP 应用的难点问题，如包超时重传导致的拥堵以及难以直接从Hadoop MapReduce 模块中拿到调度流所必须的信息等等。测试结果表明，在使用我们的调度器后，MapReduce 任务的性能在SDN 环境下提升了5%。

关键词：云计算，大数据，SDN，HADOOP，网络优化

ABSTRACT

KEY WORDS: Cloud computing, Bigdata, SDN, HADOOP, network optimization

Contents

1 项目背景	5
1.1 项目的意义和关键问题	5
1.2 研究方向	5
2 项目平台的选择	6
2.1 Hadoop	6
2.1.1 为什么选择Hadoop	6
2.1.2 Mapreduce计算框架	6
2.1.3 Shuffle阶段	8
2.2 SDN	8
2.2.1 Floodlight	8
2.2.2 OVS	9
3 基于Coflow的大数据网络优化	11
3.1 coflow相关知识介绍	11
3.2 coflow调度算法设计	13
3.3 coflow调度算法的性能表现	13
4 实验平台搭建	15
4.1 实验环境	15
4.2 Hadoop平台搭建	16
4.3 SDN开发环境搭建	18
4.4 网络拓扑	19
5 系统设计与开发	19
5.1 系统整体架构设计	19
5.2 MapReduce Fetcher	19
5.3 Python Controller (PyCon)	21
5.3.1 HTTPServer 类	21
5.3.2 配置队列, 进行限速	21
5.3.3 获取网络信息	21
5.3.4 流表预配置	21
5.3.5 下发流表	22
5.3.6 路由算法模块	22
5.3.7 Coflow 调度	22
5.4 对Floodlight 的改进	22
5.4.1 Packet-in 数据包处理模块	22

5.4.2 REST API	23
6 项目的成果	23
6.1 实验仿真测试	23
6.2 实验结果	23
6.3 结果分析	25
7 项目总结	26
8 参考文献	26

1 项目背景

1.1 项目的意义和关键问题

大数据应用正深刻改变人们的生活，已经成为当前学术界、工业界的关注焦点。大数据应用开发云平台纷纷推出大数据计算框架。如Amazon的EC2，微软的Azure。虽然云平台上的大数据框架方便了第三方开发者和用户，但其性能上还存在较多问题，尤其是网络的问题。在2014 Daytona GraySort排序赛上，基于Spark的系统它使用了207个EC2节点在23分钟内排序了100TB的数据而夺冠。而上届冠军Hadoop用了2100台Yahoo内置的机器，花了72分钟，这性能提升不言而喻！更重要的是这次比赛证实Shuffle真正的瓶颈在于网络。传统大数据网络架构是数据库服务器将应用服务器请求的数据通过网络传播到应用服务器上，处理后，将改写的数据再写回数据库服务器，而且处理过程中会出现大量的中间结果也需要网络传播。当数据量较大时，很可能堵塞网络。因而对于进一步提升大数据处理性能，研究云平台上大数据网络问题显得尤为重要。

1.2 研究方向

软件定义网络（SDN） 软件定义网络（SDN）是当前和未来网络研究的一个重要方向，通过将网络的数据转发层和逻辑控制层分离，提高了网络的灵活性和可编程性。可以在控制层面设计应用感知的控制策略，如网络接入、数据转发路由、流量工程等，从而提高应用的性能。因此，我们拟通过SDN来优化云平台上大数据框架的网络性能。

coflow调度 考虑到当前数据中心用并行计算的框架处理大规模的数据，充分考虑数据流的相关性（coflow），设计调度算法

2 项目平台的选择

2.1 Hadoop

2.1.1 为什么选择Hadoop

数据并行计算框架很多，如：Mapreduce，Spark，Google Dataflow等等，我们为什么选择Hadoop？首先，认识一下Hadoop，Hadoop是一种计算集群，它将数据分析的工作分配到多个集群节点上，从而并行处理数据。经过调研，Hadoop用于大数据处理，主要有如下几个优势：

- 1、灵活的可扩展性** 要充分利用大数据最大的优势就需要实时或接近实时地对海量数据进行分析处理。大数据分析面临的一个巨大的难题是数据量的不断增加。而Hadoop集群的并行处理能力能明显提高分析速度，但随着要分析的数据量的增加，集群的处理能力会受到影响。Hadoop通过增加集群节点，可以线性地扩展集群以处理更大的数据集。另外，在集群负载下降时，也可以减少节点，以高效使用计算资源。所以Hadoop的弹性很好
- 2、Hadoop的设计适合大数据处理** 大数据一般都是分布广泛的并且是非结构化的。而Hadoop非常适合处理这类数据，因为Hadoop的mapreduce的计算框架工作原理是将数据拆分成片，并将每个“分片”分配到特定的集群节点上进行分析。数据不必均匀分布，因为每个数据分片都是在独立的集群节点上进行单独处理。
- 3、Hadoop成本低** Hadoop的软件是开源的，同时，Hadoop支持商用硬件，可以运行在一般商业机器构成的大型集群上，如：亚马逊弹性计算云（Amazon EC2）等，不必花费重金购买服务器级别的硬件设备。所以我们可以低成本的实现计算能力强的Hadoop集群，性价比很高。
- 4、容错能力强** 在Hadoop集群进行大数据处理分析过程中，当一个数据分片发送到某个节点进行分析时，该数据在集群其它节点上会存有副本。通过备份的方式，即使一个节点发生故障，数据可以快速的恢复，继续进行分析处理。故障检测和自动恢复是Hadoop最初的设计目标，所以Hadoop很健壮。

由于Hadoop具有上述优势，使得Hadoop在学术界和工业界都大受欢迎。

2.1.2 Mapreduce计算框架

Mapreduce是什么？怎样完成大数据处理？

Hadoop的设计思路源于Google的GFS和MapReduce。它是一个开源软件框架，通过在集群计算机中使用MapReduce这个简单的编程模型，可编写和运行分布式应用程序处理大规模数据。

MapReduce架构的大数据处理的操作流程如下：

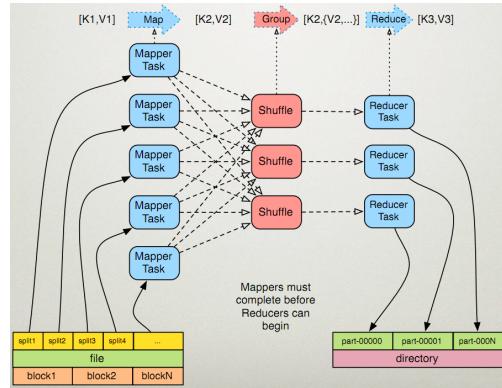


Figure 1: Mapreduce计算框架

1. 首先调用MapReduce库的输入流模块，将输入文件分成M个数据片度，每个数据片段的大小可以通过可选的参数来控制。然后用户程序在机群中创建大量的程序副本。程序副本中包含一个特殊的master程序。其它的程序都是worker程序，由master分配任务。有M个Map任务和R个Reduce任务将被分配，master将一个Map任务或Reduce任务分配给一个空闲的worker。
2. 被分配了map任务的worker程序读取相关的输入数据片段，从输入的数据片段中解析出key/value pair，然后把key/value pair传递给用户自定义的Map函数，由Map函数生成并输出的中间key/value pair，并缓存在内存中。
3. 缓存中的key/value pair通过调用Partition模块分成R个区域，之后周期性的写入到本地磁盘上。缓存的key/value pair 在本地磁盘上的存储位置将被传给master，由master负责把这些存储位置再传送给Reduce worker。
4. 当Reduce worker程序接收到master程序发来的数据存储位置信息后，使用RPC从Map worker所在主机的磁盘上读取这些缓存数据。当Reduce worker 读取了所有的中间数据后，通过对key进行排序后使得具有相同key值的数据聚合在一起。由于许多不同的key值会映射到相同的Reduce 任务上，因此必须进行排序。如果中间数据太大无法在内存中完成排序，那么就要在外部进行排序。
5. Reduce worker程序遍历排序后的中间数据，对于每一个唯一的中间key值，Reduce worker程序将这个key值和它相关的中间value值的集合传递给用户自定义的Reduce函数。Reduce函数的输出被追加到所属分区的输出文件。
6. 当所有的Map和Reduce任务都完成之后，master唤醒用户程序。在这个时候，在用户程序里的对MapReduce调用才返回。在成功完成任务之后，MapReduce的输出存放在R个输出文件中（对应每个Reduce任务产生一个输出文件，文件名由用户指定）。一般情况下，用户不需要将这R个输出文件合并成一个文件-他们经常把这些文件作为另外一个MapReduce的输入，或者在另外一个可以处理多个分割文件的分布式应用中使用。

2.1.3 Shuffle阶段

之前我们说shuffle阶段是制约大数据处理的网络瓶颈，下面我们将对Mapreduce的shuffle阶段进行详细的介绍。整体的Shuffle过程包含以下几个部分：Map 端Shuffle、Sort 阶段、Reduce 端Shuffle。即是说：Shuffle 过程横跨map 和reduce 两端，中间包含sort 阶段。

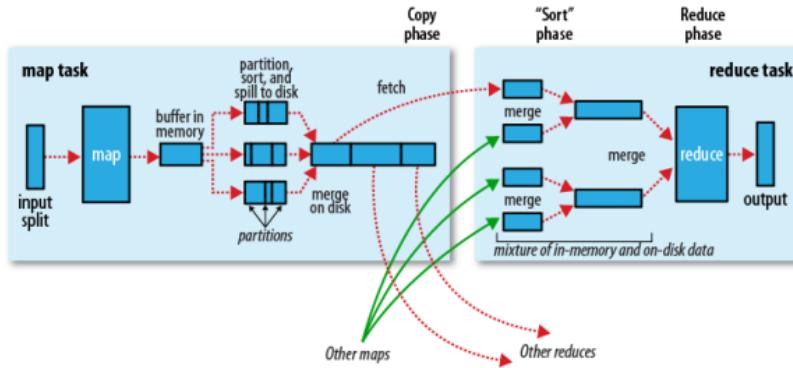


Figure 2: shuffle阶段

在Hadoop集群中，大部分map task与reduce task的执行是在不同的节点上，Reducer通过HTTP方式得到map阶段输出文件的分区，所以很多情况下Reduce 执行时需要跨节点去拉取其它节点上的map task结果。在Mapreduce的Shuffle阶段，如果集群正在运行的job有很多，而且需要跨节点拉取数据时，会产生大量的数据在网络中传输，对集群内部的网络资源消耗会很严重，尽可能地减少对带宽的不必要消耗，并且保证完整地从map task 端拉取数据到reduce端。

2.2 SDN

2.2.1 Floodlight

SDN（软件定义网络）利用OpenFlow协议，把路由器的控制平面（control plane）从数据平面（data plane）中分离出来，以软件方式实现。这个架构可以让网络管理员，在不改动硬件设备的前提下，通过集中式的控制器（Controller）以标准化的接口对各种网络设备进行管理和配置，那么这将为网络资源的设计、管理和使用提供更多的可能性，为控制网络流量提供了新的方法。

控制器作为SDN网络中的重要组成部分，我们选择Floodlight作为SDN控制器。因为Floodlight是目前主流的SDN控制器之一，它的稳定性、易用性已经得到SDN专业人士一致好评，由于其完全开源，这让SDN网络世界变得更加有活力。能集中地灵活控制SDN网络，为核心网络及应用创新提供了良好的扩展平台。

Floodlight控制器是一个企业级的，使用Java开发的OpenFlow协议的控制器。OpenFlow是一个由Open Networking Foundation (ONF)管理的开放标准。它定义了一种协议让远程控制器通过路由器可以修改网络设备的行为，使用定义良好的转发指令集。Floodlight被设计为同支持OpenFlow标准的设备（交换机，路由器，虚拟交换机）一起工作。

Floodlight controller模块为多数应用实现了一些通用的功能:

1. 发现网络状态和事件（拓扑结构，设备，流量）
2. 能够控制网络交换机（network switches）通信
3. 管理floodlight模块，共享存储，线程，测试等资源
4. 提供一个web界面和debug服务器（Python）

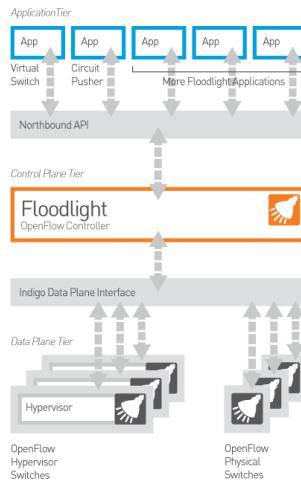


Figure 3: Floodlight控制器架构图

Floodlight控制器工作过程如下:

1. 控制器与交换机建立ofchannel通道，控制器通过ofchannel控制和管理交换机。
2. 当交换机收到一个数据包且流表中没有匹配条目，交换机会将数据包封装在packet-in消息发送给控制器，此时数据包会缓存在交换机中等待处理。
3. 控制器收到packet-in消息后，可以发送flow-mod消息向交换机写一个流表项，并且将flow-mod消息中buffer-id字段设置为packet-in消息中的buffer-id值。从而控制器向交换机写入了一条与数据包相关的流表项，并且指定该数据包按照该流表项的action列表处理。但是并不是所有的数据包都需要向交换机中添加一条流表项来匹配处理，网络中还存在多种数据包，它出现的数量很少（如ARP,IGMP等），以至于没有必要通过流表项来指定这一类数据包的处理法，此时控制器可以使用packet-out消息，告诉交换机某一个数据包如何处理。

2.2.2 OVS

Open vSwitch是一个由Nicira Networks主导的开源项目，通过运行在虚拟化平台上的虚拟交换机，为本台物理机上的VM提供二层网络接入，跟云中的其它物理交换机一样工作在Layer 2层。Open vSwitch充分考虑了在不同虚拟化平台间的移植性，采用平台

无关的C语言开发。并且遵循Apache2.0许可；我们有传统的物理交换机，为什么还要开发Open vSwitch呢？

在传统数据中心中，网络管理员习惯了每台物理机的网络接入均可见并且可配置。通过在交换机某端口的策略配置，可以很好控制指定物理机的网络接入，访问策略，网络隔离，流量监控，数据包分析，QoS配置，流量优化等。但是在云平台上，如果没有网络虚拟化技术的支持，管理员只能看到被桥接的物理网卡，其上川流不息地跑着n台VM的数据包。仅凭物理交换机支持，管理员无法区分这些包属于哪个OS哪个用户。而且难以满足以下需求：

网络隔离 物理网络管理员早已习惯了把不同的用户组放在不同的VLAN中，例如研发部门、销售部门、财务部门，做到二层网络隔离。Open vSwitch通过在host上虚拟出一个软件交换机，等于在物理交换机上级联了一台新的交换机，所有VM通过级联交换机接入，让管理员能够像配置物理交换机一样把同一台host上的众多VM分配到不同VLAN中去；

QoS配置 在共享同一个物理网卡的众多VM中，我们期望给每台VM配置不同的速度和带宽，以保证核心业务VM的网络性能。通过在Open vSwitch端口上，给各个VM配置QoS，可以实现物理交换机的traffic queuing和traffic shaping功能。

流量监控 物理交换机通过xxFlow技术对数据包采样，记录关键域，发往Analyzer处理。进而实现包括网络监控、应用软件监控、用户监控、网络规划、安全分析、会计和结算、以及网络流量数据库分析和挖掘在内的各项操作。例如，NetFlow流量统计可以采集的数据非常丰富，包括：数据流时戳、源IP地址和目的IP地址、源端口号和目的端口号、输入接口号和输出接口号、下一跳IP地址、信息流中的总字节数、信息流中的数据包数量、信息流中的第一个和最后一个数据包时戳、源AS和目的AS，及前置掩码序号等。

xxFlow因其方便、快捷、动态、高效的特点，为越来越多的网管人员所接受，成为互联网安全管理的重要手段，特别是在较大网络的管理中，更能体现出其独特优势。有了Open vSwitch，作为网管的你，可以把xxFlow的强大淋漓尽致地应用在VM上！

数据包分析 物理交换机的一大卖点，当对某一端口的数据包感兴趣时（for trouble shooting, etc），可以配置各种span（SPAN, RSPAN, ERSPAN），把该端口的数据包复制转发到指定端口，通过抓包工具进行分析。Open vSwitch官网列出了对SPAN, RSPAN, and GRE-tunneled mirrors 的支持。

Open vSwitch 引入了以下模块，很好地满足以上需求：

1. ovs-openflowd — OpenFlow交换机；
2. ovs-controller — OpenFlow控制器；
3. ovs-ofctl — Open Flow 的命令行配置接口；

4. ovs-pki — 创建和管理公钥框架;
5. tcpdump的补丁— 解析OpenFlow的消息;

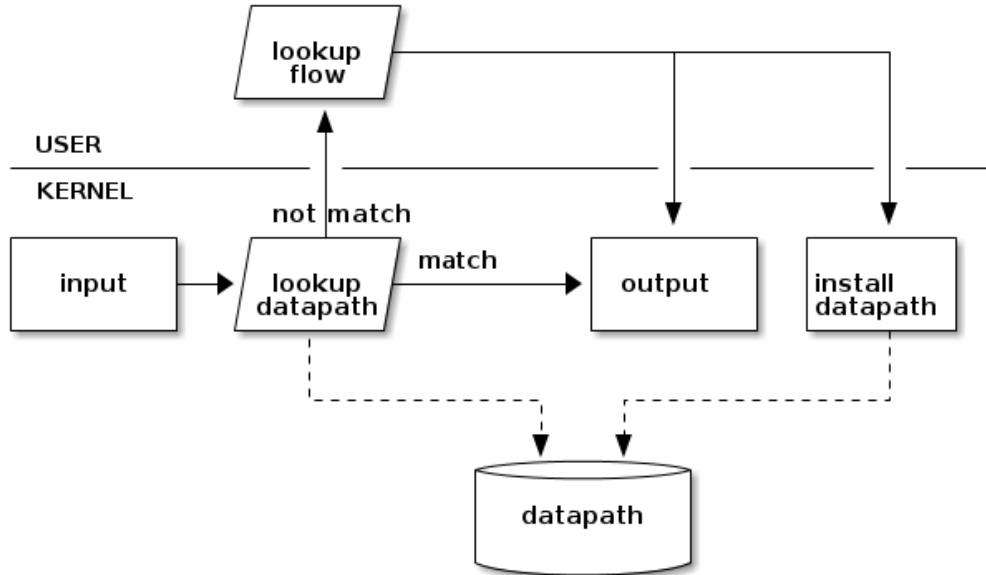


Figure 4: OVS工作流图

当Open vSwitch的一个接口收到数据包后，会按照上述流程图处理：收到数据包后，会交给datapath内核模块处理，当匹配到对应的datapath 会直接输出，如果没有匹配到，会交给用户态的ovs-vswitchd查询flow，用户态处理后，会把处理完的数据包输出到正确的端口，并且设置新的datapath规则，后续数据包可以通过新的datapath规则实现快速转发。

3 基于Coflow的大数据网络优化

3.1 coflow相关知识介绍

考虑到当前数据中心用并行计算的框架处理大规模的数据。许多数据密集型的应用都是和网络绑定的，但是对一些特别的通信请求，网络层的优化仍然是不可知道的，这通常会影响到应用层的表现。尽管数据密集型应用的架构有所不同，但是他们的通信相似甚至相同的，都要经过集群之间一系列的连续的计算过程。通常一个通信阶段的所有流都完成这个通信阶段才算完成。最近提出的coflow的概念就代表一组满足相同的通信请求（最小化完成时间、所有的流满足deadline）并行流的集合，这个可以通过应用感知的网络调度算法实现。更进一步，我们可以通过优化coflow的完成时间整个任务的响应时间。

为什么说考虑coflow级别的调度算法有利于优化任务的完成时间呢？我们看下面这个例子：

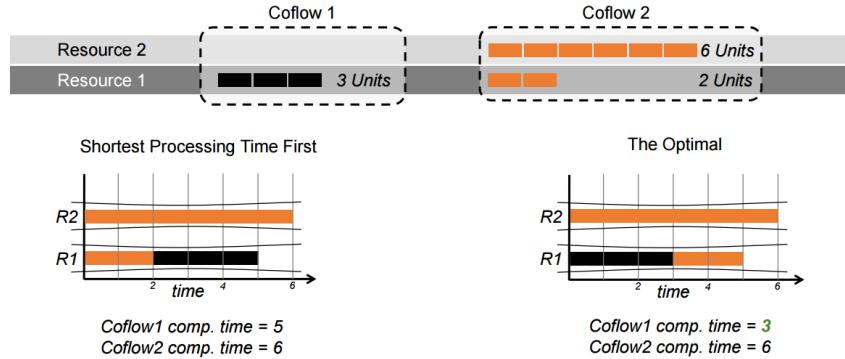


Figure 5: coflow完成时间的比较

在本例中，Coflow1用黑色表示，含有一条流，需要1个单位时间传输完成；Coflow2用黄色表示，含有两条流，一条流flow1需要6个单位时间传输完成，另一条流flow2需要2个单位时间传输完成。Coflow1的一条流和Coflow2的flow2公用资源。如果在流级别按照最短处理时间优先的调度算法（Figure 5的左侧图），Coflow1的完成时间为5，Coflow2的完成时间为6；若我们采用Coflow级别的调度算法，按照Coflow的最短完成时间调度，我们会发现虽然Coflow2的完成时间为6，但是Coflow1的完成时间变为了3，从而缩短了Coflow的平均完成时间。

有人设计了varys来提高大数据处理过程中网络通信的性能，他们以最小化Coflow的完成时间和满足deadline为目标实现了FIFO、SCF和SEBF等启发式算法。采用varys 仿真器coflowsim完成Facebook真实数据的调度测试。他们采用的是交换结构，如下图：

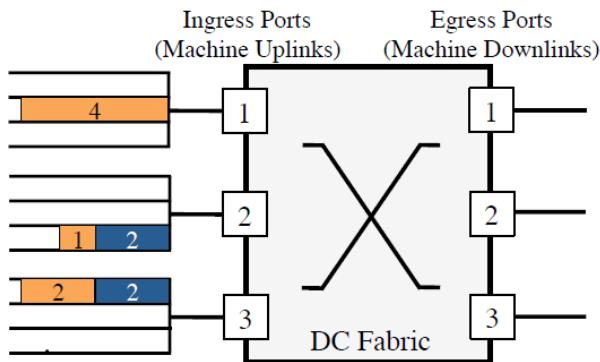


Figure 6: varys中的交换结构

上面是一个交换结构，考虑的是无阻塞的网络，也就是说：不管网络处于何种状态，任何时刻都可以在交换网络中建立一个连接，只要这个连接的起点、终点是空闲的，而不会影响网络中已建立起来的连接。对于数据中心网络来说，无阻塞就是实现任意服务器之间可以线速交互流量的方式，作为数据中心交换机，无阻塞可以支撑数据中

心网络内部大量的内部流量，满足这些流量的高效传输。

3.2 coflow调度算法设计

varys是交换结构，考虑无阻塞的情况。我们在设计coflow调度算法考虑路由，在COflow级别我们考虑最小化Coflow的完成时间，在Coflow内部的flow级别，我们考虑最大流优先，最短路径路由。具体算法如下：

Algorithm 1: 最小化coflow的完成时间的调度算法

```
1: 生成Coflow任务  
2: for  $t = 0; t \leq M; t \rightarrow t + 1$  do  
3:   if 当前部署的任务流已经处理完成 then  
4:     释放网络资源;  
5:   end if  
6:   统计当前时刻到达的Coflow;  
7:   if 有Coflow到达 then  
8:     根据当前资源，调用流级别的调度算法MFF-SP2，计算Coflow的完成时间;  
9:     根据Coflow的完成时间，进行升序排序;  
10:    Coflow完成时间最短的先部署  
11:    更新网络资源  
12:   else  
13:     进入下一个时刻  
14:   end if  
15: end for
```

Algorithm 2: 流级别的调度算法——最大流优先最短路径路由（MFF-SP）

```
1: 调用当前网络拓扑  
2: 对Coflow中的所有flow按照flow数据量大小，降序排列;  
3: for  $flow = 0; flow \leq flow\_number; flow \rightarrow flow + 1$  do  
4:   if flow正在等待调度 then  
5:     根据flow的源和目的节点利用最短路径计算其路由路径;  
6:     找到路由路径中剩余带宽最少的link;  
7:     根据剩余带宽最少的link的带宽计算Coflow的最小完成时间;  
8:   else  
9:     进行下一条流的计算  
10:  end if  
11: end for
```

3.3 coflow调度算法的性能表现

我们完成了两种Coflow调度算法的设计：

1. 先来的先调度 (FCFS: first come first scheduling)

2. 最小化Coflow的完成时间 (MCCTF: minimum cct (coflow completion time) first)

针对两种算法我们跑了一些仿真结果:

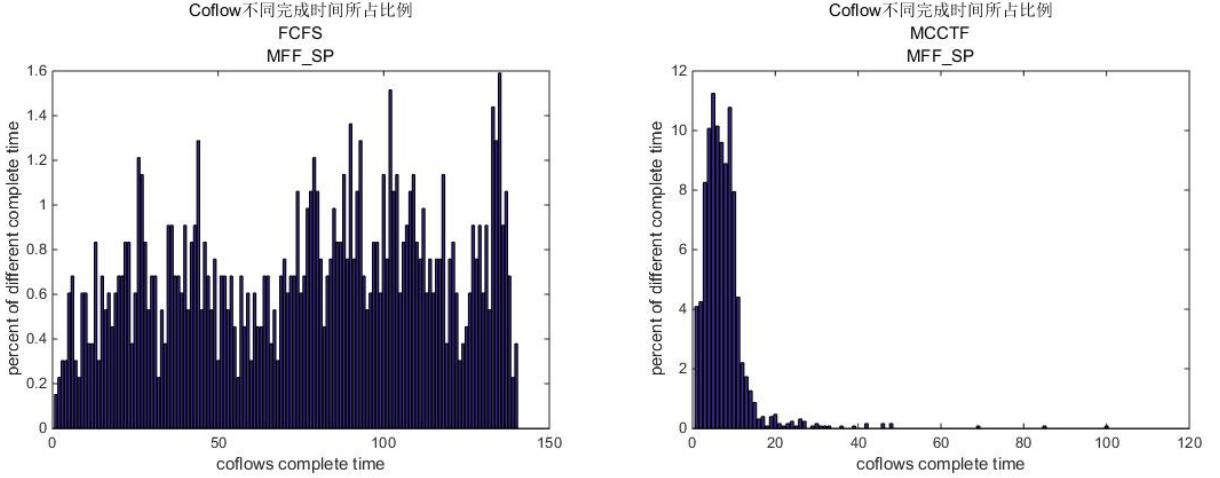


Figure 7: FCFS和MCCFT两种算法的Coflow完成时间对比

根据图 (Figure 7) 显示的结果，我们发现在我们调用Coflow调度算法FCFS时，Coflow的完成时间分布在0~140个单位时间之间，而在调用我们设计的Coflow调度算法MCCFT时，在15个单位时间以内的Coflow 占据全部Coflow 的95% 以上。

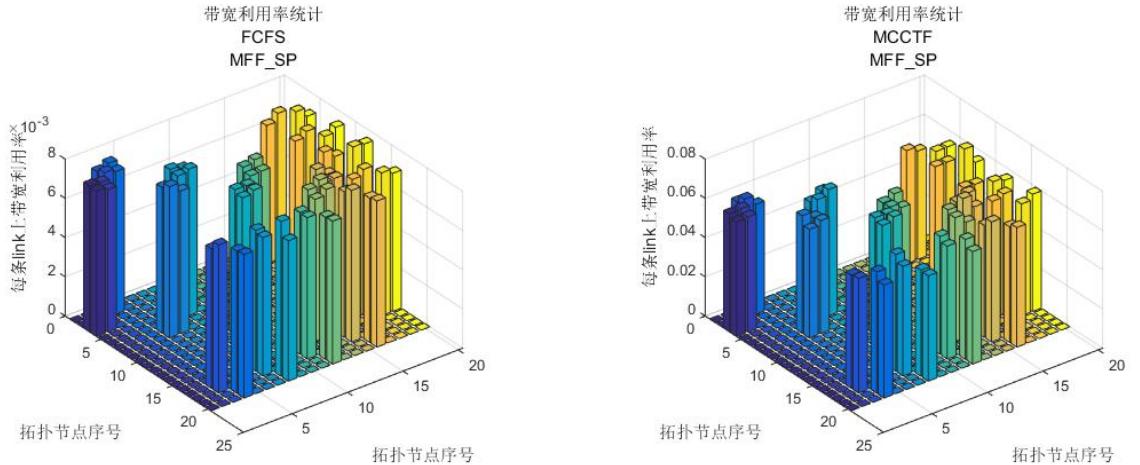


Figure 8: FCFS和MCCFT两种算法的每条链路上的带宽利用率对比

根据图 (Figure 8) 显示的结果，我们发现Coflow调度算法MCCFT的链路带宽利用率要比FCFS高出10倍左右

根据图 (Figure 9) 中的显示结果，我们可以看到算法MCCTF的Coflow的完成时间明显比算法FCFS的完成时间短，说明我们设计的算法是有效果的，最小化Coflow 的完成时间的Coflow调度算法确实可以缩短任务的平均完成时间。同时也验证了我们之前对Coflow完成时间优化的分析，如图3.1。

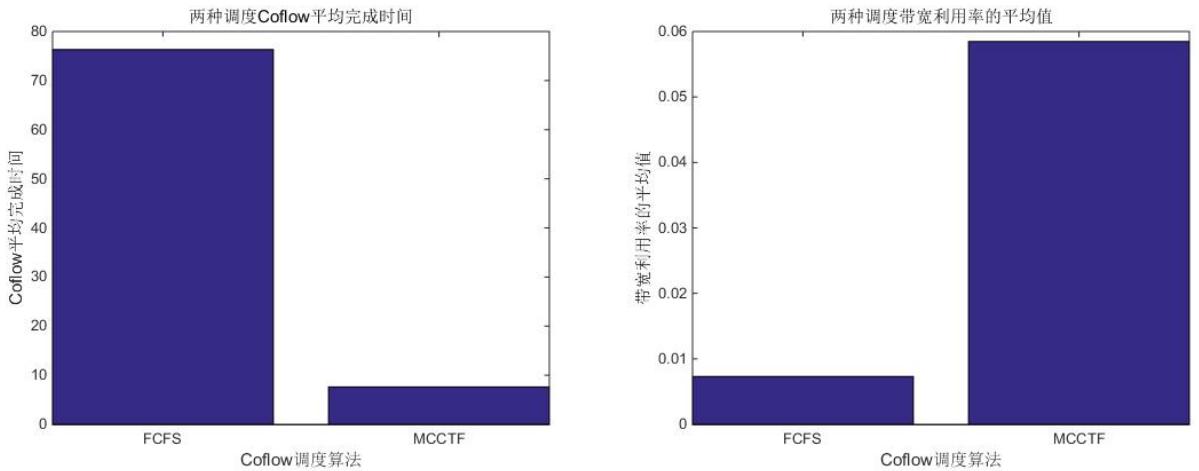


Figure 9: FCFS和MCCFT两种算法的Coflow平均带宽利用率以及平均完成时间对比

综上三组对比图我们发现，我们设计的Coflow调度算法MCCFT相比传统的Flow级别的调度算法更加优越，因为MCCFT可以实现更小的Coflow完成时间，更高的带宽利用率。而且我们发现越是数据密集型的数据，我们的算法优势越是明显

4 实验平台搭建

4.1 实验环境



Figure 10: 真实的实验环境

服务器的硬件以及软件的配置：机器的型号，内存大小，网卡，协议版本等等

Table 1: 系统软硬件及协议配置

名称	配置
服务器配置	CPU: 16 Intel(R) Xeon(R) CPU E5620 @ 2.40GHz MEM: 64G DISK: 1000G 网卡: 1000 Mbps
软件及协议	Ubuntu 14.04 LTS Floodlight 1.2 Modified Open vSwitch 2.02 OpenFlow 1.3 Hadoop 2.7.1 Python 2.7.11

4.2 Hadoop平台搭建

我们使用了4台Xeon E5620 64GB RAM 服务器作为我们的YARN MapReduce 任务执行节点，并对YARN、Mapreduce 以及HDFS 进行了配置。

为了使得整个平台的数据流量只走我们的SDN 网络，我们分别对如下文件进行了如下配置：

```
hadoop@IPL212:~/hadoop/hadoop-2.7.1/etc/hadoop$ ls -lah *-site.xml
-rw-r--r-- 1 hadoop hadoop 1006 Feb 24 22:31 core-site.xml
-rw-rw-r-- 1 hadoop hadoop 1.7K May  7 07:52 hdfs-site.xml
-rw-r--r-- 1 hadoop hadoop 620 Feb 24 22:31 httpfs-site.xml
-rw-r--r-- 1 hadoop hadoop 5.4K Feb 24 22:31 kms-site.xml
-rw-r--r-- 1 hadoop hadoop 2.4K Apr 19 16:04 mapred-site.xml
-rw-r--r-- 1 hadoop hadoop 2.8K Apr 19 20:53 yarn-site.xml
```

Figure 11: 以服务器212为例，配置文件清单

YARN 平台配置文件yarn-site.xml

- 1.设置YARN 默认允许使用的系统资源量，若不进行设置，则只能使用16GB 内存；
- 2.固定Node Manager 的地址为本机OpenFlow 网段IP (of-IPL212)；
- 3.固定Resource Manager 的地址为Master 的OpenFlow 网段IP (of-master)；

MapReduce 配置mapred-site.xml

设置单个Map 和Reduce 任务可以占用的内存大小，防止单个任务的所需内存不足或分块任务太多而导致整体速率下降；

HDFS 配置hdfs-site.xml

设置强制所有HDFS 客户端只监听OpenFlow 端的端口（默认是0.0.0.0全部监听），避免

```

23 <property>
24   <name>yarn.scheduler.minimum-allocation-mb</name>
25   <value>10</value>
26 </property>
27
28 <property>
29   <name>yarn.nodemanager.resource.memory-mb</name>
30   <value>65536</value>
31 </property>

```

Figure 12: yarn配置，允许使用的资源

```

48 <property>
49   <description>The hostname of the NM.</description>
50   <name>yarn.nodemanager.hostname</name>
51   <value>of-IPL212</value>
52 </property>
53 <property>
54   <name>yarn.nodemanager.bind-host</name>
55   <value>of-IPL212</value>
56 </property>
57 <property>
58   <description>The hostname of the timeline service web application.</description>
59   <name>yarn.timeline-service.hostname</name>
60   <value>of-IPL212</value>
61 </property>
62 <property>
63   <name>yarn.timeline-service.bind-host</name>
64   <value>of-IPL212</value>
65 </property>
-->

```

Figure 13: yarn 配置，指定nodemanager地址

```

72 <property>
73   <name>yarn.resourcemanager.address</name>
74   <value>of-master:8032</value>
75 </property>
76
77 <property>
78   <name>yarn.resourcemanager.scheduler.address</name>
79   <value>of-master:8030</value>
80 </property>
81
82 <property>
83   <name>yarn.resourcemanager.resource-tracker.address</name>
84   <value>of-master:8035</value>
85 </property>
86
87 <property>
88   <name>yarn.resourcemanager.admin.address</name>
89   <value>of-master:8033</value>
90 </property>
91
92 <property>
93   <name>yarn.resourcemanager.webapp.address</name>
94   <value>of-master:8088</value>
95 </property>

```

Figure 14: yarn配置指定resourcemanager地址

数据流量进入控制网络。

Master-Slave 关系配置

```

20 <property>
21   <name>mapreduce.framework.name</name>
22   <value>yarn</value>
23 </property>
24
25 <property>
26 <name>mapreduce.map.memory.mb</name>
27 <value>500</value>
28 </property>
29 <property>
30 <name>mapreduce.reduce.memory.mb</name>
31 <value>500</value>
32 </property>

```

Figure 15: MR设置使用yarn管理tasks，单个task占用内存大小

```

19 <configuration>
20 <property>
21   <name>dfs.namenode.rpc-bind-host</name>
22   <value>of-master</value>
23 </property>
24 <property>
25   <name>dfs.namenode.servicerpc-bind-host</name>
26   <value>of-master</value>
27 </property>
28 <property>
29   <name>dfs.namenode.http-bind-host</name>
30   <value>of-master</value>
31 </property>
32 <property>
33   <name>dfs.namenode.https-bind-host</name>
34   <value>of-master</value>
35 </property>

```

Figure 16: HDFS强制流量走SDN网络

在主节点Master 上的slaves 文件中加入所有服务器的列表。图7

```

1 of-IPL212
2 of-IPL213
3 of-IPL211
4 of-IPL201
-

```

Figure 17: slaves

4.3 SDN开发环境搭建

交换机采用Open vSwitch，运行在Ubuntu Linux 下。控制器采用Floodlight，获取和控制整个SDN 网络，并且与我们的PyCon 进行通信。

将多个虚拟交换机ovs挂载到了floodlight控制器上，同时保留一个被动监听端口以供PyCon 进行额外的操作：

```
ovs-vsctl set-controller ovsbridgename tcp:floodlight-ip:6633 ptcp:6666
```

允许远程控制虚拟交换机控制器，以便PyCon 控制：

```
ovs-vsctl set-manager ptcp:6640
```

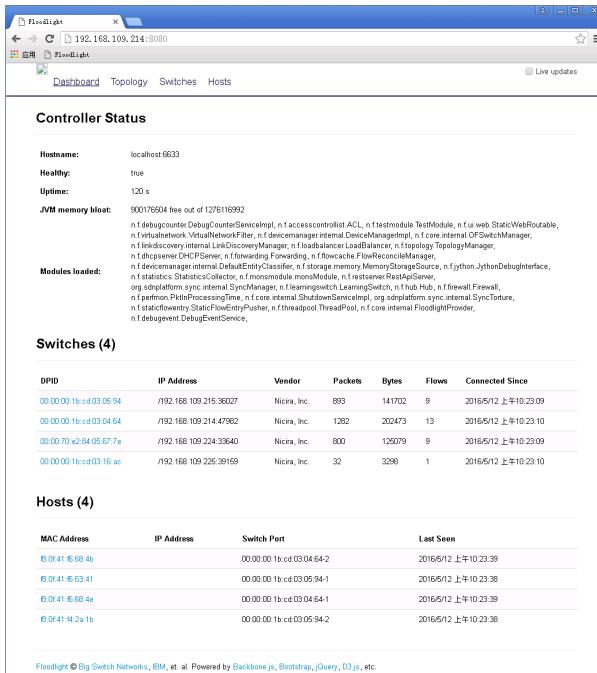


Figure 18: Floodlight Web 监视界面

利用ovs完成hadoop平台上的数据导流工作

4.4 网络拓扑

5 系统设计与开发

5.1 系统整体架构设计

5.2 MapReduce Fetcher

Map任务结束之后，主要进行Copy、Sort、Reduce三个步骤；其中Copy阶段，就是从执行各个Map任务的节点获取map的输出文件。这是由ReduceTask.ReduceCopier类来负责。ReduceCopier对象负责将Map函数的输出拷贝至Reduce所在机器。在这个过程中，Reduce 进程会启动一些数据复制线程(Fetcher)，通过HTTP GET 方法请求由HTTP Server 管理的Shuffle 数据块，这些数据块是先前成功完成的Map Task 的输出文件。这些等待Shuffle 的数据块可能在内存中也有可能在HDFS 中，取决于服务器当前的内存是否

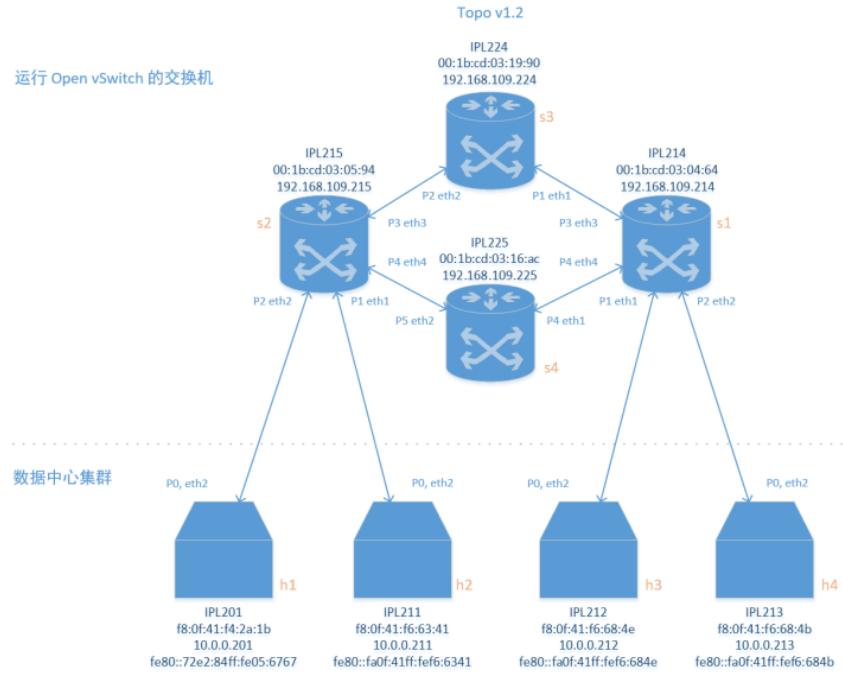


Figure 19: 完整详细的网络拓扑结构

数据处理工作流图

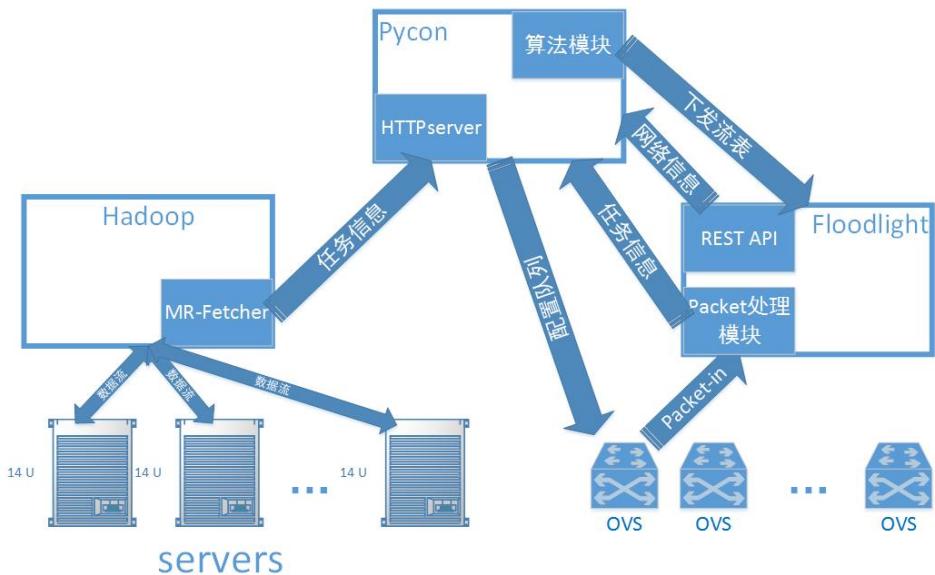


Figure 20: 基于SDN+Hadoop的系统架构设计

充足以及对Hadoop 的配置。如果大小超过一定阈值就写到磁盘，否则放入内存。为了利用SDN controller建立全局的调度策略，优化代码插入位置，在NM获得全局调度信息而不用这样一个一个发送，以提高效率。我们做出了一个初步可用的HTTP Client POST，向PyCon控制器（SDN controller）POST必要的数据，如：源主机、目的主机、数据长度以及MapID等信息。

5.3 Python Controller (PyCon)

5.3.1 HTTPServer 类

在Hadoop 中很多地方都用到了Servlet， 并且使用Jetty 作为Servlet 的容器来提供HTTP的服务， 其主要是通过org.apache.hadoop.HTTP.HTTPServer 类实现的， HTTPServer 类是对Jetty 的简单封装， 通过调用HTTPServer 类的addServlet 方法增加可以实现增加Servlet 到Jetty 的功能。

Hadoop 各个组件都会用到HTTPServer, datanode/namenode, resourcemanager等。

Hadoop 内嵌了HTTP 服务器Jetty， 主要有以下两方面的作用

1. 基于REST API 的Web 访问接口， 用于展示Hadoop 的内部状态
2. 参与Hadoop 集群的运行和管理

5.3.2 配置队列，进行限速

网络研究主要通过两个途径提高QoS：一个是节点控制；另一个是网络控制。节点控制的策略包括流量整形、节点缓冲区管理、队列调度等。网络控制通过对路由的控制来提高网络的性能，其目标是为进入网络的数据流（业务）选择满足其服务质量要求的路径，同时保证网络资源的有效利用。

为了提供更好的QoS，我们编写了一个OVS 队列配置模块来实现最小瓶颈算法，可以在PyCon 启动时自动完成队列的配置操作，以供算法使用。

5.3.3 获取网络信息

采用Floodlight REST API 获取当前受控制的SDN 网络的带宽使用信息等。不过，直接调用REST API 获取到的信息不是直接可用的，而是一个JSON 格式的文本串。我们编写了一段程序将其自动转化为便于处理的矩阵形式。

5.3.4 流表预配置

SDN 网络配置流表相比较常规网络较为缓慢，因为当一个数据包没有匹配的流表，交换机会将数据包发送到控制器来请求得到一个回应，得到回应之后才能继续发送数据包，这个过程的延迟较大，大约是ms 级别的延迟，这对于常规延迟要求较低的TCP 网络来说是无法容忍的。在我们实际测试的过程中也发现，如果不进行一个流表的预配置，由于延迟较高，TCP 发送端将认为数据包丢失，会进行数据包重传，且较为严重（大约占到Shuffle 流量的六分之一）。为了缓解这个问题，我们灵活采用FL API 获得的拓扑信息对网络进行了分析，在交换机上预配置了一系列的流表。大大缓解了实际SDN 网络中运行传统应用时的数据包重传问题。

5.3.5 下发流表

使用Floodlight 的StaticFlowPusher REST API 进行流表的下发。默认超时时间是5秒，以避免流表过多匹配效率下降的问题。

5.3.6 路由算法模块

我们从修改过的Hadoop Fetcher 模块和Floodlight 控制器货物获取即将开始传输的Flow 的信息，然后调用路径计算模块进行路由规划。目前，PyCon 路径计算模块实现了以下两种路由算法：

5.3.7 Coflow 调度

我们从Hadoop Fetcher 获得了任务的Job 编号，用于识别Coflow 信息。然后在采用最小瓶颈算法时将不同的Coflow 装入不同的队列实现流速控制。但是在实际实现中遇到了一些瓶颈，因此我们先以MATLAB 仿真结果作为展示。

1.加权最短路径算法

思路：先使用Floodlight 的API 获得当前整体网络带宽使用信息，然后将返回的信息处理成矩阵，以带宽作为权重，调用Dijkstra 进行计算，得到一条路径。

2.最小瓶颈算法

思路：要使Flow 的传输时间最短。计算一条流在网络中所有可行的路径，选择一个带宽最大化，传输时间最短的路径，然后进行流表的下发。

细节：为了实现带宽最大化，我们使用了OpenFlow 的队列特性。在Linux 上通过OVS 调用内核的QoS Queue 机制实现。队列的配置由PyCon 完成，不经过Floodlight。

5.4 对Floodlight 的改进

5.4.1 Packet-in 数据包处理模块

Packet-in 数据包是OpenFlow 协议中当交换机没有对应的匹配流表时，自动向控制器发送的一个数据包，目的是为了获取有效的路由信息，得到控制器的回复之后就可以按照指令传送数据，使数据包可以被顺利发送。

思路：配置流表，让特定数据包以Packet-in 包的形式传到Floodlight 控制器。我们编写了一个新模块，使得我们可以顺利提取到Hadoop Fetcher 无法拿到的端口信息。

具体做法：添加了一个packet-in 数据包处理模块。可以提取TCP Source Port 和Payload 内的Request URI 并且发送给PyCon，之所以这么做是因为在Hadoop MR 模块

源码内无法通过`HTTPURLConnection`直接拿到Source Port

5.4.2 REST API

我们使用了Floodlight 提供的众多开放的API，比如拓扑信息，Host 信息，网络速率，流表下发与删除等。

但是，由于Floodlight 目前仍在进一步开发中，功能并不完善。我们额外采用了OVS 的一些特性进行一些特殊操作，比如直接对OVS 交换机进行流表操作而不经过Floodlight，这样我们得以实现一些特殊的功能比如下发`actions=normal`（Floodlight 无法直接下发之类特殊操作）。

6 项目成果

6.1 实验仿真测试

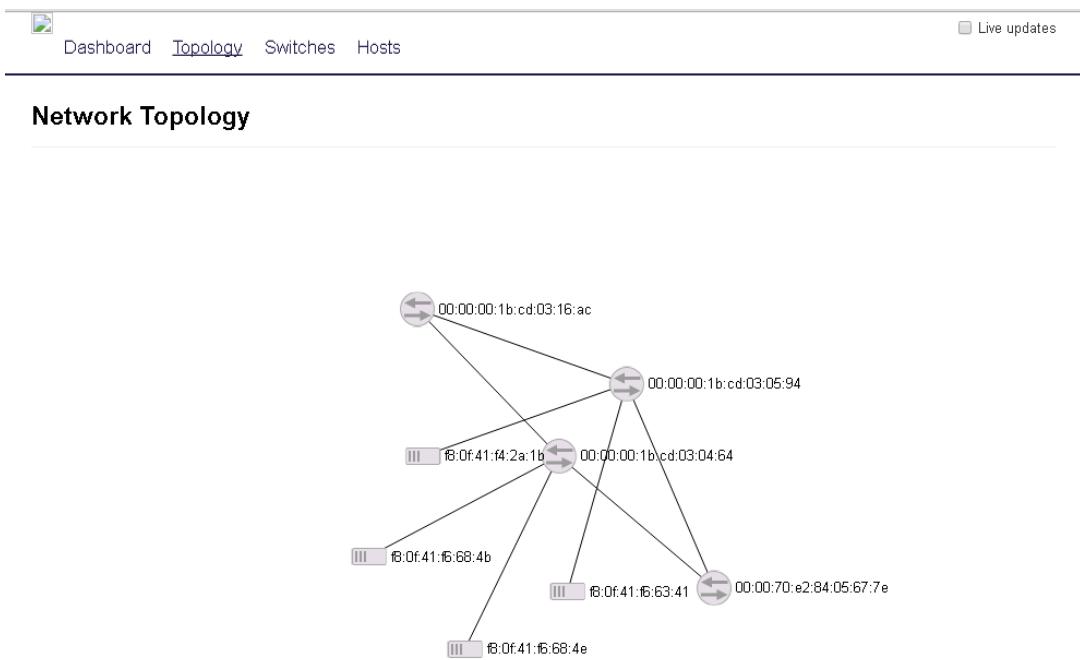


Figure 21: 实验测试网络拓扑结构。此图由Floodlight 使用生成树算法自动产生，说明拓扑配置无误

6.2 实验结果

由于我们主要进行的是Hadoop 的Shuffle 网络优化，我们选择了实际应用和测试中网络传输量最大的TeraSort 进行测试。

TeraSort 是Hadoop MapReduce 上的一个典型的采用Map-Reduce 架构的范例程序。通常

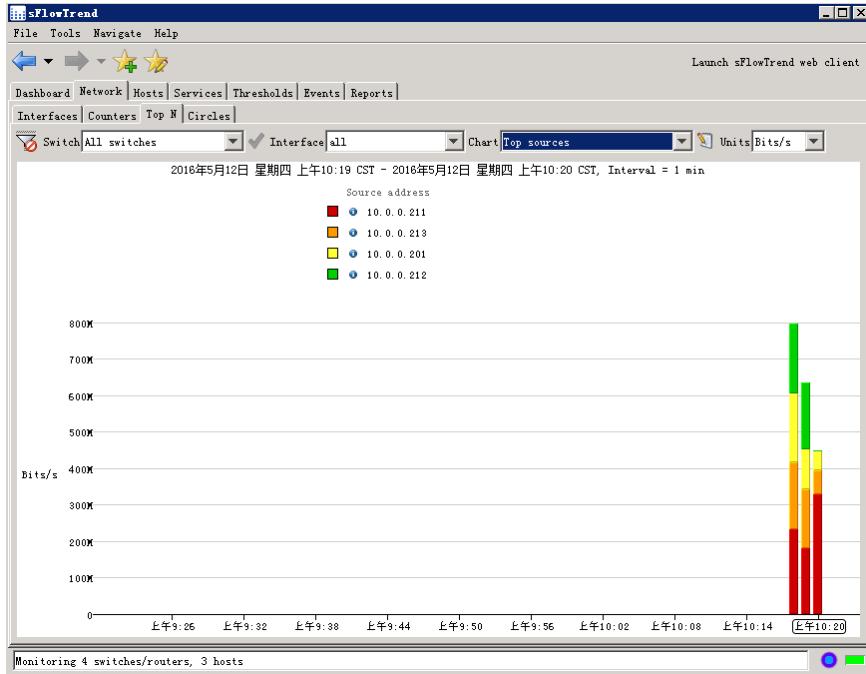


Figure 22: 服务器上的实时流量状况。图为刚执行一个TeraSort 时的网络流量，可以看到四端口总计高达800Mbps的吞吐量

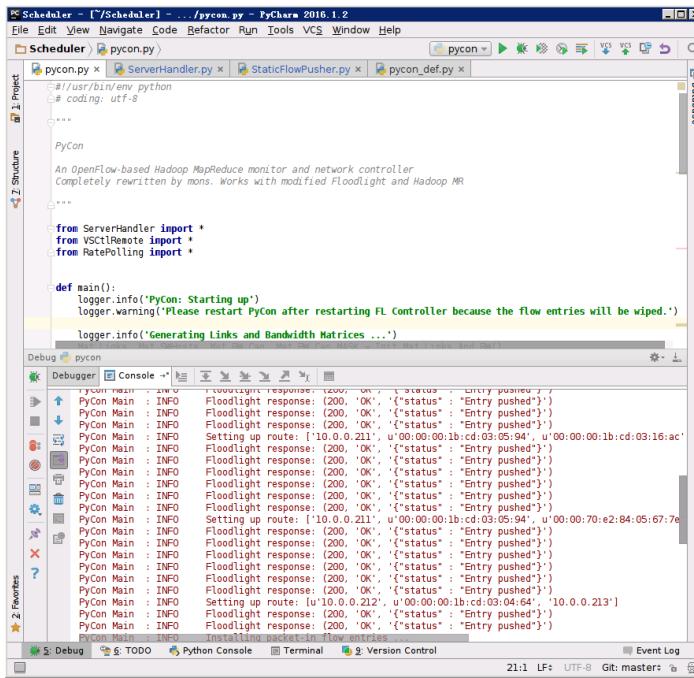


Figure 23: PyCon的运行情况。调试窗口显示的是Floodlight 流表下发成功信息

的大规模数据集群中也会采用1TB（1024GB）的数据集大小对数据中心性能进行测试。但由于我们的系统规模不大，仅4台服务器与4台交换机，并且是在新颖的SDN架构上对Hadoop 大数据应用的研究，我们采用的数据集大小仅为4GB。

实验参数：数据中心内部所有连接速率均为1 Gbps，关闭了MapReduce 的Map 数据压缩选项，生成排序内容时的数据分块大小为4（对应4 台Host）。实际排序时，所使用的Map Container 的数量为32 个（视为32 个分块）。

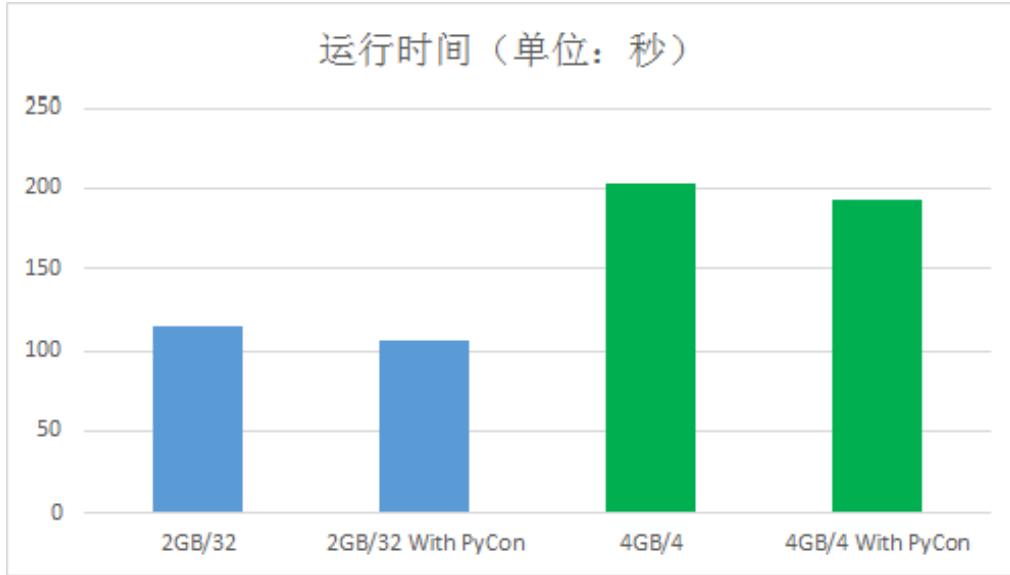


Figure 24: 实验测试结果

由于我们是在测试SDN 对整体应用性能的优化效果，我们对TeraSort 排序的总运行时间进行了测定，从提交任务、分配任务到Reduce 结果返回结果。实验结果表明，未启用PyCon 时，完成数据排序的时间平均为203 秒；启用PyCon 后，数据排序操作时间平均为193 秒，约有5% 的性能提升。

6.3 结果分析

提升性能的因素有以下几点：缓解了直接套用SDN 网络而不进行有效配置时TCP 重传导致的网络拥塞；Floodlight 默认并不会进行负载均衡操作，而PyCon 采用的算法集成了负载均衡的功能，使得传输效率大大提升。

性能提升并不显著的原因：由于测试规模受限，数据集较小，无法显著体现优化的性能；

实际数据中心内部的带宽普遍较大（1 Gbps），普遍在100MB 以内数据块的传输只需要1 秒以内就可以完成，即使只走1 条路，也需要多几秒便可以传输完成；

在Reduce 阶段，有且仅有一个容器可以进行工作，进行各个排序子块数据的合并，导致Reduce 步骤本身在整个任务运行时间中所占的比重较大，而传输时间是一个小量。

SDN性能瓶颈分析

通过分析sdn网络的工作流程，可知控制器通过响应packet-in消息发送packet-out/flow-mod消息的速度是非常重要的，它的快慢直接影响了控制器拓扑发现，流表下发，mac地址学习能力，甚至整个网络的性能。而且SDN网络中通常采用反应式流安装，控制器的响应时间直接影响着流安装的处理速度，本文将重点测试在负载不

同的情况下控制器处理packet-in消息的吞吐量和响应时间。同时也关注控制器支持创建openflow连接的能力与拓扑更新的速度。(4).虽然我们引用了Coflow 概念并在系统中进行了实现，但是由于在测试环境中同一时间只运行了一个任务。

7 项目总结

在本项目中我们以优化大数据处理性能为背景，将注意力集中到网络性能的优化。最初我们希望基于openstack平台，利用Ryu SDN控制器集中管理数据中心中的虚拟机，来优化大数据处理的性能，基于这个想法，我们提出以下几方面的研究方案：

1. 虚拟机的位置优化，动态迁移虚拟机实现负载均衡；
2. 设计集中控制调控算法，来实现代码传输、数据传输的协同工作；
3. 设计实时算法来调控网络传输路径，实现网络带宽优化；
4. 设计实时算法实现数据从硬盘到内存的预加载算法；
5. 通过用户态协议栈（DPDK）来加速网络处理；

在开发过程中，我们发现这其中的工作量实在太大，正如老师所说，上述几个方案的研究够三个博士毕业的工作了。所以我们将我们的想法收敛，经过组内的讨论和调研，在中期检查后，我们将注意力转到应用层，利用SDN 和云管理工具（Hypervisor）的全局控制能力，设计流级别的调度算法来改善网络性能，于是我们引入了Coflow的概念。另外，由于在openstack平台下，网络配置困难，所以我们考虑先不用虚拟机，而是在直接在服务器上搭建hadoop平台。基于这个平台，我们完成了Coflow调度算法的设计，并在matlab内进行了完整地仿真实验，以及性能的分析。另外在真实系统中，我们实现了利用SDN集中调控hadoop中的任务流，同时进行了实验测试，在任务的完成时间上我们得到5%的性能提升。

回顾一年来的工作，从最初的认识大数据处理，到平台的选择，算法的设计，仿真、实现、测试以及最后的结果分析总结，我们经历了很多的周折，不断的重新选择，不断的调整方向，最终收敛到目前的工作，感谢这各项目的给予我们的锻炼，收获很多，感谢耐心帮助过我们的师兄师姐，感谢我们的指导老师对我们的帮助和支持。

8 参考文献

References