软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(10):2220−2234 [doi: 10.13328/j.cnki.jos.004679]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种支持细粒度并行的 SDN 虚拟化编程框架*

宋 平[1], 刘 轶[1], 刘 驰[1], 张晶晶[1], 钱德沛[1], 郝沁汾[2]

[1](北京航空航天大学 计算机学院,北京 100191)

[2](华为技术有限公司,广东 深圳 518129)

通讯作者: 宋平, E-mail: ping.song@jsi.buaa.edu.cn

摘 要: 软件定义网络(software defined network,简称 SDN)通过集中式的控制器提高了网络的可编程性,成为近年来网络领域非常热门的话题.以 Openflow 网络为代表的软件定义网络将逻辑控制与数据转发相隔离,为网络虚拟化技术提供了良好的平台.集中式的抽象与控制使得 SDN 虚拟化框架的处理效率成为主要瓶颈.现有的 SDN 虚拟化框架由于缺乏对细粒度并行的支持,为编程人员充分利用多核/众核资源、控制更大规模的网络带来了极大的挑战.为了提高 SDN 虚拟化框架的处理效率,提出一种新的 SDN 虚拟化编程框架,通过新颖的 API 和运行时,在框架内部支持细粒度的并行处理.该框架通过对网络中流和网络资源进行抽象,使开发人员可以直接通过划分流空间来定义不同的虚拟网络,利用无锁的编程方式对共享的网络资源和流进行操作.实验结果表明,该框架在逻辑控制的执行效率方面具有良好的可扩展性,可以创建出更大规模的虚拟网络,并对其进行更为复杂的控制.

关键词: SDN 虚拟化;事件编程;细粒度并行;众核处理器

中图法分类号: TP311

中文引用格式: 宋平,刘轶,刘驰,张晶晶,钱德沛,郝沁汾.一种支持细粒度并行的 SDN 虚拟化编程框架.软件学报,2014,25(10):2220−2234. http://www.jos.org.cn/1000-9825/4679.htm

英文引用格式: Song P, Liu Y, Liu C, Zhang JJ, Qian DP, Hao QF. Fine-Grained parallel SDN virtualization programming framework. Ruan Jian Xue Bao/Journal of Software, 2014,25(10):2220−2234 (in Chinese). http://www.jos.org.cn/1000-9825/4679.htm

Fine-Grained Parallel SDN Virtualization Programming Framework

SONG Ping[1], LIU Yi[1], LIU Chi[1], ZHANG Jing-Jing[1], QIAN De-Pei[1], HAO Qin-Fen[2]

[1](Department of Computer Science and Technology, BeiHang University, Beijing 100191, China)

[2](Huawei Technologies Co., Ltd., Shenzhen 518129, China)

Corresponding author: SONG Ping, E-mail: ping.song@jsi.buaa.edu.cn

Abstract: Software defined network (SDN), which introduces centralized controllers to drastically increase network programmability, has been a hot topic in the network domain. Software defined network separates control plane from data plane of network equipment, establishing a good platform for network virtualization. As the network scales up, the performance of SDN virtualization framework becomes a key bottleneck. Still, current SDN virtualization frameworks lack support for fine-grained parallelism, making them challenging for developers to fully exploit many cores to virtualize large networks. This paper presents a novel API and runtime for fine-grained parallel programming in SDN virtualization framework. By abstracting flows and network resources, the framework programming model enables developers to easily write programs to directly define various virtual networks and parallelly operate the network resource or flow objects by a lock-free manner. Experimental results show that the presented framework has a better logical control performance, allowing one to implement rich functional virtual networks.

Key words: SDN virtualization; event-based programming; fine-grained parallelism; many-core processor

* 基金项目: 国家自然科学基金(61133004); 国家高技术研究发展计划(863)(2012AA01A302); 华为合作项目(YB2012120105)
收稿时间: 2014-02-21; 修改时间: 2014-07-07; 定稿时间: 2014-07-31

软件定义网络(software defined network,简称 SDN)的出现,使得可编程网络的思想重新受到重视.通过集中式的逻辑控制,软件定义网络将更多的网络功能转变成用户可定义的软件,允许开发人员简单、灵活地编写程序,运行于控制器端,实现对整个网络的控制.软件定义网络广泛的应用性可以从最近大量实际用例中看出:可编程的网络实验床[1,2]、数据中心网络[3−5]、企业级网络[6,7]、无线网络[8]、网络测量系统[9]等.

与存储虚拟化、服务器虚拟化类似,网络虚拟化技术对底层的物理网络资源进行抽象,使得多个虚拟网络之间透明地使用底层的物理网络资源.在软件定义网络中,集中式的控制器可以直接地获取整个网络的拓扑信息、交换机状态以及链路信息等,为网络虚拟化技术中网络资源的抽象提供了良好的支持.作为两种技术的天然结合,SDN 网络虚拟化技术不仅可以有效地增强网络资源的共享,而且可以使网络控制变得更加灵活、智能.

SDN 虚拟化技术主要是通过软件定义网络中的控制器获取网络信息;然后,根据这些信息对网络资源进行抽象,并进行划分与隔离;利用编程人员定义的控制逻辑使用虚拟网络资源.随着云计算技术的兴起,大型数据中心需要进行网络资源的虚拟化管理,如按需分配带宽、最优路由等 QoS 策略,且随着网络规模的扩大,事件处理的复杂度也随之增大,使得 SDN 虚拟化框架的处理效率成为了主要瓶颈[10].现有的 SDN 虚拟化框架,如 FlowVisor[1],ADVisor[11],OVN[12]等,从不同方面来解决 SDN 虚拟化中的问题,但是这些框架在不同程度上都存在处理效率的可扩展性问题,使框架难以应用于大规模的网络环境中,限制了构建虚拟网络的规模以及功能.

随着处理器核数的增加,在未来的众核环境中,编程人员可以利用几百甚至几千个处理器核以及充足的内存和 I/O 资源,提高 SDN 虚拟化框架的执行效率.在这样的背景下,现有的 SDN 虚拟化框架存在一个重要的问题亟待解决:即在框架内部对 SDN 事件处理过程中,缺乏细粒度并行的支持.利用强大的众核资源,SDN 虚拟化框架可以对更大规模的物理网络进行抽象,生成规模更大的虚拟网络,并对这些虚拟网络进行更为复杂的逻辑控制.但是,由于现有的虚拟化框架缺乏良好的性能可扩展性,使得如何利用众核资源控制更大规模的网络成为了编程人员需要面对的最大挑战.

为此,我们提出了新的解决方案:在 SDN 虚拟化框架内部,利用事件处理过程中的细粒度并行,提高框架性能的可扩展性.与此同时,在事件驱动方法的基础上,提供更高层次的编程接口简化程序编写.在本文中,我们设计了一种支持细粒度并行的SDN虚拟化框架,它允许编程人员编写并行过程,访问共享网络资源以及对SDN事件进行处理.作为一种支持细粒度并行的 SDN 虚拟化框架,本框架做出的贡献主要有以下几点:

(1) 针对流和网络资源处理的编程接口.本框架提供了直接的编程接口,对网络中的流和网络资源进行抽象和处理.根据对不同类型的流的定义,直接构建不同的虚拟网络.利用提供的状态接口,编程人员可以直接对网络资源进行抽象.通过与不同类型的 SDN 事件进行绑定,将事件处理过程转化为对不同数据对象(流对象和网络资源对象)的处理过程.另外,在事件处理方法内部,支持对不同的对象进行并行处理与同步,实现事件处理过程的细粒度并行;

(2) 简单地共享数据访问方式.在细粒度并行执行过程中,需要保证对共享网络资源操作的正确性.利用本框架提供的 API,可以使编程人员无锁地编写程序、访问易变的虚拟网络资源,如带宽或延迟等,以及处理过程中产生的局部共享数据.在运行时内部的锁机制自动地对共享数据进行同步,减少了程序编写过程中的错误;

(3) 支持细粒度并行的运行时.为了支持上层的并行模型,本框架设计了一种新的针对共享内存的支持细粒度并行的运行时.框架运行时不仅可以并行地收发消息和处理事件,而且使用专门的计算线程对事件处理进行加速,提高了 SDN 虚拟化框架的处理效率.

据我们所知,本框架是面向 SDN 虚拟化、针对共享内存多核/众核环境的第一个支持细粒度事件并行处理框架.另外,针对 SDN 虚拟化框架内部的共享网络资源,我们首次在框架中提供了统一的接口,使用无锁的编程方式进行处理.

1 问题分析与动机

现有的 SDN 虚拟化框架通常使用第三方控制器对虚拟网络进行逻辑控制.现有的控制器通常使用粗粒度

并行的方式来处理SDN事件,即使用多个工作线程并行地处理不同的SDN事件,在事件处理过程内部串行地执行与事件类型绑定的事件处理方法(event handler).事件处理过程内部,复杂的事件处理方法将会带来较高的处理时延,而现有第三方控制器中的工作线程不仅负责事件的处理,而且负责收发消息,因此,较长的处理时延会阻塞异步I/O的执行,导致框架的逻辑控制层无法正常地执行.

为了对上述问题进行验证,本文对常用的第三方控制器(NOX[8],Beacon[13])中线程的执行效率(responses/s)进行测试,观察框架线程的执行效率随着事件处理方法执行时延增长的变化曲线.测试过程中使用了一个工作线程执行事件处理过程和消息收发.通过调整事件处理方法的执行时间,测试100s内框架工作线程的执行效率.测试结果如图1所示:随着事件处理时延的增长,不同框架线程的处理效率逐渐下降.当事件处理时延大于100μs时,NOX线程的效率为0(在图1中为负∞);当事件处理时延大于1ms时,Beacon线程的效率也降为0(在图1中为负∞).测试结果说明:在现有第三方控制器所使用的粗粒度并行方式中,工作线程难以满足处理时延较长的事件处理需求.

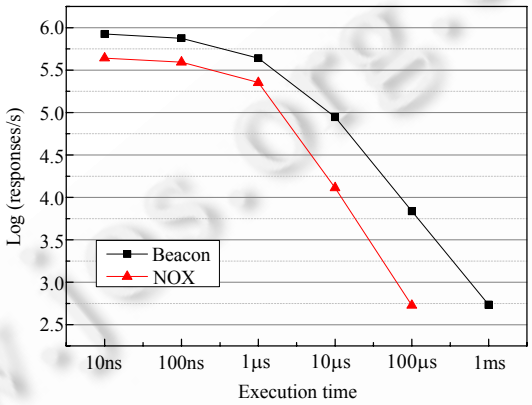


Fig.1 Decreasing throughput with the increasing execution time of the event handler

图1 随着事件处理方法执行时间的增长,框架吞吐量的变化曲线

在实际的SDN应用中,复杂的事件处理逻辑限制了网络的规模,需要更细粒度的并行支持.例如:在基于SDN的多媒体传输应用OpenQoS[14]中,基于服务质量(QoS)的路由计算具有较高的时间复杂度,OpenQoS使用LARAC算法对每个多媒体流进行路由计算,其时间复杂度为$O([n+m\log m]^2)$.该算法随着交换节点个数n和链路个数m的增长,具有较大的时间开销.为了满足更大规模网络下的性能需求,OpenQoS需要执行并行的路由算法[14].基于SDN的数据中心网络MicroTE[15],使用预先计算可行路径的方式以提高计算效率.但是,当拓扑结构发生变化时,MicroTE所使用的Bin-packing启发式算法同样具有较高的计算复杂度($O(PN\log N+PE+P\log P)$,其中,P为可行路径个数,N为交换节点个数,E为链路个数).当处理大规模的网络时,MicroTE明确提出了需要使用并行的算法提高计算效率[15].与此同时,现有研究[16]已提出并行的算法,通过细粒度的方式提高基于QoS的路由计算效率.结合上文的实验分析可以看出:由于现有SDN虚拟化框架的逻辑控制层缺乏对细粒度并行的良好支持,使得虚拟网络的规模受到限制,并且工作线程难以满足实际应用中复杂的事件处理需求.

另外,现有的SDN虚拟化框架在编程方面还存在如下两个问题:

- 首先,现有的SDN虚拟化框架缺乏全面、直接的编程接口定义虚拟网络和网络资源.FlowVisor,IVOF[17]等框架关注交换节点、网络设备、链接等网络资源的抽象,隐式的虚拟网络定义增加了编程人员的编程难度[12].libNetVirt[18]虽然隐藏了网络划分,但是编程人员难以对网络进行深层次的控制.OVN使用基于流的虚拟网络定义方式,编程简单,但缺乏直接的接口对虚拟网络资源进行操作;
- 其次,现有的SDN虚拟化框架缺乏直接的接口对虚拟网络资源进行定义和操作.虚拟网络资源(如交换节点、带宽、转发表等)通常在框架内部预先定义,并直接提供给编程人员使用.在框架运行过程中,由

运行时隔离不同虚拟网络对物理网络资源的占用.随着 SDN 技术的发展,将会产生更为丰富的网络资源,而现有框架预定义的网络资源有限,难以适用于未来 SDN 网络.

为了解决上述问题,面向多核/众核环境,我们设计了一种支持细粒度并行的 SDN 虚拟化框架.在框架的逻辑控制层,通过如下两种并行方式支持细粒度并行:

(1) 支持同一事件多种处理方法的并行执行.一个类型的 SDN 事件可能绑定多种处理方法,多种处理方法一起构成了对该类型 SDN 事件的处理逻辑.对于不存在依赖关系的事件处理方法,本框架通过标记相同优先级的方式进行并行执行,并且由运行时保证同步;

(2) 支持事件处理方法内部的并行执行.针对逻辑复杂的事件处理方法,本框架支持并行编程接口,在事件处理方法执行过程中,动态地生成多个子任务并行执行.

另外,利用本框架提供的编程接口,编程人员不仅可以通过划分流空间的方式直接定义虚拟网络及其处理逻辑,并且可以自定义虚拟网络资源及其操作方法,供不同的虚拟网络使用.框架的运行时根据所接收的不同类型的 SDN 消息,产生对不同虚拟网络中的流的处理任务以及对虚拟网络资源的处理任务,并且利用多个线程并行地加以处理.

2 并行编程模型

本框架并行编程模型是一种面向 SDN 虚拟化、基于 SDN 事件产生与消费的并行编程模型.与现有虚拟化框架的编程模型不同,本框架指导编程人员从一个新的角度虚拟化网络,并对虚拟的网络进行更为复杂的逻辑控制.

2.1 模型概述

如图 2 所示,本框架编程模型由如下组件构成.

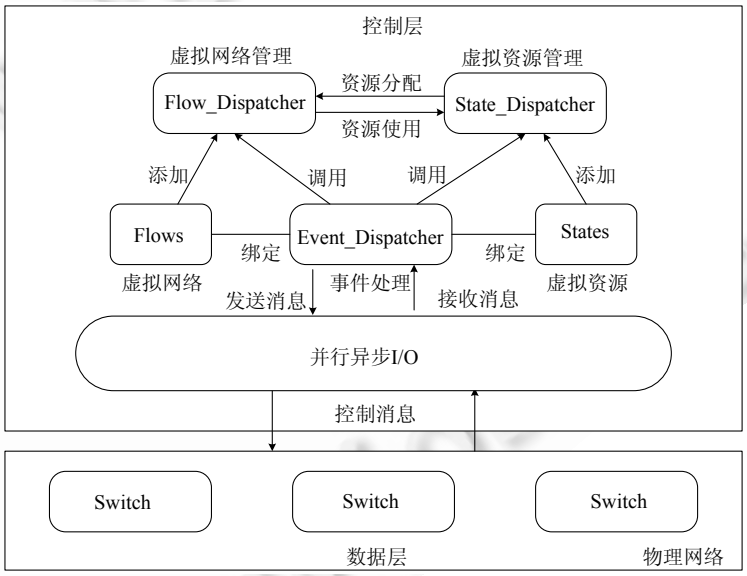


Fig.2 Achitecture of the parallel programming model

图 2 并行编程模型结构图

- Flows,编程人员定义的虚拟网络,与 OVN 生成的虚拟网络类似,该网络由一组特定类型的流构成;
- States,由编程人员定义的虚拟网络资源.作为全局共享的数据对象,这些资源将在框架初始化时生成,由 State_Dispatcher 管理,供不同的虚拟网络使用;

- Flow_Dispatcher,虚拟网络管理组件.该组件保存编程人员定义的虚拟网络的信息,当物理网络中的流消息到达时,根据其所属的虚拟网络,生成对应的流对象进行处理;
- State_Dispatcher,虚拟网络资源管理组件.该组件负责维护所有的虚拟网络资源,并对资源进行隔离,在提供给不同虚拟网络使用的同时,保证操作的一致性.另外,当物理网络中与虚拟网络资源相关的事件到达时,生成对应的任务,对网络资源进行维护;
- Event_Dispatcher,包含不同类型事件所对应的处理方法.该组件接收物理网络发送的消息,将消息转化为对应类型的事件,并根据具体事件对特定的流对象或状态对象进行操作.

2.2 流的抽象

作为 SDN 虚拟化框架最重要的工作,本文所提框架指导编程人员从流的角度对网络进行划分并构建虚拟网络.框架中的虚拟网络由一组相同类型的流构成,例如 HTTP 流的 TCP 源端口号或 TCP 目的端口号为 80.

本框架提供了一个流的抽象类 Base_flow,用于定义一组特定类型的流,对网络进行划分.Base_flow 类包含了流的基本信息,例如包头域(header field)、流的名称(flow name)以及一个流分片(flow slice),其中,流分片用于网络划分.另外,Base_flow 类还具有一个统一的虚方法 start,通过重写该方法,编程人员可以对虚拟网络的处理逻辑进行定义.

如下面代码所示,为了定义一个虚拟网络,编程人员需要通过继承的方式,定义一组特定类型的流.

- 首先,编程人员需要在 configure 方法中配置流的信息,如下面第 7 行、第 8 行所示,将所有 tcp 目的端口号为 22 的流添加到流分片中并定义流的名称;
- 其次,需要重定义 start 方法,编写该虚拟网络的处理逻辑.

```
1:  class Tcp_Flow: public Base_flow{
2:  public:
3:      /* Tcp_Flow 初始化*/
4:          Tcp_Flow(·) {… …}
5:      /* 添加流空间 */
6:          void configure(·){
7:              flow_slice.add(tcp_dst_port,22,true);
8:              flow_name="tcp_flow";
9:          }
10: /* 重定义 start(·)*/
11:     void start(·){
12:         /* 定义处理逻辑*/
13:     }
14: }
```

通过将创建的流的实例添加到 Flow_Dispatcher 中,框架将会自动地产生虚拟网络,并对该网络进行逻辑控制.当物理网络中的流消息到达时,框架会根据该流的信息直接生成对应类型的流的对象,并对该对象进行处理.为了减少内存占用,当流对象执行完成后将会被自动删除.但这种做法仍然存在一个问题:在物理交换机中的流表被修改之前,同一个流中的多个数据包可能会被重复发送到正在运行的框架.针对这一情况,如果将已处理的流对象立即删除,则有可能导致同一个流被重复处理,浪费资源,甚至产生逻辑错误.因此,框架运行时将短暂地保存已处理的流对象,当上述情况出现时,会根据该流对象的处理结果直接转发重复的数据包.

2.3 网络资源的抽象

与现有的 SDN 虚拟化框架不同,本文所提框架支持自定义的虚拟网络资源.利用本框架提供的 Base_state 类,编程人员可以定义任意的网络资源,并且编写对网络资源的操作方法,供虚拟网络使用.

与 Base_flow 类相似,Base_state 类是一个基础模板类,用于定义一个被虚拟网络共享的网络资源.Base_state 类中包含了该资源的基本信息:资源名称、特定的数据对象以及一个 global 参数决定该对象是否作为全局共享资源.Base_state 类中还包含了一个串行队列,用于保存对该网络资源的所有操作.

下面的代码示例展示了保存网络拓扑信息的 Topology 类的定义.利用 Base_state 类定义一个网络资源,编程人员需要声明保存该资源数据对象的类型(第 1 行)、设置该资源的名称(第 7 行)以及编写维护该资源所需要的方法(第 10 行~第 12 行).为了实现对网络资源更加灵活的处理,在本框架中,全局共享的网络资源对于各个虚拟网络不再透明,编程人员需要自己编写方法以保证一个特定网络资源内部的隔离性.所有的虚拟网络资源统一由 State_Dispatcher 管理,State_Dispatcher 负责维护并向虚拟网络分配网络资源,与此同时,保证不同虚拟网络对虚拟网络资源操作的一致性.

```
1:  class Topology: public Base_state ⟨Matrix⟩{
2:  public:
3:      /* Topology 初始化*/
4:          Topology(·) {… …}
5:      /*  配置资源信息*/
6:          void configure(·){
7:              state_name=“topology”;
8:          }
9:      /*定义资源的相关操作方法  */
10:         void add_link(·) {… …}
11:         void add_switch(·) {… …}
12:         … ….
13: }
```

2.4 事件处理接口

本文所提框架提供了新的事件处理接口 register,该接口可以将任意一个 SDN 事件类型与任意一种事件处理方法进行绑定,当该事件触发时,将调用对应的处理方法.

下面的代码展示了本框架提供的事件处理的两种方式:

- 第 1 种方式(如第 1 行、第 2 行所示)将一个特定的数据对象 object 以及处理该对象的方法 function 与一个特定的事件类型 E 绑定.当 E 类型事件被触发时,将调用 function 方法对数据对象 object 进行处理;
- 第 2 种方式(如第 3 行、第 4 行所示)是现有框架常用的方式,将一种处理方法与一个特定的事件类型 E 绑定.所不同的是,local 参数将执行该方法的线程的类型,如果为 true,则该方法将被 I/O 线程直接执行;否则,将产生任务交由计算线程执行.

上述方法中的 priority 参数定义了该方法的优先级,所有的 function 方法在执行结束后将返回一个布尔类型的返回值,如果为 false,则停止对该事件的执行.在本框架运行时内部,绑定同一个事件类型的多种处理方法会根据 local 和 priority 参数并行执行,具体实现方式将在后续章节给出介绍.

```
1: template ⟨typename E,typename R⟩
2: register (R object, & R::function,int priority);
3: template ⟨typename E⟩
4: register (bool function,int priority,bool local);
```

2.5 并行的数据对象操作

本文所提框架中有两类数据对象:根据不同类型的流产生的流对象;保存虚拟网络资源的状态对象.该框架提供了简单的接口,在底层事件处理过程中,对这两类数据对象进行并行的操作与同步.

如下面的代码所示,handle 方法是针对数据对象的并行操作方法.当 B 类实例对象调用 handle 方法时,需要指定处理该对象的方法 method,R 类型返回值 return_value 以及该方法所需参数.在 handle 方法执行过程中,会动态地产生任务,并根据实例对象的类型 B 由运行时中不同的线程并行执行.如果为流对象,产生的任务将直接放入计算线程的本地队列中,由多个计算线程并行执行;如果为全局共享资源对象(该对象的 global 参数为 true),则直接将生成的任务放入该对象的串行队列中,由状态线程串行执行;如果为细粒度并行中产生的局部共享对象(该对象的 global 参数为 false),则产生的任务将依然被放入计算线程的本地队列中,并且由运行时保证操作该数据的一致性.框架通过 wait 方法进行同步.handle 方法会返回生成任务的 TaskId.wait 方法通过获取这个 ID,与该任务进行同步.框架同样提供了 wait_all 方法,对产生的多个任务进行同步.

```
1: template 〈typename R,typename B,argtypes...〉
2: TaskId handle〈R〉 (R &return_value, & B::method,args…);
3: void wait (TaskId id);
```

为了防止多个 handle 方法同时对同一个返回值 return_value 修改产生的冲突,本框架要求返回值的类型 R 必须为 Base_state 类的子类.当多个 handle 方法并发执行时,运行时会保证对统一返回值进行同步操作的正确性.另外,本框架还提供了无返回值的 handle 方法.

2.6 框架示例程序

为了进一步对本框架进行说明,我们重新编写了 Learning Switch 程序[19]作为框架的示例程序.

在 Learning Switch 程序中,将网络中所有交换机的 mac 地址转发表作为全局共享的网络资源,并进行动态维护.当网络中非 LLDP 类型的流到达时,根据当前 mac 地址转发表的信息进行学习或者转发.

图 3 展示了 Learning Switch 程序详细的代码示例.

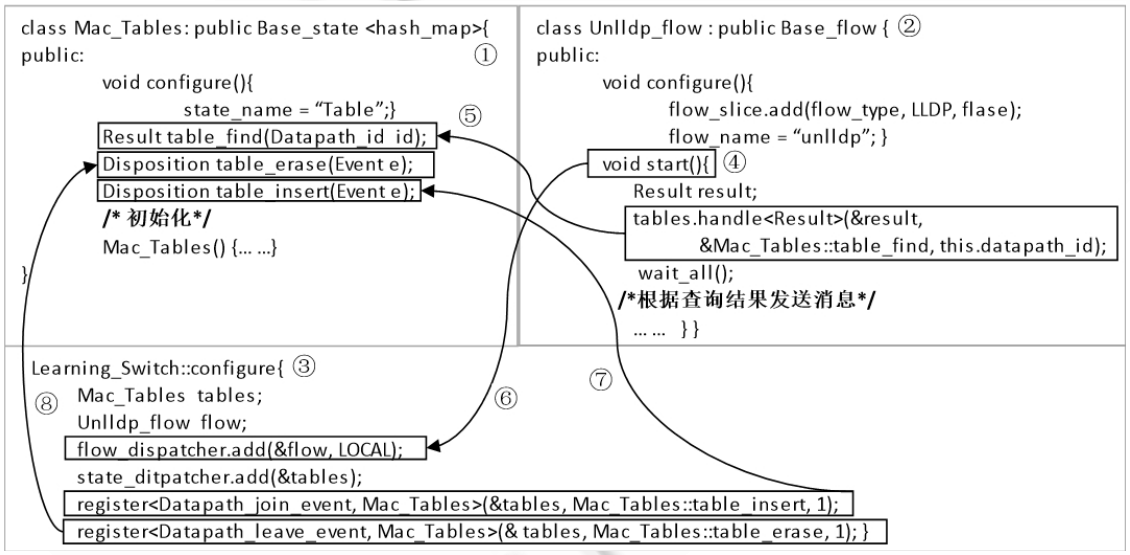


Fig.3 Example program: Learning Switch

图 3 示例程序:Learning Switch

图 3 所示代码示例分为 3 部分:全局共享资源 Mac_Tables 的定义(①)、非 LLDP 虚拟网络 Unlldp_flow 的定义(②)以及程序的配置信息(③).图中所有 configure 方法仅在框架初始化阶段执行.通过重写 start 方法对虚拟网络的处理逻辑进行定义(④),当物理网络中的数据流到达时,首先判断其类型是否为 LLDP,如果是,则丢弃该流;否则生成 Unlldp_flow 类型的流对象,执行 start 方法.在 start 方法执行过程中,将会动态地生成一个任务,根据交换机 ID 号查询对应的 mac 地址转发表(⑤).该任务将被直接放入 Mac_Tables 类中的串行队列中.另外,图 3

中的 Result 类型为 Base_state 类的子类,其实例对象 result 为局部的数据对象,不会被其他虚拟网络访问.

在程序配置过程中(③),定义的虚拟网络以及网络资源将被添加到运行时的管理模块中,并且与底层 SDN 事件进行关联.当 flow_dispatcher 接收到一个流对象时,会根据流对象的配置信息划分流空间,产生新的虚拟网络,将产生的虚拟网络与其处理逻辑进行关联(⑥).当 tables 添加到 state_dispatcher 后,将被定义为全局共享的网络资源,被所有的虚拟网络共享.通过 register 方法将交换机加入和离开事件与 tables 关联(⑦,⑧),当物理网络中有交换机加入或离开时,将会对 tables 进行相应的操作,并由运行时保证操作的一致性.

3 细粒度并行的表达

与现有的 SDN 虚拟化框架不同,本文所提框架通过对细粒度并行的支持,提高虚拟网络的逻辑处理能力.本节主要从两个方面展示如何利用本框架,在虚拟网络的逻辑处理过程中实现细粒度的并行操作:同一事件类型的多个事件处理方法之间的并行;事件处理方法执行过程中的并行.

3.1 多个事件处理方法的并行执行

绑定同一个事件类型的多个事件处理方法之间可能不存在依赖关系,因此,可以通过并行地执行事件处理方法提高框架处理效率.利用框架提供的 register 方法,绑定同一事件类型的多个处理方法之间的依赖关系通过优先级参数确定.对于具有相同优先级的事件处理方法,在事件处理过程中将被并行地执行.

框架初始化阶段将会执行所有 register 方法.不同类型的 SDN 事件可能与一个事件处理方法或一个事件处理方法链表相关联.框架运行时使用 DAG 图对事件处理方法链表中的多种方法按照不同的优先级进行保存.具有相同优先级的方法将被分配到 DAG 图中的相同层次.优先级按照 DAG 图从上到下的顺序递减.在框架运行过程中,当事件被触发时,会获取对应的 DAG 图,I/O 线程将会按照优先级从大到小顺序执行.在执行过程中,会根据事件处理方法的 local 值或操作对象的类型,产生不同的任务与计算线程和状态线程协作执行.通过这样的方式,编程人员可以最大化事件处理的并行度.

下面的代码展示了当新的交换机加入到物理网络时,对虚拟资源的更新过程.代码中,3 种事件处理方法与 Datapath_join_Event 绑定.当该事件被触发时,I/O 线程首先执行前两种方法.在执行过程中,将动态产生两个任务分别发送到 topology 和 bandwidth 的串行队列中,由特定的状态线程对并行执行.I/O 线程将等待,直到两个任务执行结束.然后,生成新的任务发送给计算线程,根据新的网络状态,对预先计算的路径信息进行重路由.由于该事件处理方法是最后一个处理方法,I/O 线程将处理其他事件,而由计算线程完成该事件的处理.

```
1: //处理交换机加入事件
2: register⟨Datapath_join_Event,Topology⟩ (topology,Topology::handle_datapath_join,1);
3: register⟨Datapath_join_Event,Bandwidth⟩ (bandwidth,Bandwidth::handle_datapath_join,1);
4: register⟨Datapath_join_Event⟩ (Rerouting,2,FALSE);
```

3.2 事件处理方法内部的并行执行

框架为编程人员提供了流和资源的模板类构建虚拟网络,定义网络资源.运行时,内部通过产生不同的数据对象,管理虚拟网络以及网络资源.借鉴数据并行的思想[20],可以通过并行处理不同的数据对象进一步提高程序并行度和框架处理效率.编程人员通过 handle 方法实现事件处理方法执行过程中的数据并行和任务并行.

如下面的代码所示,我们对串行的 K 近似算法[21]进行了并行化处理,并通过本框架加以实现.在算法执行过程中,共享的网络资源利用数据并行的方式进行处理,如第 3 行~第 5 行、第 7 行所示.生成的任务将被发送到对应网络资源对象的串行队列中.对于计算量较大的边的近似值计算和路径查询,我们利用任务并行减少算法执行时间,如第 9 行~第 15 行所示.该过程中,Result 对象为局部共享的变量,当执行 handler 方法时,将生成多个任务由计算线程并行执行.

```
1:   /*获取当前网络资源信息,并保存到 result 中*/
2:       Result result;
```

```
3:      topology.handle⟨Topology⟩(& result._topo, & Topology::get_topo);
4:      delay.handle⟨Delay⟩(& result._delay, & Delay::get_delay);
5:      bandwidth.handle⟨Bandwidth⟩(& result._bandwidth, & Bandwidth::get_bandwidth);
6:   /*等待查询结束*/
7:      wait_all(·);
8:   /*生成 max 个任务,并行地计算每条边的近似值,并等待所有任务完成*/
9:      for (int i=0; i<max; i++)
10:        result.handle(& Result::approximation,i);
11:     wait_all(·);
12: /*生成 max 个任务进行并行路径查找*/
13:     for (int i=0; i<max; i++)
14:        result.handle(& Result::para_dijkstra,i);
15:     wait_all(·);
16: /*根据路径信息,占用带宽资源*/
17:     bandwidth.handle(&Bandwidth::update, path);
```

4 系统实现

本文所提框架主要包含两部分:针对虚拟网络构建,网络资源抽象的编程模型;支持编程模型,对触发的事件进行并行处理的运行时.本节主要对运行时的实现进行介绍.

本框架利用 Openflow 协议[22]作为交换机与框架之间的通信协议.框架运行时内的线程共分为 3 类:I/O 线程、计算线程以及状态线程.不同类型的线程具有不同的行为,负责不同的工作.I/O 线程负责对 Openflow 消息进行收发,触发 Openflow 事件,执行处理复杂度低的事件处理方法;计算线程负责对处理复杂度高的事件处理方法进行并行执行;状态线程负责执行对全局共享的网络资源的操作.

图 4 中,$T_1^{I/O},T_2^{I/O},...,T_n^{I/O}$ 表示 I/O 线程,$T_1^{cal},T_2^{cal},...,T_m^{cal}$ 表示计算线程,$T_1^{State},T_2^{State},...,T_k^{State}$ 表示状态线程,*Switches* 表示交换机链接的集合,$States_i$ 表示全局共享网络资源的集合.如图 4 所示:框架运行时,I/O 线程并行地对 Openflow 消息进行收发,每个交换机链接与特定的 I/O 线程进行绑定.根据接收的 Openflow 消息类型触发对应类型的 Openflow 事件,并根据事件的类型获取对应的事件处理方法列表,按照优先级顺序进行执行.在同一优先级中,首先将 local 值为 false 的事件处理方法生成新的任务交由计算线程执行,然后顺序执行 local 值为 true 的事件处理方法.当该优先级内所有的事件处理方法执行完成后且所有方法返回值为 true 时,I/O 线程将会继续执行;否则,放弃对该事件的处理.当执行最后一个优先级时,I/O 线程不会等待 local 值为 false 的事件处理方法执行完成.

状态线程负责对全局共享网络资源进行统一的处理.每个全局共享资源对象与特定的状态线程进行绑定.每个状态线程通过轮询的方式执行与其相关联的全局共享资源对象的串行操作队列.共享资源对象的操作任务被串行执行,并且进入队列的顺序是不确定的.

与现有框架不同,本框架提供了额外的计算线程,对复杂度高的事件处理方法进行计算加速.每个计算线程具有一个本地任务队列.在执行过程中,计算线程首先从其本地任务队列获取任务并执行;如果队列为空,则利用任务窃取[23]的方式获取其他计算线程任务队列中的任务;如果窃取失败,则从总任务队列获取新的任务.在任务执行过程中,利用 handle 方法可以动态地产生对某个数据对象的处理任务,如果是全局共享的网络资源对象,则将产生的任务发送到该对象的串行队列中;否则,将生成的任务放到本地任务队列中,通过内部的锁机制保证执行过程的正确性.

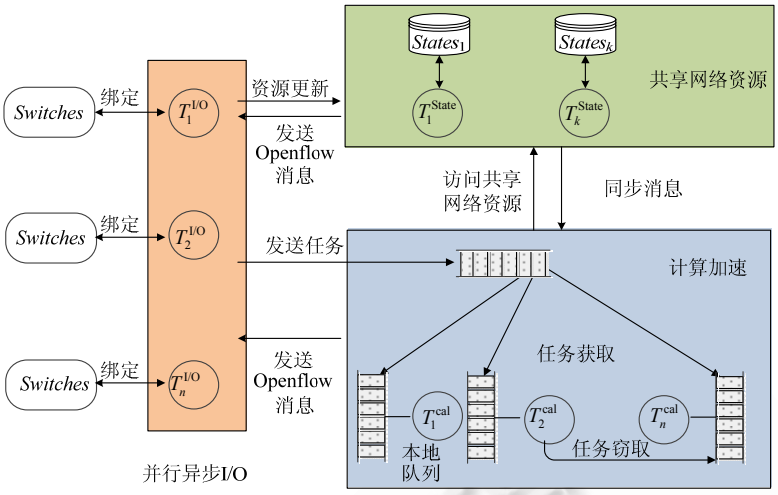


Fig.4 Various threads' cooperation in the framework runtime

图 4 运行时内部线程间的协作

5 实 验

本文所提框架利用细粒度并行提高虚拟网络的逻辑处理能力,使框架可以应用于大规模的软件定义网络,构建更大规模、功能更加丰富的虚拟网络.为了测试框架的处理性能,主要是虚拟网络的逻辑处理能力,我们从两个方面进行验证:首先,测试了框架执行复杂度低的事件处理方法时的效率,并与常用的第三方逻辑控制器的效率进行了比较;其次,对于本框架特有的细粒度并行,当执行复杂度高的事件处理方法时,针对不同算法,我们测试了本框架在不同并行度下的执行效率,并与粗粒度并行的执行效率进行比较.

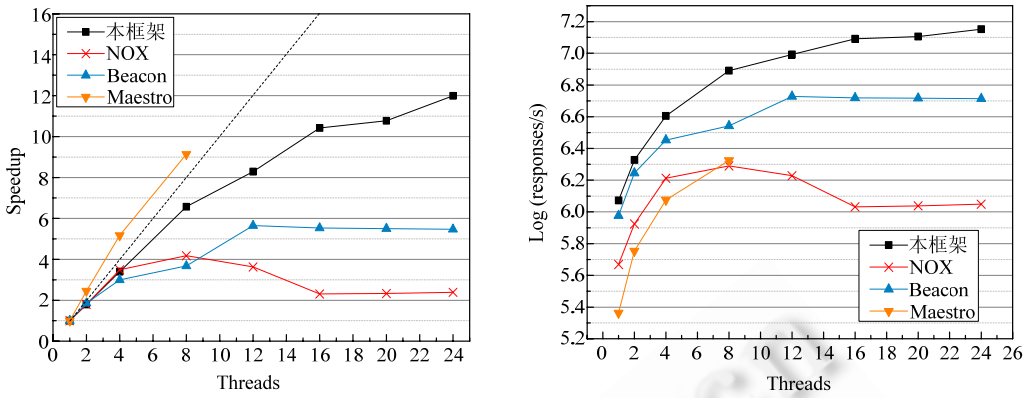
本节所有的实验均在一台四路 Intel Xeon 服务器上执行.每路上具有一个 Intel(R) Xeon(R) E7-4087 处理器,每个处理器包含 6 个核,主频为 1.87GHz,并且支持超线程.该服务器共有 24 个处理器核,共 48 个并行线程.我们使用针对 Openflow 控制器性能的标准测试程序 Cbench[24]对框架性能(responses/s)进行测试.在本节进行的实验中,每次与控制器连接,Cbench 都进行 13 组测试,其中,前 3 组作为预热阶段不计入测试结果,每组数据的测试时间设置为 10 000ms.每次测试取后 10 组结果的平均值,最终得出每秒平均的响应次数.测试结果表明,本框架具有更高的虚拟网络的逻辑处理能力和良好的可扩展性.

5.1 低复杂度事件处理方法的执行效率

为了与现有多线程逻辑控制器[8,13,25]进行比较,我们使用粗粒度的并行方式执行 Learning Switch 应用(如图 3 所示),通过增加 I/O 线程个数,测量框架逻辑处理能力的增长.每个 I/O 线程与不同的处理器核进行绑定.Cbench 绑定在专用的处理器核上,并通过片上网络与控制器进行通信.利用 Cbench 模拟了 64 个 Openflow 交换机,使用吞吐量模式对框架的性能进行测试.根据测试结果,通过公式(1)计算加速比.

$$Speedup = \frac{Multi\text{-}Thread\ Controller\ Performance}{SingleThread\ Controller\ Performance} \tag{1}$$

在执行 Learning Switch 程序时,使用了 1 个状态线程对共享哈希表进行处理,该线程没有加入到工作线程个数中.从图 5 的测试结果中可以清楚地看出:当使用粗粒度并行执行复杂度低的事件处理方法时,本框架的逻辑处理能力相比现有第三方控制器具有更高的性能以及更好的可扩展性.如图 5(b)所示,随着线程个数的增长,与现有的第三方逻辑控制器相比,本框架具有更高的性能.当线程个数小于 8 时,Maestro 的性能要低于本框架;当 I/O 线程个数大于 8 时,NOX 的性能逐渐降低,并趋于平稳;当线程个数超过 12 时,Beacon 的性能也不再增长.由于 Maestro 自身不支持 8 个以上个数的线程,因此当线程个数大于 8 时,没有 Maestro 的测试数据.



(a) 框架加速比 (b) 框架吞吐量

Fig.5 Lightweight event handler execution performance

图 5 低复杂度事件处理方法的执行效率

Table 1 Cache hit rates of different controllers

表 1 不同控制器的 cache 命中率

线程个数	本框架	Beacon	Maestro	NOX
1	0.699 8	0.671 8	0.813 9	0.662 9
2	0.698 9	0.633 6	0.760 6	0.660 2
4	0.690 9	0.608 7	0.766 1	0.647 3
8	0.700 2	0.601 9	0.746 4	0.640 1

如图 5(a)所示,与 Beacon 和 NOX 相比,本框架具有更好的扩展性.当线程个数从 1 增加到 8 时,Maestro 出现了超线性加速比.主要原因是:Maestro 运行时内部特殊的具有优先级的输入输出方式,获得了更高的 cache 命中率,见表 1.虽然 Maestro 出现了超线性加速比,但其性能要低于本框架.由于框架运行时将根据到达的流消息直接生成对应的流对象进行执行,省略了流事件触发的过程,因此可以更有效地利用更多的工作线程提高框架的逻辑处理能力.

5.2 高复杂度事件处理方法的执行效率

现有的 SDN 虚拟化框架以及第三方的逻辑控制器,在对虚拟网络进行控制时都不支持细粒度的并行操作,难以处理复杂度较高的逻辑处理过程.针对两种复杂度较高的 QoS 路由算法(QPAS[16],*K*-Approx[21]),我们利用本框架实现了两种算法的串行和并行版本,并分别使用粗粒度的方式执行串行的算法,使用细粒度并行的方式执行并行的算法.

在本次测试中,我们使用一台测试机运行 Cbench,并通过一台华为交换机与服务器上一个 10Gbps 的端口相连.为了更加准确地测试两种方法的执行效率,我们使用 Cbench 的延迟模式进行测试,并且利用 Cbench 模拟了 3 000 个交换机.在服务器端,我们构建了一个具有 3 000 个交换节点的胖树结构的虚拟网络,并且随机生成了每条链路的带宽和延迟信息.由于不同的链路状态将会导致不同的执行效率,因此在实验过程中,链路信息保持不变.

粗粒度并行的测试过程为:当流消息到达时,使用 I/O 线程直接进行处理,而不进行计算加速.在流对象处理过程中,首先获取当前的拓扑结构、带宽信息以及延迟信息,根据这些信息,使用串行算法进行计算.计算完成后,向 Cbench 端发送一条 Openflow 消息.与粗粒度并行的测试过程不同的是,细粒度并行在每个流事件的处理过程中使用并行的算法进行计算.如第 3.2 节所示,并行 *K*-Approx 算法在执行过程中分为如下两个步骤.

第 1 步,根据拓扑结构、延迟和带宽信息计算每条边的近似值.在计算过程中生成 *N* 个任务,分别计算拓扑矩阵的不同部分,并使用 wait_all 方法进行同步.该过程中任务之间相互独立,不存在共享数据;

第 2 步,根据归约后的矩阵生成 *N* 个任务,并行地计算最短路径,并使用 wait_all 方法进行同步.在路径计算过程中,不同的任务在执行过程中会访问一个共享的保存中间计算结果的队列.该队列作为局部共享变量,保存在一个 Base_state 类的子类中.通过 Base_state 类中内部锁机制,互斥地对该队列进行访问.

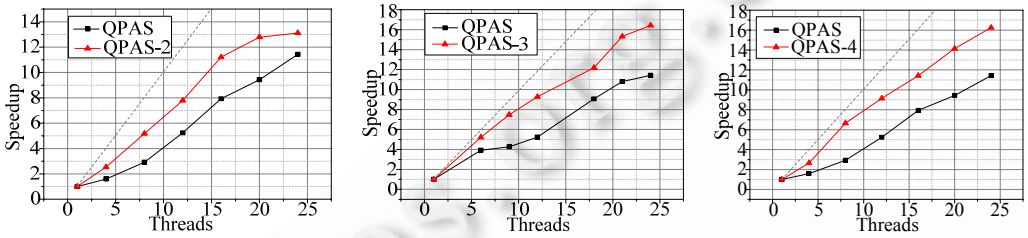
并行 QPAS 算法的执行过程同样分为两步:

第 1 步,根据拓扑结构、延迟和带宽信息生成 N 个任务,并行地搜索 QoS 度量 Pareto 子集,根据约束条件删除不满足条件的边,并通过 wait_all 方法进行同步,得到一个约简后的矩阵.该步骤执行过程中,不同任务之间相互独立;

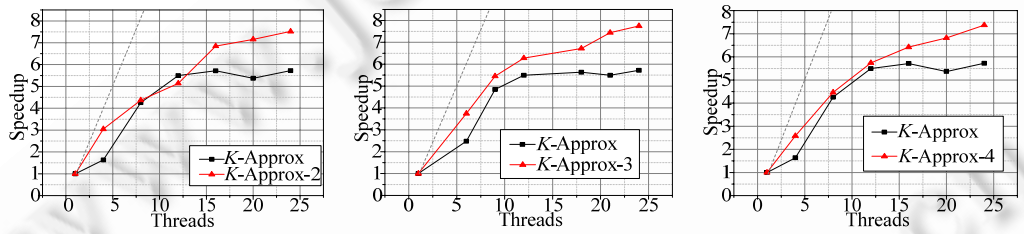
该并行算法的第 2 步与并行 *K*-Approx 算法的第 2 步相同.

N 为并行 *K*-Approx 算法和并行 QPAS 算法的并行度.通过人为地调整 N 的大小,测试在不同的并行度下,框架性能随计算线程个数增长的变化情况.在测试细粒度并行时,运行时内包含一个 I/O 线程对消息进行收发,一个状态线程对共享数据进行处理,利用多个计算线程对流处理过程进行加速,计算线程之间通过任务窃取并行地执行生成的子任务.所有线程绑定到不同的处理器核上.根据测试结果,利用公式(1)计算加速比.

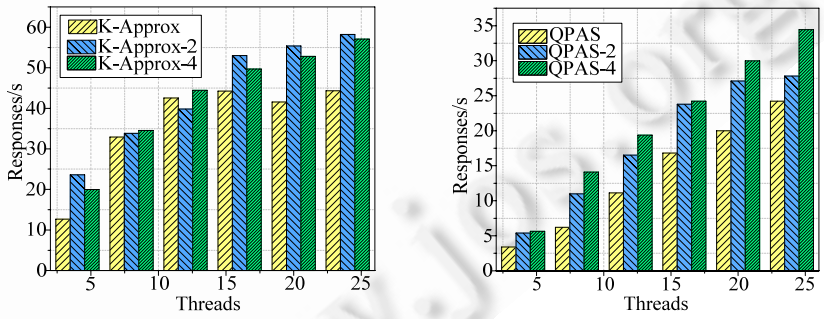
对于 QPAS 算法和 *K*-Approx 算法,与粗粒度并行方式相比,细粒度并行方式随着线程个数的增多具有更高的性能(如图 6(c)所示)以及更好的扩展性(如图 6(a)和图 6(b)所示).



(a) QPAS 算法的加速比:QPAS 表示串行 QPAS 算法,QPAS-*i* 表示并行度为 *i* 的并行 QPAS 算法



(b) *K* 近似算法的加速比:*K*-Approx 表示串行 *K*-Approx 算法,*K*-Approx-*i* 表示并行度为 *i* 的并行 *K*-Approx 算法



(c) 不同并行度下 QoS 路由算法的性能比较

Fig.6 Test results of heavyweight event handler execution

图 6 高复杂度事件处理方法的执行效率

产生如上结果的主要原因是:并行化的算法可以有效地提高算法的执行效率.另外,细粒度并行方式在执行过程中被处理的流对象的个数相对较少,因此,每个计算线程所占用的内存较少.对于 QPAS 算法,如图 6(a)所示:

随着线程个数的增加,加速比持续地增加.但对于 *K*-Approx 算法,当线程个数增加到 15 时,加速比的增长趋于平缓,如图 6(b)所示.主要原因是:在 QoS 算法执行过程中,*K*-Approx 算法动态生成的任务具有较短的执行时间,从而导致了计算线程更加频繁的任务窃取过程,增加了执行开销.从图 6(c)中可以看出:随着线程个数的增长,与串行 QoS 算法相比,并行的 QoS 算法具有较低的处理时延.另外,在并行算法执行过程中,不同的并行度导致并行算法的处理时延存在差异.在不同个数的线程下,随着执行过程中并行任务个数的增多,并行 QPAS 算法的执行效率逐渐增长.当线程个数大于 16 时,并行 *K*-Approx 算法在并行度为 2 时,取得更高的执行效率.

6 相关工作

FlowVisor[1]应用于物理网络与第三方控制器之间,通过对实际网络进行分片(slice)实现网络的虚拟化功能.每个网络分片拥有不同的转发逻辑策略,可以利用多个第三方控制器独立地对每个分片进行逻辑控制,保证了分片之间不受干扰.在 FlowVisor 基础上,ADVisor[11]和 AutoSlice[26]丰富了 FlowVisor 的功能,简化了框架部署难度.IVOF[17]框架通过构建虚拟交换机构建虚拟网络,虚拟交换机由不同类型的控制器进行控制,整个框架具有统一的组件进行管理.上述框架从不同方面解决了 SDN 虚拟化框架应用中的问题,但是这些框架使用分布式的处理方式,其执行过程中,网络信息的同步会带来较大的网络流量与处理开销,从而增加流的处理延迟.另外,这些框架依赖于第三方的控制器进行逻辑控制,限制了虚拟网络的功能与规模.

与上述框架相比,本框架使用单控制器对网络进行控制,很大程度上减少了网络信息同步的开销,而且本框架具有更高的逻辑处理能力,可以构建更大规模的虚拟网络.

OVN[12]与 libNetVirt[18]等框架整合了网络虚拟和逻辑控制,使用单一的控制器对整个网络进行控制,简化了虚拟网络构建,利用片上网络进行通信,减少了通信时延.OVN 是一种基于流的网络虚拟化框架,编程人员利用该框架可以直接对流空间进行划分,构建虚拟网络,并编写对应的处理逻辑对不同的虚拟网络进行控制.libNetVirt 通过集中式的控制器控制所有 OpenFlow 交换机,可以把整个 OpenFlow 网络虚拟成一个单独的网络结点,为服务提供者提供网络资源.与 OVN 类似,本框架同样通过对网络中流的抽象构建虚拟网络;但与 OVN 和 libNetVirt 不同,本框架侧重于利用资源丰富的众核环境对更大规模的物理网络进行抽象以及逻辑控制.利用细粒度并行的方式,本框架可以有效地提升虚拟化框架的处理效率.另外,编程人员利用本框架可以自定义网络资源,使框架具有更好的适用性.

为了解决原始单线程控制器的效率问题,现有的多线程控制器,如 NOX[8],Beacon[13],Maestro[25]等,利用粗粒度并行的方式,使用多个线程进行并行 I/O 以及事件处理.由本文第 1 节可以看出:现有的多线程控制器在事件处理过程内部不支持并行操作,无法满足复杂度较高的处理需求.本框架利用细粒度的并行,利用额外的计算线程对计算复杂过程加速,可以有效地提高逻辑控制的效率,丰富了虚拟网络的功能.

在 SDN 虚拟化以外的其他领域,同样存在异步并行编程模型.TigerQuoll[27]是针对 JavaScript 的并行编程模型,通过基于事件的产生和消费对进行并行的执行.F#[28]是通用的异步编程模型,利用 F#,可以在事件的处理过程中生成任务,实现处理过程之间的控制与交互.与这些模型不同,在事件处理过程中,本框架利用数据驱动的方式对共享数据进行统一的处理.

7 结束语

软件定义网络为网络虚拟化提供了良好的平台.SDN 网络虚拟化技术不仅可以有效地增强网络资源的共享,而且可以使网络控制变得更加灵活和智能.在本文中,我们介绍了一种面向 SDN 网络虚拟化的支持细粒度并行的编程框架.本框架提供了直接的接口对流空间进行划分以构建虚拟网络,自定义网络资源.在事件驱动的基础上,利用细粒度并行的方式提高虚拟网络的逻辑处理能力.另外,本框架还提供了统一的共享数据访问方式,利用无锁的编程方式访问共享的网络资源,简化编程.通过实验可以看出,本框架具有较高的逻辑处理效率和良好的可扩展性.因此,开发人员利用本框架可以充分地探索当前多核/众核环境中潜在的并行性,提高 SDN 虚拟化框架的处理效率,实现对更大规模网络的抽象,构建功能更加丰富的虚拟网络.在未来的工作中,我们还

将针对一些通用的并行编程模型或框架进行改造,进一步验证本框架的性能和可扩展性.

References:

[1] Sherwood R, Chan M, Gibb G, Flajslik M, Handigol N, Huang TY, Kazemian P, Kobayashi M, Underhill D, Yap KK, Appenzeller G, McKeown N. Carving research slices out of your production networks with OpenFlow. ACM SIGCOMM Computer Communication Review, 2010,40(1):129−130. [doi: 10.1145/1672308.1672333]

[2] Han SW, Kim N, Kim JW. Designing a virtualized testbed for dynamic multimedia service composition. In: Proc. of the 4th Int'l Conf. on Future Internet Technologies. New York: ACM Press, 2009. 1−4. [doi: 10.1145/1555697.1555705]

[3] Al-Fares M, Radhakrishnan S, Raghavan B, Huang N, Vahdat A. Hedera: Dynamic flow scheduling for data center networks. In: Proc. of the 7th USENIX Conf. on Networked Systems Design and Implementation (NSDI 2010). Berkeley: USENIX Association, 2010. 19. https://www.usenix.org/legacy/events/nsdi10/

[4] Niranjan MR, Pamboris A, Farrington N, Huang N, Miri P, Radhakrishnan S, Subramanya V, Vahdat A. PortLand: A scalable fault-tolerant layer 2 data center network fabric. ACM SIGCOMM Computer Communication Review, 2009,39(4):39−50. [doi: 10.1145/1594977.1592575]

[5] Tavakoli A, Casado M, Koponen T, Shenker S. Applying NOX to the datacenter. In: Proc. of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII). New York: ACM SIGCOMM, 2009. 1. http://conferences.sigcomm.org/hotnets/2009/

[6] Feamster N, Nayak A, Kim H, Clark R, Mundada Y, Ramachandran A, Tariq MB. Decoupling policy from configuration in campus and enterprise networks. In: Proc. of the 17th IEEE Workshop on Local and Metropolitan Area Networks (LANMAN). Piscataway: IEEE, 2010. 1−6. [doi: 10.1109/LANMAN.2010.5507162]

[7] Curtis AR, Mogul JC, Tourrilhes J, Yalagandula P, Sharma P, Banerjee S. Devoflow: Scaling flow management for high-performance enterprise networks. In: Proc. of the ACM SIGCOMM 2011 Conf. New York: ACM Press, 2011. 254−265. [doi: 10.1145/2043164.2018466]

[8] Gude N, Koponen T, Pettit J, Pfaff B, Casado M, McKeown N. NOX: Towards an operating system for networks. ACM SIGCOMM Computer Communication Review, 2008,38(3):105−110. [doi: 10.1145/1384609.1384625]

[9] Tootoonchian A, Ghobadi M, Ganjali Y. OpenTM: Traffic matrix estimator for OpenFlow networks. In: Proc. of the 11th Int'l Conf. on Passive and Active Measurement (PAM 2010). Berlin: Springer-Verlag, 2010. 201−210. [doi: 10.1007/978-3-642-12334-4_21]

[10] Zuo QY, Chen M, Zhao GS, Xing CY, Zhang GM, Jiang PC. Research on OpenFlow-based SDN technologies. Ruan Jian Xue Bao/Journal of Software, 2013,24(5):1078−1097 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4390.htm [doi: 10.3724/SP.J.1001.2013.04390]

[11] Salvadori E, Corin RD, Broglio A, Gerola M. Generalizing virtual network topologies in OpenFlow-based networks. In: Proc. of the IEEE Global Telecommunications Conf. (GLOBECOM 2011). Piscataway: IEEE, 2011. 1−6. [doi: 10.1109/GLOCOM.2011.6134525]

[12] Kontesidou G, Zarifis K. Openflow virtual networking: A flow-based network virtualization architecture [MS. Thesis]. Stockholm: Royal Institute of Technology, 2009.

[13] Erickson D. The Beacon OpenFlow controller. In: Proc. of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN 2013). New York: ACM Press, 2013. 13−18. [doi: 10.1145/2491185.2491189]

[14] Egilmez HE, Dane ST, Bagci KT, Tekalp AM. Openqos: An openflow controller design for multimedia delivery with end-toend quality of service over software-defined networks. In: Proc. of the Signal & Information Processing Association Annual Summit and Conf. (APSIPA ASC), 2012 Asia-Pacific. Piscataway: IEEE, 2012. 1−8. http://www.apsipa2012.org/

[15] Benson T, Anand A, Akella A, Zhang M. MicroTE: Fine grained traffic engineering for data centers. In: Proc. of the CoNEXT. New York: ACM Press, 2011. 8. [doi: 10.1145/2079296.2079304]

[16] Qin Y, Xiao WJ, Huang H, Liang BL, Zhao CG, Wei WH. A parallel QoS routing optimization based on Pareto subsets. Chinese Journal of Computers, 2009, 32(3):463−472 (in Chinese with English abstract).

[17] Sonkoly B, Gulyás A, Czentye J, Kurucz K, Vaszkun G, Kern A, Jocha D, Takács A. Integrated OpenFlow virtualization framework with flexible data, control and management functions. In: Proc. of the 31th IEEE Annual Int'l Conf. on Computer Communications (INFOCOM 2012). Piscataway: IEEE, 2012. http://infocom2012.ieee-infocom.org/

[18] Turull D, Hidell M, Sjodin P. libNetVirt: The network virtualization library. In: Proc. of the 2012 IEEE Int'l Conf. on Communications (ICC). Piscataway: IEEE, 2012. 5543−5547. [doi: 10.1109/ICC.2012.6364673]

[19] Voellmy A, Ford B, Hudak P, Yang RY. Scaling software-defined network controllers on multicore servers. ACM SIGCOMM Computer Communication Review (Special October Issue SIGCOMM 2012), 2012,42(4):289−290.

[20] Bal HE, Heines M. Approaches for integrating task and data parallelism. IEEE Concurrency, 1998,6(3):74−84.

[21] Xue GL, Sen A, Zhang WY, Tang J, Thulasiraman K. Finding a path subject to many additive QoS constraints. IEEE/ACM Trans. on Networking, 2007,15(1):201−211. [doi: 10.1109/TNET.2006.890089]

[22] OpenFlow Switch Specification Version 1.0.0. ONF, 2011. https://www.opennetworking.org/

[23] Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM), 1999,46(5): 720−748. [doi: 10.1145/324133.324234]

[24] Tootoonchian A, Gorbunov S. Cbench: An OpenFlow controller benchmarker. 2011. http://www.openflow.org/wk/index.php/Oflops

[25] Cai Z, Cox AL, Ng TSE. Maestro: A system for scalable OpenFlow control. Technical Report, TR10-08, Rice University, 2010.

[26] Bozakov Z, Papadimitriou P. Autoslice: Automated and scalable slicing for software-defined networks. In: Proc. of the 2012 ACM Conf. on CoNEXT Student Workshop (CoNEXT Student 2012). New York: ACM Press, 2012. 3−4. [doi: 10.1145/2413247.2413251]

[27] Bonetta D, Binder W, Pautasso C. TigerQuoll: Parallel event-based JavaScript. In: Proc. of the 18th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PpoPP 2013). New York: ACM Press, 2013. 251−260. [doi: 10.1145/2442516.2442541]

[28] Syme D, Petricek T, Lomov D. The F# asynchronous programming model. In: Proc. of the 13th Int'l Conf. on Practical Aspects of Declarative Languages (PADL 2011). Berlin: Springer-Verlag, 2011. 175−189. [doi: 10.1007/978-3-642-18378-2_15]

附中文参考文献:

[10] 左青云,陈鸣,赵广松,邢长友,张国敏,蒋培成.基于 OpenFlow 的 SDN 技术研究.软件学报,2013,24(5):1078−1097. http://www.jos.org.cn/1000-9825/4390.htm [doi: 10.3724/SP.J.1001.2013.04390]

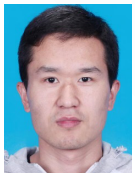
[16] 秦勇,肖文俊,黄翰,梁本来,赵成贵,魏文红.一种基于 QoS 度量的 Pareto 并行路由寻优方法.计算机学报,2009,32(2):463−472.

宋平(1987－),男,河北三河人,博士生,主要研究领域为面向特定领域的并行编程.
E-mail: ping.song@jsi.buaa.edu.cn

张晶晶(1990－),女,硕士生,主要研究领域为高性能计算.
E-mail: yishiweikong@126.com

刘轶(1968－),男,博士,教授,博士生导师,CCF 会员,主要研究领域为并行计算,计算机网络.
E-mail: yi.liu@jsi.buaa.edu.cn

钱德沛(1952－),男,教授,博士生导师,CCF 会士,主要研究领域为体系结构,网格计算,计算机网络.
E-mail: depeiq@buaa.edu.cn



刘驰(1990－),男,硕士生,主要研究领域为高性能计算.
E-mail: chi.liu@jsi.buaa.edu.cn

郝沁汾(1969－),男,博士,副教授,主要研究领域为高性能计算,计算机体系结构.
E-mail: Haoqinfen@huawei.com