支持增量式编程的多模态网络环境

崔子熙,田 乐,崔鹏帅,胡宇翔,伊 鹏,邬江兴
(中国人民解放军战略支援部队信息工程大学信息技术研究所,河南郑州 450001)

摘 要: 当前,多模态网络编程模型与底层硬件紧耦合、强相关,导致网络程序呈现扁平化和单片化特征.因此,持续开发模态程序效率低下且极易出错,制约了网元设备的可用性和可靠性.为此,本文提出面向多模态网络的编程环境(PINet's Programming Environment,PPE),支持增量式开发网络协议与功能.基于"巨型交换机"思想,PPE提出了一种平台无关的编程模型及语言,支持模块化编程和跨平台移植,通过模块单元的灵活组合提高网络程序的开发效率.同时,针对上述模型设计了前后端分离的编译系统框架.该系统自动化解析并组合分布式的模态程序,通过优化报文处理逻辑自动适配硬件资源约束.实验结果表明,在不影响硬件性能的基础上,PPE能够降低20%的程序开发量,同时引入编译时延和资源开销在合理范围内.

关键词: 编程模型;多模态网络;可编程数据平面;模块化;增量式编程;网络模态
基金项目: 国家重点研发计划课题(No.2022YFB2901501)
中图分类号: TP393 **文献标识码:** A **文章编号:** 0372-2112(2024)04-1230-09
电子学报URL:http://www.ejournal.org.cn **DOI:**10.12263/DZXB.20230852

Enabling Incremental Programming in PINet Environment

CUI Zi-xi, TIAN Le, CUI Peng-shuai, HU Yu-xiang, YI Peng, WU Jiang-xing
(*Institute of Information Technology, PLA Strategic Support Force Information Engineering University, Zhengzhou, Henan* 450001, *China*)

Abstract: At present, programming abstractions of polymorphic network (PINet) are tightly coupled to the underlying hardware, and thus programs are monolithic and target-specific. As a result, it is inefficiency and error-prone to develop programs continuously, which compromises the availability and reliability of hardware devices. In this paper, we present the PINet's programming environment (PPE) that aims to enable incremental development of protocols and functions. Based on the abstraction of one-big-switch, PPE proposes a target-independent model & language to support modularity and portability. It improves development efficiency by powerful forms of program composition. Correspondingly, the compiler framework is separated from front-end and back-end, so that it can automate the composition and analysis of the distributed programs. The packet processing logic is optimized to fit the resource constraints of hardware. The experimental results show that, PPE can reduce the lines of code by 20% without the affect to hardware performance, while introducing reasonable compilation delay and resource overhead.

Key words: programming abstraction; polymorphic network; programmable data plane; modular; incremental programming; network modality
Foundation Item(s): National Key Research and Development Program of China (No.2022YFB2901501)

1 引言

"人-机-物"万物智联时代,多模态网络提出"技术体系与支撑环境分离"的网络范式,由多元化技术体系满足垂直行业的多样化需求,加速网络创新发展[1].为了实现多种网络模态(协议体系与功能应用)在同一物理环境内共生共存、独立演进,多模态网络在具体部署环境中引入可编程数据平面技术,主要包括可编程交换设备与网络编程语言[2].目前存在CPU、ASIC(Application Specific Integrated Circuit)、FGPA(Field Programmable Gate Array)等多种异构的硬件平台,支持使用P4(Programming Protocol-independent Packet Processors)[3]语言自定义网络协议与功能,为构建多模态网络基础设施提供丰富灵活的选择[4].

尽管基于P4的编程环境能够加速多模态网络应用

收稿日期:2023-09-11;修回日期:2023-11-01;责任编辑:覃怀银

创新,但是在实际应用环境中面临以下三个方面的挑战:(1)可移植性:现有的编程模型与底层硬件架构紧耦合,导致代码中存在大量低级的操作指令与功能对象,破坏了程序的可移植性;(2)可组合性:P4编程呈现扁平化特征,打破了现有网络分层处理的理念.集中式的协议解析逻辑与后续报文处理算法完全分离,导致单一的数据结构贯穿整个程序,增加了代码单元之间的耦合关系;(3)可扩展性:低级的编程模型要求用户充分了解底层设备规格和实现细节,未能向上提供统一的资源封装与编排功能.面对大规模网络服务场景,用户需要逐设备调试异构平台的硬件资源,降低了网络模态部署的可扩展性.

针对P4程序呈现出扁平化和单片化的缺点,现有工作纷纷对网络编程模型及语法进行改造,同时优化编译过程.P4Visor[5]提出一种轻量级程序合并方案,通过自定义逻辑指令和算法树分析来实现多程序组合,但未能解决程序的移植性问题.P4I/O[6]提出基于用户意图的网络编程框架,使用高度抽象的自然语言描述用网意图,但要求编译系统提前开发大量的模板,工作原理缺乏普适性.μP4框架[7]致力于将模块化特征引入P4编程,但编译器必须对解析器代码和控制流代码进行同质化转换,带来严重的资源开销和性能损失.Lyra[8]重在解决面向数据中心网络应用的跨平台编译与管理问题,目标硬件限定于异构的ASIC芯片,且未能实现增量式部署.

面向多模态网络的运行逻辑和部署需求,本文提出多模态网络编程环境(PINet's Programming Environment,PPE)的概念,旨在支持协议与功能增量式部署.具体来说,PPE提出一种高级编程模型,允许将任意可编程网元抽象为服从可重构匹配表(Reconfigurable Match Tables,RMT)架构的交换节点.模态开发阶段,程序员从复杂的硬件细节中抽象出流处理的逻辑本质,着重构建平台无关且独立自包含的基本模块单元.核心在于,模块单元可以灵活组合,构造多种具备应用价值的模态程序.针对编程模型的新特征,PPE对现有编译框架进行改进,自动将模态程序适配至各种异构硬件平台,有效降低多模态网络应用的开发与部署成本.

2　网络编程模型概述

2.1　面向多模态环境的增量式编程

在多模态网络环境中,增量式编程具备以下含义:(1)任意的用户程序都可以作为子模块并入当前主程序,无需进行手动适配;(2)当子模块从主程序中添加或删除时,对应的协议与功能也将自动添加或删除,无需进行程序重构;(3)用户程序可以无缝移植到适配的目标平台,无需关心底层硬件细节.

现有编程环境无法支持增量式特征的根本原因在于未能提供高级编程模型.因此,PPE基于"巨型交换机"(One Big Switch,OBS)思想对P4编程框架进行进一步抽象与改进,设计了一种面向多模态网络的编程模型mP4(meta P4).该模型支持用户快速构建差异化模态程序,降低网络应用的开发与部署成本.

2.2　多模态网络编程模型

OBS思想广泛应用于大规模可编程网络的管控方案中[9~11].在此基础上,多模态网络编程模型充分实践"高内聚,低耦合"的模块化编程思想,具备以下特征.

(1)逻辑独立:如图1所示,模块单元是典型的I/O处理系统,依次经过解析单元、处理单元和封装单元.其中,解析单元完全面向处理单元的需求,避免不必要的协议解析,保证模块的独立自包含特征.

(2)数据独立:模块可以拥有独立的数据命名空间(即"模块接口"),内部的协议类型、元数据和报文处理逻辑被隐藏起来,对外仅声明参数列表,实现了增量式开发的关键——模块可以灵活组合、多次复用,直到构成用户期望的模态程序.

(3)平台无关:模块内应当不包含任何与特定平台绑定的报文操作指令,也无需指定具体的流水线阶段(Ingress或Egress).针对复杂的网络处理功能(如哈希操作),编程模型提供统一的接口进行调用,完全消除网络程序与物理架构的耦合性.

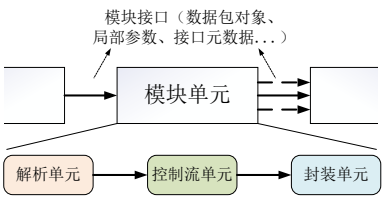

图1　基于mP4模型的模块单元抽象

模块化思想也体现在编译流程中(如图2所示).首先,开发者按照用网意图定义各自模态下的网络协议与算法,使用编程模型提供的高级语法开发模块单元,进而组合为完整的模态程序.然后,前端编译系统以模态程序和资源配置为输入,生成平台无关的流水线中间表示(Intermediate Representation,IR).最后,后端编译系统综合单片流水线,将IR程序映射到底层资源,生成可执行的硬件资源配置(如二进制程序).上述三个过程分别对应模态开发、模态编排与模态部署阶段,初步揭示了多模态网络编程环境的运行机理.

3　编程语言

基于mP4模型的网络程序主要包含三个部分:(1)

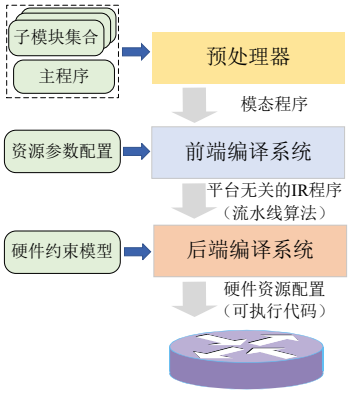

图2 多模态网络编译系统架构与流程

模态域定义;(2)模块及其接口定义;(3)流水线(主程序)定义. 本节以图3为示例,流水线Router包含Filtering和Forwarding两个模块,前者完成基于VLAN标签的报文过滤功能,而后者实现基于IPv4和IPv6协议的三层转发逻辑.

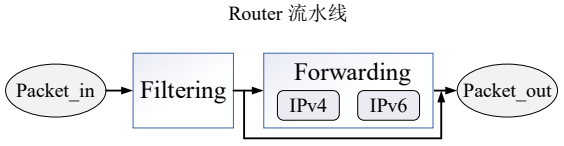

图3 示例程序逻辑图

3.1 语法与语义

从形式上看,多模态网络编程提供了一种“重逻辑、轻解析”的语言,重点关注模块内的核心算法及模块之间的数据交互,支持增量式开发网络程序.

mP4程序首先确定网络协议类型,构建模态域的基本概念. 模态域代表一系列固定的协议类型及解析逻辑,集成在编译工具链内,覆盖典型的多模态网络应用场景. 域内,协议类型与头部解析图(Header Parsing Graph,HPG)的节点一一对应,节点之间的流转关系确定. 图4显示了一种IP协议体系下的报文类型及解析逻辑树,包含了VLAN、TCP、UDP等常见的头部字段,记为FabricParser.

module关键字对应于编程模型提出的模块单元. 图5中,行1声明了IPv4模块及参数列表,则Forwarding模块可以直接对其进行调用并传递参数(行23). 模块定义时,程序员仅仅声明执行本单元逻辑所必要的协议类型(行17),却有效避免了RMT架构无法完成分布式解析带来的程序扁平化特征. 逻辑上,Forwarding模块仅解析二层帧头部,只有进入IPv4模块的数据包才会继续解析三层协议,确立了模块单元之间的独立自包含特征.

接口是模块组合的“粘合剂”,为程序员提供必要的数据类型. 图5中,行5~11定义了接口IForwarding,

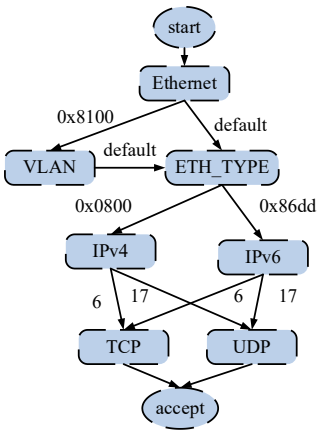

图4 协议类型及解析逻辑图(IP模态)

```
1  module IPv4(inout bit<16> io_nh);   // 子模块签名
2  module IPv6(inout bit<16> io_nh);
3
4  // 接口定义
5  interface IForwarding {
6        struct hdr_t {    // 报文头部类型
7              ethernet_t       ethernet;
8              ...
9        }
10       struct metadata_t { ... } // 用户元数据类型
11 }
12
13 module Forwarding() implements IForwarding {
14 // 编译器按照接口定义对模块进行参数扩展
15 // ( )    =>    ( hdr_t hdr,    im_t im,
16 //                 metadata_t meta )
17       parser { hdr.ethernet, hdr. eth_type }
18
19       control {
20             bit<16> meta_nh = 0; // 参数初始化
21             switch ( hdr. eth_type.value ) {
22                   //将 meta_nh 传递至子模块的 io_nh
23                   0x0800:    IPv4.apply(meta_nh);
24                   0x86dd:    IPv6.apply(meta_nh);
25                   default:    return;
26             }
27             …
28       }
29       // 无协议增减时，编译器自动补全以下代码
30       // deparser { hdr.ethernet, hdr. eth_type }
31 }
```

图5 模块及接口定义

包含协议头部和用户元数据两种结构体(固定元数据类型im_t由编程模型提供). 编译阶段,编译器自动对Forwarding模块的参数列表进行扩展(如行14~16所示),因此在模块内访问hdr和meta变量是合法操作. 本质上,接口为模块单元提供局部的数据命名空间,取消了核心元数据的全局作用域,能够有效提高网络程序的数据安全与可扩展性.

主程序内使用pipeline关键字定义完整的报文处理流水线,具备部署至数据平面的应用价值. 形式上,流水线等价于“大模块”,即按需对已有的模块组合,构建复杂的数据流算法. 区别在于,流水线需要提供确定的头部解析代码,指导底层设备完成协议识别功能. 图6

中,Router流水线指定FabricParser作为头部解析模板,无需重复定义.一个主程序中可以包含多个流水线定义,达到网络模态共存的目标.

```
pipeline Router using FabricParser {
    bool skip_forwarding = true;
    Filtering.apply(skip_forwarding);
    if (meta_skip_forwarding == false) {
        Forwarding.apply();
    }
}
pipeline Loadbalancer{…}
```

图6　主程序中流水线定义

3.2　复杂逻辑表达

模块单元默认为单播转发模式,而表达多播转发与非线性转发逻辑需要程序员显式添加修饰符.同时,mP4模型提供了一系列高级对象及方法以简明地表达复杂处理逻辑,进一步确立平台无关的编程特征.

对于使用多播修饰符的模块单元,编程模型提供了mc_t类型对象,用以控制RMT架构中报文缓存与复制引擎(Packet buffer and Replication Engine,PRE)的行为.如图7所示,该对象通过实例方法设置报文的多播表编号,取代了对固定元数据的直接访问.之后,程序员主动调用方法创建数据包副本.此时,编译器确保目标硬件立即按照多播表配置完成报文复制操作.

```
@Multicast   //多播逻辑修饰符
module IPv4(hdr_t hdr, im_t im,
                    metadata_t meta, mc_t mce) {
    table mcGroupTable {
        // map<ipv4_addr_t, grp_id_t>
        mce.set_mc_group(grp_id);
        ...
    }
    mcGroupTable.apply();
    mce.apply();
}
```

图7　多播转发逻辑示例

为了禁止流水线直接操作数据包负载,P4编程隐藏了报文对象的概念;换言之,P4语言只能描述"单报文"处理系统的逻辑.mP4模型保留了上述系统约束,但通过非线性逻辑修饰符提供了具象化的数据包对象(pkt_t类型).如图8所示,该对象可作为内置copy_from函数的参数,直接表达报文克隆操作,无需调用低级的操作指令(如V1model架构提供的clone方法).数据包扩展为mP4模型带来"多线程处理"特征,报文句柄可以像C语言中子线程变量一样灵活访问,提高了网络编程语言的表现力和准确性.

```
@Orchcast   // 非线性逻辑修饰符
module forwarding(pkt_t p, hdr_t hdr,
                  im_t im, metadata_t meta) {
    ...
    // 流水线生成复制报文 pt
    pt = copy_from(p);
    // p 和 pt 分别进入不同的逻辑模块
    prog.apply(p);
    test.apply(pt);
    // 多报文控制
    p.im.set_out_port(DROP);
    pt.im.set_out_port(CPU_PORT);
}
```

图8　非线性转发逻辑示例

4　多模态网络编译系统

4.1　总体框架

PPE的编译框架由三部分组成.

(1)预处理.预处理器对绑定接口的子模块(视为基本编译单元)执行分布式解析:首先,执行类型检查和语法树分析,生成变量列表;然后,在模块内执行接口扩展操作,确认接口提供的数据覆盖内部算法需求;最后,按照主程序定义对实例化的子模块进行链接,合成完整的模态程序.实际部署时,预处理器可以集成至前端编译系统中.

(2)前端编译.即使输入程序固定,不同的资源需求也会导致系统编译结果存在巨大差异,因此模态程序通常仅包含核心算法逻辑,详细的资源参数由额外的配置文件导入.前端编译器对主程序内的流水线进行解析与优化,生成平台无关的IR程序,具体包含主程序接口合成(4.2节)、协议解析代码合成(4.3节)和流水线特征分析(4.4节)三个步骤.

(3)后端编译.后端编译系统将IR程序向下转译为平台相关的代码块,然后将其编排至合理的流水线阶段(4.5节),最终调用平台编译器生成硬件流水线配置.

4.2　主程序接口合成

编译阶段,用户流水线被视为程序分析的对象.由于主程序调用的子模块相互独立,前端编译器首先合成统一的接口.合成接口是区别于报文负载的数据结构,其中hdr、im和meta等数据结构代表了当前报文对象的可操作域(又被称为"byte-stack"),还原了数据流处理的扁平化特征.

为了有效组合模块单元,μP4必须将中间模块的协议解析代码转译为控制流代码,进而卸载至"匹配-动作"单元(Match-Action Unit, MAU)执行,带来不可忽视的性能与资源损耗.本方案则避免对解析器代码和控制流代码进行同质化——编译器从模块提取所有解析字段和封装字段(即parser和deparser),分别流水线的首尾两端合成集中式列表L_p和L_e,消除其中的重复项.

相应地，主程序的解析逻辑代码（手动添加或自动生成）必须完全覆盖L_p中的所有字段，而封装代码则需要对L_e列表内的字段进行排序，保证报文封装顺序的正确性（例如，VLAN头部封装在IPv4头部之前）. 对于组合后可能出现先封装后解析的协议字段，编译器选择在集中式列表中对其进行保留并标记为算法的“内部协议”.

4.3 解析代码合成

主程序接口合成之后，编译器能够按需为流水线合成协议解析代码. 以Router程序为例，其对应的协议解析树属于图4所示HPG的子图，则直接裁剪TCP和UDP节点即可.

形式上，编译器有能力将主程序的HPG继续拆分为解析子图，及时分发到各个子模块中，便于下一步完成流水线特征分析. 对于一些尚处于研究阶段的新型可编程架构（如文献[12]提出的IPSA架构），分布式解析子图可以实现每阶段的协议按需解析，解析成本分摊在各个处理单元内部，从硬件层面支持服务不中断的增量式编译与部署.

4.4 流水线特征分析

当前RMT架构支持的逻辑控制语句不包含循环结构，因此每个流水线算法都是树形结构. 编译器对语法树执行静态分析，计算流水线的特征，便于对部署程序的硬件资源消耗进行初步评估. 流水线的特征值主要包括：(1)算法直径：算法流程图PFG的最大深度；(2)协议解析长度：解析器提取协议字段长度的最大值；(3)头部堆栈长度：算法执行过程中存储头部字段需要的最大空间.

图9为Router算法展开后的处理流程图（Processing Flow Graph，PFG），每个节点（A~G）代表对应IR程序中的匹配表. 编译器对PFG执行深度优先算法，扫描出树形图的最大深度，即为算法直径. 显然，Router的算法直径为5.

对任意流水线算法a，前端编译器在接口合成过程中维护集中式解析列表$L_p(a)$及对应的解析逻辑. 因此，编译器对其HPG执行加权的图遍历，可直接得出算法对应的协议解析长度$l(a)$. 例如Router算法的最大协议解析长度为58 byte，对应的解析路径是eth(12)→vlan_tag(4)→eth_type(2)→ipv6(40). 由于“内部协议”的存在，实际的最大协议解析长度可能小于编译器求解的结果，即满足$\bar{l}(a) \leqslant l(a) \leqslant L_p(a)$.

算法的头部堆栈长度描述了合成接口中hdr对象的空间大小，依据不同的流处理路径而动态变化，因为程序中可以随时添加合法的协议字段. 此时封装列表$L_e(a)$中存在不属于$L_p(a)$的协议字段集合P，即

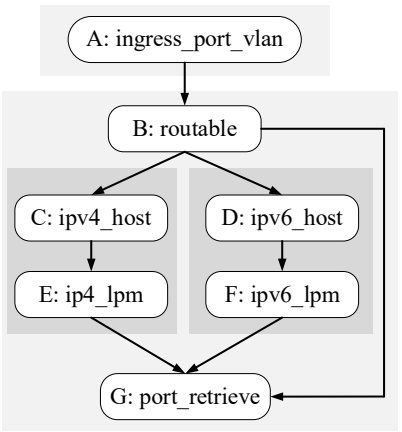

图9 Router程序展开后的处理流程图

$L_e(a) \neq L_p(a) \cap L_e(a)$. 相反，主动删除已经被解析的协议字段并不改变头部堆栈大小，这是因为RMT架构仅仅将该协议字段的有效标志置零，其内存空间不会被回收. 为了求解头部堆栈长度的最大值，需要对算法执行递归遍历，找出所有“协议增量路径”.

假设算法ψ对应的PFG存在$N(a)$个不同的协议增量路径，则其头部堆栈长度为

$$B(a) = \max_{x \in N(a)} \{ l(x) + \Delta(x) \} \tag{1}$$

其中，$l(x)$为路径x上所有模块生成的解析子图对应的最大解析长度；而$\Delta(x)$为路径x上协议增量的最大值. 显然，当P为空集时，$N(a)=0$，此时可直接推得$B(a)=l(a)$. 而一般情况下，任意子项取最大值并不代表总和为最大值，最坏情况下需要遍历所有的协议增量路径.

求解流水线特征值有助于编译器针对底层资源进行程序优化. 第一，算法直径与程序部署占用的MAU资源呈正相关[13]，如果算法程序的直径远远超出目标平台提供的逻辑阶段，则考虑通过优化算法逻辑来缩短直径，使模块化程序变得紧凑. 例如，将不存在逻辑依赖关系的匹配表合并后并行执行. 第二，头部堆栈空间占用底层硬件的包头矢量（Packet Header Vector，PHV）资源，容量固定为8-bits、16-bits、32-bits三种规格，因此可以通过元数据位宽与容器大小对齐以优化PHV利用率. 第三，编译器检查目标平台的解析资源能否满足最大的协议解析长度，若否，则需要更换目标平台或精简程序模块.

4.5 单片流水线编译

作为编译流程的关键一步，集成在后端的流水线编译器将mP4程序指令向下转译为平台相关的低级代码，同时适配平台架构约束，完成算法PFG到硬件流水线的映射. 本文以Tofino交换芯片支持的TNA架构为例对该过程进行说明.

TNA架构依次经过入口流水线IGP、流量管理器TM和出口流水线EGP三个组件. 其中,IGP与EGP相互独立且拥有相同的硬件资源(如内存容量和流水线阶段). 面向TNA架构的后端编译器对转译后的IR程序进行扫描,捕获目标平台对出端口、队列参数等数据的使用限制. 例如,元数据enq_qdepth记录了数据包进入TM组件时的队列深度,相关指令必须分配至在EGP阶段执行. 此外,还需要考虑硬件架构对流水线阶段数、每阶段匹配表并行数和每阶段内存容量的限制. 必要时,编译器为程序添加报文再循环指令(resubmit或recirculate),开辟新的流水线约束空间.

令$R(G_a)$表示流水线算法a的循环次数,其中,G_a为表示算法a经前端编译生成的PFG;v_i表示G_a的第i个节点. 假定单条硬件流水线(IGP或EGP)拥有N个处理阶段,其中阶段n内映射的匹配表集合为$V(n)$. 定义$S(v_i)$为离散函数,指示节点v_i映射的流水线阶段,则求解目标为符合架构约束的S集合. 为此,本文将该过程建模为整数线性规划(Integer Linear Programming,ILP)问题.

约束1:PFG逻辑顺序 对于PFG单条路径上任意两个相邻节点,依照逻辑顺序分配在不同的流水线阶段. 即

$$S(v_i) \neq S(v_j),\ \text{if}\ v_i \succ v_j \tag{2}$$

其中,符号$\succ$表示逻辑顺序关系.

约束2:流水线串行 连续部署在单条流水线的v_i集合,其算法直径不能超过N. 即

$$0 \leqslant S(v_i) \leqslant N-1 \tag{3}$$

约束3:内存容量 分配单个处理阶段的节点集合,其匹配表项总量不能超过内存容量. 即

$$\forall n,\ \sum_{v_i \in V(n)} s(v_i) \leqslant s,\ \sum_{v_i \in V(n)} t(v_i) \leqslant t \tag{4}$$

其中,s和t分别表示MAU的SRAM和TCAM容量.

约束4:匹配表并行 每个处理阶段内至多卸载M个并行的节点. 即

$$\forall n,\ \left|V(n)\right| \leqslant M \tag{5}$$

目标1 对数据包执行再循环指令的直接代价是引入双倍的报文处理时延并减少交换平台的总吞吐量,因此编排后流水线的回环次数应尽可能少. 即

$$\min R(I_a) \tag{6}$$

目标2 程序占用的流水阶段数应尽可能少,为增量式的功能和资源扩展提供便利. 令$a(n)$表示阶段n是否被占用的布尔函数,则

$$\min \sum_{n=0}^{N-1} a(n) \tag{7}$$

基于上述问题模型,系统调用开源的工具(如Or-tools)进行求解. 当问题不存在解时,则执行4.4节描述的程序优化方案. 本质上,PPE的后端编译器提前对硬件资源进行编排,尽可能避免厂商提供的平台编译器(通常内置启发式算法)无法求解可行的流水线配置. 转译后的IR程序恢复平台相关的特征,可以编译生成可执行的二进制代码,也可以逆向转化为P4代码.

5　实验验证

基于mP4语言的多模态网络编程环境在p4c项目和μP4项目的基础上进行原型开发,大约涉及9 000行C++代码. 得益于高级编程模型的高兼容性和可扩展性,集成后的编译系统能够适配多种不同型号的可编程网络平台,包括国产的盛科交换芯片. 作为统一配置节点,所有的编译过程都在一台浪潮NF5280M5系列服务器完成,搭载Intel Xeon Silver 4114 CPU和64 GB内存,运行Ubuntu 20.04操作系统. 本文重点测试多模态网络编程语法和编译系统在开发效率、资源开销和编译性能三个方面的表现.

5.1　开发效率

为了验证高级编程模型在开发效率方面的优势,实验设计了多种IP模态域下的功能程序,既包括基于IPv4的虚拟隧道协议VxLAN、基于IPv6的分段路由协议SRv6、基于MPLS的流量工程协议,也包括地址转化(NAT)和访问控制列表(ACL)等面向多层协议的网络功能.

表1展示了5个由上述模块单元灵活组合构成的功能程序C1~C5. 其中C1即为图3示例程序. 配合不同的后端编译组件,上述程序均可编译至基于V1Model的多核CPU平台和基于TNA架构的ASIC平台.

从C1到C2的程序开发过程充分证明本文所提模型能够支持增量式编程特征. 首先,C2开发了BdFor-Vxlan和VxlanLogic两个子模块,与已有的IPv4模块组

表1　基于不同模块的程序组合

模块单元	C1	C2	C3	C4	C5
Filtering	√	√			√
Forwarding	√	√	√	√	√
IPv4	√	√	√	√	√
IPv6	√	√	√	√	√
MPLS					√
VxLAN		√			
SRv6			√		√
NAT				√	√
ACL					√

合为完整的VxLAN报文处理逻辑;然后,外部控制流调用Vxlan模块即可完成功能升级. 如图10所示,由虚拟隧道封装引入的协议头部,其作用域限定在IVxlan接口中. 因此,当Vxlan模块被从主程序中添加或删除时,其附属的协议类型和元数据字段也被自动添加或删除,避免了程序重构的成本.

```
1  interface IVxlan {
2      header vxlan_t {...}
3      struct hdr_t {...}
4  }
5
6  module Vxlan(inout bit<16> io_nh)
7                  implements IVxlan {
8      BdForVxlan.apply();
9      VxlanLogic.apply();
10     IPv4.apply(io_nh);
11 }
12
13 module Forwarding() implements IForwarding {
14     switch (hdr.eth_type) {
15         // 外部仅替换调用模块
16         0x0800:   Vxlan.apply(meta_nh);
17         0x08dd:   IPv6.apply(meta_nh);
18         default:  return;
19     }
20     …
21 }
```

图10 Router程序增量式开发

5.2 资源开销

与扁平化特征的单片程序相比,模块化编程往往带来额外的硬件资源开销. 针对上述5种功能组合,分别将P4程序、μP4程序和mP4程序面向TNA架构编译,通过SDE提供的可视化工具(如p4i)对芯片资源开销进行分析评估.

表2显示了三种编程方案在PHV、逻辑阶段数(stage)和内存容量(memory)方面的开销. 其中内存容量指标以P4程序为基准进行归一化处理.

编译失败的情况经常发生在原生P4程序中,其原因在于ASIC平台编译器无法有效分配PHV和Stage资源,要求有经验的程序员进行手动优化,最终导致实际部署时间大约提高了两个数量级,即从$O(\min)$到$O(\text{day})$. 相反,在两种高级编程方案中,编译器主动对头部堆栈空间进行优化,避免协议字段碎片化并使PHV资源易于分配. 对于C4组合,μP4编程和mP4编程分别占用了68%和71%的PHV资源,较P4编程减小了15%和12%. 然而,μP4程序的编译包含解析代码转译过程,显著增加了算法所需的逻辑阶段数,最终导致C5的程序直径超出流水线阈值而编译失败. 实验证明,与解析器资源相比,高速硬件平台的MAU是更加稀缺的资源,往往对程序的编译结果产生重大影响.

表2 三种方案的资源开销对比

程序组合	P4(优化后)		μP4			mP4		
	PHV	stage	PHV	stage	memory w.r.t P4	PHV	stage	memory w.r.t P4
C1	34%	4	35%	7	+2%	37%	4	−1%
C2	67%	8	61%	12	+12%	63%	9	+2%
C3	51%	6	46%	7	+8%	49%	5	0
C4	83%	5	68%	8	+5%	71%	6	+1%
C5	92%	10	NA:编译失败			88%	11	+3%

由表2可得,mP4程序与最优的P4程序在逻辑阶段数和内存容量方面基本持平,均未超出TNA架构模型的限制. 对于C2组合,mP4编程的内存使用量较μP4编程减小了10%. 结果表明,本文提出的高级编程方案不但克服了μP4方案的缺陷,而且通过算法分析过程提供了针对目标平台的程序优化能力,减少了人工调试的潜在成本.

5.3 编译性能

本小节针对IPv4标识、SRv6标识、命名标识(Named Data Networking, NDN)、地理标识(Geo Networking, GEO)和身份标识(Mobility First, MF)5个复杂通信场景进行程序重构与开发,将测试程序分别部署至同一台Edgecore Wedge 100BF-65X交换机,使用Spirent测试仪测量硬件流水线的吞吐量. 重点对比P4程序与mP4程序在核心代码量与编译时间的结果,同时关注程序部署后的硬件性能.

由表3可以看出,本文提出的高级编程模型能够使程序的核心代码量平均降低20%,同时引入的编译时延在可接收的范围内. 对于IPv4模态,mP4程序的编译时间比P4程序的编译时间多出1.96 s,这是因为后端编译需要执行额外的代码转译和算法分析过程. 综合考虑程序开发与调试时间,mP4程序的增量式编程效率更高. 图11展示了其中四种程序部署后100 Gbps网口的吞吐量,两种编译配置的转发性能均可达到最大线速. 结果说明,使用PPE提供的高级语言进行网络编程不会对数据平面的交换性能产生明

表3 两种方案的编译性能对比

模态应用	P4		mP4	
	核心代码量	编译时间/s	核心代码量	编译时间/s
IPv4	458	4.35	364	6.31
SRv6	173	2.73	153	3.28
NDN	381	3.99	289	5.03
GEO	264	3.35	184	3.82
MF	94	1.86	88	1.92

显的负面影响.

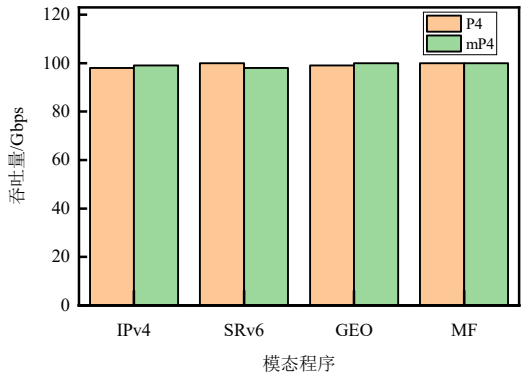

图11　程序部署后转发性能对比

6　结束语

针对扁平化和单片化的编程模式难以支持网络模态增量式开发和部署的问题,本文设计了一种高度模块化且平台无关的数据平面编程模型mP4. 该模型在语言规范上支持用户进行增量式编程,通过模块单元的灵活组合快速生成平台无关的模态程序. 相应地,编译系统采用前后端分离架构,在程序组合的过程中自动完成算法逻辑优化,避免程序员陷入模态功能重构和硬件资源调试的困境. 实验证明,在合理的资源开销范围内,基于高级编程模型的多模态网络编程能够有效降低数据平面功能开发与维护成本.

参考文献

[1] 邬江兴, 胡宇翔. 网络技术体系与支撑环境分离的发展范式[J]. 信息通信技术与政策, 2021, 47(8): 1-11.
WU J X, HU Y X. The development paradigm of separation between network technical system and supporting environment[J]. Information and Communications Technology and Policy, 2021, 47(8): 1-11. (in Chinese)

[2] 胡宇翔, 崔子熙, 李子勇, 等. 基于领域专用软硬件协同的多模态网络环境构造技术[J]. 通信学报, 2022, 43(4): 3-13.
HU Y X, CUI Z X, LI Z Y, et al. Construction technologies of polymorphic network environment based on codesign of domain-specific software/hardware[J]. Journal on Communications, 2022, 43(4): 3-13. (in Chinese)

[3] 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.

[4] HU Y X, LI D, SUN P H, et al. Polymorphic smart network: An open, flexible and universal architecture for future heterogeneous networks[J]. IEEE Transactions on Network Science and Engineering, 2020, 7(4): 2515-2525.

[5] ZHENG P, BENSON T, HU C C. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs[C]//Proceedings of the 14th International Conference on Emerging Networking Experiments and Technologies. New York: ACM, 2018: 98-111.

[6] RIFTADI M, KUIPERS F. P4I/O: Intent-based networking with P4[C]//2019 IEEE Conference on Network Softwarization. Paris: IEEE, 2019: 438-443.

[7] SONI H, RIFAI M, KUMAR P, et al. Composing dataplane programs with μP4[C]//Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communi-cation. New York: ACM, 2020: 329-343.

[8] GAO J Q, ZHAI E N, LIU H H, et al. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous ASICs[C]//Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communi-cation. New York: ACM, 2020: 435-450.

[9] VOELLMY A, WANG J C, YANG Y R, et al. Maple: Simplifying SDN programming using algorithmic policies [J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 87-98.

[10] 崔子熙, 胡宇翔, 兰巨龙, 等. 基于流分类的数据中心网络负载均衡机制[J]. 电子学报, 2021, 49(3): 559-565.
CUI Z X, HU Y X, LAN J L, et al. Load balancing based on flow classification for datacenter network[J]. Acta Electronica Sinica, 2021, 49(3): 559-565. (in Chinese)

[11] BOL P D, LUNARDI R, DE FRANÇA B, et al. Modular switch deployment in programmable forwarding planes with switch (de) composer[C]//Proceedings of the SIGCOMM Poster and Demo Sessions. New York: ACM, 2021: 30-32.

[12] FENG Y, CHEN Z K, SONG H Y, et al. Enabling in-situ programmability in network data plane: From architecture to language[C]//19th USENIX Symposium on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2022: 635-649.

[13] LI Y F, GAO J Q, ZHAI E N, et al. Cetus: Releasing p4

programmers from the chore of trial and error compiling [C]//19th USENIX Symposium on Networked Systems Design and Implementation. Berkeley: USENIX Association, 2022: 371-385.

作者简介

崔子熙 男,1996年7月出生,河南焦作人. 现为战略支援部队信息工程大学博士研究生. 主要研究方向为可编程数据平面、软件定义网络.
E-mail: czxndsc@163.com

田 乐 男,1987年11月出生,陕西咸阳人. 现为战略支援部队信息工程大学副研究员. 主要研究方向为可编程数据平面、软件定义网络.

崔鹏帅 男,1990年4月出生,河南安阳人. 现为战略支援部队信息工程大学副研究员. 主要研究方向为网络空间安全、新型网络架构.

胡宇翔 男,1982年11月出生,河南周口人. 现为战略支援部队信息工程大学教授、博士生导师. 主要研究方向为新型网络架构、网络空间安全.

伊 鹏 男,1977年11月出生,湖北黄冈人. 现为战略支援部队信息工程大学研究员、博士生导师. 主要研究方向为网络空间安全.

邬江兴 男,1953年9月出生,安徽金寨人. 现为中国工程院院士,战略支援部队信息工程大学教授. 主要研究方向为网络空间安全、信息技术. 中国电子学会会员编号:E190043780M.