同济大学
TONGJI UNIVERSITY

博士学位论文

同济大学学位论文 LaTeX 模板
使用示例说明与参考

（本论文由我要努力想办法撑到两行的著名国家杰出青年基金
(No.123456789) 支持）

姓　　名：同济人
学　　号：201804
所在院系：同济大学 Linux 用户组
学科门类：工学
学科专业：电子控制计算机
指导教师：陈杰教授
副指导教师：裴刚教授 （校外）

二〇一九年九月

同濟大學
TONGJI UNIVERSITY

A dissertation submitted to
Tongji University in conformity with the requirements for
the degree of Doctor of Philosophy

A Simple Sample of Tongji Thesis Using TongjiThesis

(Supported by the Natural Science Foundation of China for
Distinguished Young Scholars, Grant No.123456789)

Candidate :	Tongji Ren
Student Number :	201804
School/Department :	TONGJILUG
Discipline :	Gong Xue
Major :	DianziControlComputerScience
Supervisor :	Prof. Jie Chen
Associate Supervisor :	Prof. Gang Pei (XiaoWai)

September, 2019

学位论文版权使用授权书

本人完全了解同济大学关于收集、保存、使用学位论文的规定，同意如下各项内容：按照学校要求提交学位论文的印刷本和电子版本；学校有权保存学位论文的印刷本和电子版，并采用影印、缩印、扫描、数字化或其它手段保存论文；学校有权提供目录检索以及提供本学位论文全文或者部分的阅览服务；学校有权按有关规定向国家有关部门或者机构送交论文的复印件和电子版；在不以盈利为目的的前提下，学校可以适当复制论文的部分或全部内容用于学术活动。

学位论文作者签名：

年　　月　　日

同济大学学位论文原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，进行研究工作所取得的成果。除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人创作的、已公开发表或者没有公开发表的作品的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。本学位论文原创性声明的法律责任由本人承担。

学位论文作者签名：

年　　月　　日

摘要

论文的摘要是对论文研究内容和成果的高度概括。摘要应对论文所研究的问题及其研究目的进行描述，对研究方法和过程进行简单介绍，对研究成果和所得结论进行概括。摘要应具有独立性和自明性，其内容应包含与论文全文同等量的主要信息。使读者即使不阅读全文，通过摘要就能了解论文的总体内容和主要成果。

论文摘要的书写应力求精确、简明。切忌写成对论文书写内容进行提要的形式，尤其要避免“第 1 章……；第 2 章……；……”这种或类似的陈述方式。

本文介绍同济大学论文模板 TongjiThesis 的使用方法。本模板符合学校的硕士、博士论文格式要求。

本文的创新点主要有：

- 用例子来解释模板的使用方法；
- 用废话来填充无关紧要的部分；
- 一边学习摸索一边编写新代码。

关键词是为了文献标引工作、用以表示全文主要内容信息的单词或术语。关键词不超过 5 个，每个关键词中间用分号分隔。（模板作者注：关键词分隔符不用考虑，模板会自动处理。英文关键词同理。）

关键词：TEX，LATEX，CJK，模板，论文

ABSTRACT

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summarization of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words "the first chapter", "the second chapter" and the like should be avoided in the abstract.

Key words are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 key words, with semicolons used in between to separate one another.

Key Words: TEX, LATEX, CJK, template, thesis

目录

插图索引

表格索引

第 1 章　绪论

1.1　引言

软件定义网络（Software Defined Network，SDN）将网络控制平面和数据平面进行分离，并通过逻辑上集中式的 SDN 控制器向上层应用提供网络全局视图，使灵活、高效的网络管理变得可能[mckeown2008openflow, feamster2013road, b4]。对于底层数据平面中的网络设备，通过南向接口（例如 OpenFlow 协议[mckeown2008openflow]），SDN 控制器上的应用可以对其进行配置，进而控制网络中数据包的传输。然而，让上层应用直接调用底层网络设备的低级配置接口对其进行配置，则会导致系统非常容易出错，影响可拓展性。因此，研究人员在 SDN 中提出高级编程模型的概念[foster2011frenetic, maple, reich2013modular]，对上层应用提供对网络配置的高级抽象接口，并自动地将上层应用表达的高级抽象网络策略转化为底层网络设备的低级配置。然而，随着 SDN 概念的普及，以及底层网络设备技术的迅速发展，SDN 高级编程模型不断迎接着新的挑战。

一方面，随着 SDN 概念的普及，上层应用希望 SDN 高级编程模型提供的高级抽象接口可以不断地支持新的特性。从最原始的对单个交换机的配置，到可以对网络中多个交换机的配置，再到可以对网络中交换机和特定网络功能节点（如防火墙）的统一配置。从对网络管理的意图[berde2014onos]，到高级 SDN 程序（即逻辑上可以应用于网络中所有数据包的 onPacket 函数），再不断地丰富高级 SDN 程序的语法（如支持循环结构，支持路由代数[gao2018t] 等）。这些上层应用迫切希望的新的特性，给高级编程模型所提的自动地生成底层网络设备的低级配置带来了巨大挑战。

另一方面，随着底层网络设备技术的迅速发展，底层网络设备在不断变得高效、灵活的同时，其架构也变得越来越复杂。从 OpenFlow1.0[openflow1] 的单流表架构，到 OpenFlow1.1[openflow1-1] 的多流表流水线架构，再到协议无关并可自定义的多流表流水线架构的 P4[1]。最初的单流表架构虽然简单，但由于多个匹配域的叉乘（cross-product），单流表架构并不高效。多流表流水线架构解决了匹配域的叉乘问题，但其架构也变得复杂从而影响流表规则的下发：要考虑给哪一个流表下发。P4 的协议无关并可自定义的多流表流水线架构实现了非常灵活的网络配置，但使用时不光要考虑流表规则的下发，而且（实际上要先于流表规则下发）要考虑生成多流表流水线结构。支持有状态操作的可编程交换机[moshref2014flow, bianchi2014openstate] 其强大的灵活性也伴随着复杂的架构。这些新诞生的底层网络架构同样给高级

编程模型带来了巨大挑战。

随着高级抽象接口支持的特性越来越多，底层网络设备架构变得越来越复杂，两者之间结构的差异也会变得越来越大。而 SDN 高级编程模型始终需要考虑的问题则是，如何高效地填充两者之间不断变大的鸿沟，即尽可能地将上层应用表达的高级抽象网络策略转化为符合底层网络设备架构的高效的低级配置。

在这章中，我们先从面向单交换机和面向全网络两个大方向进行研究背景介绍。其中对于面向单交换机，我们先介绍可编程数据通路（datapath）的研究，再介绍面向单交换机编程模型的研究；对于面向全网络，我们先介绍面向一般网络编程模型的研究，再介绍针对特定网络编程模型的研究。再结合介绍内容，给出论文中主要的研究挑战以及相关解决方案。最后给出论文整体架构。

1.2 研究背景：面向单交换机

1.2.1 可编程数据通路的研究背景

可编程数据通路的研究可以分为两大类：基于硬件的可编程数据通路，基于软件的可编程数据通路。

基于硬件的可编程数据通路：灵活的数据包分类（即可以利用通配符对数据包匹配域的任意组合进行匹配）是实现可编程数据通路的基础。基于 TCAM 的三元匹配功能（即 0，1，x），可以非常高效地实现数据包分类[yu2005efficient, lakshminarayanan2005algorithms, song2005efficient]。然而，由于其的电力消耗，以及占用的电路空间，TCAM 不具备很好的可拓展性。Naous 等人[naous2008implementing]将数据通路中的匹配表分为准确匹配和通匹配，并分别利用 SRAM 和 TCAM 进行实现。最终他们在 NetFPGA 平台上实现了 OpenFlow 交换机。Jiang 等人[jiang2011scalable] 基于决策树的数据包分类算法，在单个 FPGA 上放置了 10K 条可以支持 5 个匹配域的规则或 1K 条可以支持 12 个匹配域的规则。

由于单纯利用交换机芯片无法实现灵活的数据包处理，研究者开始考虑利用交换机中的其他资源（如 CPU）来处理一部分数据包操作。Luo 等人[luo2009accelerating]利用基于网络处理器的加速卡实现 OpenFlow 交换机，并降低了百分之二十的网络数据包延迟。Lu 等人[lu2012using] 利用 CPU 来辅助 ASIC 交换机对数据流的处理。具体来说，他们利用 CPU 实现基于流转发的大容量匹配表以及利用 DRAM 存储网络中突发数据包。Mogul[mogul2012hey] 等人利用 CPU 实现交换机中的软件定义计数器，以达到对计数器相关信息的灵活操作。类似的，他们利用 DRAM 存储计数器，并通过 CPU 对其进行更新操作。数据包匹配后会在 buffer 中存储匹配记录，然后 CPU 根据 buffer 记录更新 DRAM 中数据。

对于单流表匹配结构对流表容量要求较大（由于匹配域叉乘），使用起来并不高效等问题，研究人员考虑用多流表流水线作为可编程数据通路的基本结构。OF-DPA[OF-DPA] 是 Broadcom 提出的 OpenFlow 数据平面抽象。OF-DPA 与 OpenFlow v1.3.4 兼容，其底层为多个具有固定匹配域流表的流水线结构。PicOS[PicOS] 与 OF-DPA 类似，在底层为多个具有固定匹配域流表的流水线情况下，向上兼容 OpenFlow 协议。FlowAdapter[pan2013flowadapter] 设计了具有三层的交换机架构。其中上层为 OpenFlow 软件数据平面，并通过中间的 Flow Adapter 转化为底层的 OpenFlow 硬件数据平面。

对于上面所述的多流表流水线架构，用户只可以下发流规则，不可以更改流水线结构。因此，为了追求更灵活的可编程数据通路，研究人员开始考虑自定义的多流表流水线架构。Forwarding Metamorphosis[rmt] 提出 RMT 架构，表示 Reconfigurable Match Tables。RMT 架构包含 ingress processing、egress processing、以及中间的队列结构。其中 processing 由 parser、deparser、以及中间的多个 stage 组成。一个流水线上的逻辑流表可以由一个或多个 stage 共同实现，因此 RMT 可以灵活地实现具有不同结构的多流表流水线。同时其 parser 也可以自定义。最终，RMT 实现了协议无关可自定义的多流表流水线架构。dRMT[chole2017drmt] 删除了 RMT 中 stage 与流表的绑定关系，即任何 stage 可以访问任何流表；删除了 stage 与 stage 的依赖关系，即数据包可以进入任意 stage 而无需经过前面的 stage，实现了更灵活更高效的数据包处理。为了考虑更丰富的数据包操作，PISA[pisa] 概括了 RMT 架构并实现了更灵活的数据包操作指令（atoms）。OpenState[bianchi2014openstate] 考虑了有状态的操作。

基于软件的可编程数据通路：Linux 内核[linux]，DPDK[dpdk]，Netmap[rizzo2012netmap]，Click[kohler2000click] 等工作需要对底层实现具有相当的了解才能在上面构造软件交换机，这对网络程序员需要快速适应这些平台并开发新特性增加了困难。Open vSwitch（OVS）[pfaff2015design] 考虑了向其添加流表规则的接口，但不可以自定义地配置协议以及操作。Oko[chaignon2018oko] 通过 Berkeley Packet Filter (BPF) 对 Open vSwitch 进行拓展，使其可以支持有状态过滤器。Pisces[shahbaz2016pisces] 是一个可编程的协议无关的软件交换机，程序员可以在上面通过 P4 语言进行配置，并可以输出以 OVS 为目标的底层代码。

1.2.2　面向单交换机编程模型的研究背景

2006 年，IETF 网络配置工作组提出了 NETCONF[enns2006netconf] 作为用于修改网络设备配置的管理协议。NETCONF 允许网络设备发布可通过其发送以及接受可扩展配置数据的 API。另一种广泛部署的网络设备管理协议是 SNMP[hare2011simple]。

SNMP 使用结构化管理接口（SMI），以获取存储于管理信息库（MIB）中的数据。通过 SNMP，网络管理员可以改变 MIB 中的变量以修改配置设置。

OpenFlow[mckeown2008openflow] 是目前 SDN 中所使用最广的南向接口标准。OpenFlow 对于支持 OpenFlow 转发设备提供了一个共同说明，以及对于 SDN 数据平面和控制平面之间的通信通道提供了标准。从 OpenFlow1.1[openflow1-1] 开始，OpenFlow 标准增加了多表、组表以及流水线处理技术。对于支持 OpenFlow 多流表交换机，当数据包到达时，和第一个流表中的流项进行匹配。匹配成功时执行相应操作，可以是应用指令或者跳转至指定其他流表，实现数据包在多个流表之间的转移。组表可以用来实现多播。从 OpenFlow1.3[openflow1-3] 开始增加 Meter 表，用来实现简单的 QoS 操作。

OVSDB[pfaff2013open] 的设计是向 Open vSwitch 提供一个高级的管理接口。除了 OpenFlow 的基本配置流表的能力外，通过 OVSDB 可以创建多个虚拟交换机的实例，以及向交换机接口设置 QoS 策略。

目前面向多流表流水线的交换机配置语言包括 Concurrent Netcore[schlesinger2014concurrent] 以及 P4[1]。Concurrent Netcore 可以用来指定路由策略以及多流表中数据包处理的顺序。P4 作为目前占有统治地位的配置自定义多流表流水线的语言，P4 不光可以支持自定义的流水线结构，而且可以自定义数据包的 parser 以实现协议无关的流水线。

与 P4 需要手动地指定每一个流表的格式以及流水线的结构，Packet transactions[sivaraman2016packet] 向用户提供了一个高级编程模型，可以用来自动地生成底层数据通路的结构。通过该高级编程模型，用户可以编写 onPacket 函数并提交给 Packet transactions，后者会自动生成底层数据通路结构以及配置。在生成底层数据通路结构时，Packet transactions 会合并用户 onPacket 函数中的语句。

1.3 研究背景：面向全网络

1.3.1 面向一般网络编程模型的研究背景

一般网络这里是相对特定网络（如车联网等）而言的概念。我们考虑面向一般网络的编程模型不会对网络实现的功能以及应用场景进行限制。

在 1990 年代中期，Active Networking[tennenhouse1997survey, tennenhouse1996towards] 提出了通过可编程网络基础架构，可以实现定制化的服务。主要有两种方法被考虑：1. 通过用户可编程交换机并使用 in-band 负责数据传输，out-of-band 负责管理信道；2. 将需要执行的程序附带在数据包上，并在交换机或路由器上执行对应的程序。

基本 SDN 控制器如[gude2008nox, erickson2013beacon, medved2014opendaylight, shalimov2013advanced]，向下通过 OpenFlow 协议与底层交换机通信，获取网络状态信息并传给控制器上的应用程序。通过响应式模式，上层应用收到底层数据包后，进行相关处理，再通过控制器提供的接口对相关网络设备进行配置。ONOS[berde2014onos] 提供意图（intent）的方式对网络中数据流的传输进行控制。

具有 SDN 编程抽象的系统如 FML[hinrichs2009practical]，一种基于流的管理语言提供了高级编程模式来指定网络安全相关配置。Onix[koponen2010onix] 引入了 NIB（Network Information Base）抽象，以便应用程序通过读写存储在 NIB 中的键值对来修改流表。Frenetic[foster2011frenetic]，Pyretic[reich2013modular]，Nettle[voellmy2011nettle]，提供了 SDN 编程语言。其中 Frenetic 的 NetCore 支持特定的策略组合形式，如信息收集和数据流控制。Pyretic 拓展了 Frenetic 提供了基于模块化的 SDN 编程语言。Nettle 则是基于函数式相应式编程（FRP）对网络进行控制。Maple[maple] 提出了基于算法式的 SDN 编程语言，并提供对数据包的读、测试等基本 API。通过记录数据包执行的踪迹，构建踪迹树，然后基于踪迹树自动生成底层流表。McClurg 等人[mcclurg2016event] 提出网络事件结构的模型，保证了在事件到达时对网络状态的更新过程中也可以满足的指定的不变性质。

SNAP[snap] 考虑将高级 SDN 程序转化为多个有状态交换机的配置。但是其编程模型为 One-Big-Switch 模型，即用户无法指定数据包在网络传输的具体路径。

1.3.2　面向特定网络编程模型的研究背景

我们这里对特定网络分为三种情况：1. 交换机和中间盒（或 NFV）的混合网络；2. 实现特定功能的网络（如 Paxos）；3. 针对特定场景的网络（如车联网，物联网等）。

混合网络：Qazi 等人[qazi2013simple] 通过 SDN 保证数据流可以正确地传过事先规定好的中间盒序列。Fayazbakhsh 等人[fayazbakhsh2014enforcing] 解决了当中间盒对数据包头进行修改后，数据包仍然可以按照指定路径进行转发。Trident[gao2018t] 考虑利用网络中间盒可以处理数据包更高层信息的功能，设计统一的高级编程模型，使网络数据包的传输可以依赖更高层信息（如数据包对应流的 http 信息）。为了实现类似的功能（即数据包根据高层信息进行转发），Mekky 等人[mekky2014application] 采取拓展 Open vSwitch 的方法，使交换机存储状态信息。

实现特定功能的网络：通过利用可编程交换机，一些传统的分布式算法可以高效地在网络中实现。NetPaxos[dang2015netpaxos, dang2016paxos] 利用 P4 语言对可编程交换机进行配置，使其分别扮演 Paxos[lamport2001paxos] 的 acceptor，coordinator 等角色，通过订制包头的数据包在交换机之间的传输，实现分布式一致性算法。基于类似的

想法，Netcache[jin2017netcache] 和 Netchain[jin2018netchain] 则利用可编程交换机流表的性质，将键值对的存储在网络中，实现分布式键值对存储。Typhoon[cho2017typhoon] 利用底层 DPDK 数据平面，实现基于 SDN 的实时大数据流处理框架。Beckett 等人[beckett2016don] 针对 BGP 场景，将 BGP 路由的约束转化为底层交换机的配置。

特定场景网络：Zhu 等人[zhu2018sdn] 考虑对利用 SDN 技术实现车联网中紧急消息的快速分发功能。Hare 等人[hare2012policy] 设计一个集中式的策略框架来管理车辆网络的光谱资源来保证用户需要的数据传输性能。Mobile fog[hong2013mobile] 提出面向物联网的编程模型。Mobile fog 中的一个应用由多个进程组成，每个进程映射到网络中的一个节点，如核心节点，fog 节点，边缘节点。进而这些进程根据节点之间的层级关系，也组成一个逻辑上的层级关系（如父子节点关系）。通过层级关系，进程之间可以发送数据，共同完成任务。

1.4 论文研究挑战

根据上节对研究背景的叙述，我们把本论文研究的挑战主要分为两部分: 面向单交换机编程模型的研究和面向全网络编程模型的研究。而每一部分都有两个基本问题：SDN 程序数据平面实现问题和针对特定场景情况的优化问题。其中实现问题是指：给定一个 SDN 程序，一个数据平面是否可以正确表达该 SDN 程序。正确是指：对于任意数据包，SDN 程序对数据包返回的结果和数据平面对数据包返回的结果一致。下面我们针对上述的两部分来具体说明论文研究挑战。

第一，在面向单交换机编程模型的研究中，已有的相关工作可以分为如下：1. 通过低级配置接口生成具有固定结构（如单流表或 OF-DPA）数据通路的配置；2. 将高级程序转化为具有可定制结构（如 RMT）数据通路的配置。但是缺少将高级程序转化为具有固定结构数据通路的配置的相关工作。而在转化之前，需要考虑的是该固定结构数据通路是否可以实现高级程序。因此，如何判断一个高级 SDN 程序是否可以在具有固定结构数据通路上实现，是一个挑战。除此之外，当 SDN 程序具有循环结构时，如何将该 SDN 程序实现在可定制结构数据通路上并没有相关工作。因此高级 SDN 程序中循环结构在可定制结构数据平面的高效实现，是一个挑战。

第二，在面向全网络编程模型的研究中，对于一般网络，大量工作是以响应式模式（即不会主动生成配置）对网络进行管理，因此效率并不高。SNAP 实现了主动编译生成数据平面配置，但其程序并不灵活。因此，给定一个高级灵活的 SDN 程序，如何生成面向全网络的数据平面配置是一个挑战。除此之外，针对不同场景，如软件定义联合[mishra2017comparing] 网络的低时延要求，以及车联网中车的移动性要求，如何优化数据平面配置是一个挑战。

1.5　论文研究内容

针对上一节中探讨的两方面的挑战，本文进行有针对性的研究，主要分为以下五个部分：

第一，针对判断高级 SDN 程序是否可以在具有固定结构数据通路实现问题，提出了将高级 SDN 程序和底层数据通路统一的特征空间。并将实现问题转化为空间中的比较问题，提出数据通路编程容量理论来系统地解决问题。

第二，针对高级 SDN 程序中循环结构在可定制结构数据通路的高效实现问题，提出了重复软件流水线转换。通过转换，在计算循环结构的最佳数据通路结构时显示出更高的效率。

第三，针对高级灵活的 SDN 程序面向全网络数据平面实现问题，当网络节点全部为可编程交换机时，提出将程序拆分并部署到不同交换机的方案，并算出给定目标下的最优部署；当网络节点存在固定功能节点时（如防火墙），提出程序正确性的定义，并通过系统路径约束的概念保证程序的正确。

第四，针对软件定义联合网络的具体场景，设计高级编程系统，并基于共享本地状态的方法，优化系统性能：降低数据传输时延，提高传输带宽。

第五，针对车联网的具体场景，提出软件定义车联网的架构以及编程框架。考虑车联网中车移动性特点，提出优化的规则下发方法，减少生成的规则数量。

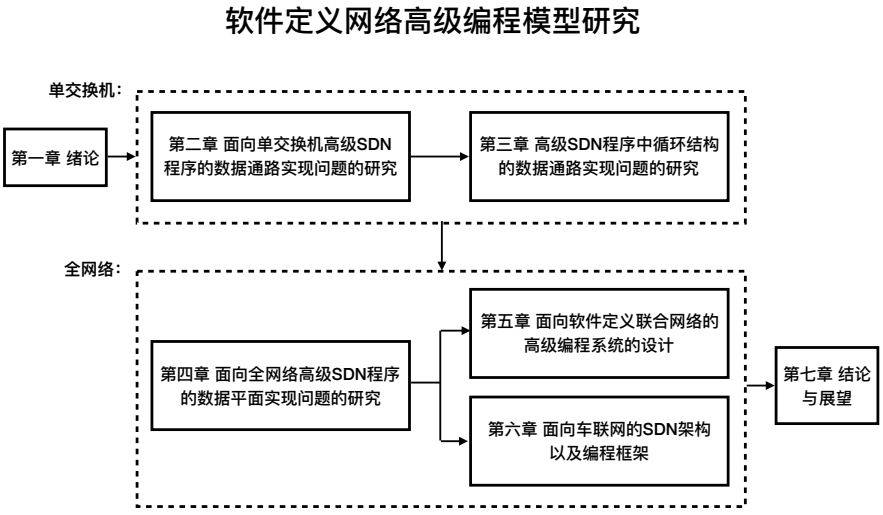


图 1.1　论文框架

1.6　论文结构

如图 1.1所示，本论文总共由七部分组成：第一章为绪论，先介绍了目前软件定义网络高级编程模型研究的背景。并根据背景给出论文挑战，最后列出论文内容。

第二章，第三章均为面向单交换机情况。其中，第二章研究面向单交换机的高级 SDN 程序数据通路实现问题，并给出特征空间以及数据通路编程容量理论。待第二章的基本实现问题讨论完后，第三章考虑高级 SDN 程序中循环结构，并给出循环结构在可定制结构数据通路的高效实现方法。

第四章，第五章，第六章均为面向全网络情况。其中，第四章研究面向全网络的高级 SDN 程序数据通路实现问题。待第四章的基本实现问题讨论完后，第五章考虑针对软件定义联合网络的数据平面优化，第六章考虑软件定义车联网架构以及编程框架。

最后，第七章总结了研究内容并对未来工作进行展望。在正文之后，附上参考文献，博士期间个人论文发表情况。

第 2 章　cap 章节

2.1　引言

SDN 的一个主要研究方向是可编程、高效的数据通路 (例如, OpenFlow1.3[openflow1-3], OF-DPA[OF-DPA], P4[1])。只有实现可编程性，SDN 数据通路才能支持各种不断发展的应用场景。同时，数据通路必须高效，能够满足高吞吐量和高成本效益等要求。在过去的几年中，多流表流水线已经成为 SDN 数据通路的一个关键部分 (例如, Domino[sivaraman2016packet], Forwarding Metamorphosis[rmt])。

然而，高效数据通路的一个问题是，它们必须经常以低级语言（甚至配置）进行编程，导致开发低效。例如，TCAM 对于实现高吞吐量至关重要，然而它不支持逻辑非。因此，SDN 的第二个主要研究方向是数据通路无关的、高级编程，并通过抽象来隐藏底层数据通路细节。在过去几年中，也出现了多个高级 SDN 编程模型（例如, Frenetic[foster2011frenetic], Maple[maple] ）。

随着这两个研究方向的发展，一个基本问题则是：给定的高级程序能否在给定的低级数据通路上实现。对这一问题的良好理解将有助于高级 SDN 程序的设计和数据通路的设计。给定一个固定的数据通路（例如，一个固定的流水线结构，OF-DPA），数据通路的供应商可以提供相关的高级程序的指南。反过来，给定一组高级程序，可以来设计支持这些程序的最紧凑的数据通路。即使对于可重新配置的数据通路（例如, P4），由于重新配置是昂贵且耗时的，因此可以基于对该问题的理解指导设计更健壮的数据通路。考虑到所有可以在给定可编程数据通路上实现的高级程序可以看作是数据通路的容量，我们将这一基本问题定义为 SDN 数据通路编程容量问题。

然而，解决数据通路容量问题并非易事。考虑一个简单的数据通路，Simple-DP。如图 2.1所示，它是最简单的数据通路之一，由构成流水线的三个表组成，其中第一个表 (`t1`) 在源 IP 上匹配，并且可以跳到以下两个表，这两个表都在目标 IP 上匹配。

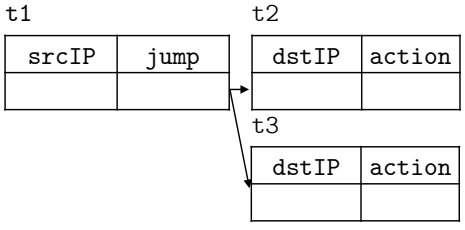


图 2.1　一个简单的数据通路示例: Simple-DP

我们再考虑下面两个简单的高级 SDN 程序，它们都采用算法性、事件驱动

的编程模型（下面会给出有关编程模型的更多详细信息）。我们可以发现第一个程序可以通过 Simple-DP 实现，而第二个程序则不能。

```
//  Routing Function: secureL3Route
L1: def secureL3Route(Addr srcIP, Addr dstIP):
L2:   if srcIP == 10.0.0.1:
L3:     return Forward(port=shortestPath(dstIP))
L4:   else:
L5:     return Drop();
```

```
//  Program: twoHostL3Route
L1: def twoHostL3Route(Addr srcIP, Addr dstIP):
L2:   if srcIP == 10.0.0.1:
L3:     return Forward(port=shortestPath(dstIP))
L4:   elif srcIP == 10.0.0.2:
L5:     return Forward(port=securePath(dstIP))
L6:   else:
L7:     return Drop();
```

尽管前面的数据通路和高级程序是最简单的，但它们已经可以说明解决 SDN 数据通路容量问题并不明显。由于需要实现多个服务，一般数据通路和高级程序可能要复杂得多，因此它们在分析中可能会带来严峻的挑战。本章的目标是试图得到一个系统的方法来解决 SDN 数据通路容量问题。

本文的贡献可以概括如下。首先，我们提出了一个统一的特征函数空间来消除程序结构和流水线布局的复杂性和不一致性。其次，我们在这个函数空间中定义了一个比较运算，用来判断一个高级程序是否可以在给定的流水线上实现。

2.2 相关工作

高级 SDN 程序编译器：过去几年，人们提出多种允许程序员用高级语言编写 SDN 程序，并将这些程序编译成流表的系统。因为它们需要检查高级程序到交换机流表的转换，因此这些系统与本章工作相关。我们将这些系统分为两类：无层和分层。

无层系统 (如 SNAP[arashloo2016snap], FML[hinrichs2009practical], FlowLog[flowlog], Maple[maple])，要求程序员通过编写数据包处理函数的形式，指定数据包的转发行为。然后 SDN 控制器使用这些函数来配置和更新网络状态。这样的系统启发了流水线容量定理中函数的概念。但是，这些系统在没有实际执行的情况下无法验证提交的函数是否可以写入给定的流水线结构。

分层系统如 Frenetic 家族 (Frenetic[foster2011frenetic], Pyretic[reich2013modular]) 提供了一个两层编程模型，其中控制器程序指定感兴趣的事件，然后在这些事件发生时

通过计算新的网络策略来响应。同样，这样的系统无法验证给定控制器程序的输出是否可以写入交换机的流水线。

流水线定制语言：流水线定制语言 (如 P4[1], PISCES[shahbaz2016pisces], Concurrent NetCore[schlesinger2014concurrent]) 和我们的流水线容量定理之间有一些相似之处，同样对流水线行为进行了分析以及保证。例如，Concurrent NetCore 的类型系统确保用于部署到流水线的任何程序都具有某些属性[schlesinger2014concurrent]，而 PISCES 的交换机规范允许编译器分析流水线并优化其性能[shahbaz2016pisces]。然而保证流水线属性或提高性能与验证编译是否可行是不同的。

流水线设计：流水线设计方案，如 Jose 等人[Jose-et-al] 考虑的将数据包程序编译到可重构交换机、Sun 等人[Sun-et-al] 的软件定义流表流水线、FlowAdapter[pan2013flowadapter] 和 Domino[sivaraman2016packet] 显然与我们的流水线容量定理相关，因为它们检查硬件约束下的流水线布局设计。然而，Jose 等人，Sun 等人，和 FlowAdapter 着重于将逻辑查找表或多流表流水线映射到物理表，而我们的流水线容量定理侧重于通用程序，而 Domino 考虑的是较弱的硬件约束条件 (如有状态操作)。

2.3　模型

我们首先给出高级 SDN 程序和低级数据通路的模型。由于 SDN 的主要关注点是路由，因此我们这里将高级 SDN 程序称为路由函数。由于多流表流水线是 SDN 数据通路的最先进技术，因此我们这里将流水线作为数据通路对象进行研究。

2.3.1　路由函数模型

路由函数：我们将一个路由函数表示为 f。我们考虑它是一个逻辑上集中的、确定的函数，并用高级语言编写。SDN 控制器对进入该控制器网络的每个数据包逻辑上执行该函数[maple]，以计算出该数据包的全网范围的路由。

每个 f 在数据包上的执行要读取数据包的一组属性（称为匹配字段）$\mathcal{M} = \langle m_1, ..., m_n \rangle$ (例如, `<srcIP, dstIP, ...>`)。我们用 M 表示数据包匹配字段在 $\mathcal{M}$ 中的一个子集。此外，我们用 $dom(M)$ 来标识一组匹配字段 M 的域。f 的执行会从一组（对数据包的）有效操作 $\mathcal{R}$ 返回一个路由操作，(例如, `Drop`, `Forward(port=2)`):

$$f : dom(\mathcal{M}) \rightarrow \mathcal{R}.$$

这样的函数所在的空间标识为 $\mathcal{F}$。

示例：我们用如下的路由函数 onPkt 来演示我们路由函数模型的一些关键特性。

```
\\  Routing function: onPkt
    Map hostTbl[key: dstIP, value: switch]
    Map condTbl[key: (dstIP, port), value: cond]
    Map routeTbl[key: (switch, cond), value: outPort]
L1: def onPkt(Type ethType, Addr srcIP, Port srcPort, \
          Addr dstIP, Port dstPort):
L2: if (ethType != IPv4):
L3:   return Drop()
L4: if (verify(srcPort, srcIP)):
L5:   dstCond = condTbl[dstIP, dstPort]
L6:   dstSw = hostTbl[dstIP]
L7:   return Forward(port = routeTbl[dstCond, dstSw])
L8: return Drop()
```

具体来说，onPkt 读取匹配字段 $\mathcal{M}$ = ⟨ethType,srcIP,srcPort,dstIP,dstPort⟩ 并将 $\mathcal{M}$ 域中的每个值映射到 $\mathcal{R}$ = {Drop(),Forward(port=x)} 中的路由操作。虽然我们将 onPkt 以命令式函数写出，需要强调的是我们的模型是完全通用的，并且没有指定编程范式。

具体地说，onPkt 的前三行声明了键值表：hostTable 和 condTable。它们分别把每个 IP 地址和一个连接交换机以及主机条件 (例如, 身份验证状态) 进行关联，而 routeTable 将（交换机、条件）映射到其转发端口。在 onPkt 的主体部分上，L2 和 L3 检测并丢弃非 IPv4 流量，而 L8 则丢弃来自未验证的终端的流量。如果验证通过，L5 到 L7 设置 dstCond 和 dstSw 变量，然后基于这两个变量从 routeTbl 返回路由操作。

路由函数数据流图：由于一般的路由函数可以具有任意的、复杂的控制结构，我们将路由函数转换为数据流图（Dataflow Graph，DFG）以更好地表示其结构。我们把 f 的 DFG 表示为 G_f。

具体来说，要计算 f 的 G_f，我们必须删除 f 中所有的控制流依赖。这些依赖通过以下转换被删除：

- 通过将 f 转换为静态单赋值形式（Static Single-Assignment，SSA）来移除赋值语句顺序依赖；

- 通过将分支的条件值赋给 guard 变量来移除分支，并将对这些 guard 的依赖性附加到 `if` 和 `else` 块中的所有语句；
- 通过将程序循环转换为黑盒函数来删除它们，黑盒函数的输入为循环中读取的所有变量，输出为循环中写入的所有变量。

例如，我们的示例路由函数 `onPkt` 转换如下：

```
L1: def onPkt(...):
L2: g0 = (ethType != IPv4)
L3: if  g0: return Drop()
L4: g1 = verify(srcPort, srcIP)):
L5: if  g1: dstCond = condTbl[dstIP, dstPort]
L6: if  g1: dstSw = hostTbl[dstIP]
L7: if  g1: return Forward(port = routeTbl[...])
L8: if !g1: return Drop()
```

注意 `onPkt` 的 `L2` 处的 `if` 语句已经从条件表达式变成了 guard 变量 `g0`，`g0` 又被附加给了 `L3`，这一行之前是在 `if` 块内的。

在给出了这个转换后，我们定义 f 的 G_f:

定义 2.1： 一个路由函数 f 的数据流图 DFG $G_f = (V_f, E_f)$ 是一个从转换的 f 生成的点带权的有向无环图，其中:

- V_f 中的每个点 v_f 是 f 的一个变量；
- 一个 v_f 的权重是它的域值范围大小；
- 当两个变量之间存在一条 E_f 中的有向边时，意味着源变量出现在目的变量的赋值声明中。

作为示例，我们给出 `onPkt` 的 DFG 如下:

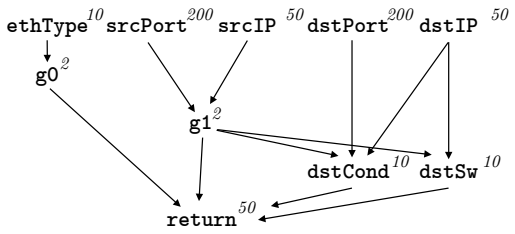


图 2.2　路由函数 `onPkt` 的 DFG $G_{\texttt{onPkt}}$

可以看到，点 `dstSw` 是由其赋值声明中的两个变量 `g1` 和 `dstIP` 派生得到的。该点的权重 10 表示 `dstSw` 的域值范围大小。

2.3.2 流水线模型

我们考虑目前最新的数据通路设计：多流表流水线架构。我们首先对一个流水线 pl 中的一个表 t 进行建模，然后给出对于流水线的清晰定义。

流水线中的表：每个流水线中的表 $t \in pl$ 是精准匹配的 match-action 表。t 的匹配操作可以是：（1）一个路由操作；（2）对 t 的输出寄存器 $r(t)$ 的写操作以及对 pl 中的后续表的跳操作；（3）对 pl 中的后续表的简单跳操作。并不是所有的 t 都输出路由操作。我们把输出路由操作的 t 称为出口表。

每个 t 匹配着一组输入 $I(t)$。输入 $I(t)$ 包含数据包的匹配字段 $m_i \in \mathcal{M}$ 和前序表的输出寄存器 $r(t)$。t 的关键限制是它可以包含的最多规则数量和 $r(t)$ 的比特位长度，在这里我们对应地表示为 $maxrules(t)$ 和 $bits(r(t))$

流水线：一个流水线 pl 是关于表集合 $\{t_i\}$ 的单根有向无环图。pl 中的一条边 (t_i, t_j) 表示到达 t_i 的数据包可以跳转到 t_j。

每个通过 pl 的数据包从 pl 的根节点出发，然后在 pl 中被处理，最终传到出口表。因此，每个包在 pl 中传输的过程可以映射到 pl 中的一条路径，以及对该数据包进行的路由操作（属于 $\mathcal{R}$）。

一个数据包在 pl 中通过的路径和它的出口表输出的操作由这个数据包的匹配字段组 $\mathcal{M}$ 在每个 $t_i \in pl$ 中的匹配情况决定。基于此，pl 也可以概括为一个从 $dom(\mathcal{M})$ 到 $\mathcal{R}$ 的映射，这个映射依赖于 pl 的内容，即流表。

我们把所有流水线 pl 所在的空间标识为 P。

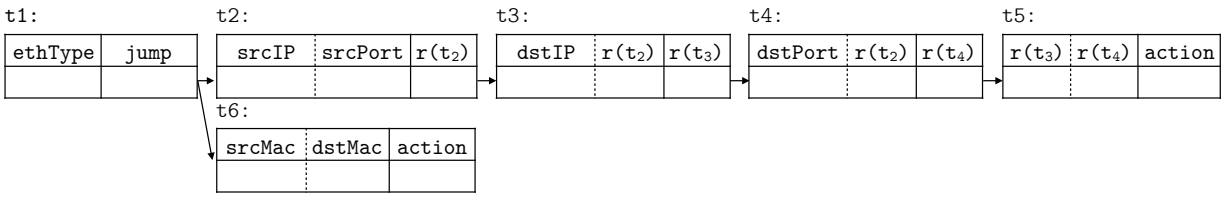


图 2.3 示例数据通路, ExampleDP

示例：我们现在给一个示例流水线 ExampleDP（如图 2.3所示），来说明我们的流水线模型。要注意的是在这个例子中，一个流表的匹配列在它的左边，然后写入到它右边的寄存器中，而表中字段 output 表示这个表包含了输出路由操作。

考虑流表 t2∈ExampleDP.t2 是个精准匹配表，它的输入 I(t2) 是 srcIP 和 srcPort，它的输出寄存器是 r(t2)。

关于 t2 的关键计算限制是它的最大规则数 $maxrules$(t2)) 和他的输出寄存器的大小 $bits$(r(t2))。

符号	定义
路由函数相关符号	
f	路由函数
$\mathcal{F}$	路由函数空间
m_i	数据包的匹配字段
$\mathcal{M}$	$\forall\, m_i$ 的集合
$dom(M)$	M 中有效值的域
$\mathcal{R}$	$\forall$ 有效路由行为的集合
流水线相关符号	
pl	流水线
$\mathcal{P}$	流水线函数空间
t_i	流水线中的表
$r(t_i)$	t_i 的输出寄存器
$bits(r(t_i))$	t_i 的输出寄存器 bit 位长度
$I(t_i)$	t_i 的输入
$maxrules(t_i)$	t_i 可以包含的最大 # 规则数

表 2.1　在我们 ‘‘主要结果以及相关证明" 一节中使用的符号标识列表。

2.4　主要结果以及相关证明

基于函数模型和流水线模型，我们现在给出关于函数 f 是否可以由流水线 pl 实现的主要结果。然后我们再给出结果的相关证明。关键的一些符号在表**??**中写出以供参考。

2.4.1　概述

本节的目标是得到一种系统方法来验证路由函数 f 是否可以由流水线 pl（我们称之为 $f \Longrightarrow pl$）实现。存在的一个主要挑战是路由函数和流水线的表示方式是不同的，并且这两种表示方式都具有巨大的复杂性和多样性。可以想象将每个路由函数 f 看作函数空间 $\mathcal{F}$ 中的一个点，将每个流水线 pl 看作函数空间 $\mathcal{P}$ 中的一个点。

我们的主要贡献是引入了一个创新的、统一的、规范化的函数空间 C，我们称为特征函数空间。每个路由函数 f 通过将 τ 映射到特征函数 $\tau(f) \in C$ 来表示 f 的计算负载。另一方面，每个流水线 pl，则被映射到一组特征函数 $\kappa(pl) \subset C$，表示这个流水线的一组计算能力。图 2.4给出了这种映射结构。

鉴于 $\tau(f)$ 和 $\kappa(pl)$ 都定义在了同一空间 C（其中 $\tau(f)$ 对应点，$\kappa(pl)$ 对应一组点），人们就可以通过比较 $\kappa(pl)$ 中的每个元素与 $\tau(f)$，从而判断这个计算负载是否可以被一个计算能力 ‘‘覆盖"，进而导出了我们的基本容量理论，即 $\exists\, c \in \kappa(pl) \geq \tau(f), f \Longrightarrow pl$。

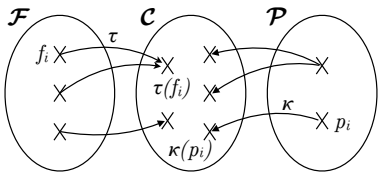


图 2.4　空间 $\mathcal{F}, \mathcal{P}$ 和 $\mathcal{C}$ 及其它们之间的映射表示。

2.4.2　特征函数

我们首先来定义一个通用的特征函数 c。

定义 2.2： 一个特征函数 c 是一个从数据包的匹配字段的子集 M 到一个向量的映射：

$$c(M) \triangleq < scope(M),\ ec(M) > .$$

我们将 $c(M)$ 的向量的两个部分对应地称为 $c(M)[scope]$ 和 $c(M)[ec]$。

给定两个特征函数，我们可以比较它们。

定义 2.3： 我们定义 c_i 支配 c_j, 标识为 $c_i \geq c_j$ 如下:

$$c_i \geq c_j \triangleq \ \forall\, n \in \{scope, ec\},$$
$$\forall\, M \in 2^{\mathcal{M}},\ c_i(M)[n] \geq c_j(M)[n].$$

为了确保我们的容量理论，我们需要将一组特征方程和一个单独的特征方程作比较。

定义 2.4： 一组特征方程 C_i 支配一个特征方程 c_j, 标示为 $C_i \trianglerighteq c_j$，如果存在一个 $c_i \in C_i$ 支配 c_j

$$C_i \trianglerighteq c_j \triangleq \exists\, c_i \in C_i : c_i \geq c_j.$$

2.4.3　路由函数的特征

给定特征函数的概念，我们现在导出路由 f 的特征函数，表示为 $\tau(f)$。

定义 2.5： 关于数据包匹配字段 M 的一个路由函数的特征方程的范围（scope）定义为 M 中有效值域的大小:

$$\tau(f)(M)[scope] \triangleq dom(M)$$

$\tau(f)(M)[ec]$ 是我们根据 “f 等价" 概念建立的属性:

定义 2.6： 我们定义 f 等价（标识为 $\sim_f$），为 M 中两个值 $v_i(M)$ 和 $v_j(M)$ 之间的一种联系，即不能被 f 所区分:

$$v_i(M) \sim_f v_j(M) \triangleq \forall\, v_k(\mathcal{M} - M) \in dom(\mathcal{M} - M),$$
$$f(v_i(M), v_k(\mathcal{M} - M)) = f(v_j(M), v_k(\mathcal{M} - M)).$$

我们关于 f 等价的定义可以自然地导出 f 等价类的定义。

定义 2.7： 一个 f 等价类（标识为 $[v_i(M)]_f$），是一个给定的 M 中的变量 $v_i(M)$ 的所有和它 f 等价的变量的组:

$$[v_i(M)]_f \triangleq \{v_j(M) \in dom(M) : v_i(M) \sim_f v_j(M)\}.$$

计算等价类的数量可以得出 f 等价类数的概念。

定义 2.8： M 中 f 等价类的数量（标识为 $|dom(M)/\sim_f|$），是 M 的一组等价类的基数。

现在来看我们关于 $\tau(f)(M)[ec]$ 的定义。

定义 2.9： M 的路由函数特征函数的 ec（equivalent class）是 M 的 f 等价类集合的基数。

$$\tau(f)(M)[ec] \triangleq |dom(M)/\sim_f|$$

定义 2.10： 路由函数的特征函数 $\tau(f)$ 表示 f 的计算负载:

$$\tau(f)(M) \triangleq (dom(M),\ |dom(M)/\sim_f|).$$

虽然 $\tau(f)$ 非常强大，但它是不切实际的，因为直接计算等价类数需要非常大的计算。因此，我们通过定义路由函数 $\tau_G(f)$ 的边界特征函数来限制 $\tau(f)$，该函数很容易从 f 的 DFG 导出。该函数刻画了 f 的计算负载的一个上界：$\tau_G(f)$ 支配 $\tau(f)$。

$\tau_G(f)[scope]$ 的定义如之前所述。然而，我们不计算 $\tau_G(f)[ec]$，而是用 G_f（即 f 的 DFG）中特定的顶点割值来确定其上限。我们现在开始说明这个割值。

定义 2.11： 设 $V_f(M)$ 是 G_f 中的点 $m_i \in M$，$D_f(M)$ 是从 $V_f(M)$ 派生的 G_f 中的点。M 的顶点最小割集，$G_f.vertexMinCut(M)$，是将 $V_f(M)$ 从 $D_f(\mathcal{M} - M)$ 中分离出来的最小割集中顶点的权重的乘积。

给定这个割集，我们定义 $\tau_G(f)$ 如下:

定义 2.12： 一个路由函数的特征函数 $\tau_G(f)$ 刻画 f 的计算负载上限; $\tau_G(f)$ 支配 $\tau(f)$:

$$\tau_G(f)(M) \triangleq (dom(M),\ G_f.vertexMinCut(M)).$$

示例：现在用我们的示例路由函数 `onPkt` 来说明这些概念。

考虑 `onPkt` 的匹配字段 `srcIP` 和 `srcPort`。每个都只被读取一次：在 L4 上，由布尔函数 `isVerified` 读取。因此，虽然 `srcIP` 和 `srcPort` 可能分别有许多 f 等价类，(`srcIP`, `srcPort`) 只有两个：`isVerified` 的值计算为 0 和它的值计算为 1。

假设 `onPkt` 是一个小型商业网络的路由函数，其外层是一个连接有 50 个主机的 NAT，每个主机运行着一些受限制的应用，这个限制是指只能使用 200 个标准端口。给定这个条件下，dom(`srcIP`, `srcPort`) = 10000，以及 τ(`onPkt`)(`srcIP`, `srcPort`) = (10000, 2)。

虽然 (`srcIP`, `srcPort`) 的等价类数是直接的，但 `onPkt` 输入的大多数其他子集的等价类数并不明显。因此，我们将 τ(`onPkt`) 用 τ_G(`onPkt`) 约束，它的计算使用 `onPkt` 的 DFG（$G_{\texttt{onPkt}}$）。如图 2.5所示。

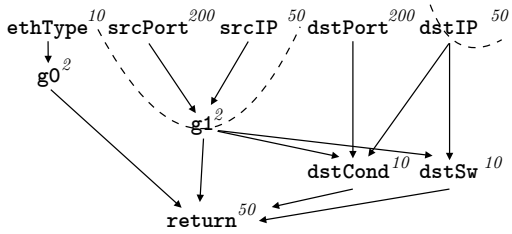


图 2.5 路由函数 `onPkt` 的 DFG $G_{\texttt{onPkt}}$ 和割集 (`srcIP`, `srcPort`, `dstPort`)

我们考虑 `onPkt` 的输入为 (`srcIP`, `srcPort`, `dstIP`) 的等价类数量。我们取它们的点和 `onPkt` 的其他输入: (`ethType`, `dstPort`, `g0`, `dstCond`, `return`) 派生的每个点之间的 $G_{\texttt{onPkt}}$ 中的顶点最小割集。图 2.5中用虚线表示了这个顶点最小割集。

该割集中的顶点 (`g1`, `dstIP`) 的权重为 50 和 2，因此 τ_G(`srcIP`, `srcPort`, `dstIP`) = (50000, 100)。

2.4.4 流水线的特性

我们现在定义流水线 pl 的特征函数集 $\kappa(pl)$。我们从定义通过流水线 pl 的路径 ρ 开始。

定义 2.13： 一个路径 ρ, 在 pl 中是一个通过 pl 的有向无环图的路径 $\langle t_1, ..., t_n \rangle$，其中 t_1 是 pl 的根表，t_n 是 pl 的出口表。

作为示例，ExampleDP 有两个路径 ⟨t1,t2,t3,t4,t5⟩ 和 ⟨t1,t6⟩，我们用 ρ_{L2} 和 ρ_{L3} 分别表示。

我们定义，$\forall\, \rho \in pl$, $\kappa_\rho(pl)$ 是一个流水线中一条路径的特征方程。

定义 2.14： pl 的特征方程组 $\kappa(pl)$ 是 $\forall\, \rho \in pl$ 的集合:

$$\kappa(pl)(M) \triangleq \{c \in C : c = \kappa_\rho(\rho)\ \forall\, \rho \in pl\}.$$

我们现在通过引入以下定义来构造路径 ρ 的特征函数：

定义 2.15： 表 $t_i \in \rho$ 的输入闭包 $\bar{M}_\rho(t_i)$ 是 t_i 可以获得信息的输入集合：

$$\bar{M}_\rho(t_i) \triangleq \{m_i \in \mathcal{M} : m_i \in I(t_i) \vee m_i \in \bar{M}_\rho(t_j)\ s.t.\ r(t_j) \in I(t_i)\}.$$

定义 2.16： ρ 的 M 的闭包 $\bar{\bar{M}}_\rho(M)$ 是由 $t_i \in \rho$ 与输入闭包 M 组成的。

$$\bar{\bar{M}}_\rho(M) \triangleq \{t_i \in \rho : \bar{M}_\rho(t_i) = M\}.$$

通过这些定义，我们现在定义 ρ 的特征方程如下：

定义 2.17： ρ 的特征函数 $\kappa_\rho(\rho)$ 表征了 ρ 的计算容量。

$\kappa_\rho(\rho)[scope]$ 是 ρ 可以读取的 M 的最大数量，$\kappa_\rho(\rho)[ec]$ 是 ρ 可以分辨的 M 的等价类的最大数量。

$$\kappa_\rho(\rho)(M) \triangleq \begin{cases} \bar{\bar{M}}_\rho(M) \neq \emptyset & (min[maxrules(t_i) : t_i \in \bar{\bar{M}}_\rho(M)], \\ & \quad min[2^{bits(r(t_i))} : t_i \in \bar{\bar{M}}_\rho(M)]) \\ \bar{\bar{M}}_\rho(M) = \emptyset \wedge \exists\, m_i \in M : & \\ m_i \notin \bigcup_{t_i \in \rho} \bar{M}(t_i, \rho) & (1,1) \\ otherwise, & (\mathrm{T},\mathrm{T}). \end{cases}$$

示例：与之前一样，我们使用示例流水线 ExampleDP 直观地说明流水线的特征函数。

ExampleDP 包含两个 ρ: ρ_{L2} 和 ρ_{L3}。考虑表 t4，仅包含在 ρ_{L3} 中。由于 t4 读取 dstIP 和 r(t2)，而 t2 依次读取 srcIP 和 srcPort，因此输入闭包 $\bar{M}_{\rho_{\mathrm{L3}}}$(t4) 为 (srcIP, srcPort, dstIP)。闭包 $\bar{\bar{M}}_{\rho_{\mathrm{L3}}}$(srcIP, srcPort, dstIP) 关于 t4 在 ρ_{L3} 的输入是 {t4}。

因此，$\kappa_{\rho_{\text{L3}}}(\bar{M}_{\rho_{\text{L3}}}) = \kappa_{\rho_{\text{L3}}}(\texttt{srcIP}, \texttt{srcPort}, \texttt{dstIP}) = (maxRules(\texttt{t4}), 2^{bits(r(\texttt{t4}))})$. 在 t4 有 2^{20} 规则数并且一个 16bit 输出寄存器这种情况下，$\kappa_{\rho_{\text{L3}}}(\bar{M}_{\rho_{\text{L3}}}) = (2^{20}, 2^{16})$.

进一步，我们考虑 ExampleDP 的匹配域的子集 (srcMac, dstMac)。ρ_{L3} 并不包含 srcMac 以及 dstMac，因此它只可以实现不包含 srcMac 以及 dstMac 的函数（或赋值为常数）。常数只有一个值，一个等价类。因此，对于任何输出包含 srcMac，$\kappa_{\rho_{\text{L3}}}$ 的值为 (1,1)。

最后，我们考虑 ExampleDP 的匹配域的子集 (srcIP, srcPort)。srcIP 和 srcPort 都被 ρ_{L3} 所读，但 (srcIP, srcPort) 并不是任何 $t_i \in \rho_{\text{L3}}$ 的输入闭包。在这种情况下，在判断数据通路的实现问题上不需要考虑 (srcIP, srcPort)，因此，$\kappa_{\rho_{\text{L3}}}$(srcIP, srcPort) = (T,T)，表示我们在和路由函数的 τ 比较时可以跳过。

2.4.5 数据通路编程容量理论

结合前面关于路由函数和流水线以及它们特征函数的定义，我们最终得到了我们的核心结论：给定的 f 是否可以在给定的 pl 中实现的一个充分条件。

定理 2.1 (流水线实现定理)： 当 $\kappa(pl)$（即 pl 的特征函数集）支配 $\tau(f)$（即 f 的特征函数）时，一个路由函数 f 可以在流水线 pl 上实现。我们有:

$$\kappa(pl) \trianglerighteq \tau(f) \Rightarrow f \Longrightarrow pl.$$

作为推论，因为 $\tau_G(f) > \tau(f)$，流水线实现定理可以推广到 $\tau_G(f)$。

示例： 我们使用 onPkt 和 ExampleDP 来说明我们的流水线实现定理。具体来说，我们的流水线实现定理指出 κ(ExampleDP) $\trianglerighteq$ τ(onPkt) $\Rightarrow$ ExampleDP$\Longrightarrow$onPkt。

进一步，当 $\kappa_\rho(\rho_{\text{L2}}) > \tau_G$(onPkt) 或者 $\kappa_\rho(\rho_{\text{L3}}) > \tau_G$(onPkt) 时，$\kappa$(ExampleDP) $\trianglerighteq$ τ(onPkt) 为真。我们通过比较每对特征函数给出的每个向量的每个分量来验证每个条件。举例来说，τ(onPkt)(srcIP, srcPort, dstIP) = (50000, 100), $\kappa_\rho(\rho_{\text{L3}})$(srcIP, srcPort, dstIP) = $(2^{20}, 2^{16})$,因此输入集 (srcIP, srcPort, dstIP) 不会阻碍 onPkt 在 ρ_{L3} 上实现。

严密性： 虽然定理只提供了一个充分条件，但在一些情况下可以建立更严密的结果，即充分必要的条件。具体来说，我们有以下结果：

定义 2.18： 当 pl 的 DAG 没有分支时，我们称 pl 为无分支流水线。

定理 2.2 (无分支流水线实现定理)： 如果 pl 是无分支流水线，且 pl 的表容量足够大，并且每个匹配字段 $m_i \in \mathcal{M}$ 正好出现在 pl 的每一张表中，$\kappa(pl) \trianglerighteq \tau_G(f) \Leftrightarrow f \Longrightarrow pl$.

2.4.6　证明

2.4.6.1　流水线实现定理的证明

我们现在提出流水线实现定理的证明。这些证明的结构如下。首先，我们建立一种编码机制对 $m \in \mathcal{M}$ 进行编码，并给予的足够信息，以完全执行给定的 f。其次，在 $\kappa(pl) \trianglerighteq \tau_G(f)$ 前提下，证明使用我们的编码的流水线可以在 pl 中实现 f。最后，我们通过一个扩展：如果 $\kappa(pl) \trianglerighteq \tau(f), f \Longrightarrow pl$ 来证明 $\tau_G(f) > \tau(f)$。

我们将基于 f 的 G_f 的 $G_f.vertexMinCut(M)$ 中的点。

定义 2.19： 最小割集 $\mu_f(M)$ 是 f 的 G_f 用 $G_f.vertexMinCut(M)$ 分割得到的点。

给定 $\mu_f(M)$ 的值为 $v_i(\mu_f(M))$ 且 $\mu_f(M)$ 的值的域为 $dom(\mu_f(M))$。

引理 2.1： 给定 $G_f.vertexMinCut(M)$，我们可以在不知道给定的值 $v_i(\mu_f(M))$ 的情况下计算 f。

虽然 $\mu_f(M)$ 可以有效表示 M 的值，$v_i(M)$，但我们可以通过引入码字（codeword）的概念来对它进行压缩，从而使通过流水线的传输最大化。

定义 2.20： f 的输入 M 的码字 $\chi_f(M)$ 是一组整数，对应于 M 的 f 等价类。

得到一个码字 $\in \chi_f(M)$ 等同于得到 M $v_i(M)$ 的值，因为可以确定地将码字映射回 $v_i(M)$ 的等价类中的值。我们现在定义什么是 ‘‘计算 M 的码字"，我们将在证明中使用它：

定义 2.21： 如果我们可以计算 M 的码字 $\chi_f(M), \forall\, v_i(M) \in dom(M)$ 我们可以计算与 $v_i(M)$ 的等价类相关联的码字。

在引理 2.2中，我们的码字给了我们一个 M 的传输要求的边界。

引理 2.2： 一张流表 t_i 只需要关于 M 的 $log_2(\lceil |dom(\mu_f(M))| - 1 \rceil)$ 比特的信息就可以正确执行 f。

证明 我们可以将任意 $v_i(M) \in dom(M)$ 编码为 $\chi_f(M)$ 中的码字，并且仍然可以传递足够的信息来计算 f。如果 $\mu_f(M)$ 可以取 $|dom(\mu_f(M))|$ 不同的值，我们可以从集合：0, ..., $|dom(\mu_f(M))| - 1$ 中为每个值分配一个唯一的码字，最多使用 $log_2(\lceil |dom(\mu_f(M))| - 1 \rceil)$ 位来表示。 □

实现定理的证明：基于我们对函数传输需求的描述，我们现在可以开始证明我们的实现定理。首先，我们将给出我们的关键潜在引理，引理 **??**，随后可以得到我们的实现定理。

引理 2.3： 如果 $\forall\ t_i \in \rho = \langle t_1, ..., t_n \rangle$ 有 $maxRules(t_i) > \tau_G(f)(\bar{M}_\rho(t_i))[dom]$，且 $2^{r(t_i)} > \tau_G(f)(\bar{M}_\rho(t_i))[ec]$，那么 $\forall t_i \in \rho = \langle t_1, ..., t_n \rangle$ 可以输出 $\chi_f(\bar{M}_\rho(t_i))$ 到 $r(t_i)$。

给出引理 2.3后, 我们现在可以证明实现定理了。

证明 给定 f 和 p，我们会证明如果 $\kappa(pl) \trianglerighteq tau(f), f \Longrightarrow pl$。考虑 $\kappa_\rho(\rho) \in \kappa(pl)$。

$\forall\ M \in \mathcal{M} : m_i \in M \rightarrow m_i \notin \bigcup_{t_i \in \rho} \bar{M}_\rho(t_i),\ \kappa_\rho(\rho)(M) = (1,\ 1)$。因此，如果 $\kappa_\rho(\rho) > \tau_G(f) \Rightarrow$ 没有被 ρ 读取的所有 m_i 会被视作常量或者根本不会被 f 读取，因此 f 是从 $\bigcup_{t_i \in \rho} \bar{M}_\rho(t_i) \rightarrow \mathcal{R}$ 得到的映射。

进一步，给定 $\kappa_\rho(\rho) > \tau_G(f)\ \forall\ t_i \in \rho,\ maxRules(t_i) > \tau_G(f)(\bar{M}_\rho(t_i))[dom]$，且 $2^{r(t_i)} > \tau_G(f)(\bar{M}_\rho(t_i))[ec]$, 那么因此根据引理 2.3 t_n 可以计算 $\chi_f(\bar{M}_\rho(t_n))$。

最后，考虑到如果一个 t_i 可以计算 $\chi_f(M_i)$，并且一个 f 是 $dom(M_i) \rightarrow \mathcal{R}$ 的映射, t_i 可以通过把 $\chi_f(M_i)$ 中的码字映射到它关联的 f 的输出的方式，计算 f 的输出 $\forall\ v_j(M_i) \in dom(M_i)$。

鉴于 t_n 是 ρ 的唯一输出, $\bar{M}_\rho(t_n) = \bigcup_{t_i \in \rho} \bar{M}_\rho(t_i)$。因此，$t_n$ 可以计算 f 的输出进一步，鉴于 t_n 是一个出口表，它总是可以将此输出传递回交换机。

因此，如果 $\kappa_\rho(\rho) > \tau_G(f), f \Longrightarrow \rho$。鉴于 $\kappa_\rho(\rho) \in k(p)$ 和 $\rho \in pl$, 我们已经证明如果 $\kappa(pl) \trianglerighteq \tau_G(f), f \Longrightarrow pl$。 □

证明我们的实现定理所需的最后一步是证明 $\tau_G(f) > \tau(f)$ 所以才有 $\kappa(pl) \trianglerighteq \tau(f) \Rightarrow f \Longrightarrow pl$。下面引理 2.4给出了这一步骤的要点。

引理 2.4： M 等价类的数量被 $dom(\mu_f(M))$ 所限定。

证明 通过反证法，假设 $\exists (f,\ M) : |dom(M)/\sim_f| > dom(\mu_f(M)$。$v_i(M)$ 在 M 中的某个等价类的每个 $v_i(M)$ 必须生成一个 $v_i(\mu_f(M))$。根据鸽子洞原理**ajtai1988complexity**，如果 M 的等价类多于 $\mu_f(M)$，则来自不同等价类的 M 的两个值必须生成相同的 $\mu_f(M)$。然而，根据引理 2.1，$\mu_f(M)$ 包含关于 M 的足够信息来确定 f 的输出值，因此 M 的这两个值必须在同一等价类中，这是一个矛盾。 □

推论 1： 任一 M 的 f 等价类的数量 $G_f.vertexMinCut(M)$ 所限。

推论 2： 特征方程 $\tau_G(f)$ 支配特征方程 $\tau(f)$。

因此，我们证明了我们的实现定理：$\kappa(pl) \trianglerighteq \tau_G(f) \Rightarrow f \Longrightarrow pl$。

2.4.6.2　无分支流水线实现定理的证明

上一小节中已经证明了我们的实现定理，$\kappa(pl) \trianglerighteq \tau_G(f) \Rightarrow f \Longrightarrow pl$，现在我们证明如果 pl 是无分支流水线，且 pl 的表容量足够大，并且每个匹配字段 $m_i \in \mathcal{M}$ 正好出现在 pl 的每一个表中，$\kappa(pl) \trianglerighteq \tau_G(f) \Leftrightarrow f \Longrightarrow pl$.

通过反证法，我们假设 $f \Longrightarrow pl$ 但存在 M 使得 $\kappa(pl)(M)[ec] < \tau_G(f)(M)[ec]$（我们省略 M 的域大小因为 pl 的表足够大）。因为 $f \Longrightarrow pl$ 且 pl 是一个无分支流水线且每一个匹配字段 $m_i \in \mathcal{M}$ 都正好出现在一个 pl 的流表中，因此我们有 $\kappa(pl)(M)[ec] \geq \prod \kappa(pl)(m_i)[ec]$（其中 $m_i \in M$）。因此我们有 $\kappa(pl)(M)[ec] \geq \tau_G(f)(M)[ec]$，而这与 $\kappa(pl)(M)[ec] < \tau_G(f)(M)[ec]$ 有矛盾。完成证明。

2.5　实验评估

我们现在对路由函数特征化的严密性（即计算出的值与真实值的差异性），路由函数特征化的时间复杂度，流水线特征化，以及流水线实现四个方面进行数值评估。所有的实验在 3.5 GHz Intel i7 处理器上运行，内存为 16 GB，系统为 Mac OSX 10.13。

2.5.1　路由函数特征化的严密性

通过比较以下一组路由函数的 τ 和 τ_G 的一个 M 等价类的数量（即 $ec(M)$），我们证明路由函数特征化的严密性。对于 τ 的 $ec(M)$ 计算将采用 f 等价类的定义进行计算。

```
\\  Routing function: simpleRoute
L1: def simpleRoute(Addr srcIP, Addr dstIP):
L2:   srcSw = hostTbl[srcIP]
L3:   dstSw = hostTbl[dstIP]
L4:   route = routeTbl[srcSw, dstSw]
L5:   return route
```

我们的第一个函数 `simpleRoute` 将包的 `srcIP` 和 `dstIP` 映射到它们的主机包交换机 `dstSw` 和 `srcSw`，然后查找它们之间的路由。

```
//  Routing Function: condRoute
L1: def condRoute(srcIP, dstIP):
L2:   srcSw = hostTbl[srcIP]
L3:   dstSw = hostTbl[dstIP]
L4:   routeCond = condTbl[srcIP, dstIP]
L5:   route = routeTbl[srcSw, dstSw, routeCond]
L6:   return route
```

f	$bits(IP)$	O(hostTbl)	O(routeTbl)	O(condTbl)	O(fwdTbl)	τ(f)(srcIP)	τ(f)(dstIP)	τ_G(f)(srcIP)	τ_G(f)(dstIP)
smplR	10	100	2	N/A	N/A	100	100	100	100
smplR	10	100	30	N/A	N/A	100	100	100	100
smplR	10	100	5000	N/A	N/A	100	100	100	100
smplR	12	100	30	N/A	N/A	100	100	100	100
smplR	10	200	30	N/A	N/A	200	200	200	200
condR	10	100	30	50	N/A	1024	1024	1024	1024
condR	10	100	30	5	N/A	1024	1024	1024	1024
condR	10	100	30	1	N/A	100	100	1024	1024
scR	10	N/A	N/A	N/A	100	2	100	2	1024
b1(scR)	10	N/A	N/A	N/A	N/A	1	N/A	1	N/A
b2(scR)	10	N/A	N/A	N/A	100	1	100	1	100
onPkt	32	100	30	50	N/A	null	null	2^{32}	2^{32}

表 2.2　不同统计量下路由函数的特征化结果

我们的第二个函数 condRoute 通过引入一个路由条件变量来扩展 simpleRoute，该变量影响路由查找。

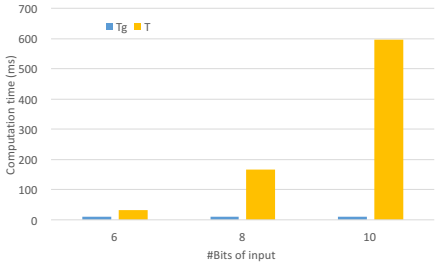
```
//  Routing Function: secureRoute
L1: def secureRoute(Addr srcIP, Addr dstIP):
L2:   if (isFiltered(srcIP)):
L3:     return Drop()
L4:   else:
L5:     route = fwdTbl[dstIP]
L6:     return route
```

我们的最后一个函数 secureRoute 将所有 srcIP 在一个过滤器列表中的数据包丢弃，并转发其余的数据包。

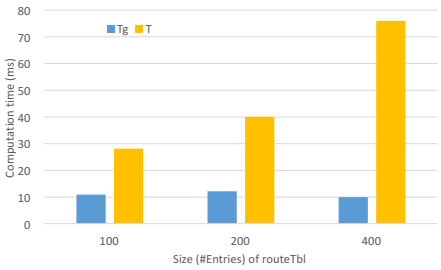
结果：结果见表 2.2。具体来说，在表 2.2中，第 2 列定义了 srcIP、dstIP 的域，第 3-6 列给出了每个表的输出范围 $O(tbls)$，第 7-10 列给出了每个函数的 $\tau(f)$ 和 $\tau_G(f)$ 中选定字段的值。行 b1(scR) 和 b2(scR) 分别表示 secureRoute 的两个分支，L2→L3 和 L2→L4→L5→L6。当一个值不适用于给定函数时，我们记录 N/A，当计算失败时，$\tau(f)$ 的值为 null。

在我们对 simpleRoute 和 condRoute 的计算中，除 $O(\texttt{condTbl}) = 1$ 的特殊情况外，τ 和 τ_G 的结果在任何情况下都几乎相同。值得注意的是，我们的 $\tau(f)$ 和 $\tau_G(f)$ 值不受 *routeTbl* O(routeTbl) 范围的影响。这说明当 hostTbl(s_1) 不等于 hostTbl(s_2) 时（s_1 和 s_2 是 *srcIP* 的两个值），$s_1 \sim_f s_2$ 的概率非常小。

进一步发现，带有控制语句的函数：secureRoute 以及 onPkt，在 τ 和 τ_G 之间有一个很大的差距。这表明 τ_G 的界在有分支程序上是较松的。然而，正如行 b1(scR) 和行 b2(scR) 所示，我们可以对程序的每一个分支计算特征函数来解决这个问题。对于每一个分支（如 b1(scR) 和行 b2(scR) 所示），τ 和 τ_G 仍然保持着严密性。



(a) 输入比特位长度变化



(b) 表大小变化

2.5.2　路由函数特征的时间复杂度

我们现在比较给定路由函数计算 τ 和 τ_G 所需的时间。如上一实验所述，τ 的计算将基于 f 等价类的定义。我们使用 `simpleRoute` 进行测试。其中 $O(\texttt{hostTbl}) = 100$，并改变输入范围（即输入比特位）和 `routeTbl` 的大小。

结果：图 **??**说明了随着输入以及 `routeTbl` 的变大，τ_G 的计算具有良好的可拓展性，即运算时间不改变。其中图 **??**表示的是在固定 `routeTbl` 大小为 100 的情况，而图 **??**表示的是固定输入外围为 6 的情况。从图 **??**可以看出，随着输入比特位的变大，τ 的计算时间呈指数级增长而 τ_G 基本不变。从图 **??**可以看出，随着 `routeTbl` 的变大，τ 的计算时间同样呈指数级增长而 τ_G 基本不变。这是由于 τ_G 的计算只依赖于 DFG 而 τ 需要对每个不同输入执行函数。

2.5.3　流水线的特征

我们现在评估流水线特征的计算时间和压缩性。其中压缩性表示特征函数的输出所占存储大小与 M 的全部可能所占存储大小的比值。我们考虑以下流水线进行评估:

1. **OF-DPA Abstract Switch 2.0：**OpenFlow 抽象数据平面抽象交换机 2.0（OF-DPA）是基于 OpenFlow 1.3.4 协议的抽象交换机模型，并在 OpenFlow 协议下允许对基于 Broadcom 的交换机进行编程。我们考虑了两个 OF-DPA 流

流水线	# 路径数量	时间 (ms)	# 有效 M	#M
ExampleDP	3	8	6	22
PicOS BR	4	13	19	$3 * (2^{24}) + 2^7$
PicOS OT	2	7	5	$2^{24} + 16$
Broadcom IPR	1	7	4	2^7
Broadcom PR	3	9	14	$2 * (2^{24}) + 2^7$

表 2.3　流水线的特征结果。

表配置：（1）桥接和路由（BR）；（2）数据中心覆盖隧道（OT），它们分别有 5 个阶段（7 个流表）和 3 个阶段（3 个表）。**OF-DPA**

2. **PicOS**：PicOS 是一个应用于白盒交换机的网络操作系统。它提供了跨 HP、Edgecore 和 Pica 交换机的可编程性。我们考虑 PicOS 提供的两个固定流水线：（1）IP 路由流水线（IPR）；（2）策略路由流水线（PR），分别包含 4 个和 5 个表对应在 4 和 5 个阶段。**PicOS**

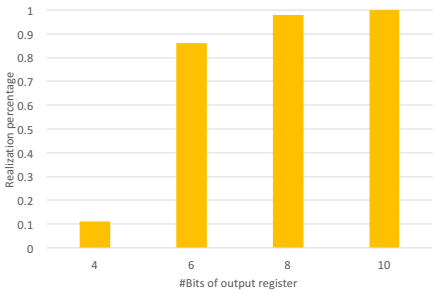
结果：表2.3是我们对评估流水线的特征结果。其中，我们说在一个流水线 pl 中 M 有效，所指的是 $\kappa_\rho(\rho)(M)$ 的值可以通过 $\kappa_\rho(\rho)(M)$ 定义的第一个公式算出。结果表明，尽管在理论上 $\mathcal{M}$ 子集的数量巨大，但实际上有效 M 并不多。同时流水线特征的计算也非常小。

2.5.4　流水线实现的分析

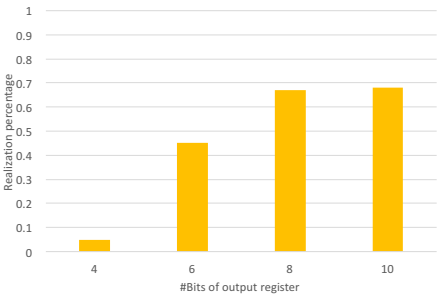
我们现在评估在流水线中成功实现函数的百分比，以查看成功实现的因素。我们考虑具有不同表内容的示例函数 `onPkt`。具体来说，对于 `condTbl` 和 `hostTbl`，我们随机设置表的输出值，范围从 10 到 30。这意味着 `dstCond` 和 `dstSW` 的域大小是从 10 到 30（即平均域大小为 20）。对于流水线方面，我们随机决定表的数量（从 2 到 4）以及每个表的输出寄存器的位长（从 4 到 10）。另外我们设置生成的流水线的匹配字段必须包含 `onPkt` 所需的五个匹配字段，并且每个匹配字段只能出现在一个表中。最后我们通过实现定理计算 `onPkt` 的成功实现百分比和生成的流水线。

结果：如图 2.7所示。其中，图**??**考虑的具有两个表的流水线；图**??**为 3 个表的流水线；图**??**为 4 个表。对于每种情况，我们变化每个表的寄存器位长度，并计算实现成功的百分比。从图 **??**的结果中，我们可以发现具有更多位寄存器的流水线可以实现更高百分比的功能。但是，当长度大于一定阈值时，成功实现百分比不会增加太多。阈值由函数中变量域的大小确定。如图**??**所示，实现成功的百分比在 4 位和 6 位寄存器长度之间的差距（即等效类的可用大小从 16 到 64）可以通过变量的平均域大小（即 20）在 16 到 64 之间的范围来解释。

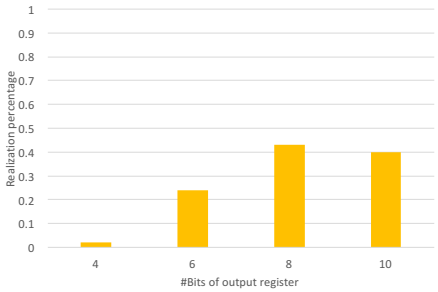
进一步，我们可以发现流水线的结构也决定了实现成功的百分比。具有 2 个



(a) 表数量 = 2



(b) 表数量 = 3



(c) 表数量 = 4

图 2.7　对于不同表数量和寄存器位长度的实现成功百分比。

表的流水线（即没有分支）具有相对较高的成功实现百分比，而具有 3 或 4 个可能包含分支的表的流水线具有较低的实现百分比，因为这些流水线的结构可能并不好（即匹配字段和表之间的随机映射）。

2.6 本章小结

本章我们针对高级 SDN 程序对于固定结构数据通路的实现问题，提出了将高级 SDN 程序和底层数据通路统一的特征空间。并将实现问题转化为空间中的比较问题，并给出流水线实现定理以及相关证明。实验表明，虽然我们的流水线实现定理给出的是充分条件，但其严密性较好，在实际情况中不会太大误差。

第 3 章　loop 章节

3.1　引言

高级 SDN 编程技术的发展在保持了编程模型的灵活性的同时，也降低了数据包处理的延时。主动式地将高级程序编译为数据通路正是一个代表性技术。基于由高级程序生成的数据通路，数据包无需再传给控制器等待处理。保持 SDN 程序的高级编程模型可以提高程序的可读性从而降低程序发生错误的风险。此外，编程模型的高度灵活性允许程序员用简单的语法，例如 `if`, `else`, `while`，描述复杂的逻辑。

然而，由于硬件条件的限制，复杂的逻辑很难部署到数据通路中，特别是在高级编程中非常有用的循环语句，在当前的数据通路体系结构 (即多流表流水线) 中却无法支持。因此，要在数据通路中部署循环部分的逻辑，必须在保持逻辑不变的情况下对它们进行转换。

一个简单的转换是基于黑盒模型。该模型将整个循环视为具有多个输入和输出的单个语句。然而，由黑盒方法生成的单语句结构不能利用数据通路中的多流表流水线。进而由于单流表中数据包匹配域的叉乘（cross-product），流表规则数量将大幅度增加[openflow1-3-1]，当流表不能再接受新的流规则时会影响系统性能。

另一个转换是从编译器领域引出的循环展开（unroll）的方法。通过展开，循环将由一串的顺序语句代替。具体地说，编译器领域中的展开方法依赖于显式的循环索引，例如 `for i in range(10)` 中的 `i`。并基于循环索引的值将循环展开为多个循环过程中的语句块。然后这些块连接在一起。但是循环展开方法是受限的。因为它只能支持循环次数明确的循环 (即静态循环条件)，例如 `for i in range(10)` 。更通用的循环例如 `while pkt.vlan is None` 或 `while i < len(pkt.segmentList)` 只能在执行的时候确定这个循环次数 (即动态循环条件)。对于动态循环条件，循环展开的方法将无法应用。

在本章中，我们提出 RSP（Repeated Software Pipeline）的循环编译转换方法。该方法的可以生成多流表结构，并支持动态循环条件。本文所提出的方法在计算循环的最佳流水线时显示出更高的效率。具体来说，我们证明了对于一类有 n 个循环次数的循环（n iteration loop）部署到流表数量有限的多流表流水线中 (设流表的最大数量为 k)，流水线设计只需考虑循环中的前 k 个迭代（我们假设 $n > k$）。当处理没有固定循环次数的循环如 `while pkt.vlan is None` 时，这个性质非常有用。

本章的贡献如下:

- 第一个对支持动态循环的高级 SDN 程序编译成多流表流水线的研究；
- 提出 RSP 转换。并证明了对于一类有 n 个循环次数的循环部署到流表数量有限的多流表流水线中 (设表的最大数量为 k)，流水线设计只需考虑循环中的前 k 个迭代。

3.2 相关工作

高级 SDN 编程模型: 对于被动式（reactive）高级 SDN 编程模型[voellmy2013maple, foster2011frenetic, monsanto2013composing]，数据包被转发到控制器并被处理。Maple[voellmy2013maple] 可以支持对数据包（包括循环）的任意处理。但是，它需要在控制器中处理数据包，而这会增加时延。对于主动式编程模型[snap, sivaraman2016packet]，尽管它们可以通过利用多流表流水线编译到数据通路，但它们的模型不支持循环。

SDN 控制平台: 有相关的 SDN 控制平台[odl, erickson2013beacon, gude2008nox, floodlight] 通过提供 API 来处理到达控制器的数据包，从而使上层应用程序可以支持循环（在处理数据包时）。然而，它们也同样增加了数据包处理的延迟。

可编程数据通路: P[P4-new] 提供编程功能，可以使用 P4 语言重新配置数据通路的流水线。然而，P4 不能在其编程模型中支持循环。

3.3 流水线设计与研究动机

在本节中，我们将首先简要介绍高级 SDN 程序中的流水线设计。然后我们将给出在这些流水线设计中处理循环的研究动机。

3.3.1 流水线设计

高级 SDN 程序用高级语言（与 OpenFlow 流规则相比）描述了对于进入网络的每个数据包要如何处理。为了保证通用性，我们考虑了 SNAP[snap] 和 packet transaction[sivaraman2016packet] 中的 “onPacket" 编程模型。其中每个语句在概念上都可以看作数据通路流水线中的流表[snap, sivaraman2016packet]。例如，考虑以下简单路由程序（SimpleRoute）:

```
L1: def onPacket(pkt):
L2:     policy = policy(pkt.srcIP)
L3:     dstSW = switch(pkt.dstIP)
```

```
L4:     route = route(policy, dstSW)
L5:     return route
```

SimpleRoute 根据每个数据包的源 IP 地址指定的策略来路由数据包。由于每个语句都可以被视为一个流表，因此程序可以被转换为具有三个流表的流水线，如图 3.1所示（即将程序主动编译为数据通路）。这里 return 语句不需要转换。需要注意的是，本章中所提的技术并不限定与特定的程序语法，而只需要满足每一个语句可以与一个流表对应即可，而对于大多数 SDN 高级程序（如[snap, sivaraman2016packet]），该条件都可以满足。

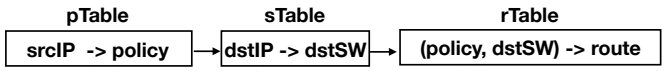


图 3.1　SimpleRoute 转换后的三张流表。每张表的结构 $a \rightarrow b$ 表示流表的匹配字段为 a 以及数据包匹配后可以设置 b 的值。

但是，为了满足 OpenFlow 交换机的硬件条件（即有限的流表数量），需要在流水线中合并一些表。例如，如果表的数量限制为 2，那么 *sTable* 和 *rTable* 可以合并为一个表，其匹配项为 *dstIP* 和 *policy*。我们将此过程称为流水线设计（即决定应合并哪些表）。具体地说，我们将初始流水线称为软件流水线，最终通过合并生成的流水线称为硬件流水线，因为它满足硬件条件的约束。

针对多个具有不同目标和约束条件的流水线设计策略，我们选择减少总流规则数量作为流水线设计的目标以及流表数量作为约束。例如，对于简单路由程序生成的三张流表，在最多允许两个流表的约束下，如果 *pTable* 和 *sTable* 的流规则数量均为 1000，*rTable* 的流规则数为 900，则优先选择合并 *sTable* 和 *rtable*（其流规则总数为 1000+1000*900）而不是合并 *pTable* 和 *sTable*（其流规则总数为 1000*1000+900）。我们可以看到第二种设计是最优的。这里我们不考虑表合并算法或相应的流规则压缩算法，因为它们已经得到了很好的研究[ge2015h, gupta2001algorithms]。

3.3.2　解决流水线设计中循环问题的研究动机

为了具有高灵活性，我们应该减少编程模型中的约束。然而，现有的相关工作[snap, sivaraman2016packet] 不能在它们的语言中支持循环。循环的实用性将在如下循环程序（SimpleLoop）中说明。

```
L1: def onPacket(pkt):
L2:     p = None; u = Max; ports = allPorts(pkt.dstIP)
L3:     for tp, tu in ports:
L4:         if u > tu:
L5:             u = tu
L6:             p = tp
```

SimpleLoop 首先初始化一个端口 (p) 及其利用率 (u)，并选择一组潜在端口（L2）。然后它遍历所有可能的端口（L3 到 L6），并比较它们的端口利用率，选择利用率最低的端口。虽然该程序可以通过提供一个用于端口选择的 API 实现，而不使用循环，但是 API 会给端口选择策略添加约束。程序员应该有足够的灵活性，并用高级语言 (即 L4 到 L6) 实现自定义的端口选择策略。

由于现有的 SDN 数据通路（例如ofdpa）不支持循环，因此为了在底层数据通路上实现需要对循环进行转换。有两种可能的转换方法：

第一种方法是黑盒方法，它将整个循环视为单个语句。SimpleLoop 中的循环可以看作是一个单独语句，进而可以视为单个流表。其中的匹配项包括 p, u, 和 *dstIP*。这种方法的问题是：1. 不能利用数据通路中的多流表流水线结构；2. 由于循环中迭代之间的数据依赖性，流表合并后相应的流表内容很难生成。

第二种方法是应用编译器领域中的循环展开技术。通过循环索引（即 i 的值）可以对循环进行展开。例如，SimpleLoop 的循环可以转换为：

(我们首先把 `for tp, tu in ports` 转换为 `for i in len(ports)`。)

```
L1: def onPacket(pkt):
L2:     ...
L3:     i = 0
L4:     ... #the loop body
L5:     i = 1
L6:     ... #the loop body
L7:     ... #i from 2 to len(ports) - 1
```

该做法的优点是转换后的程序可以应用于流水线设计。但其局限性是：1. 循环展开技术只能处理静态循环条件（例如显式的迭代次数）。而循环条件可以是动态的 (例如 `while pkt.vlan is None`)，所以这种方法是受限的；2. 即使循环可以转换为顺序语句，但长的语句序列可能会降低流水线设计的性能。

因此我们提出 RSP（Repeated Software Pipeline）的循环编译转换方法。该方法支持动态循环条件并计算循环的最佳流水线时显示出更高的效率。

3.4 重复软件流水线（RSP）转换

在本节中，我们将首先描述在流水线设计中引入循环之后的一些关键概念，然后再根据这些概念导出 RSP 转换。

3.4.1 关键概念

流水线: 由于这里的流水线包括软件流水线和硬件流水线，而软件流水线更贴近与对程序的建模，因此我们这里对上一章的流水线模型进行了拓展（即引入

输入/输出变量的概念）。流水线 pl 由一系列表 $t_1, t_2, ...t_m$ 组成。每个表 t_i 指定一组变量 $t_i.inputV$ 作为输入变量，另一组变量 $t_i.outputV$ 作为输出变量。（即到达 t_i 的数据包可以匹配 $t_i.inputV$ 并将值写入 $t_i.outputV$。）路由操作，例如 Drop 或 Forward(port=i)，属于特殊输出变量的实例。当一个数据包 pkt 在没有任何路由操作的情况下通过 t_i 后，pkt 可以跳转到表 t_j，其中 $i < j$ 保证 pl 中没有循环。我们将一组变量 $pl.inputV$ 称为 pl 的输入变量，且 $pl.inputV = (\cup_{i=1,2,...,m} t_i.inputV - \cup_{i=1,2,...,m} t_i.outputV)$，又将一组变量 $pl.outputV$ 称为 pl 的输出变量，且 $pl.outputV = (\cup_{i=1,2,...,m} t_i.outputV - \cup_{i=1,2,...,m} t_i.inputV)$。具体来说，流水线的输入、输出变量是指从外部访问流水线的变量（即它们可以在流水线之前设置，在流水线之后读取）。如果表 t 有一个输入变量是 v_a，则变量 v_a 是 t 的隐藏输出变量，这是因为如果 t 中的一条规则匹配了 v_a 的一个值，则该规则也必然输出相同的 v_a 值。

静态单赋值：给定流水线 pl，pl 的静态单赋值（SSA）形式意味着 pl 中的每个变量只能赋值一次。SSA 的一个例子可以在图 3.2中看到。从现在起，我们假设每个流水线都是具有 SSA 形式的。

循环条件和循环体: 给定高级 SDN 程序中的循环，循环条件是判断数据包是否进入循环的控制语句。循环体是应用于数据包的重复过程。为了简单但不丢失普遍性，我们考虑拥有布尔条件为循环条件、重复代码块为循环体的 *while-loop*。如果循环次数是执行时才确定的，则表示该循环条件是动态的。

3.4.2　重复软件流水线 (RSP) 转换

给定高级 SDN 程序中一个循环，RSP 转换将执行以下操作：1. 提取循环条件和循环体，并将它们链接在一起以生成新的语句集；2. 计算新语句集的软件流水线（每个语句都可以看作一个流表）；3. 连接软件流水线的多个副本并生成 RSP。举例说明，如果循环条件是 `while i < len(pkt.segmentList)` 循环体是 `test = True if check(pkt.srcAddr, pkt.segmentList[i++]) else False`，则软件流水线是 $(i, segmentList) \rightarrow None, (srcAddr, segmentList, i) \rightarrow (i, test)$。第一个表用于检查是移动到下一次循环过程还是中断循环。RSP 则是将该软件流水线的多个拷贝连接在一起。以下是 RSP 的定义：

定义 3.1： 给定有相同的布局和流规则的 n 个流水线（表示为 pl_1, pl_2, ... pl_n），我们用 pl^n 表示 n 个顺序连接 pl_1, pl_2, ... pl_n 构成的 RSP，并称为 n-RSP。

给定一个 pl^n，当一个数据包 pkt 在没有任何路由操作（如 Drop 或 Forward(port=i)）的情况下通过 pl_i 时，pkt 将到达 pl_{i+1}，其中 $1 \leq i < n$。RSP 及其 SSA 形式的示例如图 3.2所示。

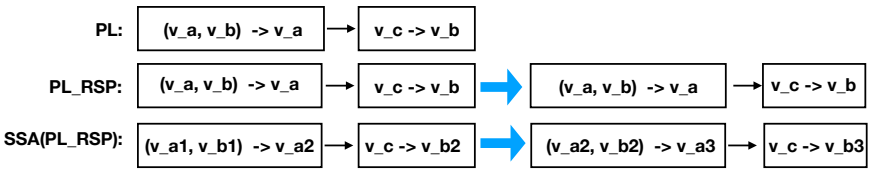


图 3.2　流水线 PL, PL 的 2-RSP 形式（PL_{RSP}），SSA 形式（$SSA(PL_{RSP})$）。

对于给定的循环及其对应的 RSP，如果一个数据包 pkt 通过重复 n 次进入该循环，然后生成一个输出，那么该输出于与 pkt 进入 n-RSP 的输出相同。直观理解是（考虑循环把 i 从 0 迭代到 n ），对于重复软件流水线中的每个软件流水线，虽然它与其它的没有区别，但是每个流水线都会读一个上一个流水线更新过 i 的值以及增加 i 的值。因此，到达不同流水线的数据包可以应用不同的规则，然后通过 n 个相同的流水线，它可以得到与循环的 n 次迭代过程后相同的输出。

在介绍了 RSP 之后，我们将其应用到流水线设计中。在下一节中，我们将对基于 RSP 的流水线设计进行分析，以说明流水线设计可以有效地完成（即 RSP 的有效性）并支持动态循环条件。

3.5　流水线设计以及 RSP 分析

在这一节中，我们首先分析了流水线的最优设计。然后，我们对 RSP 进行了分析，以证明其高效和有效性。

3.5.1　最优化流水线设计

我们根据流水线的数据流图将流水线设计归类为一个分割问题。

流水线数据流图: 给定一个流水线 pl，我们称 $DFG(pl)$ 是这个流水线 pl 的数据流图（DFG），它是一个有向无环图 (V,E)。其中点 $v \in V$ 表示流水线 pl 中的一个变量，并且如果存在 pl 中的一个表 t_i，它的 $t_i.inputV = \{v^i_{i1}, v^i_{i2}, ...v^i_{i|t_i.inputV|}\}$ 且 $t_i.outputV = \{v^o_{i1}, v^o_{i2}, ...v^o_{i|t_i.outputV|}\}$，则有 $|t_i.inputV| * |t_i.outputV|$ 条有向边 $v^i_{i1} \rightarrow v^o_{i1}$, $v^i_{i2} \rightarrow v^o_{i1}$, ..., $v^i_{i|t_i.inputV|} \rightarrow v^o_{i|t_i.outputV|}$ 在 E 中。图 3.2中 SSA 形式的流水线的 DFG 在图 3.3中所示。

源节点选择: 首先从 DFG 中的源节点（即入度为零的节点）中选出一个子集并放入 S，如果 DFG 中的一个节点 v 的所有父节点都在 S 中，则可以将 v 添加到 S 中。一个选择可以包含多次添加过程。选择完成后，从原始 DFG 中移除 S（即 S 从 DFG 中分割），然后 S 可以被视为一张流表 t。在 S 中且没有任何父节点的节点被添加到 $t.inputV$，在 S 中且没有任何子节点的节点被添加到 $t.outputV$。源节点选择的示例如图 3.3所示。其中 v_{a2} 可以添加到 $S1$，但 v_{a3} 不能（即使将 v_{a2} 添

加到 $S1$ 之后），因为它的一些父节点不在 $S1$ 中。通过将 v_{b2} 和 v_{b3} 添加到 $S2$，当前选择可以有三个表（如图 3.3中的红点线所示）：$(v_{a1}, v_{b1}) \rightarrow v_{a2}$, $v_c \rightarrow (v_{b2}, v_{b3})$, 以及 $(v_{a2}, v_{b2}) \rightarrow v_{b3}$ 。

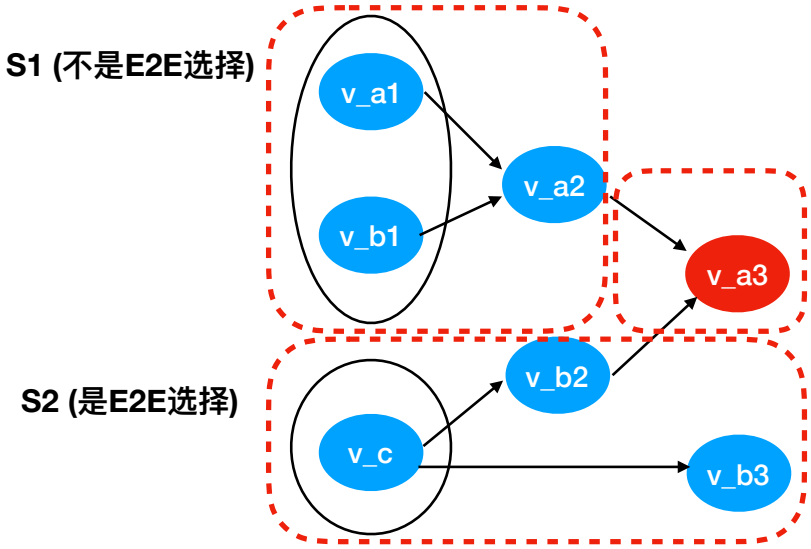


图 3.3　流水线的 DFG 和源节点选择。

流水线设计可以视为流水线 DFG 中的多次源节点选择。如果流水线设计对底层数据通路的约束为最多 k 个流表，那么最多会有 $k-1$ 源节点选择。

最优流水线设计的贪婪性: 如果流水线设计是最优的，则意味着生成的流水线具有最少的流规则数量。在这种情况下，选择的每一步都应该是‘‘贪婪的"。即，如果 v 可以添加到 S 中，则它应该被添加。我们将此属性表示为贪婪选择性质。例如，在图 3.3中，为了获得最佳的流水线，应该将 v_{b3} 添加到 $S2$ 中，否则还有另一个表：$v_c \rightarrow v_{b3}$。一个简单理解则是，如果有两个流表 t_1 和 t_2，以及 $t_2.inputV \subseteq (t_1.inputV \cup t_1.outputV)$，那么将 t_2 合并到 t_1 中不会添加额外的流规则。这意味着在优化的流水线设计中，t_2 应该合并到 t_1 中，以减少对应的数据包需要的处理步骤的数量，降低数据包的处理时延。

3.5.2　重复软件流水线的分析

基于最优流水线设计的贪婪特性，我们论证流水线设计在 RSP 上执行的有效性和效率情况（通过分析）。我们首先给出流水线优化设计的目标。

最优流水线设计的目标: 在硬件限制下（即最多可以实现 k 个表），在 pl^n（$k < n$）上计算具有最小流表规则数量的流水线。由于流水线深度影响通过流水线的数据包的时延，如果有满足约束的两条流水线具有相同数量的流规则，优先选择表数较少的流水线。

复杂度: 由于流水线设计的复杂度取决于 DFG 中的节点数量，因此 n 很大的 pl^n 的流水线设计是一个非常复杂的过程。然而，我们发现并证明复杂度依赖于 k，而不是 n（在 $k < n$ 下）。

我们首先提出三个概念：单一输出流水线、E2E 选择和全输出表。

单一输出流水线: 一个流水线 pl，其中 pl 的输出变量只出现在 pl 的最后一个表中。我们将这种流水线表示为单一输出流水线 (Single-output Pipeline，SO-PL)。SO-PL 的例子在图 3.4中所示。

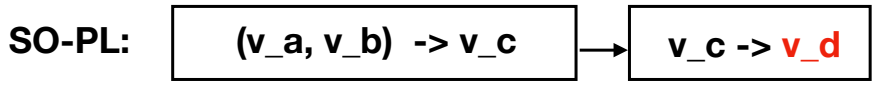


图 3.4　SO-PL 示例.

E2E 选择: 如果 DFG 上的源节点选择出的节点集合（包含所有源节点的真子集，而非全部源节点）中至少有一个 DFG 中的汇节点 v，则该选择称为 E2E（End-to-end）选择。E2E 选择的示例如图 3.3所示。其中 $S1$ 不是 E2E 选择，因为 v_{a3} 也依赖于 v_{b2}。但是 $S2$ 是 E2E，因为 v_{b3} 只依赖于 v_c。

根据 E2E 选择的定义，我们发现 SO-PL 中没有 E2E 选择。对于至少选择了一个汇顶点 v 的选择，v 的祖先应该是所有的源顶点（而不是一个真子集）。

全输出表: 给定 pl^n，如果一个表 t_x 的输出变量包含流水线 pl_i 的所有输入变量，则 t_x 是 pl^n 的全输出表。根据贪心性质，如果是最优流水线设计，则所有剩余的表都应合并到 t_x 中。如果有多个流水线的输入变量包含在 t_x 的输出变量中，我们将这些流水线的最小索引定义为 $MinIndexPL(t_x)$。考虑图 3.2中 SSA 格式流水线（$SSA(PL_{RSP})$），考虑一个流表 t_y，其输出变量为 v_{a2}, v_{b2}, v_c，则 t_y 是 $SSA(PL_{RSP}$ 的全输出表。当 $SSA(PL_{RSP}$ 的第二条流水线合并到 t_y 后，其第三条流水线（图中未显示）也合并到 t_y 中。在这种情况下，$MinIndexPL(t_y)$ 仍然是 2，即是第二条流水线的索引。

在流水线设计中，我们对全输出表的存在性得出以下定理。

定理 3.1： 存在性：如果 $DFG(pl_i)\ \forall i \in 1,2,...n$ 中没有 E2E 选择，那么通过流水线设计将 pl^n 合并到 k 个表（$k < n$），合并的 k 个表中至少存在一个全输出表 t_x。

证明 我们用反证法，即假设合并的 k 个表中不存在这样的 t_x。

基于这个假设，我们给出了一个流水线设计，它将 pl^n 合并到最少数量的表中，如图 3.5所示。对于第一个流水线 pl_1，我们选择 pl_1 的输入变量的真子集来得到表 t_1。（我们避免使用全部输入变量，因为这样会得到一个 t_x）。由于没有 E2E 选择，t_1 的输出变量不能包括 pl_1 的任何输出变量。我们考虑第二个表 t_2，其中输入变量是剩余 DFG 中的所有源节点（这与原始源节点不同，因为已经对一

组节点进行了分割）。我们发现 t_2 的输出变量不能是 pl_2 的全部输入变量，因为不能有 t_x。同样地，t_3 的输出变量不能包含 pl_3 的全部输入变量。

我们发现这种流水线设计给出的表的最小数目是 $n+1$。但是，我们有 $n+1 > k$（它超出了流表数量的限制），因此没有 t_x 的假设是错误的。 □

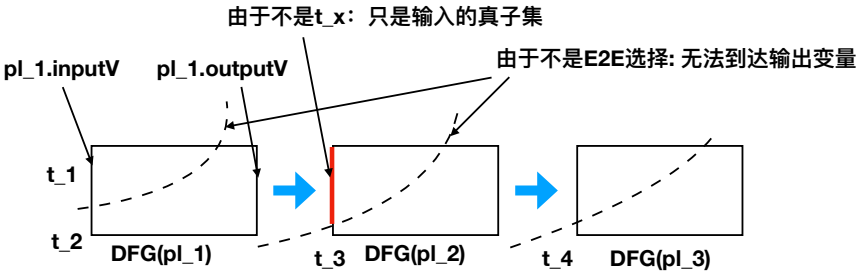


图 3.5　存在性定理的证明，图中三个流水线 (pl_1, pl_2, pl_3) 在假设下只能被合并到至少四张表 (t_1, t_2, t_3, t_4)。每个方块是一个流水线的 DFG.

存在性定理表明，如果我们要将 pl^n 合并到 $k < n$ 的 k 个表中，并且如果 pl^n 的 DFG 中没有 E2E 选择，那么合并表中至少存在一个全输出表。由于该定理不涉及流水线设计的最优性，因此可以得到多个全输出表。

基于全输出表存在性的定理，我们给出了考虑最优流水线设计的全输出表的位置定理。

定理 3.2： 位置：如果 $DFG(pl_i)\ \forall i \in 1, 2, ...n$ 中没有 E2E 选择，那么通过最优流水线设计将 pl^n 放到至多 k 个表（$k < n$）时，在最多 k 个表合并后，必须只有一个全输出表 t_x，且 $MinIndexPL(t_x) \leq k$。

证明 如果出现一个全输出表，为了优化流水线设计的需要（即贪婪性质），下面所有的表/流水线都应该合并到全输出表中。因此，只有一个全输出表 t_x。如果 $MinIndexPL(t_x) > k$，则必须将开始的 k 个流水线合并到 $k-1$ 表中，这被存在性定理排除。 □

小结: 如果每个软件流水线都是 pl^n 中的 SO-PL（即没有 E2E 选择），那么对于把 pl^n 放到至多 k 个表（$k < n$）的最优流水线设计，流水线设计应该只考虑前 k 个软件流水线，而不是所有 n 个流水线（根据位置定理的结果，用前 k 个流水线设计的结果和用前 n 个相同），这证明了其高效性，因为流水线设计的复杂程度不随着流水线数量 n 而提高。此外，对于有效性，即使不能在执行之前指定迭代次数，对于要最多合并到 k 个表的情况，也只需要考虑前 k 条流水线的最佳流水线设计。

给定一个全输出表 t_x，将后续所有表/流水线合并到 t_x 将增加 t_x 的大小。但是，在数据通路设计中，一个逻辑流表可以映射到多个硬件流表[bosshart2013forwarding]，这意味着在实现中逻辑表的大小不是问题。

放宽约束: SO-PL 约束也可以通过以下方式放宽：每个软件流水线的 DFG 至少有一个汇节点，其祖先为所有源顶点。一个例子是图 3.3中的 v_{a3}，它的祖先是所有的源顶点。我们可以看到，SO-PL 是一个特殊情况，其中所有汇节点都将所有源顶点作为祖先。一个简单的证明：我们首先假设 pl^n 可以合并到 k 个表中，而不需要创建一个全输出表，并且 pl^n 的每个软件流水线至少有一个汇节点 (v) 其祖先都是源顶点。我们移除合并后 k 个表中输出变量不是 v 的相应流规则（即，其余的流规则都是 v 的）。根据存在性定理，合并后的 k 表不存在，这意味着假设是不正确的。

在放宽后的约束条件下，即循环中至少有一个依赖于所有输入变量的输出变量，大部分循环可以作为我们提出的设计的目标。而对于不符合此要求的循环，我们可以将循环分成多个子循环（通过现有编译器技术），以便每个循环都满足约束。

3.6 实验评估

在这一部分中，我们从流水线设计的执行时间和生成的流水线的流规则数量两个方面来评估所提出的 RSP 设计。所有评估都在 16GB 内存以及 3.5 GHz Intel i7 处理器上运行，系统为 Mac OSX 10.13。

3.6.1 执行时间

实验方法: 基于最优流水线设计的分析，考虑以下简单的流水线设计算法：给定一个 DFG 和 k，递归地在 DFG 上应用源节点选择 $k-1$ 次来枚举所有可能的硬件流水线。对于 RSP 和展开技术（unrolling）的比较评估，我们随机生成具有以下条件的 DFG：对于 RSP，我们变化 RSP 中的软件流水线数（即循环的迭代次数 n）和每个流水线 DFG 中的节点数；对于展开技术，从 RSP 中生成的图的每一个软件流水线中随机移除几个节点。然后，我们比较流水线设计算法应用在 RSP 生成的图和展开技术生成的图的执行时间。因为我们没有考虑合并算法，枚举所有可能的流水线的复杂性相当于找到最佳流水线的复杂度。

实验结果: 结果如图 **??**所示。水平轴指定循环的迭代次数 (n)。图 3.6和图 3.7之间的不同点是每个软件流水线的 DFG 中的节点数 (m)（对于展开技术，它是指随机移除节点之前的节点数）。其中图 3.6中的 $m = 10$ ，图 3.7中的 $m = 20$。所有流水线设计都设 $k = 10$ 作为流表数量的限制。结果表明，与 $k < n$ 时的展开技术相比，RSP 方法缩短了执行时间 (大约十倍，当 $n = 50$ 且 $m = 10$)。当 $n = 10$ 时，RSP 和展开在流水线设计中都考虑 10 个软件流水线，因此，两种方法的执

行时间是相同的。当 $n = 100$ 时，与 RSP 方法相比，展开的执行时间过长。

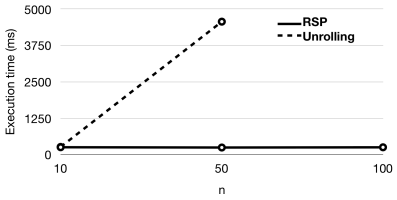


图 3.6　$m = 10$.

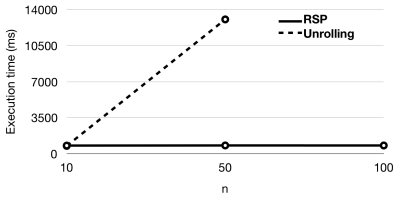


图 3.7　$m = 20$.

图 3.8　两种方法的执行时间。n 为迭代次数。m 为顶点数量。

3.6.2　流规则数量

实验方法: 为了计算流规则的数量，我们随机给每个变量（即 DFG 的节点）设置域的大小（即变量的可用值的数量）。具体来说，对于每个软件流水线的 DFG 中的源节点，我们将其值设置为 100 到 200。对于 DFG 中的内部节点，我们将值设置为 10 到 20。这是因为对于源节点，它们可能表示数据包字段。与程序中的内部变量相比，数据包字段的值范围应该更大。此外，给定一个流表，流规则数量的计算等于所有输入变量的域大小的乘积。n 和 m 的定义与上面执行时间的评估时相同。我们比较了 RSP 设计和黑盒方法（即生成单表）的流规则数量。

实验结果: 结果见表 3.1。对于黑盒方法，流规则的数量只取决于所有软件流水线的第一个 DFG（等于第一个 DFG 的所有源节点的域大小的乘积），因此对于不同的 n，值是相同的。结果表明，与单表设计方法相比，多表设计方法可以显著减少流规则的数量。值得注意的是，对于每个表，RSP 方法可能有很多未使用的流规则。例如，可以将 `while i in range(100)` 语句视为匹配 i 并包含 100 条流规则 (即 $i = 0, 1, ..., 99$) 并更改 i 的流表。当该流表作为 RSP 的第一个流表时，该表有 99 个未使用的流规则。

	n =10, m = 10	n = 50, m= 10
Single	3898434	3898434
RSP	61856	224098

表 3.1　单表方法和 RSP 方法下的流规则数。

3.7　本章小结

本章针对高级 SDN 程序中循环结构在可定制结构数据通路的高效实现问题，提出了 RSP 转换。该方法的可以生成多流表结构，并支持动态循环条件。本文所提出的方法在计算循环的最佳流水线时显示出更高的效率。具体来说，我们

证明了对于一类有 n 个循环次数的循环（n iteration loop）部署到流表数量有限的多流表流水线中 (设流表的最大数量为 k)，流水线设计只需考虑循环中的前 k 个迭代（我们假设 $n > k$）。

第 4 章　global 章节

4.1　引言

随着 SDN 技术的广泛应用以及底层可编程数据平面技术的不断发展（即从单流表到多流表流水线到自定义多流表流水线），如何实现面向全网络（即多交换机）可编程数据平面的高级 SDN 编程变得越来越重要。具体来说，我们需要达到，在保持高级 SDN 编程模型的同时，用户编写的高级 SDN 程序通过系统可以自动地编译成多个可编程数据通路的配置。

与传统 SDN 技术相比较，通过自定义多流表流水线技术（例如，P4[p4]），可编程数据通路的配置需要包括自定义的流水线结构，以及相应的流表内容。而在传统 SDN 中，由于只考虑单流表的数据平面（或具有特定功能的数据平面），其配置只包括流表内容。例如，Maple[voellmy2013maple] 通过控制器上收到的数据包，建立踪迹树的结构，并生成流表规则。但它并没有主动地生成流表规则，并且只考虑了单流表结构。Beckett 等人[beckett2016don] 利用领域特定语言，让用户可以表达 BGP 上的目标以及约束，并最终生成每个交换机的配置。但他们考虑的是特定的网络场景（即 BGP）。Tian 等人[tian2019safely] 通过意图模型，自动地生成了 ACL（Access Control List）中的规则，并且实现了自动更新。但他们只考虑了特定的网络功能（即 ACL）。

在面向全网络的可编程数据通路自动配置的工作中，Arashloo 等人[snap] 考虑有状态交换机，并将高级 SDN 程序自动地编译为多个有状态交换机的配置。但其基于的是 One-Big-Switch 模型，用户在编程时只可以看到终端主机，而没有网络拓扑细节信息（即相当于一个大的交换机）。因此用户无法让数据包沿着特定的路径进行转发。

本章中，我们将考虑基于多交换机的高级 SDN 程序数据平面实现问题。对于数据平面，我们分为两大类：完全可编程网络以及部分可编程网络。完全可编程网络是指，网络中的交换机均支持可配置的数据通路（如 P4）。而部分可编程网络是指，网络中不光包含支持可配置的数据通路，并且包含只可以实现特定功能的中间盒（如防火墙等）。对于上述两类数据平面，我们都要考虑高级 SDN 程序的实现问题。

4.2 完全可编程网络实现问题

我们考虑的面向全网络 SDN 编程模型本质上与单交换机的编程模型类似，同样参考已有的 SDN 编程模型（如[snap, sivaraman2016packet]）。但面向全网络的 SDN 程序的返回则是网络中的一条路径（即多个交换机的操作），而非单个交换机的操作。因此，对于程序的返回，我们采用路由代数的形式进行表达[gao2018t]。我们通过以下示例程序进行说明。

```
L1: def onPkt(pkt):
L2:   if pkt.srcIP in whiteList:
L3:     srcHost = hostTable(pkt.srcIP)
L4:     dstHost = hostTable(pkt.dstIP)
L5:     if pkt.dstPort in portList:
L6:       return opt(srcHost -> dstHost)
L7:     else:
L8:      return opt(srcHost -> c -> dstHost)
L9:   else: return DROP
```

图 4.1 一个简单的面向全网络的高级 SDN 程序。

如图 4.2所示，用户希望对于目的端口不在 *portList* 的数据流必须经过节点 *c*（程序中 L5-L8）。并且源 IP 不在 *whiteList* 的数据包进行丢弃（程序中 L2）。程序的返回为路由代数，如程序中的 L6 和 L8 分别表示 *srcHost* 到 *dstHost* 的最优路径（即最小跳数），*srcHost* 到 *dstHost* 并经过 *c* 的最优路径。

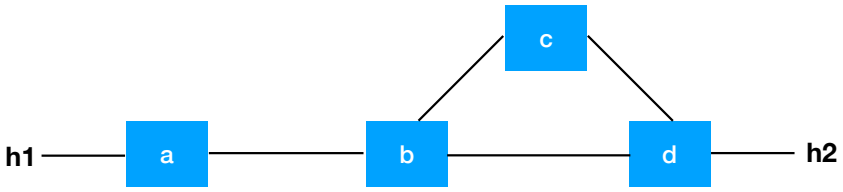


图 4.2 网络拓扑。

对于完全可编程网络，在不考虑交换机资源限制情况下，当一个高级 SDN 程序除了返回路径的语句之外都可以在单交换机上实现时，该程序也可以非常简单地在该完全可编程网络中实现。实现方法如下：将程序的所有逻辑放到边缘交换机，并通过封装数据包的方式，使数据包穿越网络。如图 **??**所示，边缘交换机为连接着主机的交换机 *a* 和 *d*，因此可以把程序的逻辑全部放在 *a* 和 *d* 上。然而当网络资源有限时（即边缘交换机流表资源不够，无法实现程序的全部逻辑），该方法是不对的。

当资源受限时，我们的目标则是将程序 ‘‘打散"，使其不同部分实现在不同交换机上。然而，确定哪些部分实现在哪些交换机上是一个问题。SNAP 考虑了类似的问题，由于其依赖于 One-Big-Switch 模型，SNAP 并不支持用户程序自定义的路径。而自定义路径会使问题变得复杂。如图 **??**所示，对于 $h2$ 发往 $h1$ 的数据流，由于交换机 d 需要判断数据包的目的端口，因此 d 必须包含程序的全部逻辑。当网络以及程序变得复杂时，我们需要一个方法将程序实现在网络中。

程序的决策树：我们首先对程序生成决策树。决策树的每一个内部节点为程序的一个语句；叶子节点为程序返回的路径。图 4.3给出了示例程序的部分决策树。决策树生成的一种做法是符号执行[king1976symbolic]，Voellmy 等人[voellmy2016magellan] 给出了相应的做法，因此我们这里不对如何生成决策树进行展开。

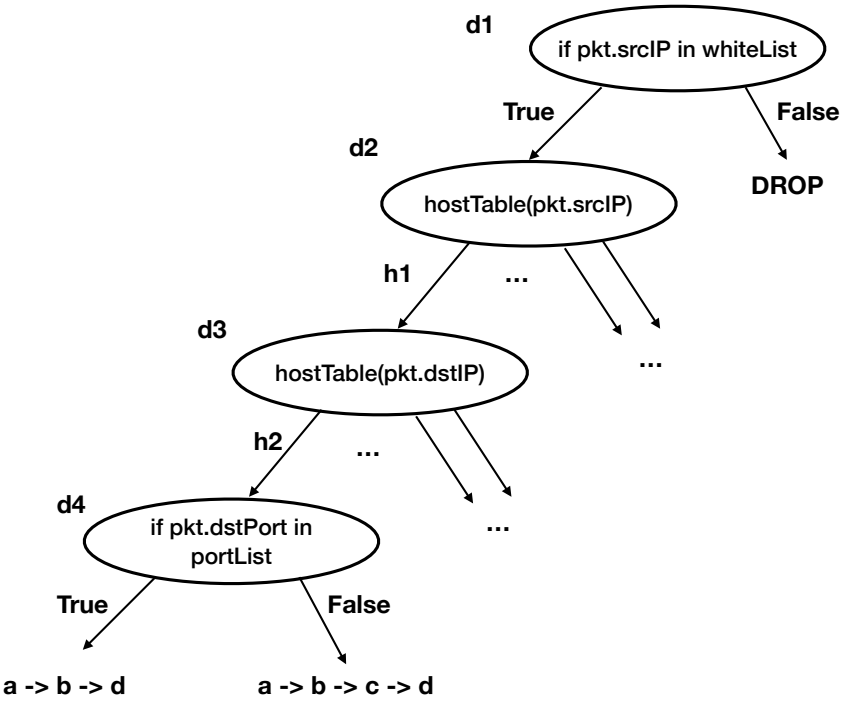


图 4.3　SDN 程序对应的决策树。

基于决策树，我们现在可以判断哪些语句必须要在一些交换机上执行。做法如下：从每个边缘交换机开始考虑。对于边缘交换机 n，若决策树的叶子节点中的路径的初始节点不是 n，则删除该叶子节点以及相应的无效边。然后从根节点开始计算所有内部节点（包括根节点）下面的叶子节点中路径的下一跳交换机，并将其标识在内部节点上（称为该节点可能的下一跳）。若一个节点可能的下一跳都相同，则说明该节点对应的语句可以在后面执行。因此，基于所有节点可能的下一跳，我们可以得到哪些语句一定要在该边缘交换机上执行。对于其余节点，则沿着同样的下一跳，在下一个交换机执行相同过程（在此之前从所有叶子节点的路径中删除 n）。最终，我们可以得到：对所有路径上的所有需要执行的

变量	含义
p_i	转发路径 i，包含一串有序的网络节点以及一串有序的决策树节点
n_j^i	p_i 中第 j 个网络节点
d_j^i	p_i 中第 j 个决策树节点
$code_i(n)$	网络节点 n 对于 p_i 的编号
N_j^i	d_j^i 可以选择的网络节点的集合
$x(d_j^i, n)$	1: d_j^i 选中网络节点 n; 0: d_j^i 未选中网络节点 n
$m(n)$	网络节点 n 最多可以实现的决策树节点数量

表 4.1　变量。

约束	含义
$xc_j^i = \sum_n x(d_j^i, n) * code_i(n)$	决策树节点选中的网络节点的编号
$\forall d_j^i, xc_j^i \le xc_j^{i+1}$	保证决策树节点之间的依赖关系
$\forall d_j^i, \exists n \in N_j^i, xc_j^i = code_i(n)$	对决策树节点可选的网络节点的范围进行约束
$\forall n, \sum_{d_j^i} x(d_j^i, n) \le m(n)$	保证网络节点资源
$\forall d_j^i \sum_n x(d_j^i, n) = 1$	保证决策树节点与网络节点的一对一映射

表 4.2　约束。

语句，它们都有相应的可执行交换机的范围。

示例：考虑图 4.3中的决策树。我们发现对于边缘交换机 a，决策树上的所有节点的可能的下一跳都相同（b）。因此，所有的语句可以不在 a 上执行。当整个决策树移到 b 时（此时叶子节点已经被更新），我们发现所有的节点的可能的下一跳都不同（均为 c 或 d）。因此，我们得到所有语句的可以执行的范围为交换机 a 或 b。

在确定每个语句（决策树上内部节点）可以执行的交换机范围后，我们可以通过混合整数线性规划进行求解。其中变量在表 5.1给出；约束在表 5.2给出。系统的目标可以是：minimize $max(\sum_{d_j^i} x(d_j^i, n))$，即最小化所有交换机最多实现的语句数量，使程序语句可以相对均匀地分布在网络中。

最终，通过计算混合整数线性规划，我们可以得到决策树中节点与网络交换机的对应关系，即对于每个交换机，我们可以得到它应该实现哪些程序语句，进而算出交换机配置。

4.3　部分可编程网络实现问题

与完全可编程网络相比，部分可编程网络中包含具有特定、固定功能网络节点，如中间盒等。而程序可以利用这些网络节点的功能对数据包进行处理。

首先给出我们设计的使用户能指定网络中的中间盒的原语。第一个是指定网络中的中间盒，`m = middlebox(name, property)`。其中 property 标识该中间

盒是否有状态。第二个是调用中间盒对数据包的处理，`m.handle(pkt)`。具体来说，我们考虑一个中间盒为一个数据包处理函数。它可以返回对收到的数据包进行处理的结果。对于一个无状态中间盒，如果两个数据包具有相同的五元组，则它们的返回结果也相同；对于一个有状态中间盒，由于数据包的处理会考虑中间盒的状态，即使两个数据包具有相同的五元组，它们的返回结果也不一定相同。

除了中间盒原语之外，为了让用户指定数据包在网络传输的路径的约束，这里也引入了路由代数作为程序的返回。图 4.4给出了对于部分可编程网络编程模型的抽象语法。对于中间盒，编程人员可以指定其属性，即有状态或无状态。中间盒的返回结果可以存储在一般变量中，在接下来的程序中调用。

$$
\begin{array}{rcll}
p & ::= & \text{onPacket(pkt) } \{I\},\ d_1,\ \ldots,\ d_n & (\textit{program}) \\
d & ::= & x^r\ =\ r & (\textit{route algebra decl}) \\
 & & \mid x^m\ =\ middlebox(n,\ s) & (\textit{middlebox decl}) \\
I & ::= & x\ =\ e & \\
 & & \mid I; I & (\textit{sequencing}) \\
 & & \mid x\ =\ x^m.handle(pkt) & (\textit{middlebox operation}) \\
 & & \mid \text{if } e^b : I \text{ then } : I & (\textit{conditional}) \\
 & & \mid \text{return } x^r \mid \text{return } r & (\textit{func return}) \\
e & ::= & c \mid x & (\textit{consts, vars}) \\
 & & \mid x^m & (\textit{middlebox vars}) \\
 & & \mid pkt.a & (\textit{packet fields}) \\
e^r & ::= & e == e \mid e \leq e \ldots & (\textit{relational}) \\
e^b & ::= & e^r \mid e^r\ \&\ e^r \mid e^r \mid e^r \ldots & (\textit{boolean}) \\
(r & \in & \textit{route algebra expressions}) & \\
(c & \in & \textit{strings}) & (\textit{consts}) \\
(a & \in & \{macSrc, ipDst \ldots\}) & (\textit{packet fields}) \\
(n & \in & \textit{strings}) & (\textit{middlebox names}) \\
(s & \in & \{stateless, stateful\}) & (\textit{properties}) \\
(x & \in & \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{y}_1, \ldots\}) & (\textit{variables})
\end{array}
$$

图 4.4　引入中间盒处理的抽象语法。

示例：基于图 4.4给出的抽象语法，我们考虑如下引入了中间盒处理的高级 SDN 程序。同时网络场景为图 4.6所示。

现在我们来看图 4.5中的程序。其中 L2(L3) 指定了转发路径约束，即路径从 h_1 至 h_2 并需经过节点 b（c）。在返回语句中通过 *any* 的函数（路由代数 **??**），标

```
L1: mFW = middlebox("firewall", "stateless")
L2: PATH1 = h1 -> b -> h2 //b: waypoint
L3: PATH2 = h1 -> c -> h2 //c: waypoint
L4: //any: picking any path
L5: def onPacket(pkt):
L6:   if pkt.srcAddr == h1 & pkt.dsrAddr == h2:
L7:     if mFW.handle(pkt) == SENSITIVE:
L8:       return any(PATH1)
L9:     else:
L10:      return any(PATH2)
L11:  else: return DROP
```

图 4.5　引入中间盒处理的程序。

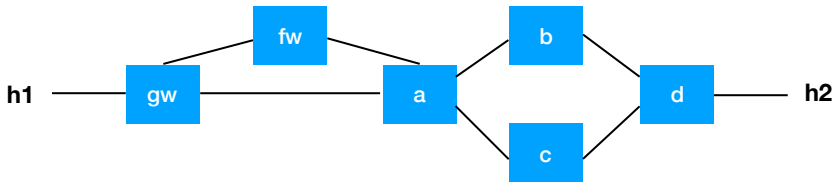


图 4.6　引入中间盒的网络拓扑。

明选出满足约束的任何一条路径。虽然编程模型非常直接简单，但由于引入中间盒对数据包的处理操作，而中间盒需要映射到网络的物理节点，系统需要保证程序的正确性。

正确性：我们采用如下对于正确性的定义：对于任何数据包 pkt，其在网络中的真实转发路径必须要符合程序中返回的对数据包 pkt 的路径。例如，如果程序员使用 opt 函数来选择一个满足约束的最少跳数路径，即 `opt(PATH1)`，则真实转发路径应为网络中的 gw, a, b, d。然而，由于数据包必须要经过防火墙，其真实转发路径必须包含防火墙 fw，从而不符合程序中返回的路径。

为了保证正确性，一个简单的做法是让程序员显式地在路径约束上包含需要的中间盒节点。在上述场景中，其应为 `PATH1 = h1 -> fw -> b -> h2`。然而，让程序员来保证路径约束和网络中真实转发的一致性的做法，会给程序员带来额外的负担。

系统路径约束：为了解决上述问题，我们引入系统路径约束（System Path Constraint，SPC）的概念。在程序中，SPC 表示为一个存有路径约束的全局变量。程序中的任何一个位置计算路径时需要满足 SPC 中的路径约束。其直观理解是，当一个数据包经过程序时，某些程序语句会给数据包添加路径约束。考虑图 4.7中

添加 SPC 后的程序。程序中的第六行添加了一个从 h_1 到 h_2 的路径约束；第七行添加了一个必须要经过防火墙的路径约束。因此，在第七行后，SPC 全局变量包含如下约束：从 h_1 到 h_2 以及经过防火墙 fw。第八行中使用 *opt* 函数计算路径，其中 “+’’ 表示连接 SPC 的路径约束和 PATH1 的路径约束，而其结果为 gw, fw, a, b, d。

```
L1: mFW = middlebox("firewall", "stateless")
L2: PATH1 = h1 -> b -> h2 //b: waypoint
L3: PATH2 = h1 -> c -> h2 //c: waypoint
L4: //any: picking any path
L5: def onPacket(pkt):
L6:   if pkt.srcAddr == h1 & pkt.dsrAddr == h2:
L7:     if mFW.handle(pkt) == SENSITIVE:
L8:       return opt(PATH1 + SPC)
L9:     else:
L10:      return opt(PATH2 + SPC)
L11:  else: return DROP
```

图 4.7　加入 SPC 的程序。

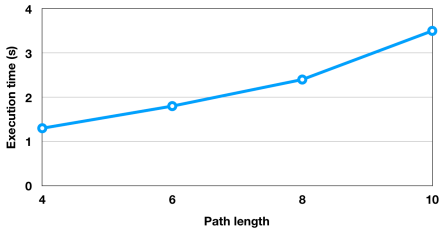
网络实现：在保证正确性的情况下，部分可编程网络实现问题等价于添加 SPC 约束后的完全网络实现问题。而 SPC 约束的增加只是在计算决策树中叶子节点路径时需要考虑，因此我们这里不再对部分可编程网络实现问题进行讨论。

4.4　实验评估

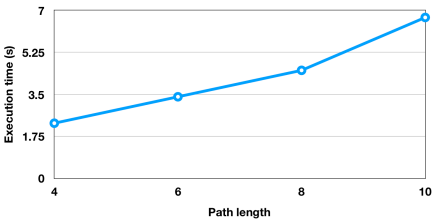
我们这里将对计算决策树中节点与网络交换机的对应关系中的混合整数线性规划的执行时间进行评估。评估是在 16GB 内存以及 3.5 GHz Intel i7 处理器上运行，系统为 Mac OSX 10.13。

实验方法：我们假设决策树具有完美二叉树结构，并通过改变决策树的大小，对相应的叶子节点（即转发路径）的数量以及程序语句的数量进行变化。给定转发路径数量下，在随机生成的拓扑图中随机生成相应数量且满足路径长度的转发路径，并随机地填充到决策树的叶子节点中。最后根据决策树以及拓扑图计算决策树中节点与网络交换机的对应关系，并记录混合整数线性规划的执行时间。

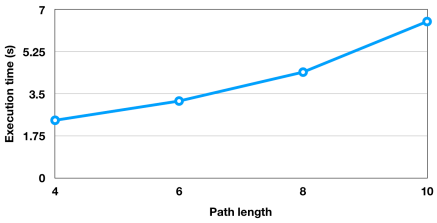
实验结果：图 4.8给出在不同场景下的执行时间。其中 #P 代表路径数量；#N 代表图中节点数量。从图 **??**可以看出，当路径长度增加时，混合整数线性规划的



(a) #P = 4, #N = 20



(b) #P = 8, #N = 20



(c) #P = 8, #N = 40

图 4.8　不同场景下的混合整数线性规划执行时间。

执行时间变长；通过比较图 **??**和图 **??**，我们可以发现，当路径数量增加时，混合整数线性规划的执行时间变长；通过比较图 **??**和图 **??**，我们可以发现，在其他变量不变情况下，拓扑的规模变大不会增长执行时间。这是因为，决策树节点的放置范围已经固定在各自所属的转发路径上，拓扑图中其他节点的增加不会影响已有决策树节点的放置范围。

4.5　本章小结

本章研究了高级灵活的 SDN 程序面向全网络数据平面实现问题。当网络节点全部为可编程交换机时（即完全可编程网络），提出将程序拆分并部署到不同交换机的方案，并算出给定目标下的最优部署；当网络节点存在固定功能节点时（如防火墙）（即部分可编程网络），考虑了程序正确性的问题。并提出系统路径约束的概念保证程序的正确。

第 5 章　stateful 章节

5.1　引言

近年来 SDN 的成功[swan, b4, edgefabric] 促使研究人员考虑将其应用于军事联合（military coalition）中，以实现一个高效、敏捷的软件定义联合（Software Defined Coalition，SDC）[vinod-sdc]。在 SDC 网络中，多个联合成员可以在一个高动态并受限（如电力）的战术网络中进行操作。

将 SDN 与军事联合结合并形成 SDC 的想法相对直观，因为通过 SDN 数据面上的灵活的数据包匹配处理和控制面上的集中式控制，联合成员可以在联合网络上实现高效、灵活的控制[p4, rmt]。然而这些突出的性质仍然无法充分满足需求。

根本原因是在一个高动态战术网络环境中，SDC 应用（例如灵活的负载均衡、DNS 放大攻击的探测）对性能（例如实时性、高效性、可靠性）有着非常苛刻的要求。在一个高动态战术网络环境中通过传统 SDN 架构运行的 SDC 应用，由于数据面和控制面之间频繁的数据传输，会受到严重的时延等问题。

最近研究人员提出了一些新的 SDC 数据平面设计[bianchi2014openstate, opensdc]。它们将对数据包进行的复杂有状态的操作（例如计数、流安全探测、拥塞控制）从控制平面卸载到数据平面，以提高 SDC 应用的性能。在这些系统中，不同的数据平面原语，如状态计数器、数据包缓存以及数据包网络中计算（in-network compute）模块被设计出来以支持不同的 SDC 应用，如有状态防火墙、主动路由保护、弹性路由等。

然而，这些系统有以下两点不足：1. 那些被卸载到数据平面上的有状态操作的结果（以下我们称之为本地状态）并没有在多个数据平面设备上进行共享，导致 SDC 网络中的巨大的资源浪费。例如，由于防火墙中间盒的有限的处理速度，其在 SDC 网络中经常是一个瓶颈。当一个数据流通过防火墙被标志为安全非敏感数据后，其数据流的后续数据包应该沿着一个高可靠高带宽的路径进行转发，而不同再次经过该防火墙。然而，由于该数据流的安全状态只保存在该防火墙设备而非共享给其他设备，在不确定该数据流是否安全情况下，一个上流设备（例如，一个网关路由器）仍然需要将所有数据包转发到该防火墙；2. 尽管已有一些分布式更新原语提供了在多个数据面设备之间共享本地状态的能力[ddp]，配置这些低层次的 SDC 数据面任然需要花费大量的时间而且非常容易出错。

为了解决以上两点不足，我们设计了 Dandelion，一个高级 SDC 编程系统。

Dandelion 提出了一系列的创新的编程原语，使用户可以灵活地制定 SDC 数据平面设备的操作。除此之外，为了将高级 SDC 程序转为高效的 SDC 数据面配置，我们采用了一个决策图的数据结构和一个高效的配置框架。转化后的配置允许 SDC 网络中的数据平面设备互相交换本地状态以达到 SDC 网络资源（例如，设备数据包处理以及设备之间数据传输能力）的高效利用。

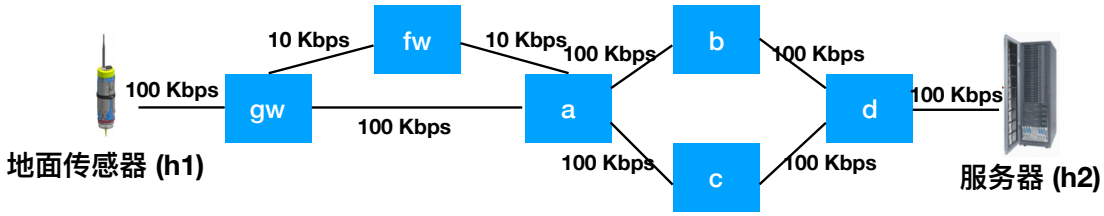
5.2 相关工作以及研究动机

相关工作: 近期一些 SDN/SDC 数据平面的工作被提出[arashloo2016snap, heorhiadi2016simplifying, soule2014merlin, benet2018mp, katta2016hula, gember2012stratos, anwer2015programming, monsa

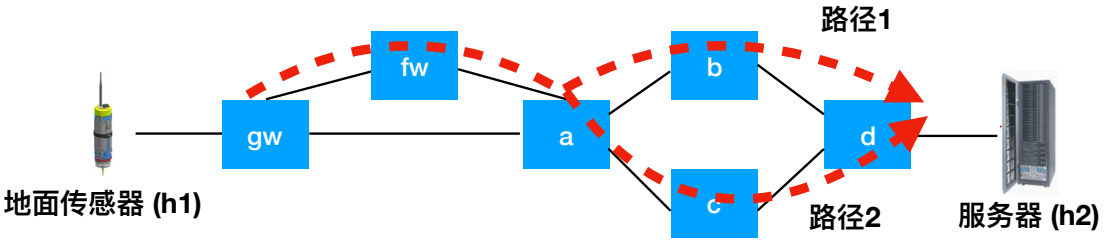
它们主要的思想是将对数据包进行复杂有状态的操作（例如，计数、流安全探测、拥塞控制）从控制平面（例如，基站）卸载到数据平面设备（例如，移动设备）上，从而提高在动态战术环境中 SDN 应用的性能。在这些设计中，被卸载的对数据包操作的结果独立地存储在每个数据平面设备上。SOL[heorhiadi2016simplifying] 和 Merlin[soule2014merlin] 在受限的网络状态环境下，通过计算路径，来解决数据平面设备的放置以及配置问题。P4CEP[kohler2018p4cep] 和 OpenSDC[opensdc] 考虑拓展数据平面的能力，从可以进行简单的数据包处理到可以进行事件处理。SNAP[arashloo2016snap] 设计一个高级网络编程系统，从而将一个高级程序转为有状态数据平面设备的配置。我们可以看出，尽管以上主要工作均关注 SDC 数据平面的设计，但这些系统的一个共同的限制是数据平面设备的本地状态无法共享给其他设备。下面的一个例子显示在 SDC 网络中这种情况会导致资源的不充分利用，进而影响 SDC 应用的性能。

为了允许在数据平面设备之间共享本地状态，DDP[ddp] 对于分布式数据平面更新设计了一些原语。Hula[katta2016hula] 和 MP-HULA[benet2018mp] 考虑的场景是数据平面设备上的负载均衡应用，并设计了一个发送以及接受探测数据包的机制以更新设备设备上的本地状态。然而，这些系统均需要用户根据不同应用手工地配置数据平面低级的原语，从而花费大量的时间并非常容易出错。

研究动机: 我们这里会通过一个例子来证明目前系统的限制，以及共享本地状态的好处。我们考虑图 5.1(a) 的一个战术网络，其中包括一个地面传感器（h_1）一个防火墙中间盒（fw）、一个计算服务器（h_2）、一个网关交换机（gw）、多个转发交换机（a-d）。链路 $gw \rightarrow fw$ 以及 $fw \rightarrow a$ 的带宽为 10 Kbps，而其他的链路均为 100 Kbps。传感器 h_1 将数据收集后，首先转发到防火墙 fw 并基于数据包的五元组（即，srcAddr、dstAddr、srcPort、dstPort、protocol）判断该数据是否为敏感数据。同时 SDC 数据收集应用程序的目标是将传感器收集上的数据传到



(a) 网络拓扑



(b) 没有本地状态共享下的数据平面配置

图 5.1　研究动机示例：一个 SDC 数据收集应用在一个具有防火墙的网络中。

服务器上。而网络管理员对于数据传输有如下策略要求：对于任何从 h_1 传到 h_2 的数据，如果是敏感的，则数据应经过交换机 b 传输；否则经过交换机 c 传输。

为了保障这样的策略要求，目前已有的数据平面系统（例如，SNAP[parashloo2016snap]）会给出如图 5.1(b) 的数据平面配置（这些数据平面系统均不支持在数据平面设备之间共享本地状态）。网关交换机 gw 将所有数据包转发至防火墙 fw；防火墙 fw 根据其五元组判断数据流是否敏感，并根据判断结果添加相应的标签给所有的数据包，最后将数据包转发至交换机 a；交换机 a 匹配数据包上的标签，并根据匹配结果转发至交换机 b（沿着路径 1）或交换机 c（沿着路径 2）。

虽然该配置是正确的，但其并没有充分利用 SDC 网络资源，从而影响数据收集应用程序的性能。一旦数据流是否敏感被防火墙 fw 判断后，该数据流的后续数据包则不需要再经过 fw。这些后续数据包应沿着路径 gw, a, c, d 或 gw, a, b, d 从而得到更高的带宽。然而，该设计在没有防火墙 fw 和网关 gw 和交换机 a 之间共享本地状态下无法实现。

以上例子证明了在数据平面设备之间共享本地状态的好处。然后为了实现共享本地状态，手动地进行低级网络配置是花费时间并且非常容易出错的。因此，我们设计 Dandelion，一个 SDC 编程系统来自动地从高级 SDC 程序转为支持共享本地状态的数据平面配置。

5.3 Dandelion 系统概述

本节中，我们首先给出 Dandelion 的编程模型，再描述 Dandelion 的架构以及从高级 SDC 程序到支持共享本地状态的数据平面配置的转化流程。

5.3.1 编程模型

由于这里考虑的场景符合上一章中的部分可编程网络，即网络中包含具有特定、固定功能网络节点，如中间盒等，因此 Dandelion 采用上一章中提出的中间盒的原语，以及引入中间盒处理的抽象语法。

在 Dandelion 系统中，交换机和中间盒的交互是双向的。通过一个隧道，一个交换机可以将一个数据包送到一个中间盒并进行处理。待处理完数据包后，中间盒可以将数据包返回给交换机并与多个其他交换机共享该数据包的本地状态。而该过程对于用户是透明的，即用户不需要手动地对数据平面进行配置实现设备之间本地状态的共享。在下一节中我们会给出 Dandelion 是如何自动地将高级 SDC 程序转为支持共享本地状态的数据平面配置。

Dandelion 程序示例：对于章节 3.3的例子，网络管理员可以写如下 Dandelion 程序，来实现根据数据包是否携带敏感数据选择不同路径转发的策略。

```
L1: mFW = middlebox("firewall", "stateless")
L2: PATH1 = h1 -> b -> h2 //b: waypoint
L3: PATH2 = h1 -> c -> h2 //c: waypoint
L4: //any: picking any path
L5: def onPacket(pkt):
L6:   if pkt.srcAddr == h1 & pkt.dsrAddr == h2:
L7:     if mFW.handle(pkt) == SENSITIVE:
L8:       return any(SPC.stable + PATH1)
L9:     else:
L10:      return any(SPC.stable + PATH2)
L11:  else: return DROP
```

图 5.2　对应研究动机例子的 Dandelion 程序。

在上一章节中我们已经讨论了程序正确性的问题，并引入了 SPC 的概念，所以这里我们不对该程序进行过多说明。值得注意的是，由于 Dandelion 希望通过本地状态的共享，实现网络资源的充分利用。这会导致在系统的不同状态时，数据流会沿着不同路径传输。因此，这里需要对 SPC 进行拓展，引入稳定 SPC（`SPC.stable`）的概念。关于 `SPC.stable` 的定义会在随后章节中说明。

5.3.2　系统架构以及工作流程

图 5.3给出了如何从高级 Dandelion 程序到支持共享本地状态的低层次数据平面配置的架构以及流程。给定一个高级 Dandelion SDC 程序，Dandelion 首先计算出对应的决策图。该决策图包含了程序对数据包的决策过程以及相应的有状态操作。其次，Dandelion 根据系统优化目标（例如，吞吐量最大化），算出网络中的转发路径。需要注意的是，只有当程序中返回的路径不是具体路径时（例如，使用 *any* 函数计算路径），系统才需要计算路径。而当返回的路径为具体路径时（例如，使用 *opt* 函数计算路径），系统不需计算路径。最后，生成相应的设备级别数据平面配置，并包含共享本地状态的配置以实现设备之间可以共享本地状态。Dandelion 利用有状态交换机来实现共享本地状态。有状态交换机的基本模型为两张匹配操作表（match-action）：第一张为状态表，即匹配数据包头并返回状态（$match \rightarrow state$）；第二张为匹配表，即匹配数据包头以及状态并返回操作（$match + state \rightarrow action$）。当数据包到达一个有状态交换机时，首先进入状态表并根据数据包头来获取相应的状态，再进入匹配表来得到操作。

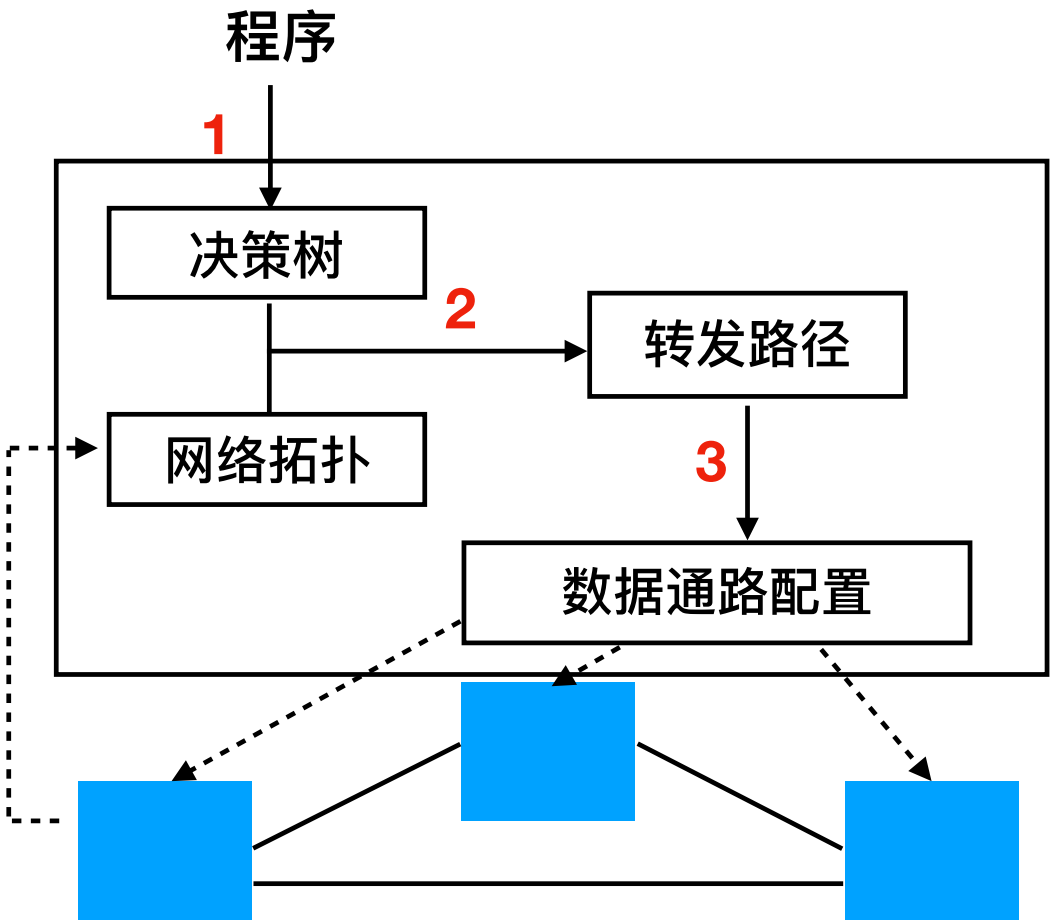


图 5.3　Dandelion 架构以及工作流程。

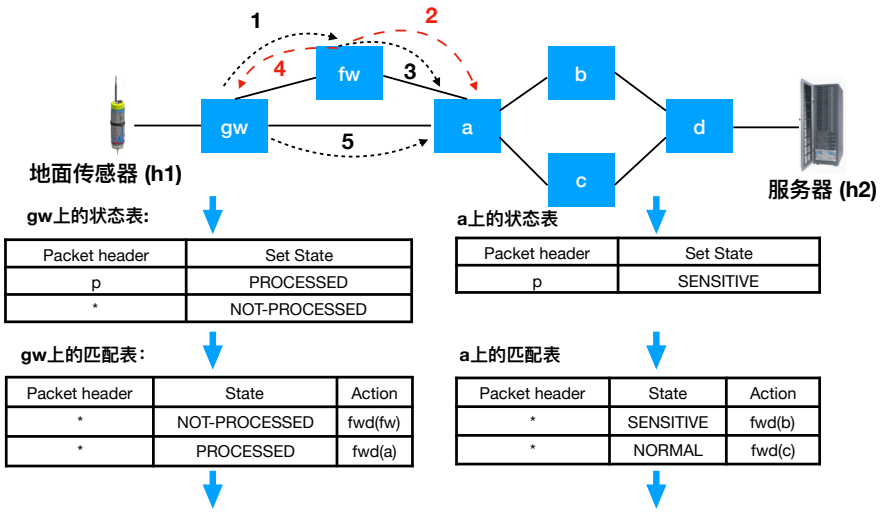


图 5.4　从图 5.2中程序转换后的配置。

支持共享本地状态的配置示例：图 5.4给出了从图 5.2的 SDC 程序生成的支持共享本地状态的数据平面配置。具体来说，通过允许共享本地状态，防火墙 fw 可以将对数据流标识的状态传给网关交换机 gw。例如，它可以发送如下状态信息：$srcAddr = h1, dstAddr = h2, srcPort = 12345, dstPort = 22, protocol = tcp \rightarrow PROCESSED$ 给 gw。网关 gw 可以将其设置在状态表中。

根据此配置，第一个到达网关 gw 的数据包 pkt 会被传送到 fw 来得到状态 *NOT-PROCESSED*（第一步）。假设 fw 识别 pkt 为 *SENSITIVE*。则 fw 会首先发送该状态信息 $pkt \rightarrow SENSITIVE$ 给交换机 a（第二步）。其次，fw 会转发 pkt 到 a（第三步）。在交换机 a，由于 pkt 的状态为 *SENSITIVE*，pkt 会转发给 b。最后，fw 共享本地状态 $pkt \rightarrow PROCESSED$ 到网关 gw（第四步）。而随后的与 pkt 具有相同匹配字段的数据包到达 gw 后会直接转发到交换机 a（第五步）。

5.4　Dandelion 设计细节

在本节中，我们给出 Dandelion 的细节部分：决策图，路径计算，以及数据平面配置的生成。

5.4.1　决策图

决策图（Decision Graph，DG）用来捕获数据包的转发决策过程以及对数据包的有状态操作。我们定义 DG 为一个单根有向无环图。该图有三种节点：数据包（packet-test）测试节点，中间盒处理（middlebox-operation）节点，以及操作（action）节点（分别映射三个基本原语：数据包测试，中间盒处理，以及路由代数）。数据包测试本质上与数据包的读取一样，为了便于理解，我们采用数据包

测试。前两个节点为 DG 的内部节点；后一个为叶子节点。如果一个节点为数据包测试节点，则其出边标识了数据包的域的范围；如果一个节点为中间盒处理节点，则其出边标识了对数据包的处理结果。例如，图 5.5给出了研究动机程序的相应的 DG。本工作中，我们不关注如何生成 DG。DG 的计算可以参考已有的相关工作[arashloo2016snapsmolka2015fast]。

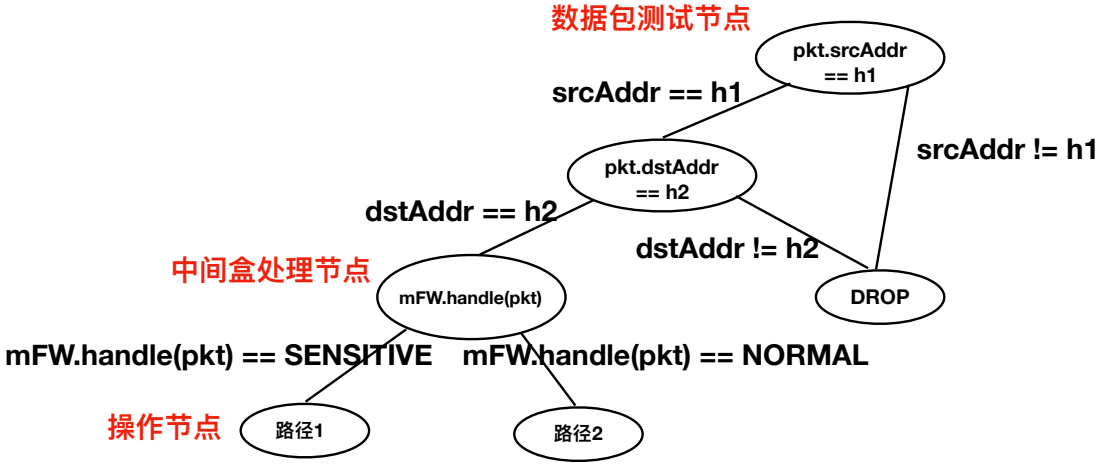


图 5.5　DG 示例。

给定一个 DG，对于从根节点到一个操作节点（即路径约束，path constraint）pc 的节点和边的序列，我们称之为 pc 的踪迹（trace），即 $T(pc)$。给定一个踪迹 $T(pc)$，我们称 $T(pc)$ 中的中间盒处理节点为 $M(pc)$。因为中间盒处理节点代表实际网络中中间盒对数据包的处理操作，为了不形成环，在 $M(pc)$ 中每个中间盒节点只可以出现一次。并通过从 $M(pc)$ 抽取代表有状态中间盒的中间盒操作节点，我们可以得到一个 $M(pc)$ 的子序列（称为 $M^s(pc)$）。该子序列 $M^s(pc)$ 中的每个节点代表有状态中间盒的操作。给定一个 $T(pc)$，对于其操作节点的 SPC 可以由 $T(pc)$（包含了数据包到达该操作节点的踪迹）简单地计算出来。

5.4.2　路径计算

由于 DG 的叶子节点代表路径约束，路径计算的目标则是根据系统性能目标对这些叶子节点计算出具体路径。（如果叶子节点为具体路径，则跳过路径计算过程。）

数据包转发模型：在路径计算之前，我们首先给出在网络中的基于 DG 的数据包转发模型。如图 5.6所示，给定在研究动机例子中的一条路径（gw, a, b），逻辑上该路径上的每一个交换机都保存该示例程序的 DG。其中 P 表示数据包测试节点；M 表示中间盒处理节点；A 表示操作节点。红色的 A 节点表示该路径的相应的约束。如图红色线所示，到达网关 gw 的数据流的第一个数据包开始遍历

DG。当数据包遇到一个中间盒节点时，通过一个隧道，它会被送到相应的中间盒。待处理完成后，中间盒会将数据包返回并共享其本地状态（即添加或修改状态表的流规则）给路径沿途上的交换机的相应的中间盒处理节点（每个节点可以被看作为一对状态和匹配表）。如果中间盒为有状态中间盒，则不设置状态，而让数据包携带状态。待遍历完 DG 后（即沿着红色线到达叶子节点），数据包得到一条路径并根据路径进行转发。由于本地状态已经共享给了其他交换机的相应的中间盒处理节点（或对于有状态中间盒，则携带在数据包上），数据包在随后的交换机中无需再送给中间盒进行处理。

如果数据包没有经过任何无状态中间盒（相比第一个数据包需要经过这些无状态中间盒），我们称该数据包的转发为稳定的（stable）。根据稳定转发的定义，我们拓展原始的 SPC 变量为 `SPC.stable`。该拓展后的变量表示只保留稳定转发路径的约束（即去除无状态中间盒）。因此，为了支持共享状态，由于 `SPC` 包括了所有数据包（含第一个数据包要经过的所有中间盒）的全部的约束，程序中应该使用 `SPC.stable` 而非 `SPC`。

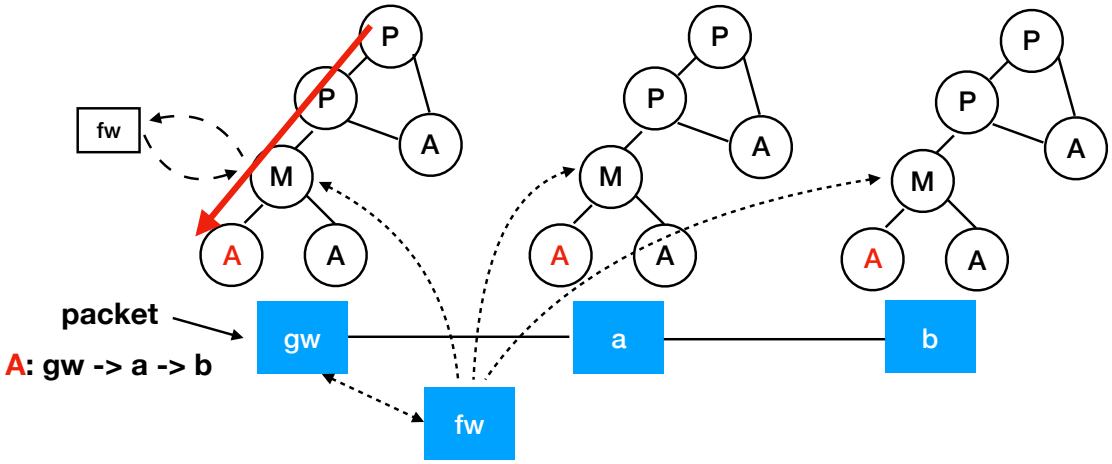


图 5.6　基于 DG 的数据包转发模型。

系统目标：基于转发模型，由于数据传输最初的一些数据包对整个数据流传输的影响较小，一个简单的路径计算（即只考虑稳定转发）如下所述。我们考虑系统目标为吞吐量的最大化，即，maximize $\sum_i b_i$。其中 i 表示路径约束 pc_i 的标号，b_i 为其吞吐量。表 5.1给出了变量的定义。表 5.2给出了约束（其中 $Path(x, y, E)$ 表示 E 中存在一条从 x 到 y 的简单路径）。本工作中，对于路径约束，我们关注基准点（waypoint）约束（因为这种约束会给路径计算带来复杂性）。我们考虑基准点约束为一组节点对。节点对的形式为 (x, y)，表示数据包要先经过 x 再到 y。（x 和 y 可以为源或目的节点。）

由于通过使用 `SPC.stable`，系统路径约束已经添加到了路径约束中，正确性已经可以保证。即数据包先得到有状态中间盒的处理结果，然后选择正确路

变量	含义
u_i, v_i	路径约束 pc_i 的源和目的节点
E	网络中所有的边（其中一个边表示为 e: $(e.src, e.dst)$）
m_e	边 e 的最大带宽
W_i	pc_i 的一组节点对
z_e^i	pc_i 选出的边 e
b_i	使用 pc_i 的流的带宽

表 5.1　变量及其含义。

约束	含义
$\forall i, \forall (x,y) \in W_i, Path(x,y,E)$	路径约束
$\forall i, Path(u_i, v_i, E)$	路径存在于 E
$\forall e \in E, \sum_i b_i * z_e^i \leq m_e$	边的带宽约束

表 5.2　约束及其含义。

径。

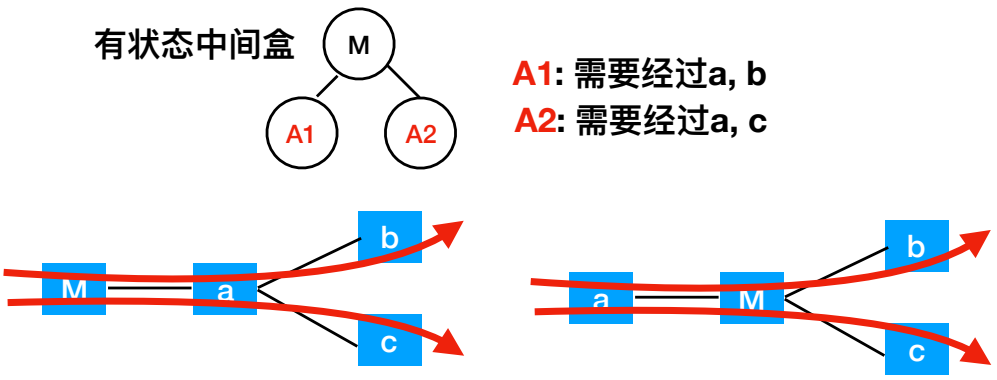


图 5.7　两种转发路径均满足要求。

然而，当前的路径约束可能会导致 ‘‘过度约束" 的问题。例如图 5.7，一个有状态中间盒处理节点 M 有两个路径约束 A_1 和 A_2 作为其 DG 中的子节点。并且 A_1 要求路径要经过交换机 a, b；A_2 要求路径要经过交换机 a, c。根据之前的讨论，对于 A_1 和 A_2，均包括路径约束 (M, a) 因为它只是简单地将 `SPC.stable` 中的基准点约束与 a, b 进行连接（同时与 a, c 进行连接）。然而，它只是一个充分条件，因为其忽略了其他满足要求的路径（例如，$a \rightarrow M \rightarrow b\ (c)$）。

基准点约束计算：相比简单地连接两个基准点约束，这里我们给出一个可以保证路径正确并且减少了忽略的路径的基准点约束计算。其计算基本思想则是对 DG 进行深度优先的后续遍历。该遍历只关注有状态中间盒节点和操作节点。当遍历到一个有状态中间盒处理节点时，我们已经得到在其决策下的所有可能的基准点约束。随后，对该点的处理为更新这些约束（每一个约束可以看作为一个有向

无环图，Directed Acyclic Graph, DAG）。上文已经说明，我们不希望添加过度约束，则更新方法如下：假设对一个中间盒处理节点 M 的处理开始前，M 的所有的约束（即 DAG）中入度为零的节点上都有一个指针，当处理 M 时，如果 M 的所有的指针指向的节点都相同（即所代表的实际网络节点相同），则同时对所有的 DAG 移动指针跳过这些节点，直到至少存在一个不相同。然后将边 (M, x) 添加到所有 DAG 中（x 表示最终指针指向的位置）。其思想是，当所有可能的路径的下一个节点都相同时，约束中跳过该节点也是正确的。如图 5.8所示，节点 M 下面有四个路径约束（DAG），而在更新时节点 a 应该被跳过。

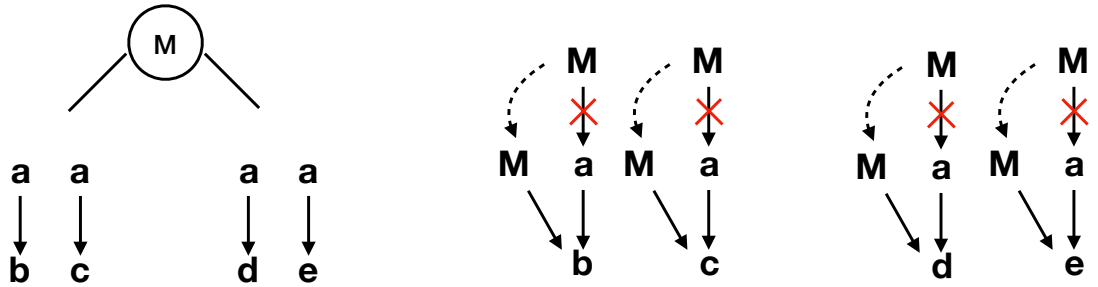


图 5.8　Skip the same nodes.

5.4.3　数据平面配置生成

删除 DG 中冗余节点：完成路径计算后，DG 中的每一个叶子节点都有一个具体路径。基于数据包转发模型，网络中所有交换机上的 DG 都相同。不难发现，如果一个交换机不属于一条路径，则相应的叶子节点应该被删除（同时删除出度为零的中间节点）。同时，通过将路径转化为交换机的下一跳，我们可以进一步删除冗余节点。考虑防火墙的例子，当路径转化为下一跳后，我们发现中间盒节点的两个叶子节点相同，即网络中的交换机 a。这意味着该中间盒处理节点对于网关 gw 没有任何意义，因为不管处理结果如何，其下一跳不变。因此，我们可以删除该节点并替换为下一跳为 a 的叶子节点。图 5.9给出了该过程。

对于交换机，基于更新后的 DG，并根据多流表 pipeline 的架构，其流表的结构以及流表内容可以简单地生成。其中一个中间盒处理节点可以看作一对状态表和匹配表。对于中间盒，如果它是无状态中间盒，则对于其数据平面，只需要配置交换机到中间盒的隧道；如果它是有状态中间盒，由于它处在计算出的转发路径之中，因此可以看作是有状态交换机，同时可以将结果嵌入到数据包中。当一个数据包在一个交换机的 DG 中遇到一个有状态中间盒节点，并且该节点无法被叶子节点替换时，该节点下的任意下一跳都是可以接受并正确的，因为它最终会到达相应的中间盒，并且从该交换机到中间盒的任意一条路径一定满足路径约束。

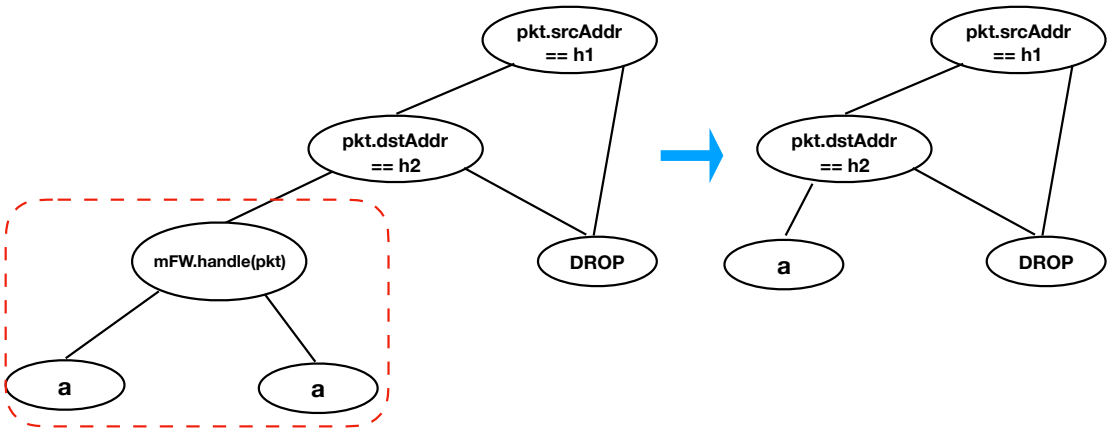


图 5.9　删除冗余中间盒节点。

消息次序：当无状态中间盒共享本地状态（即更新状态）到交换机时要考虑消息的次序问题。对于目标的交换机，需要保证更新消息要比数据包提前到达。例如，在防火墙例子中，只有第二步完成后，第三步才可以执行。因此一个简单的做法是让数据包携带更新消息。而与有状态中间盒的数据包携带状态不同的是，该消息可以添加或修改相应中间盒处理节点的状态表。

5.5　实验评估

本节中，我们先分别从时延和吞吐量两个方面验证 Dandelion 的优势。然后给出基准点约束计算部分的评估测试。实验环境是 3.5 GHz Intel i7 处理器、16 GB 内存、Mac OSX 10.13 系统。

实验方法：首先我们随机生成一个具有 25 个节点和 50 条边的网络。对于网络中的每一个边，我们设置两个随机值，分别代表时延（5 - 10 ms）和带宽（5 - 10 Mbps）。为了模拟数据流，我们在网络中随机地选择两个点分别作为数据流的源节点和目的节点。同时一个数据流可以有一个网络中的节点序列（除了本身的源和目的节点）作为其数据包处理需要的有序的中间盒节点。对于中间盒节点的选择，我们添加如下约束：中间盒节点的邻居节点数量为 2。这是因为普遍的中间盒没有路由功能，对于收到的数据包，其转发接口只有一个。作为 Dandelion 的对比，我们考虑一个传统的方法，即路径的计算对于中间盒不区分有状态或无状态。因此，该方法需要保证计算出的路径必需按正确顺序经过所有中间盒。而对于 Dandelion 中的路径计算，我们在这些中间盒中随机选择一些作为有状态中间盒，则其余为无状态中间盒。

时延：为了验证对于时延的优势，我们考虑单个数据流并变化其需要的中间盒数量。目标为计算出具有最低时延的路径。然后，我们在应用 Dandelion 和不应用 Dandelion 之间比较时延结果。

	(F=5, M=1)	(F=5, M=3)	(F=5, M=3 (S=1))
With Dandelion	1.3 (s)	1.4 (s)	4.2 (s)
Without Dandelion	4.5 (s)	10.8 (s)	11.5 (s)

表 5.3 The execution time of path computation to maximize total throughput for different scenarios.

吞吐量：为了验证对于吞吐量的优势，我们考虑多个具有相同的中间盒需求的数据流并变化中间盒的数量。目标为计算出具有最大吞吐量的路径。然后，我们在应用 Dandelion 和不应用 Dandelion 之间比较吞吐量结果。

执行时间：为了评估 Dandelion 的性能，我们分别计算两种方法（即应用 Dandelion 和不应用 Dandelion）中在计算最大吞吐量场景时的路径计算部分执行时间。

基准点约束计算：为了验证基准点约束计算的优势，我们首先设置网络中的一组节点作为一个数据流的基准点序列。然后，我们考虑低时延作为系统目标并比较应用基准点约束计算和不应用（即过度约束）的结果。在过度约束中，在经过基准点前，需要经过所有的要求的中间盒节点。

结果：如图 5.10(a) 所示，通过应用 Dandelion，时延可以被有效地降低。其中 F 表示数据流的数量；M 表示要求的中间盒的数量；S 表示有状态中间盒的数量。由于对于一个数据流的最小时延计算独立于其他数据流，因此实验只考虑一个数据流的场景。从结果中可以看出，在不应用 Dandelion 情况下，随着无状态中间盒数量的增加，时延会变大；而应用 Dandelion 情况下，时延只在有状态中间盒数量增加是变大。

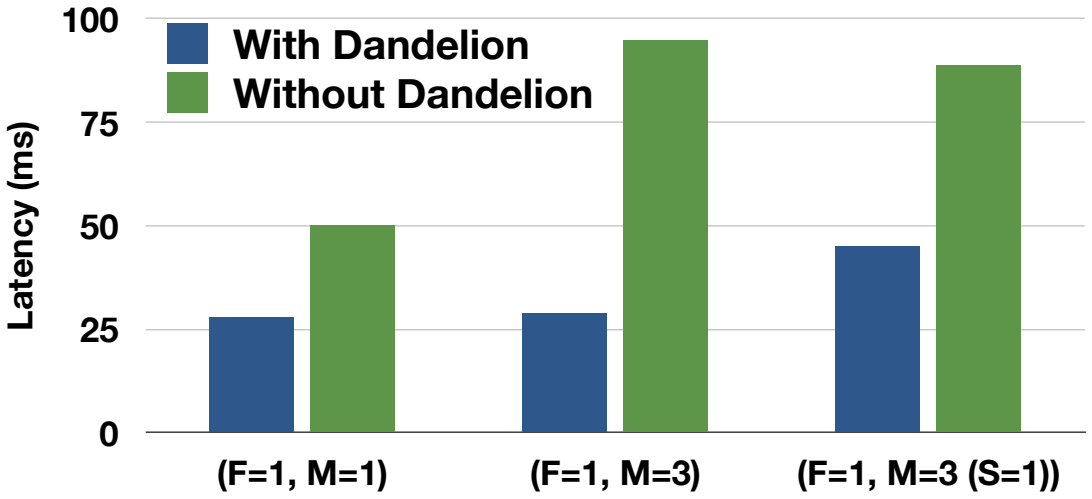
如图 5.10(b) 所示，通过应用 Dandelion，吞吐量可以有效地提高。当有五个数据流和三个中间盒时，应用 Dandelion 的吞吐量是不应用 Dandelion 的近三倍。

表格 5.3显示的是在计算最大吞吐量时路径计算部分的执行时间。由于在应用 Dandelion 情况下，约束的数量少于不应用 Dandelion 情况，因此执行时间也会相应的降低（当 F=5，M=3 时，从 10.8 秒降为 1.4 秒）

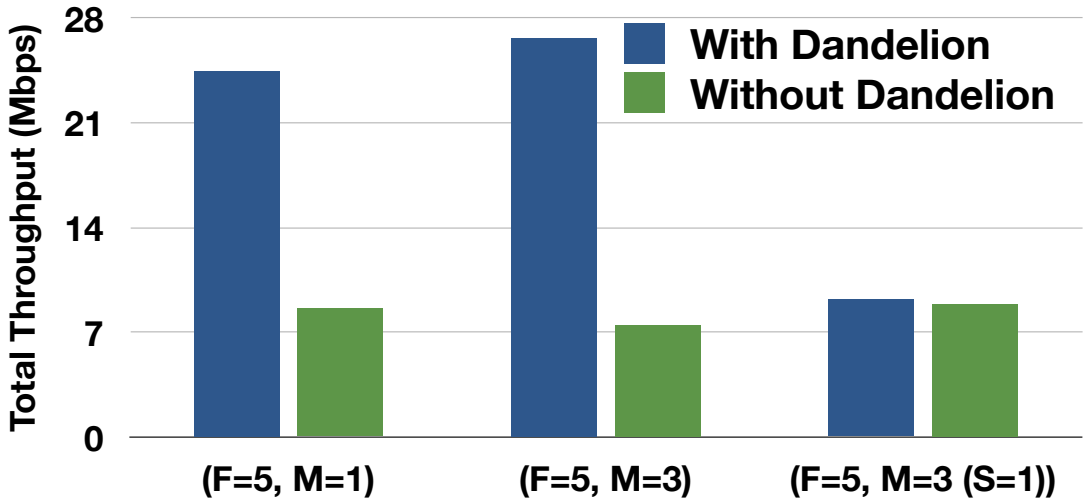
图 5.11给出的是有正确约束（即通过应用基准点约束计算）和有过度约束的时延结果。其中图 5.11(a) 对应的是只有一个基准点，而图 5.11(b) 对应的是三个基准点。从结果中我们可以看出，过度约束会增加时延，其理由是非最优的路径计算。而随着基准点的数量增加，有过度约束的时延也会变大。

5.6 本章小结

本章针对 SDC 网络设计高级编程系统 Dandelion，并基于共享本地状态的方法，优化系统性能。实验表明在某些场景下吞吐量可以实现近三倍的提高。

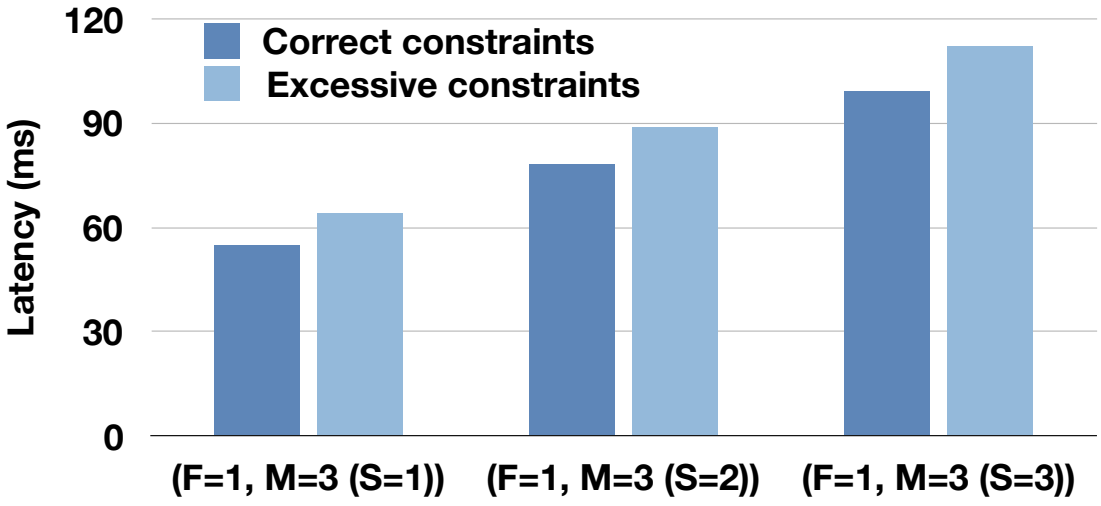


(a) 不同场景下的时延

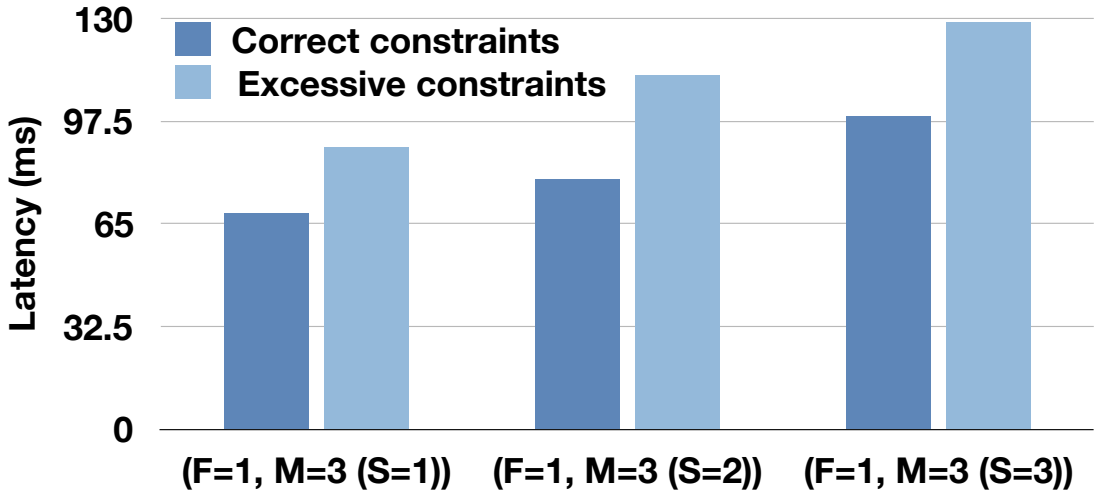


(b) 不同场景下的吞吐量

图 5.10　Dandelion 对于时延和吞吐量的优势。



(a) 基准点约束中有一个节点



(b) 基准点约束中有三个节点

图 5.11　不同基准点约束下的时延。

第 6 章　sdiv 章节

6.1　引言

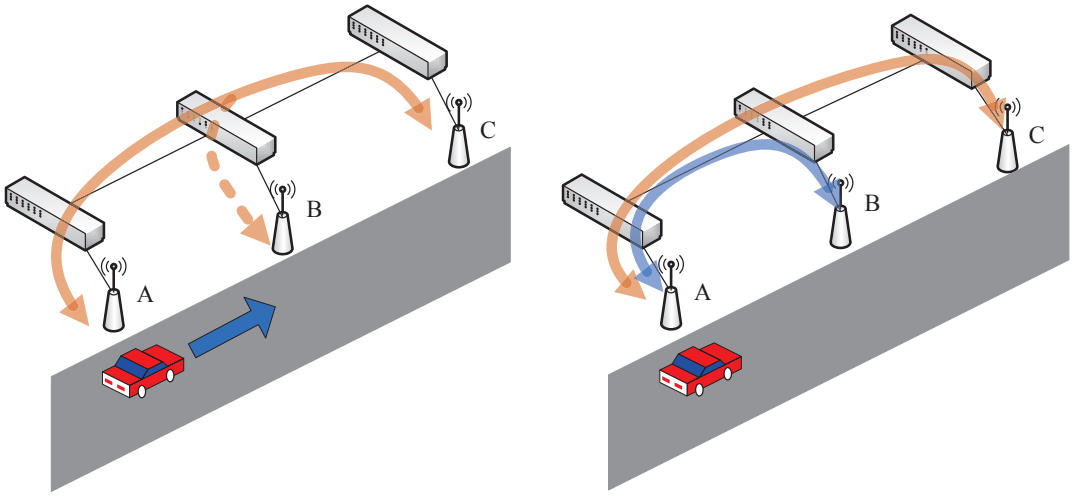
车联网（Internet of Vehicles，IoV）目前受到了学术界和工业界的巨大关注。通信技术和智能城市的巨大发展给 IoV 带来了更多可能的服务，提高了车辆驾驶的质量和安全性。为了 IoV 的进一步发展，目前已有多个研究和工业组织在实施各种相关项目[CVIS, makino2005smartway]。例如，欧盟领导的 CVIS[CVIS] 致力于发展车与车之间以及车与路旁设施之间的通信技术；Smartway[makino2005smartway] 的目标是集成多个智能传输系统并提供一个统一的平台。

虽然 IoV 有着很好的发展前景，但其中网络设备的封闭性阻碍了 IoV 上新服务的快速部署和更新。网络设备（如交换机、路由器）目前由不同的厂商开发，设备的更改需要专业人员的手工配置，使网络管理变得昂贵并且易出错。由于缺少一个开放和统一接口，无法实现网络的灵活和动态的配置，从而影响大规模 IoV 上新服务的部署和拓展。为了 IoV 的发展，一个新的网络架构是必要的。

关于新的网络架构以及下一代互联网，目前有如下相关的研究。命名数据网络[NDN] 的目标为开发一个新的互联网架构。与传统互联网的从哪里获取服务相比，命名数据网络关注获取什么服务。为了实现移动过程中的无缝隙数据传输，MobilityFirst[MobilityFirst] 将节点的移动性作为一个普遍场景进行设计。NEBULA[NEBULA] 则设计了一个基于云计算和数据中心的新的网络架构。SDN 将控制层和数据层进行了分离，并提供了配置网络的统一的接口。一个统一的配置网络设备的接口，使大规模可灵活配置的网络变得可行，从而可以加速 IoV 上新服务的部署。

考虑其开放和统一的接口，我们采用 SDN 作为 IoV 中底层网络架构，并提出软件定义车联网（Software-Defined Internet of Vehicles，SDIV）的架构。除了其开放统一的接口，SDIV 在以下方面有较大优势：1. 通过将控制平面和数据平面的分离，SDIV 有着非常高的可拓展性；2. 通过逻辑上集中式的控制器，SDIV 简化了网络的管理；3. SDIV 中的控制器可以根据当前网络状况计算数据传输的最优路径并协调车与路旁电子设备之间的通信。

关于 SDIV 上的编程框架，由于车辆的移动性，我们提出将底层每一个交换机的数据通路分为两个部分：面向静态转发路径部分和面向动态转发路径部分。其中静态部分则由高级 SDN 程序编译生成（该部分不会随着车辆移动而变化）。对于动态部分，由于车辆的不断移动，提前生成流水线结构以及流表项并不是一个好的设计。因此 SDIV 将动态转发路径部分设计为单流表的响应式下发流规



(a) (b)

图 6.1 （a）：当车辆从 *A* 移到 *B* 时，它需要重新建立连接。虚线表示连接尚没建立；（b）当车辆同时连接多个摄像头时，对于每个连接建立路径下发规则并不高效由于所有的路径目的一样（即所有数据流目的地址为 *A*）。

则。鉴于单流表的有限的流表容量以及 IoV 中的移动特性，如何构建紧凑的流表是 SDIV 的主要挑战，也是本章的主要技术内容。

本工作中，我们考虑 IoV 中的实时查询服务，来给出 SDIV 中优化的流规则设置细节。通过实时查询服务，驾驶员可以获取实时的道路或其他信息来选择正确的驾驶方向。道路信息来自于路旁的监控摄像头并传给有需求的车辆。在此场景中，具体问题如下：1. 由于车辆的移动，车辆和摄像头之间需要不断建立新的连接，从而增加了流规则的数量；2. 当车辆与多个摄像头进行连接时，由于它们具有相同的目的，对每一个连接设置流规则并不高效。图 6.1描述了上述的问题。如果控制器简单地对每一个请求都下发流规则，则交换机中流表容量会成为系统瓶颈，并最终影响实时查询服务的性能。

为了解决实时查询服务中流规则设置问题，通过借助 SDN 的集中式细粒度的流控制，我们提出了以下技术：1. 为了减少车辆发出的请求数量，我们在数据从摄像头到车辆的最后一跳时采用多播地址；2. 为了保证在车辆移动过程中数据传输不会中断，我们根据车辆的行驶状况提前在交换机中设置流规则；3. 为了减少流规则数量以及保证数据传输的正确性，我们在分支节点更改数据包头。

6.2 相关工作

目前 IoV 中的大多数服务，例如，道路安全、车辆管理、车辆导航等，依赖于车与车（V2V）之间的数据传输以及车与设备（V2I）之间的数据传输[du2015information]。已有一些研究工作验证了通过路旁 AP 以及 WiFi 提供数据的连

通性[ott2005disconnection, balasubramanian2008interactive]。

Hare 等人[hare2012policy] 设计一个集中式的策略框架来管理车辆网络的光谱资源来保证用户需要的数据传输性能。Abadi 等人（XXX）根据有限的交通信息，对交通流量进行预测以及交通拥塞控制。Wu 等人[wu2013improving] 结合 GPS 信息提高了对车辆姿态估计的准确度。Ahn 等人[ahn2012risa] 第一个提出了面向车辆网络的分布式路面状况探测以及数据分发的架构。Calafate 等人[calafate2012efficient] 研究通过计算最优数据包大小以及决定最优前向纠错，实现广播下的可靠数据传送机制。

Grassi 等人[grassi2014vanet] 通过将命名数据网络应用于车辆网络中，实现了所有计算设备上的互通。并且设备的互通独立于它们以何种方式连在一起，如有线网路，ad-hoc 网络，或延迟容忍网络。Yap 等人[yap2010blueprint] 通过将网络服务与底层设施的分离，提出 OpenFlow 无线网络以实现任何无线设施之间的互通。

Kim 等人[kim2013improving] 考虑在校园网络以及家庭网络中利用 SDN 优化网络的管理。Kanizo 等人[kanizo2013palette] 提出一个分布式的框架对较大的交换机流表进行分解来应对交换机流表容量受限问题。Voellmy 等人[voellmy2013maple] 设计 Maple 系统来发现可重用的转发策略并通过一个踪迹树的结构记录对数据包的操作来减少下发的流规则数量。

从以上讨论可以看出，虽然目前有很多 IoV 的研究工作以及相关的 SDN 的应用，但并没有考虑新服务的部署以及通过集中式的控制器优化 IoV 中数据传输。因此，我们提出 SDIV 的架构。

6.3　SDIV 架构以及工作流程

本节中，我们首先给出 SDIV 架构，并通过实时查询服务来说明其工作流程。

6.3.1　SDIV 架构

如图 6.3所示，我们提出的 SDIV 网络是一个具有三层的架构。从下至上分别为，物理层，控制层，以及应用层。

6.3.1.1　物理层

物理层包括车辆，接入点（Access Point，AP），路边电子设备，交换机，以及服务器（区别于 SDN 中的控制器）。车辆作为移动的节点通过附近 AP 与服务器进行连接。这里我们假设 AP 的数量足够多并可以覆盖所有道路。路边电子设备如摄像头收集完路况信息后需要将数据传送到服务器或车辆。交换机连接着

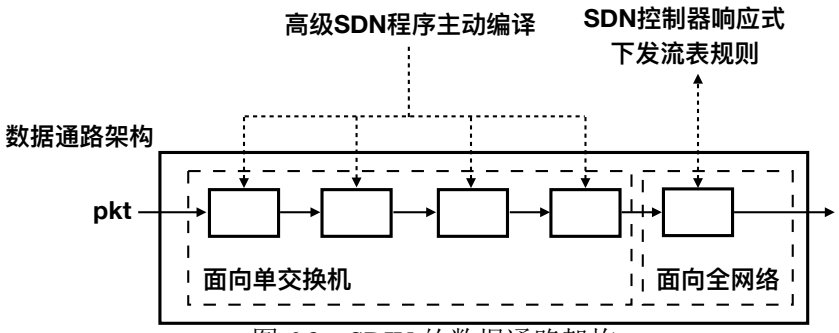


图 6.2　SDIV 的数据通路架构。

AP，摄像头以及服务器。当收到车辆发送的请求时，服务器会将相应的信息传送给车辆。一辆车可以同时连接多个服务器。

关于交换机数据通路设计，如图 6.2所示。我们提出将每一个交换机的数据通路分为两个部分：面向静态转发路径部分和面向动态转发路径部分。其中静态部分则由高级 SDN 程序编译生成。该部分具体的应用可以是：IoV 中设施之间的通信（如摄像头到服务器）或实现对数据包的非全网路由的操作（如 ACL）。对于面向动态转发路径部分，由于车辆的不断移动，提前生成流水线结构以及流表项并不是一个好的设计。无法提前生成流表项可以很好理解，因为节点的移动性，流表项应该不断变化。对于流水线方面，由于车辆的不断移动，若对动态转发路径部分使用流水线结构，则要求网络中所有节点都为相同的流水线结构。该做法既浪费网络资源，也会增加数据包时延，因为数据包经过流水线的时间与流表数量成正比。因此 SDIV 将面向动态转发路径部分设计为单流表的响应式下发流规则。

6.3.1.2　控制层

利用 SDN 技术，控制层中的控制器连接着网络中的每一个交换机（包括 AP）。通过下发 OpenFlow 流规则，控制器可以控制 IoV 中数据的传输。当网络状态发生改变时，交换机会发送相应的消息至控制器，使其获取最新的全局网络状态。通过全局网络状态信息，控制器可以将上层应用的策略，如路径选择或接入控制，转化为特定交换机上的 OpenFlow 流规则。

6.3.1.3　应用层

所有的 IoV 应用程序组成了 SDIV 应用层。应用程序复杂给 IoV 中的车辆提供服务，如实时查询服务，位置服务等。每个应用从控制层的控制器中获取网络信息，并根据自身策略进行决策。

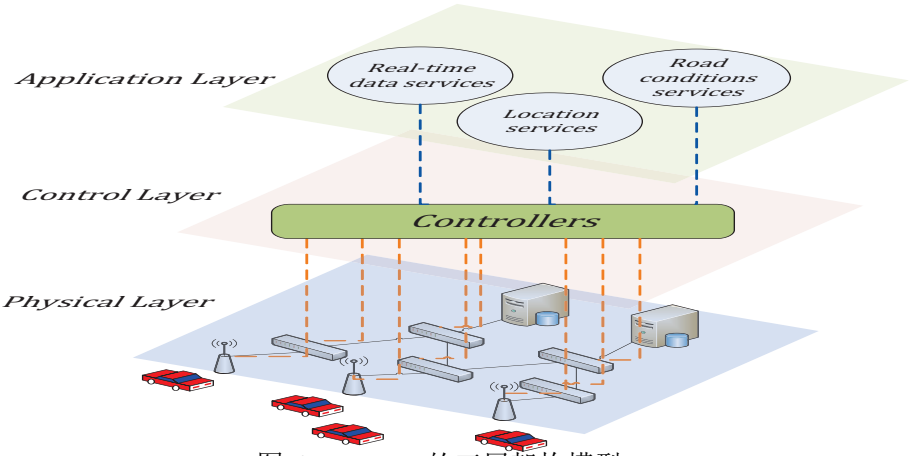


图 6.3　SDIV 的三层架构模型。

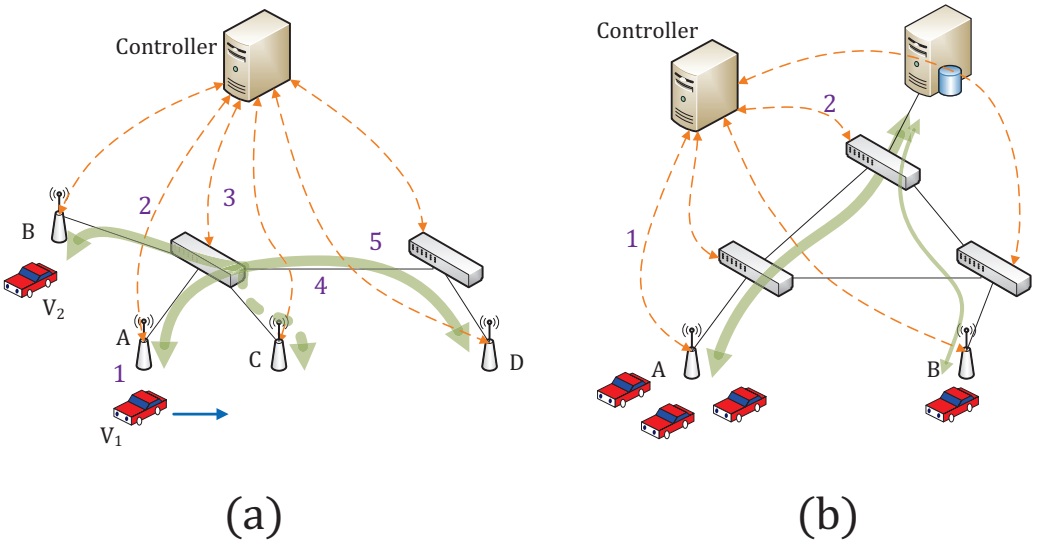


图 6.4　（a）：通过分析车辆状况，控制器可以提前下发流规则（虚线）以降低额外请求；（b）：控制器通过全局信息，实现智能带宽分配。

6.3.2　工作流程

给出 SDIV 架构后，我们这里通过两个场景来描述 SDIV 的工作流程（如图 6.4）。

图 6.4(a) 描述的是一个车辆试图接受从路况摄像头发过来的路况信息的典型场景。

第一步，车辆 V_1 发送请求至附近的路旁 AP；第二步，因为当前没有规则匹配该数据包，交换机（AP）A 转发该数据包至控制器；第三步，控制器识别数据包头，并根据车辆的请求、车辆状况（例如，位置，速度、方向等）以及网络状况下发流规则；第四步，根据下发的规则，沿途的交换机转发数据包至目的节点 D（即路况摄像头）；第五步，从摄像头 D 传送过来的数据经过上述二至四的步骤最终到达车辆 V_1。当车辆的数量增加时（即车辆 V_2 出现在了 B 附近），我们需要一个可拓展的方法去下发流规则，而非简单地重复上述步骤。同时，为了避

表 6.1 Pros and cons of traditional network technologies (multicast) and SDIV

传统技术	SDIV
劣势: 周期性广播消息	优势: 响应式方式
劣势: 路由器上保存 (S,G) 项	优势: 当需要时设置
劣势: SPT 不符合车辆行驶路线	优势: 根据车辆状态预测行驶路线
优势: 不需要控制器	劣势: 需要控制器

免由于车辆移动性带来的数据传输中断，我们需要事先在 C 下发流规则。

图 6.4(b) 描述的是车辆通过附近 AP 与服务器进行连接并上传车辆信息，如速度和位置。在该数据上传场景中，控制器通过分析哪一个 AP 收到的数据包比较多，可以判断哪一个 AP 附近车辆较多，进而可以尽心更优化的资源调度。第一步，控制器分别从两个 AP 节点 A 和 B 收集数据包。当交换机收到数据包但没有相应匹配流规则时，则发送数据包至控制器。第二步，根据目的地址，控制器下发流规则到交换机中。此时，控制器可以发现 A 附近的车辆数量较多，因此可以分配更高的带宽给 A。

除此之外，在考虑大规模的多播和车辆的移动性时，SDIV 有着更大的优势。随着连接一个摄像头的车辆增多，可以想到用多播进行数据传输达到更高效的数据传输。虽然密集模式的多播在小规模网络有很好的效果，随着数据传输范围的变大，AP 上不断增多的 (S,G) 项（S：源 IP 地址，G：组播 IP 地址）以及为了建立 SPT（Shortest Path Tree）而周期性发送的广播消息限制了数据传输的进一步拓展。为了实现快速的嫁接，由于控制层和数据层的强耦合，(S,G) 项需要被每一个路由器所保留（即使当前没有被该路由器使用）。如图 6.4(a) 所示，AP 节点 B 也需要保留 (S,G) 项为了使新来的车辆 V_2 快速获取数据。车辆的移动性也给传统的网络技术带来了困难。如图 6.4(a) 所示，当车辆 V_1 从 A 来到 C 时，车辆需要重新发送请求从而导致数据接收的中断，同时也给 D 节点的摄像头带来了困难。在我们的设计中，我们通过预测车辆可能行驶的路径解决移动性问题。我们会在预测出的路径沿途的 AP 上实现下发实现多播的流规则。而沿着 SPT 的数据传输路径进行多播无法解决移动性问题，这是因为数据传输的最短路径不一定符合车辆行驶的路径。

以下表格总结了 SDIV 和传统网络技术（多播）的优劣势比较。相比传统多播方法需要周期性广播消息以搭建 SPT，SDIV 通过利用 OpenFlow，即只有当数据包没有匹配流规则时才需要下发所需规则，实现了响应式的流规则下发方式，进而去除了周期性广播消息的需求。该响应式方式也让交换机可以保留最少数量的流规则，而非传统方式中的大量的 (S,G) 项。同时，基于全局的网络状态信息，控制器上的应用可以选择车辆可能行驶的路线并提前设置流规则进而实现数据传输的连续性。

6.4　优化的流规则下发

在描述完 SDIV 的架构以及如何工作后，我们接下来会给出流规则下发问题的解决方案，并说明简单的下发流规则会导致的后果。

在实时查询服务中，如果控制器简单地根据车辆的请求下发流规则，则由于 Ternary Content-Addressable Memory （TCAM）有限的存储容量，流表的容量大小会成为系统性能的瓶颈。因此，我们需要建立一个紧凑的流表以放入更多的决策策略。对于每个数据流的建立，都需要两条流规则来分别处理发送请求和接受数据。即使我们只考虑第二个数据流，因为请求的数据流仅在开始时使用，并且可以通过 OpenFlow 交换机中的超时机制删除，我们仍然需要为每个流至少分配一个规则。似乎每个流使用一个规则对于流表来说足够紧凑，但实际上车辆会试图同时连接多个监控摄像头。因此，我们需要在每个交换机上对每个此类数据流设置规则，即使这些数据流都具有相同的目标。总而言之，我们需要减少转发到控制器的数据包数量并同时建立紧凑的流表。

图 6.5描述了一个实时查询服务的基本工作流程。假设 V_1 想要获得道路（或十字路口）E 的状况。这里，状况可以是视频或存储在路边设备中的任何其他种类的数据。第一步，V_1 向 AP 节点 A 发送请求，然后数据流转发到 E。在确认来自 V_1 的请求后，E 将数据发送到 AP 节点 A。最后，A 将数据传输到 V_1 并完成数据传输。整个过程很简单，但可能会带来显着的性能损失。如图 6.5所示，考虑另一车辆 V_2 想要获得 E 信息的情况。同时，考虑 V_2 在 V_1 的前方。当 V_2 重复与 V_1 相同的过程时，会建立另一条从 E 到 B 的路径，这使得交换机 D 对于相同的路径安装两条规则。结果，随着连接到 E 的车辆数量的增加，沿途的交换机的流表大小可能达到最大值，从而降低了实时查询服务的性能。此外，当 V_1 移动到另一个地方时，它也需要再次发送请求。

接下来，我们考虑另一个场景来说明优化的流规则下发的重要性。V_1 想要同时从 E 和 F 获取数据。交换机 B 和 D 需要设置更多规则以使数据流传输到 A，即使两个流（$E \rightarrow A$ 和 $F \rightarrow A$）具有相同的目的地。如果 V_3 请求相同的服务，这种情况会变得更加复杂。为了使 C 下面的 V_3 接收数据，需要构建一条新路径（即，相同的数据流将在三辆车辆所连接的交换机处分开，因而该交换机成为分支节点），这再次增加了流表的大小。车辆的移动性也增加了复杂性并降低了实时查询服务的性能。

为了解决规则下发问题并减缓流表大小的增长，我们考虑将无线数据平面（用于车辆和 AP 之间的通信）与有线数据平面（用于交换机之间的通信）分开，并提出目的地驱动模型用于有线数据平面。当车辆从附近的 AP 接收数据时，它们不关心数据如何在有线数据平面中传输。相反，它们只想要一个持久的连接，

即使它们的位置随着时间而变化。为了有效地进行分离，我们引入了三种技术：1. 使用多播地址作为摄像头到车辆的最后一跳地址；2. 根据车辆的状况预先在最可能的路径中安装规则；3. 当数据包到达分支节点时修改数据包头。

6.4.1 使用多播地址作为最后一跳

我们利用多播地址来解决移动性问题并增强 SDIV 的可扩展性。每个监控摄像头都有一个唯一的组播地址，可以通过传统网络中的相同方法从 MAC 地址生成。多播地址可以在从摄像头到车辆的最后一跳中用于无线数据平面中的数据传输。车辆 V_1 接收目的地址为 E 多播地址的数据包，如图 6.5所示。使用多播地址的优点是显而易见的，即在已配备规则的 AP 范围内的每个车辆可以在不发送新请求的情况下获得实时数据。此外，在实时查询服务中，必须将第一个用户请求包转发到控制器以安装规则。如果每个请求都需要联系控制器进行规则安装，则会导致控制器成为严重的瓶颈。多播地址可以有效地解决这个问题。在第一辆车通过控制器的参与并连接摄像头后，数据流开始进行多播传输（沿着到达目的地的路径），并且对同一摄像头请求的每辆新车都会直接接收数据而不用发送请求，这降低了联系控制器的频率。例如，在图 6.5中，如果有另一车辆想要查看 E 的状况，它可以在 V_1 发送请求后不用发送请求而直接接收数据。

由于 OpenFlow 交换机中 TCAM 的大小有限，我们需要从流表中删除无用的规则。在我们的设计中，我们使用 OpenFlow 中的超时机制来删除规则。对于每一流规则，交换机都会维护一个变量，该变量表示是否在一段时间 T 秒内，没有数据包到达匹配该流规则。实际上，OpenFlow 中，通过向流表条目添加 ‘‘空闲时间" 值来实现超时机制。与相等的超时时间相比，我们根据离目的地的距离在交换机中选择不同的超时值。现实情况中，随着离目的地越近，对目的地感兴趣的车辆数量会越大。因此我们在离目的地更近的交换机设置更大的超时值。例如图 6.5所示，超时值 T_i 在交换机 i 符合约束条件：$T_A < T_B < T_D < T_E$。离信息源越近，数据流的变化越小因为这些流会合并成稳定的数据流。而稳定的数据流需要比不稳定数据流具有更长的超时。

6.4.2 路线预测并提前下发流规则

为了处理 SDIV 中车辆的移动性，有必要预测车辆选择的最可能路线，然后沿路线预先安装规则，以便在车辆移动时保持数据传输不中断。预测这种路线的一种简单方法是计算最短路径，但实际上并不总是正确。我们设计算法 $PathFind(s,d,v)$（算法6.4.2），来找到拓扑中最可能的路径。这里 s 代表当前位置，d 表示目的地，v 表示车辆的方向。$A(s,c)$ 表示 s 和 c 的角度，$D(n,d)$ 表

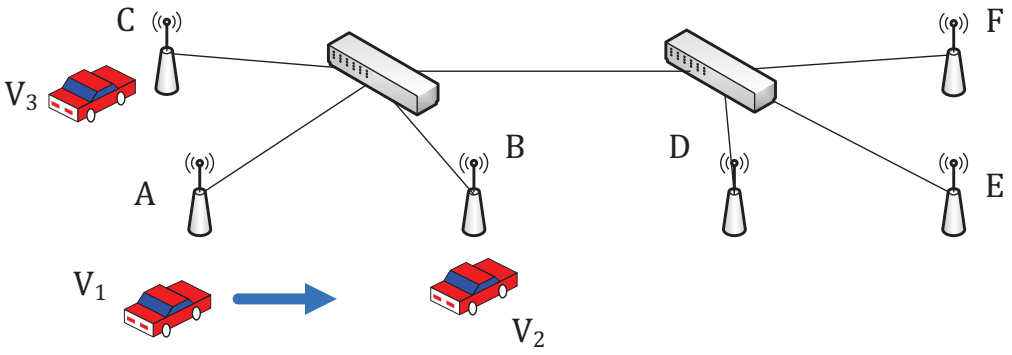


图 6.5　V_1 同时连接 E 和 F。V_2 在 V_1 到 E 的路径上并向 E 请求数据。V_3 也同时向 E 请求数据但在不同路径。

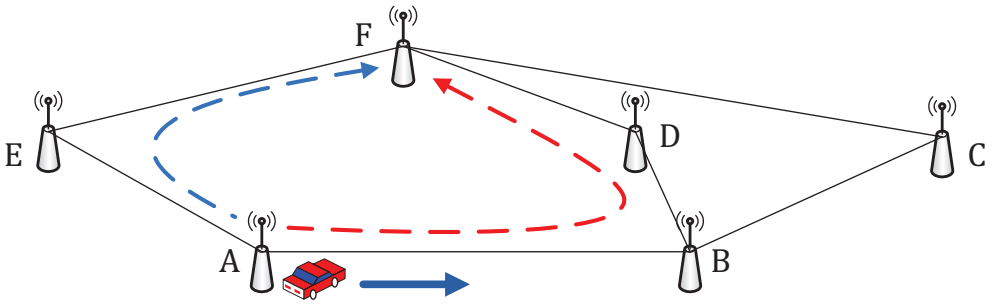


图 6.6　蓝色虚线代表当前位置到目的位置的最短路径。红色虚线代表车辆可能选择的路径。示 n 和 d 之间的欧几里德距离。

作为准备工作（第三至五行），*PathFind* 根据车辆的状况（例如，方向）过滤并得到的路径的可能的第一节点，这是因为车辆很少在街道中掉头。如图所示在十七至二十行，我们通过选择具有 $D(p,n) + D(n,d)$ 最小值（即所选位置和目的地之间的距离）的节点来递归地应用 *Find* 来计算结果路径。最后，*Find* 将目标标识为下一个节点并返回结果（第十二至十四行）。

我们在图 6.6中应用 *PathFind*。考虑一个位于 A 的车辆计划找到 F 的路线。首先，它根据当前位置的车辆方向设置 B 为 O 的值。然后发现 D 与 C 相比有更短的路径到 F。因此，它选择 D 重新开始，发现 D 直接连接到 F 并完成。结果 $A \rightarrow B \rightarrow D \rightarrow F$ 比最短路径 $A \rightarrow E \rightarrow F$ 更合理，因为实际上很少看到倒车行为。虽然路径 $A \rightarrow B \rightarrow D \rightarrow F$ 的时延比路径 $A \rightarrow E \rightarrow F$ 长，但是当车辆发现没有收到数据时，会发出更多请求。在这种情况下，车辆可能选择 B 作为其下一个位置，因此得到的路径优于最短路径，因为车辆可以在不发送另一个请求的情况下接收数据。在图 6.6中，在预测路线上的交换机上安装规则（$A \rightarrow B \rightarrow D \rightarrow E$）后，多播地址也会使 V_2 直接接收数据而不用发送一个新的要求。

6.4.3　分支节点修改数据包地址

Algorithm 1 PathFind(s, d, v)

```
1:  Algorithm PathFind(s, d, v)
2:    put s into set N, R;
3:    for each child node c of s
4:      if the angle between v and A(s, c) is less than 90 degree then
5:        put c into set O;
6:    Find(s, O, d);
7:    return R;
8:  Procedure Find(p, O, d)
9:    put p into set N;
10:   for each n in O
11:     if n is d then
12:       put n into set R;
13:       finish;
14:     put n into set N;
15:     remove n from set O;
16:     calculate D(n, d), D(p, n);
17:     if D(p, n) + D(n, d) < max then
18:       max = D(p, n) + D(n, d);
19:       r = n;
20:   put r into set R;
21:   for each child node c of r
22:     if c not in set N then
23:       put c into set O;
24:   Find(r, O, d);
25:   return;
```

在本小节中，我们将展示如何在交换机上安装规则。如算法 XXX 所示，其主要思想是我们通过利用 OpenFlow 特性在分支节点（第八至九行）添加数据包头修改操作。输入 s 表示数据源，d 表示车辆的当前位置，$path$ 由 $PathFind$ 计算生成。当数据包匹配规则并修改匹配字段时，它将在转发到特定端口之前应用修改。修改后的地址使数据包转发到直接连接到车辆的交换机。通过使用地址修改算法，我们让具有不同源地址但相同目的地地址的数据包匹配相同的规则，从而减少规则数量并减少流表大小。我们将此方法称为目标驱动模型，因为它只需要匹配目标地址。

如图 **??**所示，数据流 F_1 和 F_2 均来自监控摄像头 S_1，而 F_3 来自 S_2。F_1 由

Algorithm 2 ModifyAddress($s, d, path$)

```
1:  Algorithm ModifyAddress(s, d, path)
2:   for each node n in path do
3:    nx = the next node of n;
4:    setRule = false;
5:    for each rule r in n do
6:     no = the node connecting the output port of r;
7:     if r matching s and no != nx then
8:      act = forward to nx and modify the destination address to d;
9:      emitRule(matchFor(d), act);
10:     setRule = true;
11:    else if r matching s and no = nx then
12:     setRule = true;
13:   if setRule == false then
14:    act = forward to nx;
15:    emitRule(matchFor(d), act);
16:  return;
```

V_1 请求。F_2 和 F_3 的目标是 V_2。假设 V_1 的当前位置是 C，V_2 的位置是 D。在单播情况，F_1 和 F_2 需要至少两个规则在交换机 A 和 B；在多播情况下，F_2 和 F_3 也需要至少两条规则在交换机 A 和 B。然而这是低效的，因为 F_1 和 F_2 来自同一节点，而 F_2 和 F_3 具有相同的目的地。我们在交换机 B（作为分支节点）上修改 F_1 和 F_2 数据包头中的目标地址。通过设置 F_1 和 F_2 的目标地址（即 $F_1 \rightarrow C$，$F_2 \rightarrow D$），减少了交换机中的规则数量并可以提供更多服务。再假设 V_2 在开始时向 S_1 发送请求。每个交换机只需要一个规则来转发 F_2。而当 V_2 从 S_2 获取数据时，交换机的规则仍适用。最后，当 V_1 需要来自 S_1 的数据时，它只需在交换机 B 上额外增加一个规则并匹配 (S_1, D) 以及一个修改操作。这里我们使用两个参数来表示包头，即（源地址，目的地址），以及流表中的匹配条件。该目标驱动模式将目标地址设置为流表中的匹配条件，合并同一目标的数据流的流规则，以减小交换机中流表的大小。在图 6.5中，当 V_3 加入时，它需要在分支交换机处进行地址修改以优化规则安装。

接下来，我们通过两种情况来分析优化的规则下发：1. 一台服务器连接多台车辆；2. 一台车辆连接多台服务器。对于第一种情况，传统的规则安装仅根据源地址将数据包转发到给定端口，而在路径的每个交换机上需要一个规则。在我们提出的规则下发方法中，我们在每个交换机上需要一个规则用于转发，并且需要在分支节点处修改数据头的操作。由于修改地址操作可以与 OpenFlow 中的转

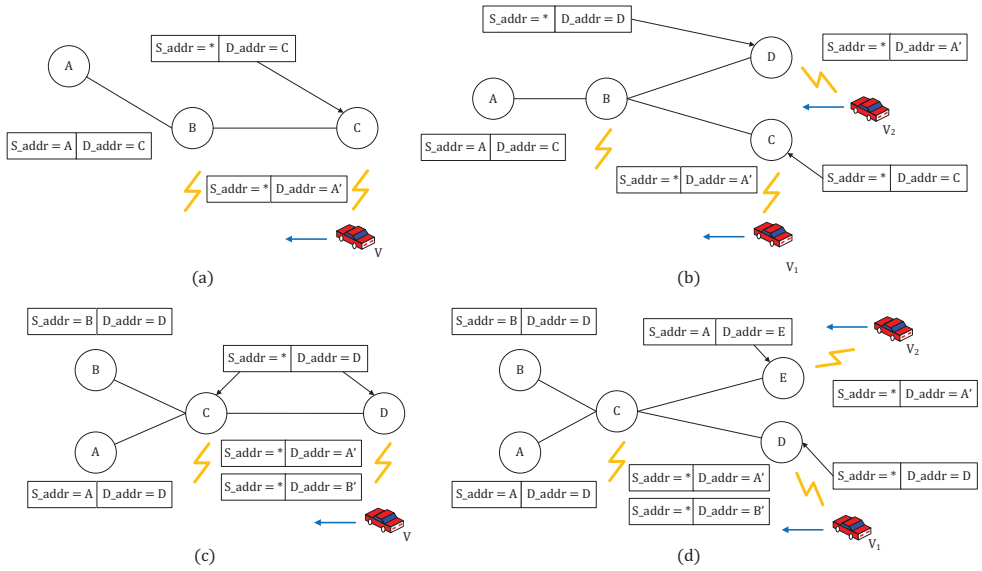


图 6.7　（a）：1 对 1 场景即 V 接受 A 的数据；（b）：N 对 1 场景即 V_1 和 V_2 同时接受 A 的数据；（c）：1 对 N 场景即 V 同时接受 A 和 B 的数据；（d）：N 对 N 场景即 V_1 接受 A 和 B 数据，而 V_2 接受 A 数据。

发操作组合作为一个规则操作，因此与传统规则下相比，规则的数量不会改变。对于第二种情况，我们假设有一辆车辆（v）连接到多个服务器（$s_1, s_2, \ldots s_n$），并且我们规定 s_i 离 v 的距离为 h_i（跳数）。设 L_i 表示用于 v 和 s_i 之间数据传输的一组链路。因此，我们有 $|L_i| = h_i$。因此传统方法需要 $\sum_{i=1}^{n} h_i$ 条规则，而通过我们优化的规则安装，我们需要 $|L|$ 条规则，其中 $L = \bigcup(L_1, L_2, \ldots L_n)$。考虑一个极端情况，任何两组链路 L_i 和 L_j 都没有共享链路，则我们有 $|L| = \sum_{i=1}^{n} h_i$。在这种情况下，我们提出的方法所需的规则数量与传统方法相同。对于其他大多数场景，$|L| < \sum_{i=1}^{n} h_i$，这意味着我们提出的方法减少了规则的数量。总而言之，我们优化的规则的最坏情况包括上述第一种情况和第二种的极端情况不会改变规则的数量，而最好的情况可以在一辆车连接到多个服务器时减少很多规则。

6.4.4　示例

这里分为四种场景来给出我们方法是如何工作。如图 6.7所示，通过分析实时查询服务中的数据传输过程，我们给出交换机上的实际流规则。

6.4.4.1　1 对 1 场景

在最简单的情况下，只有一辆车向一台监控摄像头请求实时查询服务。如图 6.7(a) 所示，V 向摄像头 A 发送请求，然后从 A 接收数据流。来自 A 的数据包是 (A, C)，B 和 C 的匹配条件都是 $(*, C)$。从 C 多播的数据包是 $(*, A')$，其中 A' 表示 A 的多播地址。在这种情况下，$PathFind(s, d, v)$ 计算出的路径是 $C \rightarrow B \rightarrow A$。

然后我们在 B 下发匹配 $(*, A')$ 的多播规则。当 V 到达 B 时，它可以不间断地接收数据。

6.4.4.2　N 对 1 场景

当摄像机连接到多个车辆时，需要修改分支节点处的目的地地址，如图 6.7(b) 中的 B 所示。对于 OpenFlow 交换机，它在一个规则上可以支持一组动作（例如，包头修改和转发动作）。因此，修改操作不需要额外的规则。开始，A 发送数据包 (A, C) 因为 C 处的 V_1 发送的请求遭遇 D 处的 V_2。当 V_2 发送请求时，控制器（图中未显示）下发新的具有修改操作的 OpenFlow 规则。最终，交换机 B 将两个包 (A, C) 和 (A, D) 转发到不同的端口。由 B，C 和 D 的转发的数据流均为 $(*, A')$。

6.4.4.3　1 对 N 场景

在该场景中车辆需要同时连接到多个摄像头。如图 6.7(c) 所示，V 想要同时从 A 和 B 接收数据。A 和 B 生成的数据包是 (A, D) 和 (B, D)。交换机 C 和 D 的匹配条件都是 $(*, D)$。而这里只需要一条规则来满足不同数据包的要求。当数据包 (A, D) 达到 C 时，规则 $(*, D)$ 与之匹配，然后转发到 D。交换机 D 遵循与 C 相同的过程。同时 C 和 D 还需要更改其数据包的目标地址以进行多播转发。C 和 D 的多播数据包是分别为 $(*, A')$ 和 $(*, B')$，其中 A' 和 B' 是多播地址。

6.4.4.4　N 对 N 场景

我们将 1 对 N 和 N 对 1 场景组合并形成一个更常见的 N 对 N 场景。如图 6.7(d) 所示，V_1 需要来自 A 和 B 的实时数据，V_2 需要来自 A 的数据。对于匹配 $(A, *)$ 的数据包，在 C 处进行数据包的修改，因为当数据源 A 同时将数据包传输到 D 和 E 时，C 是一个分支节点。来自 B 的数据包无需修改即可转发到给定端口。

N 对 N 场景作为一种普遍的情况，给出了有关如何实现优化的规则下发的详细信息。我们可以看到，我们提出的优化规则下发方法的流表大小的理论上限与车辆同时连接的设备数量有关。N_d 表示设备数量，R_0 表示传统方法的规则数量，R 表示优化的规则下发方法的规则数量。我们可以得到 $R = R_0/N_d$ 作为流表大小的上限。在这种情况下，所有设备都连接到相同的交换机。

6.5 实验评估

我们使用 Floodlight[Floodlight] 作为控制器以及 Mininet[mininet] 来构建 SDN 环境，以便评估 SDIV 中优化的流规则下发。我们在服务器上运行 Floodlight，配置为 16 AMD Opteron(tm) 处理器 6172 和 16GB 内存。服务器安装了 Linux 内核版本 2.6.32。我们在另一个单独的服务器上运行 Mininet。服务器之间通过 10Gbps 以太网连接。

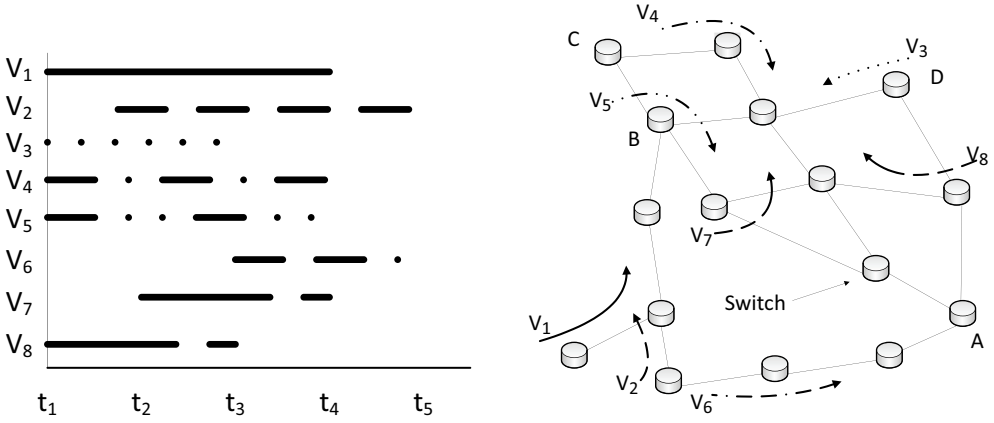
6.5.1 优化的规则下发

我们使用上海市车辆行驶线路的真实数据[shanghai] 作为我们的实验场景，以显示优化的规则下发的优势。图 6.8(a) 显示了上海人民广场周围的场景。车辆的行驶痕迹由不同时间的 GPS 数据组成。由于我们使用上海市的实际数据，车辆数量有限，但足以显示所提方法的效果。在 t_1 时，该区域只有五辆车。在 t_2，还有两辆车加入。t_3 时，V_6 出现在该区域内。车辆的出现时间和消失时间如图 6.8(b) 所示。任何两个时刻之间的间隔时间是 120 秒。由于这些车辆无法在确切时间出现，例如 t_1 和 t_2 间隔时间为 120 秒，我们会将车辆显示在 t_i 如果 $|t_i - t_{real}| < 30$ 其中 t_{real} 是车辆的实际出现时间。虽然这些 GPS 标记在同一时刻可能没有完全相同的时间戳，但通过限制 GPS 标记的时间戳和所标识的时刻为一定范围之间，可以说这些车辆非常接近这些位置。在这种情况下，我们将时间范围设置为 30 秒，以确保每个 GPS 的标记都能反映当时的真实位置。

图 6.8(c) 是图 Fig. 6.8(a) 的简化版本。如图 6.8(a) 所示，具有不同特征的箭头根据 GPS 标记中这些时间戳的顺序来说明车辆的方向。我们假设在 GPS 标记周围总有一个路边 AP（图 6.8(c) 中的交换机），以便随时可以随时传输数据。我们使简化版本中的每辆车在出现的立刻都有数据传输需求。因此，地铁 8 号线和地铁 2 号线交叉点附近的 V_1 和 V_2 想要从 B 和 C 接收数据，并且具有相同路径即沿着地铁 8 号线。V_3 位于两条道路汇合点并需要 B 和 C 的数据。V_4-V_5-V_6 具有相同的目标 A，但选择不同的路径。V_7 询问 D 的数据。V_8 只有目标 B。所有这些车辆都需要持久的数据流服务，直到他们移动到目的地。我们需要在每个必要的交换机上安装规则以满足他们的要求。

我们的对比方法是一种简单地基于源地址将数据包转发到给定端口的通用方法。通过优化的规则下发，我们合并具有相同目的（目的地址驱动模式）的流规则。结果如图 6.9(a) 所示。在 t_4 上的下降是由于我们设定的规则超时。图 6.9(b) 显示了不同时刻的车辆数量。图 6.9(c) 显示了两种方法中不同车辆的延迟时间。如果车辆同时连接两个摄像头，我们在此评估的延迟时间是最长的。如我们选择 C 而不是 B 来评估 V_1 的延迟时间。

(a)



(b)　　(c)

图 6.8　（a）：上海市人民广场附近的场景图；（b）：车辆的出现和离开时间；（c）：（a）的简化图并保护车辆方向信息。

另外，我们可以从图 6.9(c) 中看到我们优化的规则下发方法虽然需要在数据包报头中进行地址修改，但几乎不会影响数据传输的性能。我们减少了规则数量并压缩了流表中的空间，以实现更好的可管理性和可拓展性。特别是，从图 6.9(b) 看出，与对比方法相比，规则数减少了 60%。

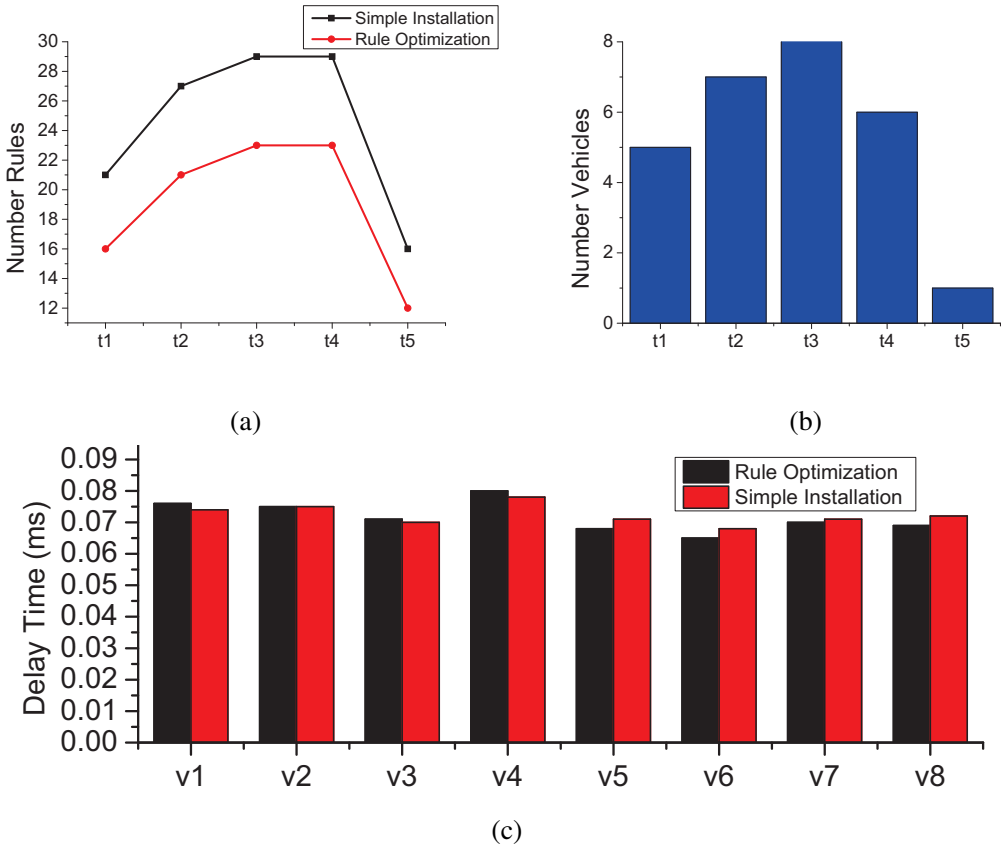


图 6.9 （a）：两个策略的规则数量差异；（b）：不同时刻的车辆数量；（c）两个策略的车辆延迟时间。

6.5.2 时延分析

我们对两种模式（1 对 N 和 N 对 1）进行研究分析，并说明它们如何以不同方式影响结果。我们将规则数和延迟时间与对比方法进行比较，以表明它不会影响数据传输的性能。图 6.10(a) 显示 N 对 1 模式，即有多个车辆连接到一个监控摄像头（B）。我们选择具有最长路径的 A 来评估延迟时间。对比方法需要在每个交换机上安装一个规则，并匹配源地址。对于所提的方法，从 B 传输到 A 的每个数据包都会修改包头（目标 IP 地址），然后转发到下一个交换机。图 6.10(b) 表明，随着交换机数量的增加，两种方法的延迟时间都变大，但没有显着差异。我们得出结论，优化的规则下发方法中的数据包修改过程不会影响性能。图 6.10(c) 显示了交换机的规则数量。红色（右侧）行表示节点 B 同时通过网络将数据流传输到每个节点的对比方法。每个交换机只需要一个规则基于 $(*, B')$（B' 是多播地址）。黑色（左侧）线描述了优化的规则下发方法，该方法修改每个分支节点处的数据包目标地址，然后将数据包转发到给定端口。每个交换机只需要一个规则。

图 6.10(d) 显示 1 对 N 模式。其中车辆 V 同时连接多个摄像机（B, C, D）。图 6.10(e) 显示了根据不同交换机数量的延迟时间。如果没有优化的规则下发，每

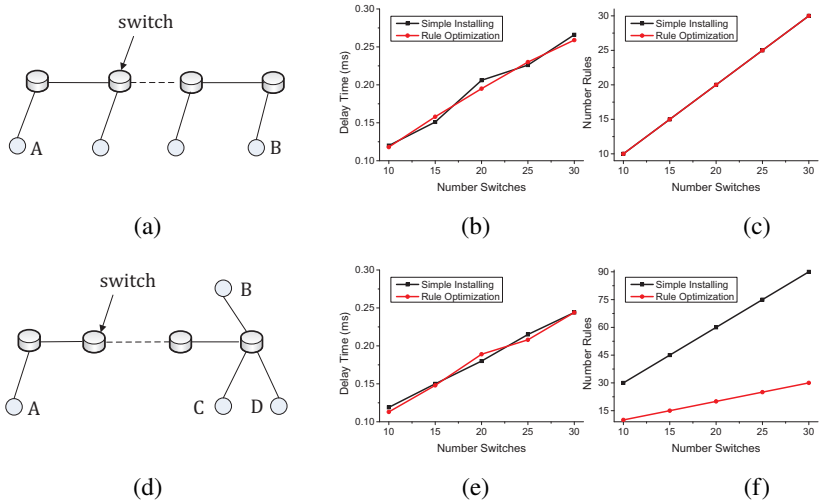


图 6.10　（a）：N 对 1 模式的拓扑；（b）：对于（a）中不同交换机数量的延迟时间；（c）：对于（a）中不同交换机数量的规则数量；（d）：1 对 N 模式的拓扑；（e）：对于（d）中不同交换机数量的延迟时间；（c）：对于（d）中不同交换机数量的规则数量。

个交换机的每个源节点都需要三个规则，$(B,*),(C,*),(D,*)$，这是源地址驱动模式（基于数据源转发数据包）。通过优化的规则下发，我们只需要一个规则，即每个交换机上的 $(*,A)$，它仍然可以支持 N 对 1 模式。图 6.10(f) 显示了两种方法的规则数量。

我们从结果中得出两个结论。首先，分支节点处的数据包修改对数据传输的性能没有影响。其次，对于优化的规则下发的最佳情况（1 对 N），压缩后规则的数量与数据源的数量成比例。对于优化的规则下发的最坏情况（N 对 1），交换机的规则数量等于对比方法的规则数量。

致谢

衷心感谢导师 xxx 教授和物理系 xxx 副教授对本人的精心指导。他们的言传身教将使我终生受益。

在美国麻省理工学院化学系进行九个月的合作研究期间，承蒙 xxx 教授热心指导与帮助，不胜感激。感谢 xx 实验室主任 xx 教授，以及实验室全体老师和同学们的热情帮助和支持！本课题承蒙国家自然科学基金资助，特此致谢。

感谢 TongjiThesis，它的存在让我的论文写作轻松自在了许多，让我的论文格式规整漂亮了许多。

参考文献

[1] BOSSHART P, DALY D, GIBB G, et al. P4: Programming protocol-independent packet processors[J]. ACM SIGCOMM Computer Communication Review, 2014, 44(3): 87-95.

个人简历、在学期间发表的学术论文与研究成果

个人简历:

xxxx 年 xx 月 xx 日出生于 xx 省 xx 县。

xxxx 年 9 月考入 xx 大学 xx 系 xx 专业，xxxx 年 7 月本科毕业并获得 xx 学士学位。

xxxx 年 9 月免试进入 xx 大学 xx 系攻读 xx 学位至今。

发表论文:

[1] Yang Y, Ren T L, Zhang L T, et al. Miniature microphone with silicon- based ferroelectric thin films. Integrated Ferroelectrics, 2003, 52:229-235. (SCI 收录, 检索号:758FZ.)

[2] 杨轶, 张宁欣, 任天令, 等. 硅基铁电微声学器件中薄膜残余应力的研究. 中国机械工程, 2005, 16(14):1289-1291. (EI 收录, 检索号:0534931 2907.)

[3] 杨轶, 张宁欣, 任天令, 等. 集成铁电器件中的关键工艺研究. 仪器仪表学报, 2003, 24(S4):192-193. (EI 源刊.)

[4] Yang Y, Ren T L, Zhu Y P, et al. PMUTs for handwriting recognition. In press. (已被 Integrated Ferroelectrics 录用. SCI 源刊.)

[5] Wu X M, Yang Y, Cai J, et al. Measurements of ferroelectric MEMS microphones. Integrated Ferroelectrics, 2005, 69:417-429. (SCI 收录, 检索号:896KM.)

[6] 贾泽, 杨轶, 陈兢, 等. 用于压电和电容微麦克风的体硅腐蚀相关研究. 压电与声光, 2006, 28(1):117-119. (EI 收录, 检索号:06129773469.)

[7] 伍晓明, 杨轶, 张宁欣, 等. 基于 MEMS 技术的集成铁电硅微麦克风. 中国集成电路, 2003, 53:59-61.

研究成果:

[1] 任天令, 杨轶, 朱一平, 等. 硅基铁电微声学传感器畴极化区域控制和电极连接的方法: 中国, CN1602118A. (中国专利公开号.)

[2] Ren T L, Yang Y, Zhu Y P, et al. Piezoelectric micro acoustic sensor based on ferroelectric materials: USA, No.11/215, 102. (美国发明专利申请号.)