中国科学技术大学
University of Science and Technology of China

P4: Programming Protocol-Independent Packet Processors

Pat Bosshart, Dan Daly, Glen Gibb, et al.

SIGCOMM CCR 2014

授课教师：赵功名

中国科大计算机学院

2025年秋·高级计算机网络

本文的重要意义

- 在 SDN / P4 生态中的历史地位
  - “OpenFlow 2.0 草案”的代表：原文明确把 P4 定位为 OpenFlow 未来演进方向的一个 strawman proposal——不再在标准里硬编码具体协议头字段，而是让交换机暴露“可编程解析 + 可编程匹配”的通用能力
  - 因此，很多人把这篇论文视为“OpenFlow 2.0 草案”的代表性工作：控制器不只是“填表项”，而是可以告诉交换机“你应该怎样处理一个包”

本文的重要意义

从“控制面可编程”走向“转发面可编程”

- 前面我们在课程里已经看到：OpenFlow 把控制平面从设备中抽离出来，让控制器可以下发转发表项，实现“控制面可编程”
- 这篇论文进一步回答：
  - **当控制面已经可编程后，数据/转发平面本身如何变得可编程?**
  - ——不只是“查哪个表”，而是“这个表长什么样、解析哪些头、执行什么动作”

Outline

I. Introduction

II. Abstract Forwarding Model

III. A Programming Language

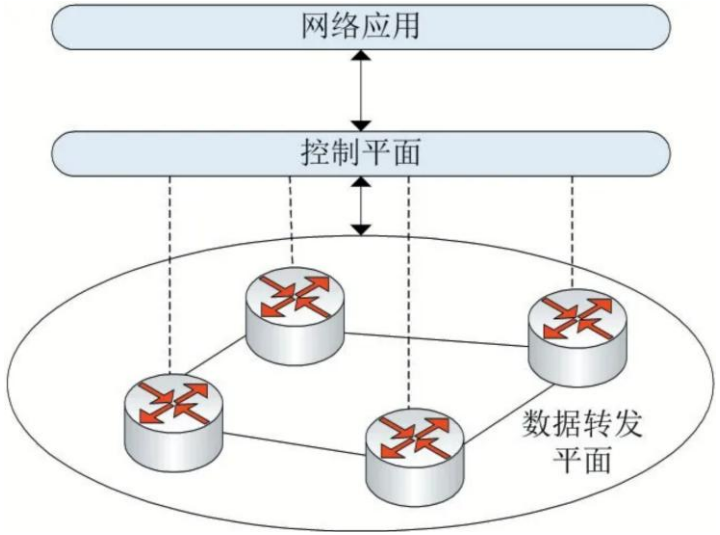
IV. P4 Language by Example

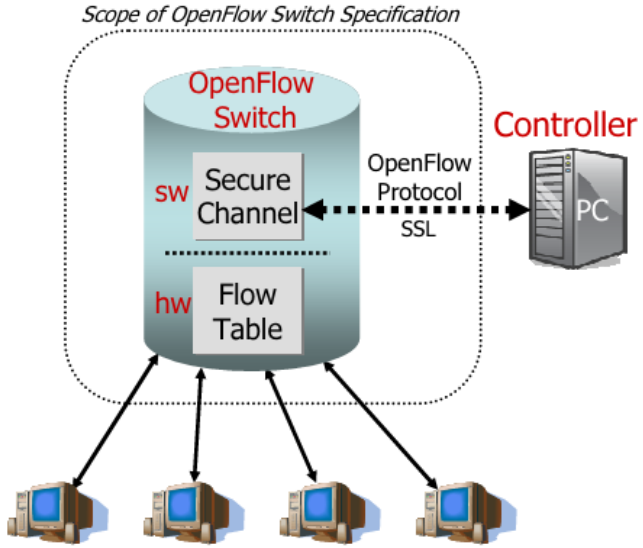
V. Compiling a P4 Program

VI. Conclusion

1. Introduction：背景

- SDN 的核心思想
  - 控制平面 / 数据平面分离：控制逻辑从设备中抽离出来，集中在控制器中实现
  - 逻辑集中控制：通过南向接口统一下发规则，控制多台交换机的转发行为

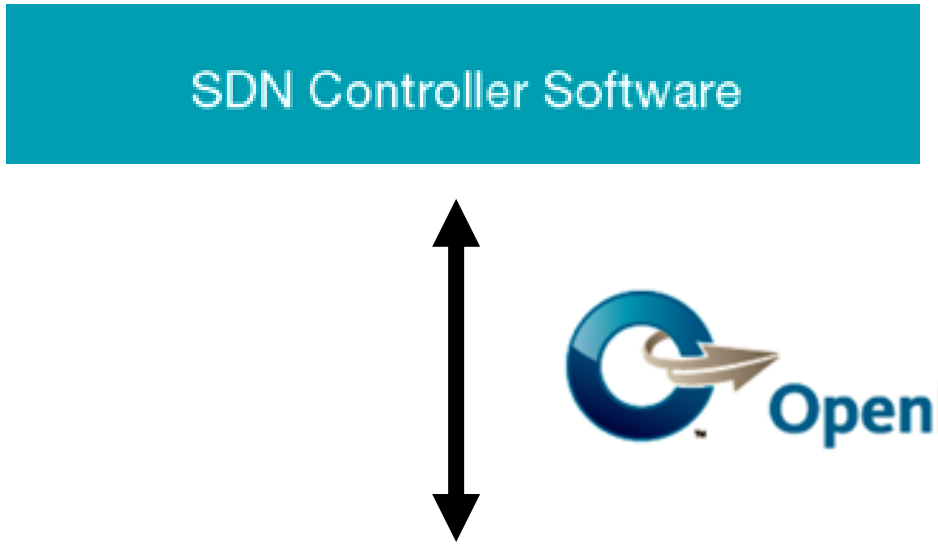




- OpenFlow 的角色
  - 控制器—交换机之间的开放接口标准：规定控制器如何向交换机下发“流表项”
  - 用 match + action 表达转发行为：在预定义的字段上匹配（五元组、端口等），执行转发、丢弃、改写等动作

1. Introduction：背景

- OpenFlow 的局限
  - 面向固定功能交换机设计：假定交换机内部流水线、匹配字段和动作类型都是预先写死的
  - 控制器只能“填表项”，不能改变交换机内部处理逻辑（解析哪些协议、表的顺序和结构等）



OpenFlow-enabled Network Device

Flow Table comparable to an instruction set

MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

1. Introduction：背景

- OpenFlow 头字段数量迅速膨胀
  - OpenFlow 标准规定了控制器可以匹配的 header fields 集合。短短几年内，这个集合就从 12 个字段 增长到 41 个字段

Version	Date	Header Fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

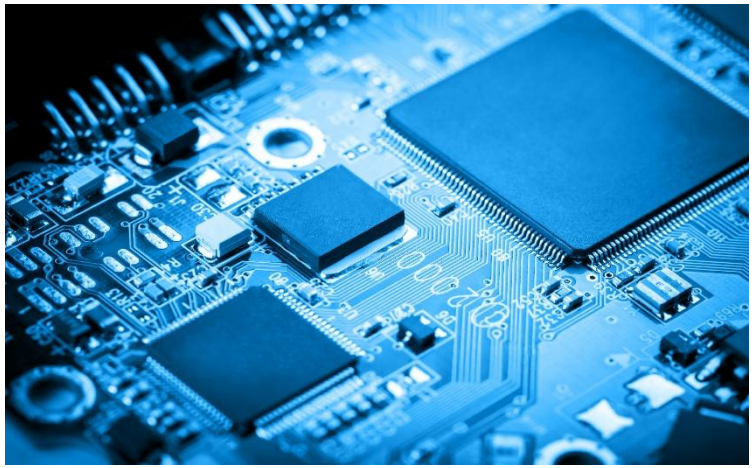
- 增长原因：新协议与新封装层不断加入
  - 数据中心与运营商网络逐渐引入更多协议和封装
  - 每出现一种新协议 / 新封装，就需要在 OpenFlow 里 新增可匹配字段，推动标准版本不断扩展
- 后果：复杂度上升，却仍然难以快速支持 新的、自定义的、实验性的协议

1. Introduction：Openflow的局限性

- 固定的协议集（Protocol-dependent）
  - OpenFlow 标准里写死了一组可解析、可匹配的头字段集合，交换机的 parser 也按这套协议硬编码
  - 想支持新的头部就必须修改标准 + 更新设备实现
- 固定的流水线结构（Fixed pipeline）
  - 标准假定交换机内部是一个**预定义好的**多级 match+action 流水线
  - 控制器的权限仅限于“向既有表里填表项”，不能增/删表，也不能重新组织流水线，更无法插入新的中间处理逻辑
- 固定的底层目标（Fixed-function targets）
  - OpenFlow 最初面向的是固定功能 ASIC 交换机，随着可编程 ASIC、软件交换机、SmartNIC 等新型“包处理器”出现，OpenFlow 的抽象难以表达这些更灵活的目标

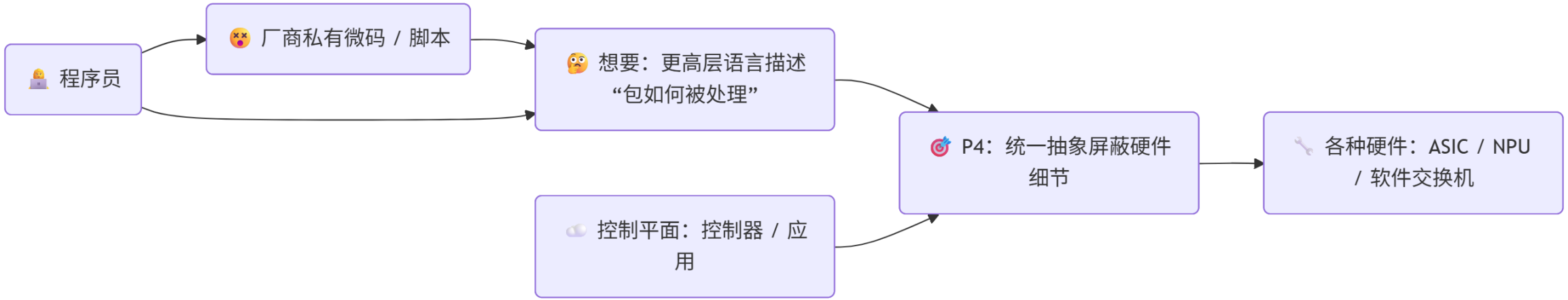
1. Introduction：新一代可编程芯片的出现带来的机遇

- 硬件趋势：可编程流水线进入 ASIC
  - 近年的交换机芯片设计已经在定制 ASIC 上实现了“可重配置的 match-action 流水线”，在 Tbps 速率下仍可工作
  - 这些芯片可以在片上重新配置解析器、匹配表和动作逻辑，不再只支持固定协议和固定转发流程
  - 典型代表就是支持可重配置匹配表（RMT）、可编程解析器的新一代芯片，为“协议无关、目标无关”的数据平面提供了硬件基础



1. Introduction：新一代可编程芯片的出现带来的机遇

- 编程难题：厂商私有、“微码式”极低层接口，编程这一代交换芯片“远非易事”
- 机会：需要一种 更高层的语言 来描述“包应如何被处理”
  - 目标是：屏蔽不同芯片的底层细节，让程序员面向统一的抽象模型编程；同时作为控制平面与数据平面的共同语言：控制器不仅“填表项”，还能通过 P4 程序定义表和流水线本身

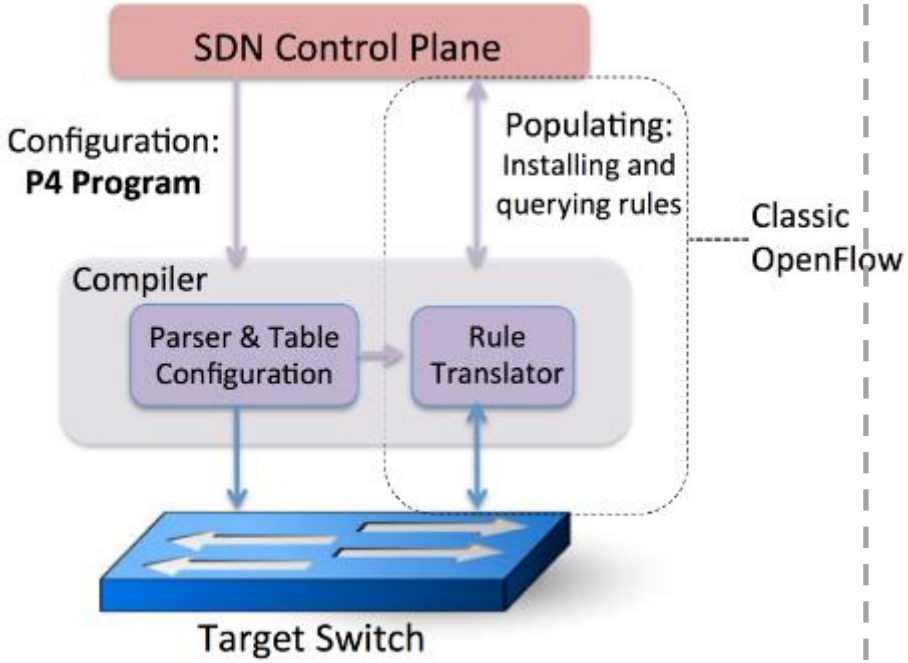


1. Introduction：P4 提出的三大目标

- （1）现场可重配置（Reconfigurability in the field）
  - 交换机部署之后，还能改“处理逻辑”而不只是改表项
  - 面向运维场景：出现新业务 / 新封装时，不必更换硬件，而是在现场重新配置数据平面
- （2）协议无关（Protocol independence）
  - P4 不预设“必须支持 IPv4/IPv6/TCP/UDP 等固定协议集合”
  - 程序员在 P4 语言里用 header 定义自己需要的头部格式，用 parser 定义解析状态机，从而支持：标准协议（IPv6、VXLAN 等）、自定义 / 实验性协议
- （3）目标无关（Target independence）
  - 写 P4 程序时，不直接面向某一颗具体芯片，而是面向抽象的“可编程包处理流水线”模型
  - 编译器负责把同一份 P4 程序映射到不同目标：可编程 ASIC / NPU / FPGA / 软件交换机等
  - 程序逻辑尽量保持不变，只在编译/配置阶段处理目标相关的资源约束与优化

1. Introduction：P4 与 OpenFlow 的关系

- P4 不替代 OpenFlow，负责描述这台交换机内部是什么样的可编程转发管线
  - 解析哪些头、有哪些表、什么顺序、每个表支持哪些动作
  - OpenFlow / 其他控制协议在其之上工作：在 P4 定义好的这些表上，控制器通过 OpenFlow,填表项，实现路由、ACL、负载均衡等具体策略
- 分工直观类比
  - P4 像是在“配置 CPU 的指令集和流水线结构”：决定这颗“数据平面处理器”能理解哪些指令/字段、流水线长什么样
  - OpenFlow / 控制器则像“发出具体指令和操作数”：在既定指令集/流水线之上，写入具体规则：匹配哪些流、如何转发或改写



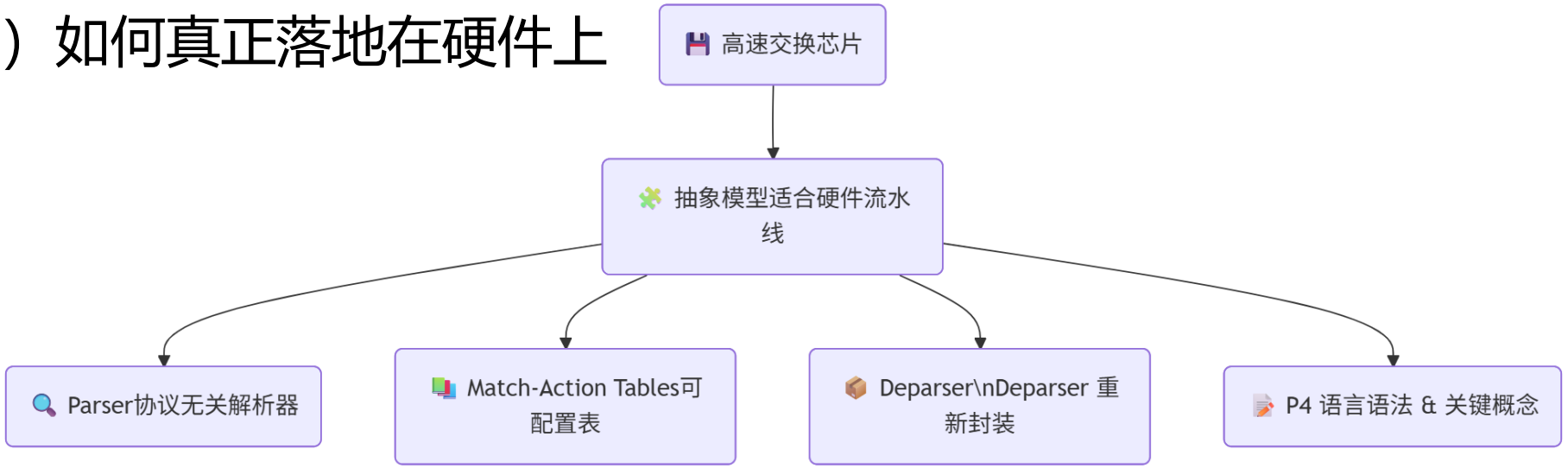
1. Introduction：关键问题

- 核心挑战：语言要站在“中间地带”
  - 一方面：表达能力要足够强
    - 能描述现实网络中的多种转发逻辑：路由、封装/解封装、ACL、隧道、计量等
  - 另一方面：又要便于硬件实现
    - 必须能高效映射到高速流水线：有限的表项数、有限的动作复杂度、固定的流水线深度
  - 问题本质
    - 语言如果太自由（像 C++ / Click），很难静态分析和做硬件映射；太受限（像 OpenFlow 1.x），又表达不了新协议和复杂处理

定位：介于 OpenFlow 与通用语言之间

1. Introduction：论文主张与设计路线

- 先提出一个适合硬件流水线的抽象模型
  - 协议无关解析器+ 一系列可配置的 match+action tables
- 再在此模型之上设计
  - 语言（P4 的语法与关键概念）
  - 编译流程（从 P4 程序到具体目标，如 ASIC / NPU / 软件交换机）
- 后续章节将围绕“模型 + 语言 + 编译器”展开，说明这个最佳位置（ sweet spot ）如何真正落地在硬件上



Outline

I. Introduction

II. Abstract Forwarding Model

III. A Programming Language

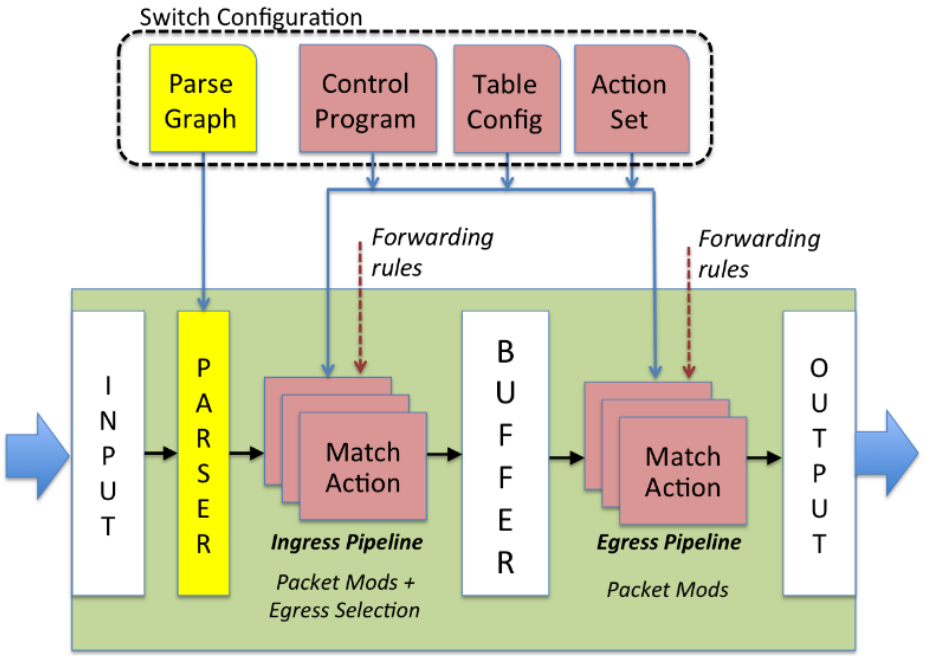
IV. P4 Language by Example

V. Compiling a P4 Program

VI. Conclusion

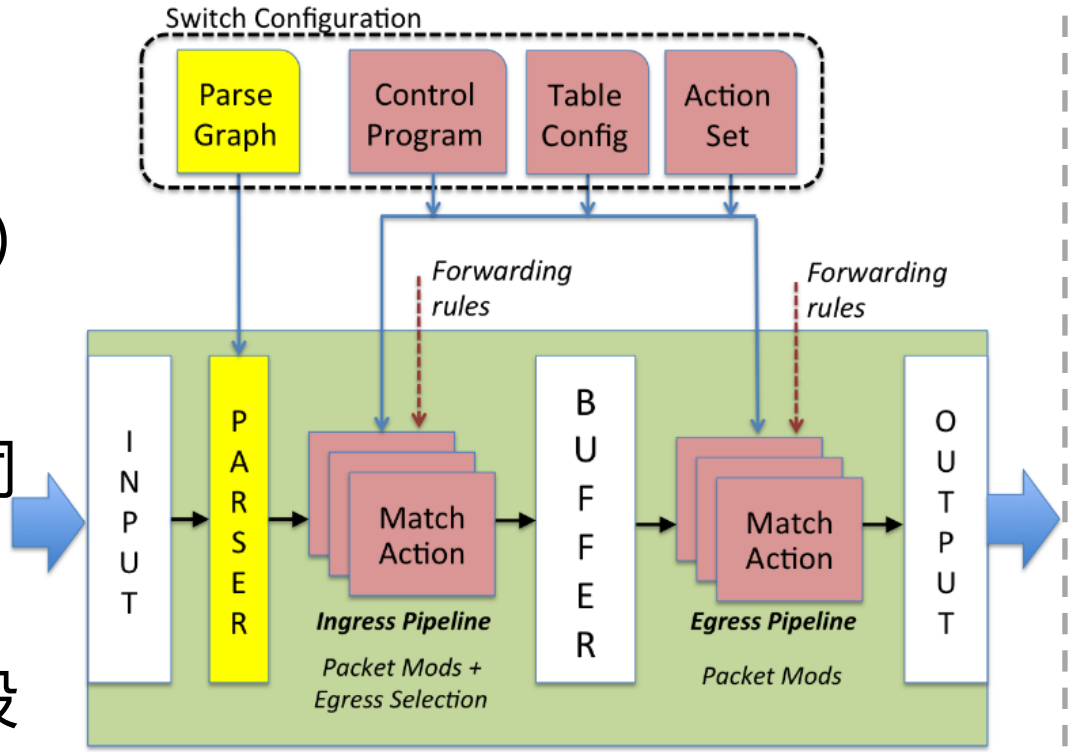
2. ABSTRACT FORWARDING MODEL

- 设计目标：统一抽象各类转发设备
  - 论文先提出一个抽象的数据平面转发模型，希望用同一套结构概括：二层交换机、三层路由器、NAT、负载均衡器、部分中间盒等
  - 后面的 P4 语言与编译流程，都默认“目标设备”符合这一抽象模型
- 模型的核心结构
  - 1. Packet Parser（解析器）
    - 按协议格式从比特流中提取各层 header，填充到 P4 中定义的 header 实例
    - 同时初始化部分元数据（如 ingress 端口、包长度等）



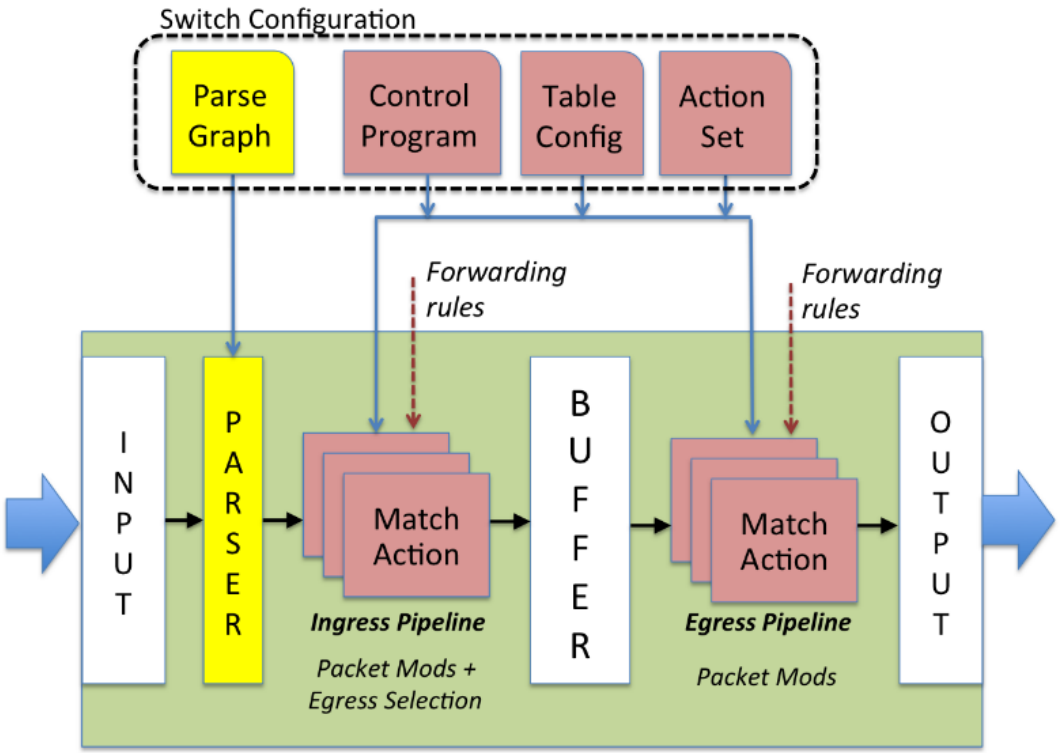
2. ABSTRACT FORWARDING MODEL

- 模型的核心结构
- 1. Parser（解析器）的角色
- 解析步骤：从比特流到“头部对象”
  - 到达交换机的包，先进入解析器（parser）
  - P4 假设底层实现了一个 状态机：
    - 从包头开始，按状态机依次跳转到不同 header
    - 在每个状态里从比特流中提取对应字段的值。解析器只关心“这一段比特是某个字段、多少位”，不解释协议语义



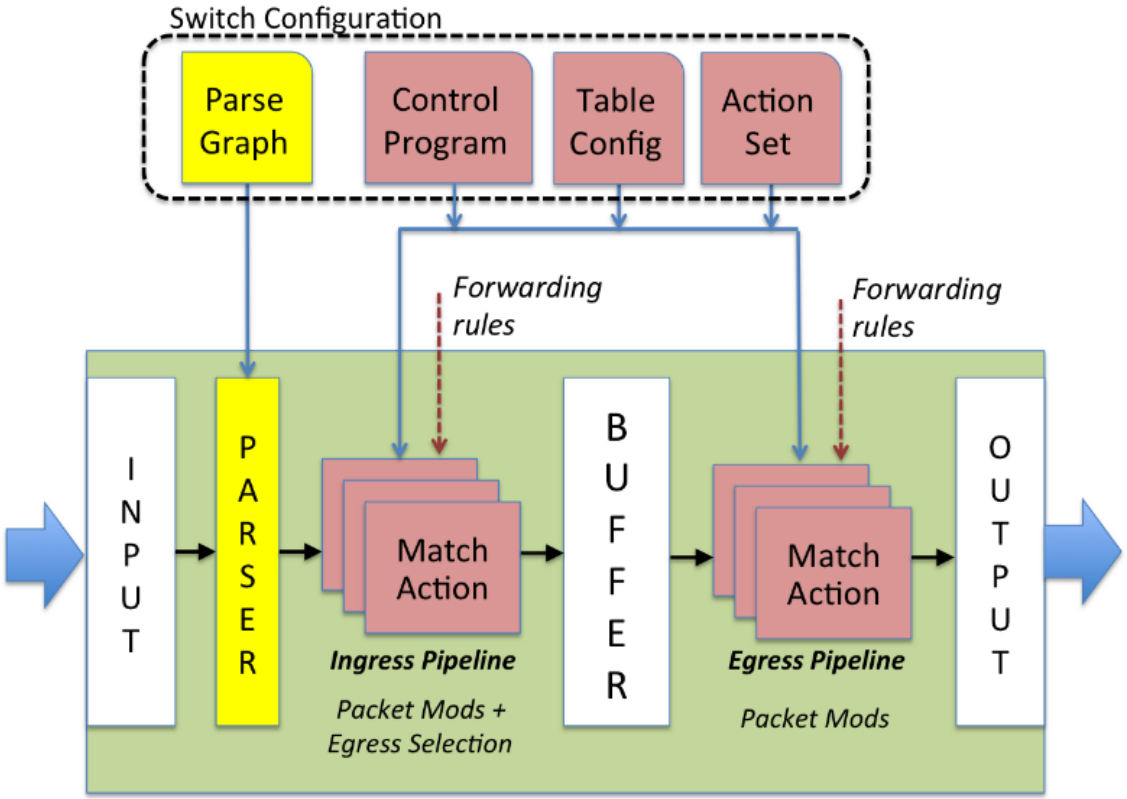
2. ABSTRACT FORWARDING MODEL

- 模型的核心结构
- 2. Ingress Match+Action Pipeline
  - 一串按顺序执行的 match-action 表
  - 基本职责：在解析后决定包的转发行为
    - 选择一个或多个 egress 端口
    - 决定放入哪个队列
    - 决定是否复制（多播/SPAN/送控制平面）或直接丢弃包
  - 可能涉及多张表：路由选择、ACL 过滤、隧道封装/打标签等逻辑，都可以串接在 ingress 流水线的一系列表中完成



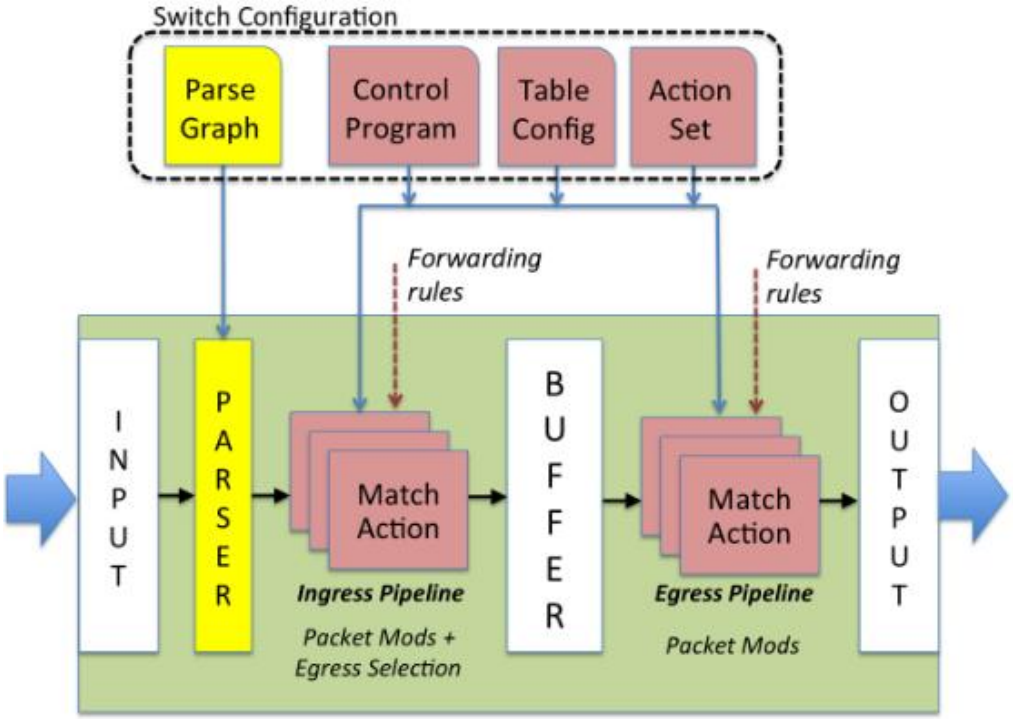
2. ABSTRACT FORWARDING MODEL

- 模型的核心结构
- 3. Egress Match+Action Pipeline
  - 基本职责：在已经确定 egress 端口之后，对每个出口实例进行进一步处理
  - 典型用途：对多播副本进行差异化修改，例如为不同下游链路设置不同的标签或优先级



2. ABSTRACT FORWARDING MODEL

- Metadata 的作用
  - Metadata 是和包一起在流水线中“流动”的中间状态
    - 保存输入端口、临时标记、策略决策结果等
  - Parser、Ingress、Egress 通过读写 metadata 来传递信息、协调处理逻辑



- 通俗理解：Metadata 就是包在交换机里一路“背着”的隐形小纸条，上面记着入口端口、选好的出口、队列号之类的信息，给表匹配和动作逻辑用，但不会真的写进报文里

2. ABSTRACT FORWARDING MODEL

- 与 OpenFlow 抽象的三点差异
- 1. 解析器（Parser）
  - OpenFlow：假定交换机有一个固定解析器，只能解析标准里写死的那几种协议头
  - P4 抽象模型：解析器本身可编程，可以在语言中定义新 header 类型和合法的 header 序列，从而识别新头部
- 2. 流水线结构（Pipeline）
  - OpenFlow：match+action 阶段默认视为串行的一条流水线，每级顺序固定
  - P4 抽象模型：允许多个 match+action stages 以串行、并行或组合的方式组织，便于表达更复杂的处理结构和并行表

2. ABSTRACT FORWARDING MODEL

- 与 OpenFlow 抽象的三点差异
- 3. 动作（Actions）
  - OpenFlow：提供一小组预定义动作（转发、丢弃、改写等），集合相对有限
  - P4 抽象模型：动作由一组协议无关的原语（protocol-independent primitives）组合而成，可以构造出更复杂的处理逻辑，同时仍适合硬件实现

2. ABSTRACT FORWARDING MODEL

- Configure vs Populate：两类操作
- 1. Configure（配置阶段）面向数据平面结构本身
  - 配置 parser 状态机：定义要识别哪些头部、合法的解析路径
  - 指定表的个数与顺序：确定整个 ingress / egress 流水线结构
  - 为每张表声明：可以匹配哪些字段（headers / metadata）支持哪些动作类型（action 原语组合）
  - 可以理解为：把 P4 程序“烧进”交换机，生成一块可编程转发管线
- 2. Populate（填表阶段）面向具体转发策略：
  - 控制器向已经定义好的表中写入 / 修改 / 删除表项
  - 表项内容包括：匹配条件（match fields）+ 选择的动作及参数
  - 可以理解为：在既定流水线上，装载当前要生效的策略配置（路由、ACL、隧道等）

2. ABSTRACT FORWARDING MODEL

- 对不同硬件的统一抽象
- 1. 固定功能 ASIC（Fixed-function ASIC）
  - 硬件的解析器和流水线结构基本写死，实际上不能真正“重配置”
  - P4 编译器的主要任务：检查这段 P4 程序是否能嵌入现有管线结构；做表合并、字段映射等“适配工作”，而不是重塑流水线
- 2. 可编程 ASIC / NPU / FPGA
  - 这类目标提供可重配置的解析器和多级 match-action 流水线
  - P4 编译器负责：
    - 生成具体的 parser 状态机配置
    - 决定物理表的划分与顺序
    - 把 P4 动作映射为底层支持的动作逻辑或微码

2. ABSTRACT FORWARDING MODEL

- 对不同硬件的统一抽象
- 3. 软件交换机（Software Switch）
  - 抽象模型可以直接映射为软件结构：
    - parser → 报文解析函数 / 模块
    - 表 → 查表数据结构（哈希表、TCAM 仿真等）
    - 动作 → 普通函数调用 / 代码片段
  - 因此，同一份 P4 程序可以在软件交换机上运行，用于开发、测试与原型验证，再迁移到硬件目标

Outline

I. Introduction

II. Abstract Forwarding Model

III. A Programming Language

IV. P4 Language by Example

V. Compiling a P4 Program

VI. Conclusion

3. A PROGRAMMING LANGUAGE

- 为什么需要新的“编程语言”？
  - 只用 OpenFlow 不行：OpenFlow 1.x 只能在标准里写死的字段上填表项，不能描述新头部格式、解析流程，也不能重构流水线结构
  - 直接用 C / C++ / Click 又太自由：Click 这类任意 C++ 模块，表达能力很强，但约束太少，很难从中自动推导出“表之间的依赖关系”和“哪些表可并行”，也难直接映射到专用硬件的 parse–match–action 流水线
- 本文的选择：为抽象模型专门造一门语言，作者因此提出：在前面抽象转发模型之上，设计一门专用语言（P4），语义紧贴 “可编程 parser + 多级 match-action + 控制流”，又足够规整，便于编译器做静态分析和硬件映射
- 一句话概括：OpenFlow 太硬，C/Click 太软

  → **P4 是为“抽象转发模型”量身定做的那块刚刚好的中间层语言**

3. A PROGRAMMING LANGUAGE

Click 模块与 P4 的对比

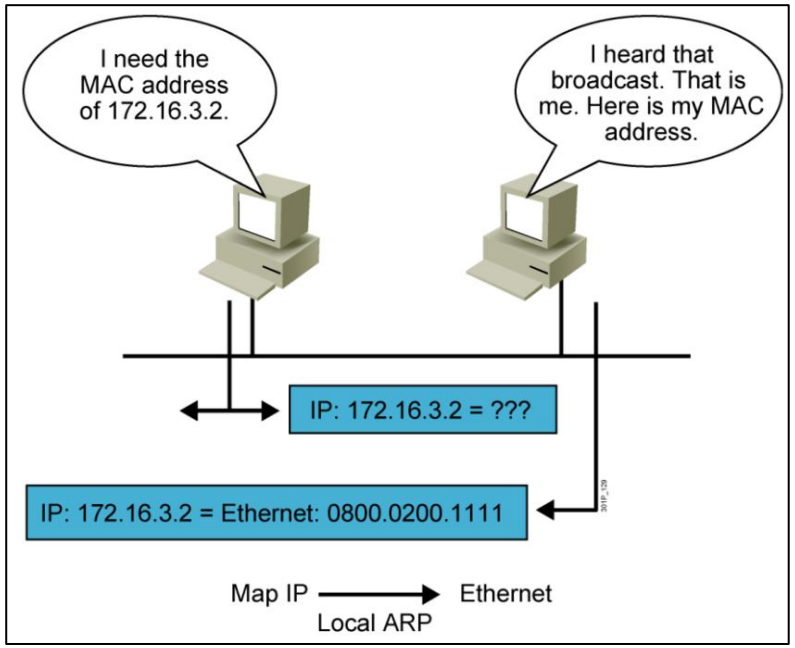
- Click 的优势：非常灵活用
  - 很多小模块随意连线，可以拼出路由器、防火墙、NAT 等各种数据平面功能。表达能力几乎接近通用 C/C++，基本不限制你想实现什么逻辑
- Click 的问题：不适合“表 + 控制器填表”模式
  - 没有统一的 match+action 表 抽象，也没有控制器远程填表的标准接口
  - 编译器看到的只是“一堆 C++ 模块 + 任意连接”，很难自动推导哪些能做成表、哪些可并行、需要多少硬件资源
  - 结果：**很灵活，但很难系统化地映射到专用的 parse–match–action 硬件流水线**

3. A PROGRAMMING LANGUAGE

数据依赖与表依赖图（TDG）

- 1. 编译器需要回答两个关键问题：哪些表之间有数据依赖，只能按顺序执行？
  - 附属关系确定哪些表可以并行执行。例如，由于IP路由表和ARP表之间的数据依赖性，需要顺序执行

右图形象地展示了ARP的过程。执行 ARP 过程的前提是知道目标主机的 IP 地址，因此在不知道下一跳 IP 的情况下是没办法得到下一跳的 MAC Addr 的



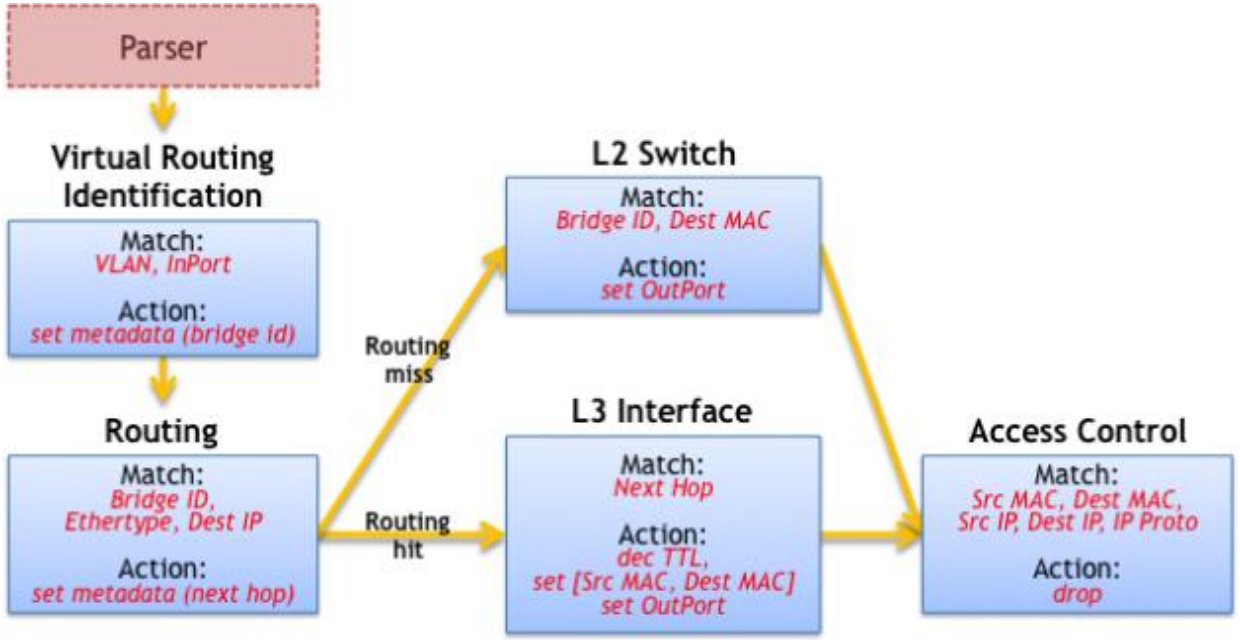
- 论文指出：必须让程序员能（显式或隐式）表达这些串行依赖关系，编译器才能合理排布流水线

3. A PROGRAMMING LANGUAGE

数据依赖与表依赖图（TDG）

- 2. Table Dependency Graph（表依赖图）
  - P4 程序首先会被编译成一个 Table Dependency Graph（TDG），如图
  - 节点：match+action 表
  - 有向边：数据依赖 —— 当一个表的输出字段被后续表用作匹配条件或动作输入时，就画一条边

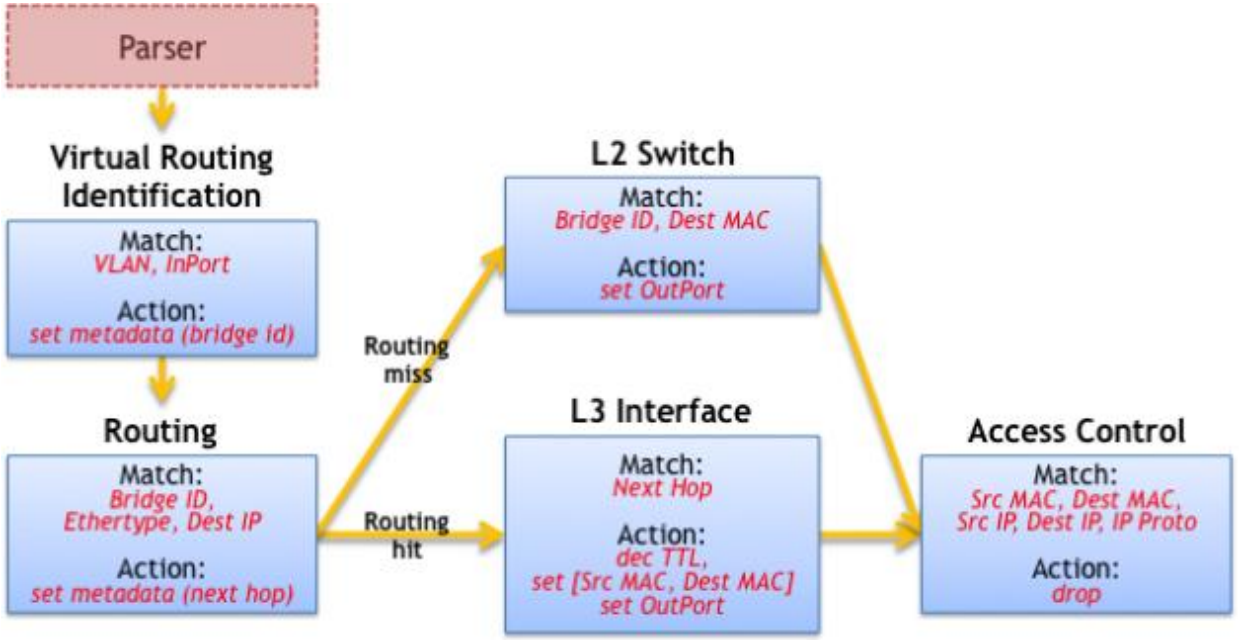
- TDG 用来描述：每张表的输入字段、动作效果、控制流关系；编译器据此判断哪些表可并行、哪些必须串行，并映射到具体流水线级



3. A PROGRAMMING LANGUAGE

数据依赖与表依赖图（TDG）

- 3. 例子：L3 路由表 → ARP 表
- 图中 L2/L3 交换机的 TDG 就给出了一个经典例子：
  - 先查路由表：若命中则从 L3 出口出；否则走 L2 口
  - 再查 Access Control：匹配中则说明控制访问配置不允许这个包出去
- 这里存在明显的数据依赖：
  - L2还是L3必须等待路由表查询结果
  - 因此表必须串行执行，在 TDG 中表现为从 Routing 指向 L2 或 L3 的边



3. A PROGRAMMING LANGUAGE

- 两阶段编译流程
- 第 1 阶段：P4 控制程序 →表之间的“关系图”TDG
  - 编译器解析控制程序中的表调用和条件语句，把命令式控制流转换为TDG表示
  - 对每张表做 读/写 字段分析，判断它依赖哪些 头/metadata、会修改哪些字段
  - **构造 Table Dependency Graph（TDG）**
- 第 2 阶段：TDG → 具体目标设备
  - 由目标相关的后端，把 TDG 映射到具体交换机资源：表的数量与顺序、表宽度与高度、TCAM/SRAM 使用等
  - 必要时**对多张逻辑表进行操作**（合并 / 拆分 / 重排）
- 为什么需要“两阶段”编译流程？简单理解：先把“程序里表之间怎么依赖”抽象成一张图**（阶段 1，跟硬件无关）**，再根据不同芯片的资源条件，把这张图落到具体流水线里面**（阶段 2，跟硬件相关）**

Outline

I. Introduction

II. Abstract Forwarding Model

III. A Programming Language

IV. P4 Language by Example

V. Compiling a P4 Program

VI. Conclusion

4. P4 Language by Example

- 示例场景：mTag 网络拓扑
  - mTag 就是论文里用 P4 定义的、带四个 8 比特层次字段的“自定义转发标签头”，类似“结合了 MPLS label + PortLand Pseudo MAC”的“小玩具”，用来演示 P4 如何定义新协议头、解析它、并在 match-action 表里根据它转发
- Why mTag? → 现存问题：
  - 主机数量增长 → 核心 L2 表溢出
  - 希望用“分层标签”简化核心转发逻辑
- mTag 思路：
  - 用 32 bit 标签表示“上行 / 下行路径信息”
  - 核心交换机只需要看标签的某一字节即可转发

4. P4 Language by Example

mTag 与现有方案对比

- MPLS优点：支持标签堆栈，可以在核心网络里用 label 简化转发
  - 问题：要在数据中心内部部署一套多标签的分发协议，实现和运维都比较复杂
- PortLand做法：通过修改主机使用的 MAC 地址 来编码“所在位置”
  - 问题：需要改写 MAC，会破坏现有运维和调试工具对 MAC 的假设；还需要在主机上部署新代理来处理 ARP 请求
- mTag 的设计：在边缘 ToR 交换机 上为报文添加一个 32-bit mTag 头部，核心只看标签转发
  - 属性：不改变主机的 MAC / IP，对终端和现有工具透明；标签可携带类似 MPLS 的“路径 / 位置”信息，核心交换机只需检查其中一个字节即可完成转发

4. P4 Language by Example

- P4 程序组成回顾：在 mTag 示例中的实例化
- 1）头部格式（Headers）
  - P4 程序首先用 header 定义：ethernet、vlan、mTag、ipv4 等头部的字段名与位宽。例如 mTag 由 up1/up2/down1/down2/ethertype 组成
- 2）解析顺序（Parser）
  - 使用 parser 描述解析状态机：从 ethernet 开始，根据 ethertype 跳转到 vlan、ipv4 或 mTag，再根据 vlan.ethertype 判断是否进入 mTag，最后解析到 ipv4 等
  - 例如，解析 IP 报文时，版本一定是最先被解析的

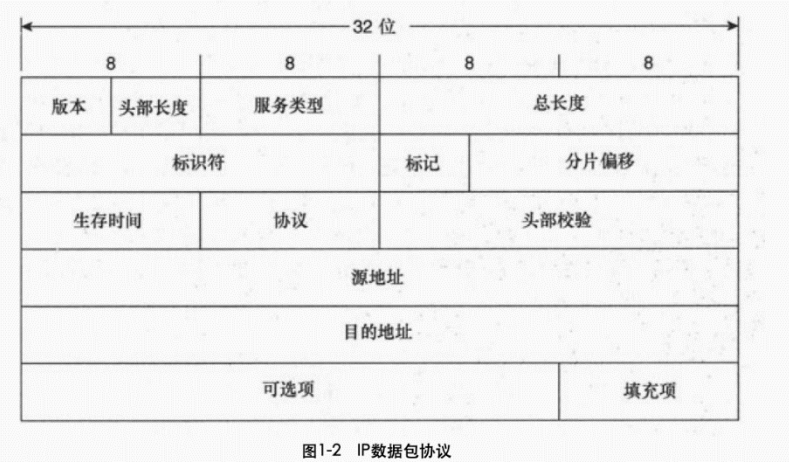


图1-2 IP数据包协议

4. P4 Language by Example

- P4 程序组成回顾：在 mTag 示例中的实例化
- 3）为边缘 / 核心交换机定义不同的表（Tables）
  - 边缘 ToR：有 mTag_table、source_check、local_switching、egress_check 等
  - 例如 mTag_table 在 ethernet.dst_addr 和 vlan.vid 上匹配，选择要添加的 mTag
- 4）定义添加 / 移除 mTag 的动作（Actions）
  - 如 add_mTag：调用 add_header(mTag)、copy_field、set_field 等原语，把 mTag 插入 VLAN 后并设置转发端口
  - 还包含与之相反的 strip 动作，用于在边缘剥离标签

4. P4 Language by Example

- mTag Header 定义
  - mTag 头结构（右图）：
    - up1、up2：上行两级聚合标识（各 8 bit）
    - down1、down2：下行两级聚合标识（各 8 bit）
    - ethertype：16 bit（记录后续协议类型）
  - 设计意图：32 比特标签足以编码多级拓扑层次
  - 核心只需根据自己的层级选择对应字段进行匹配
  - 标准以太网和 VLAN 头部格式如下

```
header mTag {
    fields {
        up1 : 8;
        up2 : 8;
        down1 : 8;
        down2 : 8;
        ethertype : 16;
    }
}
```

```
header ethernet {
    fields {
        dst_addr : 48; // width in bits
        src_addr : 48;
        ethertype : 16;
    }
}
```

```
header vlan {
    fields {
        pcp : 3;
        cfi : 1;
        vid : 12;
        ethertype : 16;
    }
}
```

4. P4 Language by Example

- Parser 作为状态机
  - 解析流程：
    - start 状态：假定先解析 Ethernet
    - ethernet 状态：根据 ethertype 决定是 VLAN 还是 IPv4
    - vlan 状态：根据 ethertype 判断是否存在 mTag
    - mTag 状态：继续解析后续 IPv4 等头部
- 特点：每个 parser state 对应一个 header 类型，通过 switch-case 描述下一步

```
parser start {
    ethernet;
}
```

```
parser ethernet {
    switch(ethertype) {
        case 0x8100: vlan;
        case 0x9100: vlan;
        case  0x800: ipv4;
        // Other cases
    }
}
```

```
parser vlan {
    switch(ethertype) {
        case 0xaaaa: mTag;
        case  0x800: ipv4;
        // Other cases
    }
}
```

```
parser mTag {
    switch(ethertype) {
        case 0x800: ipv4;
        // Other cases
    }
}
```

4. P4 Language by Example

- 表定义——mTag_table（边缘交换机）
  - 程序员描述如何在匹配+动作阶段中匹配定义的标头字段（例如，它们应该是精确匹配、范围匹配还是通配符匹配？）
  - mTag_table 读取字段：
    - ethernet.dst_addr：精确匹配
    - vlan.vid：精确匹配
  - 动作列表：
    - add_mTag（包头插入mTag头部）
  - 表大小：
    - max_size：例如 20000 条
  - 解释：每个表项决定：对某个（目的 MAC，VLAN）组合 → 应该打上什么 mTag、转到哪一端口

```
table mTag_table {
    reads {
        ethernet.dst_addr : exact;
        vlan.vid : exact;
    }
    actions {
        // At runtime, entries are programmed with params
        // for the mTag action.  See below.
        add_mTag;
    }
    max_size : 20000;
}
```

4. P4 Language by Example

- source_check 表：
  - 检查“入口端口”和“是否带 mTag”是不是匹配，如果不匹配就报警 / 丢弃，如果匹配就把 mTag 处理干净、记个状态
  - 动作：fault_to_cpu / strip_mtag / pass

```
table source_check {
    // Verify mtag only on ports to the core
    reads {
        mtag : valid; // Was mtag parsed?
        metadata.ingress_port : exact;
    }
    actions { // Each table entry specifies *one* action

        // If inappropriate mTag, send to CPU
        fault_to_cpu;

        // If mtag found, strip and record in metadata
        strip_mtag;

        // Otherwise, allow the packet to continue
        pass;
    }
    max_size : 64; // One rule per port
}
```

4. P4 Language by Example

- local_switching 表：
  - 做本地二层转发：如果目标主机就在本 ToR
  - 未命中时：说明不是本地主机，转入 mTag_table

```
table local_switching {
    // Reads destination and checks if local
    // If miss occurs, goto mtag table.
}
```

```
table egress_check {
    // Verify egress is resolved
    // Do not retag packets received with tag
    // Reads egress and whether packet was mTagged
}
```

- egress_check 表：
  - 确认出口端口是否已解析
  - 避免对已经带有 mTag 的包重新打标签

4. P4 Language by Example

- 1. 动作原语（primitive actions）示例
  - set_field：设置某个字段的值，支持按位掩码修改
  - copy_field：把一个字段的值拷贝到另一个字段
  - add_header / remove_header：增加或删除一个头部实例（整个 header）
  - increment：对字段做自增 / 自减（如 TTL–1）
  - checksum：根据若干字段重新计算校验和
- 2. 复合动作（action functions）
  - P4 程序员不会直接在表项里堆一堆原语，而是先定义“动作函数”：里面由多条原语并行执行组成一个高层动作（语义上视为一次完成）
  - 例如：add_mTag：add_header(mTag) + 一组 copy_field / set_field，最后设置 metadata.egress_spec
- 简单理解，**原语 = 指令；动作函数 = 一小段并行执行的指令组合**

4. P4 Language by Example

- add_mTag 动作详解
- （1）操作步骤（逻辑）
- add_header(mTag)：在 VLAN 头后面插入一个新的 mTag 头部
- copy_field(mTag.ethertype, vlan.ethertype)：把原来 VLAN 里的 Ethertype 拷贝到 mTag.ethertype，这样 mTag 里就记住了“原来后面跟的是什么协议”。
- set_field(vlan.ethertype, 0xaaaa)：把 VLAN 的 Ethertype 改成固定值 0xaaaa，用来在解析时标记“这里后面有一个 mTag 头”

```
action add_mTag(up1, up2, down1, down2, egr_spec) {
    add_header(mTag);
    // Copy VLAN ethertype to mTag
    copy_field(mTag.ethertype, vlan.ethertype);
    // Set VLAN's ethertype to signal mTag
    set_field(vlan.ethertype, 0xaaaa);
    set_field(mTag.up1, up1);
    set_field(mTag.up2, up2);
    set_field(mTag.down1, down1);
    set_field(mTag.down2, down2);

    // Set the destination egress port as well
    set_field(metadata.egress_spec, egr_spec);
}
```

4. P4 Language by Example

- add_mTag 动作详解
- （1）操作步骤（逻辑）
- add_header(mTag)：在 VLAN 头后面插入一个新的 mTag 头部
- copy_field(mTag.ethertype, vlan.ethertype)：把原来 VLAN 里的 Ethertype 拷贝到 mTag.ethertype，这样 mTag 里就记住了“原来后面跟的是什么协议”。
- set_field(vlan.ethertype, 0xaaaa)：把 VLAN 的 Ethertype 改成固定值 0xaaaa，用来在解析时标记“这里后面有一个 mTag 头”

设计意图（直观理解）：对外仍然是“标准以太网 + VLAN + Ethertype”的形式，只是：当接收端看到 VLAN.ethertype=0xaaaa，就知道：“VLAN 后面不是直接 IP，而是先有一个 mTag。”这样就用一个 特殊 Ethertype 值把“插了 mTag”编码进了标准以太网框架里，既方便解析器识别，又对主机和其他设备保持兼容

4. P4 Language by Example

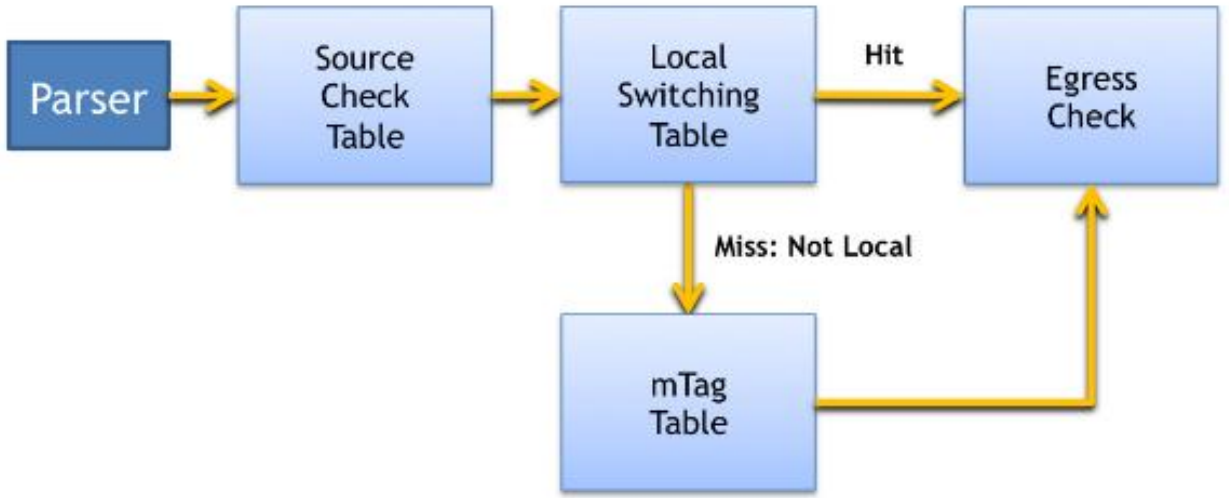
- 1. 继续设置标签字段 up1/ up2/ down1/ down2 都来自 mTag_table 的表项参数，表示该流在数据中心拓扑中的“上行 / 下行层次编码”
- 2. 设置出口端口 set_field(metadata.egress_spec, egr_spec) 同样由表项提供 egr_spec 参数，直接在动作里把 转发端口 写入 metadata，供后续 egress 处理使用
- 3. 小结：add_mTag 同时完成：给报文“封装”好 mTag 标签（四个层次字段 + ethertype）；决定这包接下来要从哪个端口发出去
- 简单理解，**add_mTag = 封装路径标签 + 选好下一跳**

```
action add_mTag(up1, up2, down1, down2, egr_spec) {
    add_header(mTag);
    // Copy VLAN ethertype to mTag
    copy_field(mTag.ethertype, vlan.ethertype);
    // Set VLAN's ethertype to signal mTag
    set_field(vlan.ethertype, 0xaaaa);
    set_field(mTag.up1, up1);
    set_field(mTag.up2, up2);
    set_field(mTag.down1, down1);
    set_field(mTag.down2, down2);

    // Set the destination egress port as well
    set_field(metadata.egress_spec, egr_spec);
}
```

4. P4 Language by Example

- 控制程序 main() 的逻辑
- 1. 先跑 source_check：检查“入口端口”和“mTag 状态”是否匹配，如果发现错误，就做标记
- 2. 如果没有 ingress_error：说明入口检查通过，继续做 local_switching，尝试按本地 MAC/VLAN 转发到直连主机
- 3. 如果此时 egress_spec 仍未定义：表示不是本地目的主机，就去查 mTag_table，给包打上 mTag，准备走核心路径
- 4. 最后执行 egress_check：统一检查：出口是否已经确定？有没有对带 mTag 进来的包错误地重新打标?



Outline

I. Introduction

II. Abstract Forwarding Model

III. A Programming Language

IV. P4 Language by Example

V. Compiling a P4 Program

VI. Conclusion

5. Compiling a P4 Program

- 编译器的任务
  - 输入：
    - P4 程序（头部定义、解析器、表、动作、控制程序）
  - 输出：
    - 目标设备的配置：
      - 解析器状态机与状态表
      - 各级流水线的表结构与动作配置
      - 计数器 / policer 等资源绑定
  - 核心挑战：
    - 在有限硬件资源约束下完成映射
    - 同时保证语义正确性与性能

5. Compiling a P4 Program

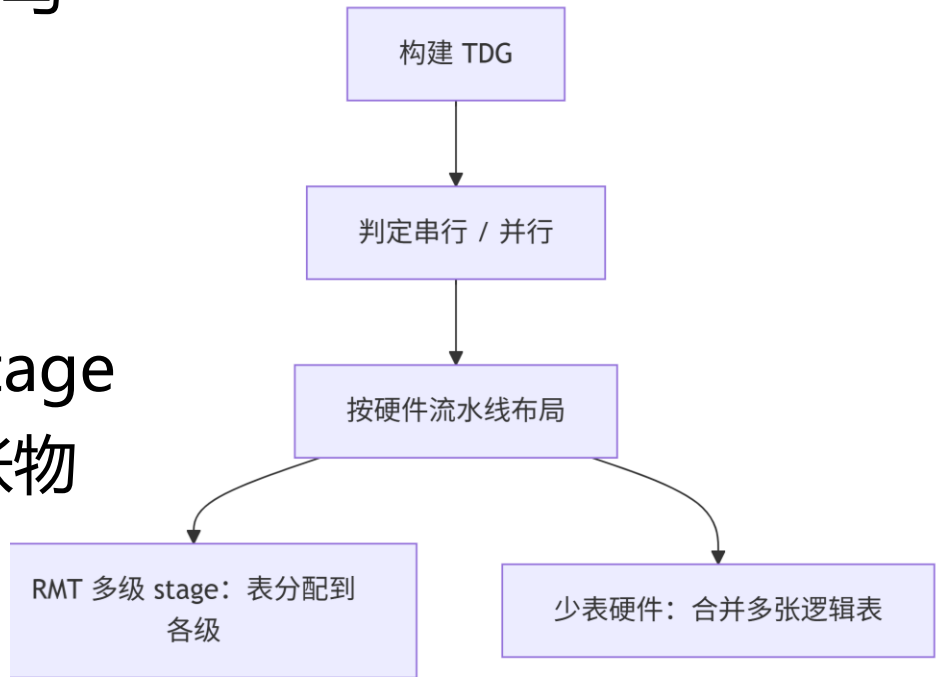
- 解析器编译示例
  - 对于可编程解析器设备：
    - 编译器生成状态表，类似：
      - 当前状态：vlan，匹配 Ethertype=0xaaaa → 下一状态：mTag
      - 当前状态：vlan，Ethertype=0x800 → 下一状态：ipv4

Current State	Lookup Value	Next State
vlan	0xaaaa	mTag
vlan	0x800	ipv4
vlan	*	stop
mTag	0x800	ipv4
mTag	*	stop

  - 对于固定解析器设备：
    - 只能验证 P4 程序的 parser 是否“与硬件支持的协议集合兼容”

5. Compiling a P4 Program

- 控制程序编译与资源映射
- 步骤：
- 1. 从 P4 控制程序构建 TDG：分析每个表的读 / 写集合
- 2. 判断哪些表可并行执行，哪些必须串行
- 3. 根据硬件流水线结构进行布局：
  - 有多级 stage 的 RMT：安排哪些表在哪一 stage
  - 只有少数表的硬件：把多张逻辑表合并为一张物理表
- 运行时：当控制器安装新规则时，可能需要“组合多张逻辑表的条件”生成新表项



5. Compiling a P4 Program

- 针对不同目标的编译策略示例
  - 软件交换机：
    - 直接映射为软件中的哈希表 / TCAM 仿真结构
  - 支持 RAM + TCAM 的 ASIC：
    - 精确匹配表（如 mTag_table）映射到 SRAM 哈希
    - 需要通配 / 前缀匹配的表映射到 TCAM
  - 支持平行表的 ASIC：
    - 编译器尽量并行执行无依赖的表，减少流水线长度

Outline

I. Introduction

II. Abstract Forwarding Model

III. A Programming Language

IV. P4 Language by Example

V. Compiling a P4 Program

VI. Conclusion

6. Conclusion

- 问题：
  - OpenFlow 面向固定功能交换机，难以支持新协议与新流水线
- 贡献：
  - 提出抽象转发模型（parser + match-action pipelines）
  - 给出 P4 语言草案，实现“协议无关 + 目标无关”的数据平面编程
  - 描述从 P4 到具体硬件的编译流程
- 影响：
  - 成为后续 P4 语言标准与可编程交换芯片的基础
- 后续工作与思考论文
  - 有意未覆盖的部分：拥塞控制原语、队列调度、流量监控等
  - 后续研究方向：如何扩展数据平面编程能力，又不破坏可编译性与线速性能如何结合 P4 做网络测量、拥塞控制、NFV 等应用