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 [**?**]，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。

2 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。

2.1 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 [**?**] 和 packet transaction [**?**] 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 [**?**, **?**]。例如，考虑以下简单路由程序：

```
L1: def onPacket(pkt):
L2:     policy = policy(pkt.srcIP)
L3:     dstSW = switch(pkt.dstIP)
L4:     route = route(policy, dstSW)
L5:     return route
```

该路由程序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 1所示（即将程序主动编译为数据路径）。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 [**?**, **?**]），该条件都可以满足。

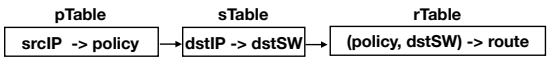

图 1: 上述简单路由程序转换后的三张流表。每张表的结构 $a \rightarrow b$ 表示流表的匹配字段为 a 以及数据包匹配后可以设置 b 的值。

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，我们选择降低总流表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 [**?**, **?**]。

2.2 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 [**?**, **?**] 不能在它们的语言中支持循环。而循环的实用性将在如下循环程序中说明。

```
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
```

该简单循环程序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以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 [**?**]）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。上述简单循环程序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，简单循环程序中的循环可以转换为：

(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 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。

3.1 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 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。

3.2 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 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 $1 \leqslant i < n$。

RSP 及其 SSA 形式的示例如图 2所示。

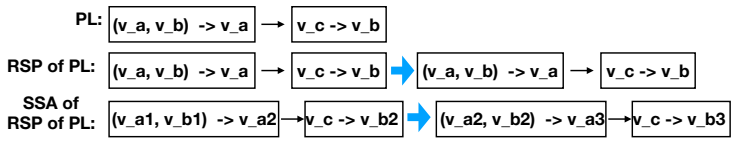

图 2: 关于流水线 (PL), PL 的 2 个 RSP 和它们的 SSA 形式的示例。

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。

4 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。

4.1 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 2中 SSA 形式的流水线的 DFG 在图 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 $(v_{a2}, v_{b2}) \rightarrow v_{b3}$ 。

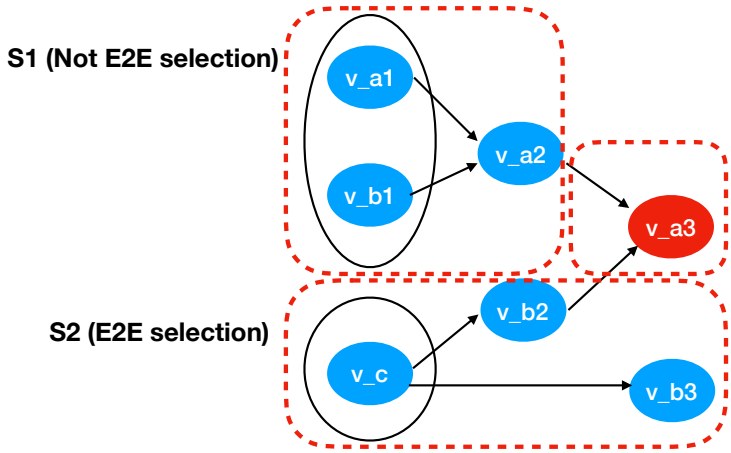

图 3: The DFG of a pipeline and the source vertices selection.

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遇到的处理步骤的数量，降低数据包的处理时延。

4.2 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 上实施的有效性和效率（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吧的时延，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 4中所示。

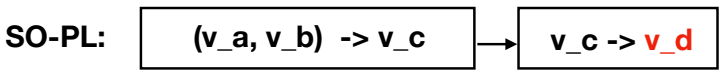

图 4: SO-PL 示例.

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 2中 SSA 格式流水线（我们称为 pl^n_{ssa}），考虑一个流表 t_y，其输出变量为 v_{a2}, v_{b2}, v_c，则 t_y 是 pl^n_{ssa} 的全输出表。当 pl^n_{ssa} 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命题。

命题 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 1 *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 ??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。

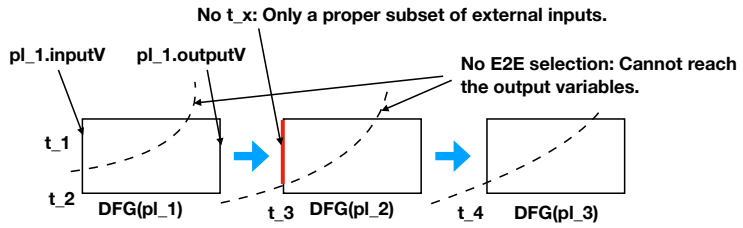

图 5: 存在性命题的证明，图中三个流水线 (pl_1, pl_2, pl_3) 在假设下只能被合并到至少四张表 (t_1, t_2, t_3, t_4) 。每个方块是一个流水线的 DFG

存在性命题表明，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。由于该命题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命题，我们给出了考虑最优流水线设计的全输出表的位置命题。

命题 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 $MinIndexPL(t_x) \leqslant k$。

证明 2 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，这被存在性命题排除。

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（根据位置命题的结果，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 [**?**]，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）。根据存在性命题，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。

5 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。

5.1 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 8所示。水平轴指定循环的迭代次数 (n)。图 6和图 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 6中的 $m = 10$, 图 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。

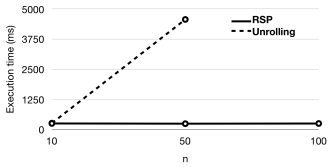

图 6: $m = 10$.

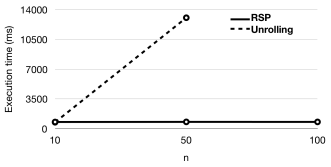

图 7: $m = 20$.

图 8: The execution time of two approaches by changing the number of iterations (n) and the number of vertices (m).

5.2 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 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 1: 单表方法和 RSP 方法下的流规则数

6 相关工作

高级 SDN 编程模型: 对于被动式（reactive）高级 SDN 编程模型 [**?**, **?**, **?**], 数据包被转发到控制器并被处理。Maple [**?**] 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 [**?**, **?**]，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 [**?**, **?**, **?**, **?**] 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 [**?**] 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。