面向现场可编程门阵列的可复用高吞吐量网络应用流水线自动化框架

Jean Bruant，OVHcloud；格勒诺布尔-阿尔卑斯大学、法国国家科学研究中心、格勒诺布尔国立综合理工学院[1]、TIMA，法国Pierre-Henri Horrein，OVHcloud，法国Olivier Muller，格勒诺布尔-阿尔卑斯大学、法国国家科学研究中心、格勒诺布尔国立综合理工学院[1]、TIMA，法国Frédéric Pétrot，格勒诺布尔-阿尔卑斯大学、法国国家科学研究中心、格勒诺布尔国立综合理工学院[1]、TIMA，法国

arXiv:2601.15151v1 [cs.AR] 21 Jan 2026

在全球通信流量持续增长的背景下，云服务提供商致力于部署可扩展的基础设施，以满足多样化的需求。作为网络基础设施的一部分，现场可编程门阵列经过专门设计，可确保实现低延迟和高吞吐量数据包处理。然而，硬件设计流程缓慢，削弱了现场可编程门阵列在不断演进的敏捷基础设施中发挥作用的能力，而这种演进涵盖从事件响应到长期转型的各个方面。在种类繁多的现场可编程门阵列上部署和维护网络功能，需要针对多个现场可编程门阵列目标平台对硬件设计进行精细调整。为解决这一问题，我们提出了*PAF*，一种基于面向流水线的设计方法的架构参数化框架。*PAF*（流水线自动化框架）基于*Chisel*实现。Chisel 是一种嵌入 Scala 的硬件构造语言（HCL），我们利用它与电路展开进行交互。将*PAF* 应用于工业级网络数据包分类系统后，结果表明它具备高效的参数化能力，能够在多个现场可编程门阵列上复用并优化同一流水线设计。此外，*PAF* 着重根据架构意图描述流水线，从而在表达复杂功能时减少代码行数。最后，*PAF* 证实，自动化并不意味着失去对架构的精细控制：与同等的穷尽式描述实现相比，它实现了相当的性能和资源使用量。

CCS 概念： • **硬件 → 硬件描述语言和编译**； 网络硬件；**可重构逻辑和现场可编程门阵列**； 硬件加速器；可演进硬件。

加法 关键词和短语：硬件构造语言、信号同步、Chisel

1 引言

云基础设施及其底层网络基础设施不断演进，以满足从事件响应、可扩展性到维护等各种需求。作为可重构硬件加速器，现场可编程门阵列能够提供网络算法所需的高吞吐量和低延迟，同时还具备快速更新能力。最大的现场可编程门阵列中有数百万个可配置逻辑单元可用，如何充分利用这些大规模并行资源是一项设计挑战。在要求严苛的网络应用场景中，设计细粒度流水线架构是为了满足流量速率要求，并对整个网络操作中的每个处理模块进行精细调优。这些模块既按照序列方式进行层次化组合——以满足数据依赖要求，也按照并行方式进行组合——以满足吞吐量要求，从而构建出更高层级的流水线，直至图1 所示的应用宏流水线级别。该图还体现了网络流水线的另一个特征：校验和计算等一些操作以比特率执行，而数据包过滤或状态存储等其他操作则以数据包速率执行。这些网络流的并存表示形式导致比特率下的主流水线与数据包速率操作之间存在强烈的同步需求，例如可基于数据包起始或数据包结束控制信号实现。

[1]格勒诺布尔-阿尔卑斯大学工程学院

作者地址：JeanBruant，OVHcloud；格勒诺布尔-阿尔卑斯大学、法国国家科学研究中心、格勒诺布尔国立综合理工学院[1]，TIMA，法国；Pierre-HenriHorrein，pierre-henri.horrein@ovhcloud.com，OVHcloud，法国里昂；Olivier Muller，olivier.muller@univ-grenoble- alpes.fr，格勒诺布尔-阿尔卑斯大学、法国国家科学研究中心、格勒诺布尔国立综合理工学院[1]，TIMA，46 avenueFelix Viallet，38031，格勒诺布尔，法国；FrédéricPétrot，frederic.petrot@univ-grenoble-alpes.fr，格勒诺布尔-阿尔卑斯大学、法国国家科学研究中心、格勒诺布尔国立综合理工学院[1]，TIMA，法国格勒诺布尔。

基于这种高级应用表示，传统硬件开发流程随后会将应用算法描述为功能等效的硬件架构。这些硬件描述不仅负责表达功能意图，还要处理与特定硬件目标相关的许多细粒度实现细节。描述将多个关注点集中在同一位置，这些关注点既受到功能约束的影响，也受到实现约束的影响。具体而言，采用传统硬件描述语言实现的流水线，会将功能意图与本地和全局信号同步紧密耦合。局部修改对整个系统的影响可能变得难以跟踪和验证，从根本上削弱设计跟上所需演进速度的能力。这种复杂的关联还会在开发阶段拖慢设计人员的进度，因为设计需要从早期草稿到可交付状态经历多次迭代。

功能参数化——即根据功能意图对设计进行参数化的能力——有助于简化演进和迭代。然而，其用途严格局限于预期范围，而且需要付出高昂的工程成本。此外，与架构参数化不同，功能参数化的主要目的并非让设计能够适配多种FPGA器件，而是调整整体功能，使其适合给定的目标。

功能参数化与架构参数化（即非功能参数化）之间的区别在很大程度上取决于上下文。定点运算中信号的宽度属于功能参数，因为它们会直接影响结果质量。为实现此类高度规则的计算而设置的寄存器级数量则纯粹属于架构参数。为了满足时序要求，需要根据目标在组合逻辑复杂度与资源使用量之间进行权衡。对于数据包处理而言，计算并非对连续样本流进行规则处理，而是基于事件：每个数据包都需要执行专门的操作，而这些操作是否取决于数据包大小则另当别论。尤其是，大多数处理步骤无法在收集.....之前确定

前述步骤的结果。核心逻辑主要由控制和同步操作组成，组合逻辑复杂度较低。在此背景下，我们旨在引入架构参数，以实现以下目标：

- 抽象掉容易出错的信号同步操作，
- 自动插入可配置的架构元素，
- 允许针对每个现场可编程门阵列目标进行定制化的架构参数化，
- 避免因冗长的参数化描述而使说明杂乱，
- 严格控制架构性能。

成功的架构参数化应在实现质量方面达到零成本抽象，同时不损害功能的可读性。本文介绍了面向流水线的硬件描述如何与可配置的信号同步策略相结合，从而实现这种架构参数化。我们的贡献如下：

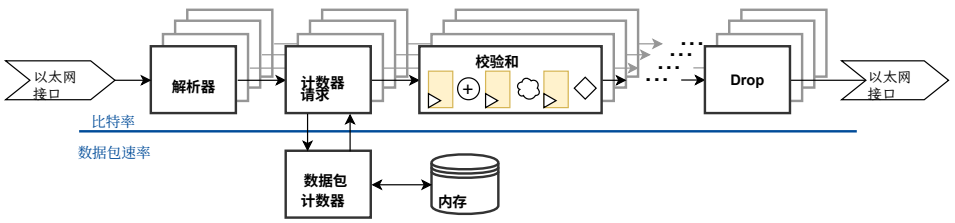


图 1. 网络应用的流水线表示

- 一种基于信号同步和非完整描述的设计方法，
- 一种基于图的硬件信号同步模型，包含多种求解策略，
- 一种实现该设计方法的框架，
- 一个工业应用案例，根据延迟、吞吐量和资源使用量，与先前的实现进行比较，评估所得电路的质量。

本文其余部分安排如下：第2 节将我们的方法与相关文献进行比较。第3 节介绍流水线硬件描述中两个相关的架构参数。第4 节和5 节详细介绍相关算法及其在我们的流水线自动化框架（*PAF*）中的实现。第6 节详细介绍一个工业数据包处理应用，以及在此场景下通过*PAF* 实现的架构参数化。第7 节进一步介绍该框架提供的自动化与分析功能，以帮助设计人员选择架构参数。最后，第8 节总结本文并介绍未来的工作。

2 相关工作

流水线范式是利用数字电路并行性的一种公认方法。早在数字设计发展的初期，面向流水线的描述和模式就已被提出 [9][12]。这类模式有助于最大限度地提高吞吐量并优化硬件资源使用量。许多研究提出了硬件流水线的抽象模型，并开发了自动化工具来生成复杂且可配置的架构。

在实践中，流水线处理是指将一项功能拆分为多个连续阶段，并在阶段之间设置中间寄存器。这一操作通常称为调度，在传统硬件描述语言（HDL）中仍需手动完成，但自动化这一过程是一个活跃的研究领域。高层次综合（HLS）将软件语言综合为硬件架构 [5][8], 通常旨在从高层算法表达式中推断出最高效的调度。研究人员已探索了多种方法，尤其是针对数据流 [21]和信号处理 [16]的方案。近期关于基于延迟不敏感设计理论 [6], 的动态调度数据流电路 [17], 的研究，通过在关键反压路径 [18]上插入缓冲区来提升电路性能。类似方法也已证明有助于提高微架构设计 [10]的灵活性。

这些技术专注于功能和算法表达式，能够为设计人员提供高度自动化，但代价是将对最终架构的大量控制权交给工具。通过使用 编译指示 ，仍可对生成的架构进行一定程度的参数化；通常需要这样做才能获得最佳结果 [7]。这种参数化也提高了设计的可复用性，只需对代码进行有限修改，便可探索各种架构。然而在实践中，编译指示 高度依赖各 FPGA 厂商及其配套工具链，因此，要在多个目标平台上复用同一设计，需要付出相当大的努力。

为了全面掌控架构及其性能，在许多情况下仍需手动进行调度，这也解释了传统硬件描述语言（如 *SystemVerilog* 和 *VHDL*）尽管已经过时，却依然广受欢迎。为克服传统硬件描述语言的局限，尤其是冗长性、表达能力和参数化方面的不足，近年来人们提出了许多替代方案。

其中，硬件构造语言（HCL），例如 Chisel [2] 或 PyMTL [20], 利用现代软件范式引入先进的配置与生成能力。HCL 并不描述电路，而是专注于描述硬件生成器，并能在相对基础的硬件描述能力之上构建自定义复杂抽象。这类生成器可以高度参数化，但 HCL 在设计上不提供架构参数化，且必须开发额外的框架才能提供此类

功能。*DFiant* 硬件构造语言 [23] 融合了源自高层次综合的流水线调度策略，并具有类似的特性与局限。其他一些形式化研究和抽象也可与我们的工作进行比较。一个 SystemC 库 [13] 引入了一种领域特定语言（DSL），基于事件驱动描述，通过事务级建模（TLM）描述流水线。它能够实例化并行和串行流水线级，从而可以快速探索设计中的调度技术。类似地，TL-Verilog [14]提供基于阶段的描述，使识别和迭代流水线级更加容易，并可将阶段实现为寄存器或连线。基于 Rust 的硬件描述语言 *Filament* [22]更进一步，提供静态流水线调度的编译时检查。这些方法虽然在面向流水线的描述方面具有很强的表达能力，却没有解决架构参数化问题，也没有旨在自动完成流水线内信号的同步。

从网络的角度来看，为了提高网络流水线描述的灵活性，已经出现了若干领域特定的研究。FlowBlaze [25] 是一种领域特定语言，提供了一种表示网络流水线的方法，重点支持有状态应用。另一种流行语言是 P4 [3], ，它依赖于清晰的关注点分离。一方面，架构表示定义了处理单元的细粒度能力；另一方面，整个应用则表示为要依次应用于每个网络数据包的操作。基于 P4 工具链的网络应用开发相比传统的硬件描述语言实现带来了实际改进，例如原生支持标准协议，并减少设计和调试所需的时间。尽管 P4 提高了网络环境中的表达能力和可复用性，但它也存在若干局限，例如依赖供应商实现，从而破坏了跨多个目标平台的可复用性 [28]。此外，它既不提供细粒度、面向流水线的硬件抽象，也不支持架构参数化。在实践中，基础硬件函数必须使用上述相同的硬件描述方法实现，因此仍然面临完全相同的局限。

与现有文献中的工作相比，我们的方法旨在探索一条介于高层次综合（HLS）的自动调度与传统硬件描述语言（HDL）和硬件构造语言（HCL）冗长表达之间的中间路径。我们希望利用 HCL 带来的高级编程范式，以及对展开过程中电路的自反性，从而显露部分架构参数。显露这些参数，首先需要构建恰当的流水线调度模型。不过，我们并不打算基于数据依赖分析实现流水线调度自动化，而是让设计人员能够清晰地描述自己对流水线调度的选择。随后，这些架构参数便可成为 HLS 工具及其设计空间探索（DSE）算法的新探索领域，以期在扩展后的设计空间中找到更多优化方案。据我们所知，这是首个提供显式用户定义流水线调度模型的方法，可借此实现架构参数化，提升设计的可复用性。此外，我们的方法并不打算创建一个接管整个设计的独立领域特定语言（DSL），而是提供一种自动化机制，仅在现有 Chisel 硬件设计中适用时集成其中。最后，流水线模型和同步解析工作都完全不依赖源语言，因此，等效的流水线自动化框架几乎可以适配任何硬件描述语言或硬件构造语言。

接下来的章节将详细说明，可复用性要求如何自然地引出架构参数化，以及为何需要将这些参数与流水线调度和功能描述解耦。

3 流水线的架构参数化

3.1 动机

为了说明架构参数化的必要性，我们考虑一个用原生 *Chisel* [2]，（一种嵌入于 *Scala* 中的硬件构造语言）编写的加法器模块。在接下来的示例中，它的用法与传统硬件描述语言非常接近，使用了相似的关键字：**模块**、**输入**、**输出**、**寄存器**、**连线**。清单 1 描述了一个简单的组合加法器。第 2–10 行将其写成标准模块。第 12 行给出了等效的内联函数，为简洁起见，后续示例都将采用这种形式。

```
1  // 传统模块描述
2  class加法 extends 模块 {
3    值 a,    b = 输入输出接口(输入(无符号整数(64.W)))
4    值 结果    = 输入输出接口(输出(无符号整数
                 (64.W)))
5
6    val sum =连线(无符号整数(64.W))
7    总和: = a  +b
8
9    结果: = 求和
10 }
11 // 等价函数式实现
12 def 加法(a:无符号整数, b:无符号整数) = a +b
```

代码清单 1. 原生 Chisel 中的组合加法

```
类 加法寄存器 继承 模块{
  变量 a,    b = 输入输出接口(输入(无符号整数(64.W)))
  变量 res     = 输入输出接口(输出(无符号整数
                 (64.W)))

  变量sum = 寄存器(无符号整数(64.W))
  总和: =a  +b

  结果 : = 和
}

定义 加法寄存器(a:无符号整数, b: 无符号整数) = RegNext(a +b)
```

代码清单 2. 原生 Chisel 中的寄存器加法

假设这个加法器的第一个版本满足当前现场可编程门阵列目标的时序要求，但在另一个目标上表现不佳。为了适应第二个目标，我们插入一个寄存器，以打断 a 和 b 输入到 res 输出之间的组合路径。代码清单 2 展示了这个极简流水线级的创建过程。与传统硬件描述语言相比，最显著的区别在于从事件驱动赋值范式转向了对导线和寄存器的面向对象声明。代码清单 1 和 2 的第 6行和第 7 行展示了这种行为。在这两种描述中，赋值保持不变（:=），位于第 7行；但第 6 行对信号 sum 的声明定义了实际的硬件原语：它可以是寄存器（**Reg**）或连线（**Wire**）。

第 12 行展示了*Chisel* 函数生成实际硬件（例如连线和寄存器）的能力。这里使用标准的**RegNext** 对象，在功能上插入一个寄存器，以存储组合加法的结果。这与传统硬件描述语言有显著区别；在传统硬件描述语言中，函数只能用于描述参数计算或纯组合逻辑。

得益于*Chisel* 的表达能力，在模块主体中只需更新一个关键字即可插入这样的寄存器。然而，最重要的副作用要到该加法器集成到模块层次结构中时才会显现：计算延迟会在毫无提示的情况下从0 个1 个时钟周期。图2 展示了一个集成示例，其中预期结果为$z = (2x + y) \oplus xy$。使用加法器（ $2x$ ）而非移位操作（ $x + x$ ）来实现 $x \ll 1$，是为了保持示例简洁，同时说明延迟补偿这一关键问题而做出的刻意选择。图2 上半部分展示了设计的第一个版本，其中两个纯组合**加法**模块与一个*1* 周期乘法器并行放置。清单3介绍了该设计的扁平化版本，该版本基于函数而非模块实例化。

图2 下半部分展示了使用addReg 替代add时，为确保整体设计正确性所需进行的更新。类似地，清单4给出了相应的扁平化版本，其中在每个求和操作后都插入了一个寄存器（延迟 1 个周期）。随后通过插入两个额外的寄存器来补偿延迟的增加：regX 用于延迟x，regMulXY 用于延迟mulXY。

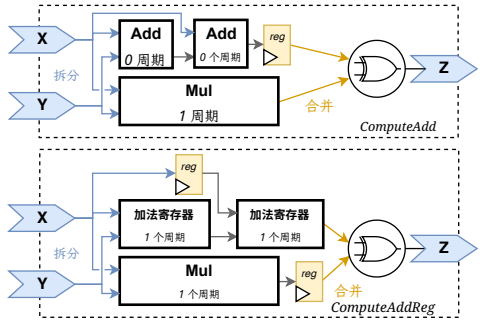


图 2。与子模块 latencyevolution 集成时可能出现的问题

```
类 FlatComputeAdd 继承 模块{
  值x, y = 输入输出接口(输入(无符号整数(5.W)))
  值 z      = 输入输出接口(输出(无符号整数
              (5.W)))

  值XY求和, XY求和 = 连线(无符号整数(5.W))



  XY求和 := 加法(x, y)

  XY求和 := 加法(XY求和, x)

  值XY乘法 = 寄存器(无符号整数(5.W))
  XY乘法: = x*y


  z := XY求和 ^XY乘法
}
```

列表 3。ComputeAdd 的扁平化版本设计，集成非寄存加法器

```
类 FlatComputeAddReg 继承 模块{
  值x,y = 输入输出接口(输入(无符号整数(5.W)))
  值 z      = 输入输出接口(输出(无符号整数
              (5.W)))

  值 XY求和, XY求和   = 连线(无符号整数(5.W))
  值 regX, regMulXY = 寄存器(无符号整数(5.W))

  XY求和 := 加法寄存器(x, y)
  寄存器X := x
  XY求和:=加法寄存器(XY求和 +regX)

  值mulXY = 寄存器(无符号整数(5.W))
  XY乘法:= x*y
  regMulXY := XY乘法

  z := XY求和 ^ regMulXY
}
```

清单 4. *ComputeAddReg*的扁平化版本设计，集成了寄存加法器

彩色线条标示出与先前 **FlatComputeAdd** 版本的差异：加法以绿色背景标示，修改内容以黄色背景标示。在原模块主体的 7 行代码中（含第 5 至 16 行），只有 3 行保持不变；更新后的版本新增了 3 行，并修改了 3 行，相当于重写了原模块主体的 66%。

实际上，设计人员只能独自负责在脑中推断延迟变化，因为工具无法自动计算这类信息。这些变化可能需要局部更新（例如这些简短的代码片段中所示），也可能需要全局更新（贯穿整个层次结构，从最内层模块一直到顶层）。为了保证设计正确性，必须手动补偿并传播延迟变化，而 EDA 工具仍无法提供任何额外帮助。

插入寄存器以打断组合路径，可能是出于可复用性的需求，例如让设计适配其他 FPGA 目标平台。虽然*Chisel* 环境允许设计人员对**加法** 模块进行参数化，以决定是否插入寄存器，但这一更改仍需手动反映到周围的组件中。导致架构参数化的不仅是可复用性需求；若要让这种参数化在大规模应用中发挥作用，还需要一定程度的自动化。此外，在实际用例中，延迟并非总是只有几个周期，有时会多达几十甚至几百个周期，因此选择等效的延迟实现方式便成为一项典型的架构参数。尤其是采用移位寄存器或存储器的实现

可以受益于显式的目标依赖原语，从而实现最高效的资源使用量和最佳的可布线性。

流水线的另一个潜在架构参数是定义阶段之间的握手，也称为协议信令。某项计算可能需要在各种场景中重复使用，而每种场景都需要自己的握手系统，例如广为人知的协议*Ready-Valid*、*Credit-Based*或*Carloni* [1]。由于使用协议信令会大幅增加示例的冗长性和复杂度，为求简洁，这里不再赘述。与插入寄存器类似，在现有硬件设计中引入或替换协议是一项繁琐且容易出错的工作，需要借助一定的自动化和流水线建模，才能高效地实现参数化。

这个简单的示例展现了传统描述中计算与实现之间错综复杂的联系，以及即使是最小的局部修改也会造成的整体影响。我们确定了两个潜在的架构参数：用于实现信号延迟的硬件原语，以及用于实现协议信令的硬件原语。然而，要开放这些参数，首先必须对流水线进行恰当建模，以便提供必要的自动化，使局部架构选择能够反映到周围环境中。为了实现这种参数化并获得相应的自动化，我们旨在围绕以下两个概念对流水线描述进行建模：

延迟感知建模信号同步要求，以自动插入 f
可配置的架构元素**协议多态**与协议无关的描述及展开时配置

3.2 关系作为一等对象

延迟和协议都通过同步来表达设计中不同部分之间的关系。为了界定这些部分的边界，我们考虑由相干信号组成的组，即对其执行直接组合运算能够得到有意义结果的信号。这类信号称为已同步信号，而设计中的每个此类部分称为时区。这些时区及其各自的关系，是我们自定义流水线同步模型的基本元素。

文献中此前已详细介绍了若干流水线同步模型 [19, 24]，并将其与我们的模型进行了比较。特别是，我们的模型与 Leiserson 提出的用于重定时的图方法 [19] 有关。我们还引入了流水线的图表示，以帮助说明数据信号之间的关系。原始研究利用这种图表示来建模寄存器之间的组合延迟，并通过移动一些寄存器来截断最长的组合路径，以期缩短时钟周期。相较之下，我们对相干信号组之间的关系（延迟和协议）进行建模，并旨在使这些所谓时区之间的同步路径可配置。我们的方法不仅节点的含义不同（组合运算符与信号组），对图的用法也有所不同。在原始方法中，图完全由描述约束，相关操作旨在更新边的值，同时保持图的整体一致性。而在我们的方法中，图的初始版本由设计人员描述，其中包含一些部分定义的边，我们旨在自动实现这些边。

由于我们无意提供基于信号依赖关系的自动调度，因此必须适当定义连续时区之间的主要关系。不过，我们认为，时区之间的关系并不总是需要详尽描述。特别是，沿流水线操作进行的简单但通常冗长的信号传播，其实现可以自动化并设为可配置。同样，协议信令的实现也可以

根据同步模型实现自动化。为了建立一个能够提供这些特性的信号同步模型，我们保留以下定义：

信号 （电气）承载电气变化并对信息片段进行编码的载体 .
在电路中，信号以其唯一名称标识，其唯一属性是类型，范围从简单的布尔型到复杂的结构化类型。一个信号只有一个驱动器（源），并且可以驱动电路中的一个或多个组件（接收端）。

时区一组完全同步的信号。
从功能角度来看，对这些信号直接进行组合运算是合理的，无需引入额外的延迟或握手。特别是，所有这些信号共享完全相同的协议信号，例如有效性和反压信号。

关系 给定的两个时区之间的同步要求。
关系并不对实际连接信号的运行逻辑进行建模，而是包含两个属性：延迟和协议信令。协议未定义，意味着不存在底层要求，因此实际实现时可以配置任意协议。延迟未定义，意味着存在一个缺失关系，需要根据用户配置的实现策略，由同步解析算法加以解决。在当前模型中，此类缺失关系 仅用于表示信号在其原始驱动器与延迟的下游引用之间的传播（参见信号（流水线））。关系并不总是对信号之间实际的硬件连接进行建模。具体而言，抽象的等价关系 可以由一系列连续的关系组成。

PipeStep 表示源时区与接收端时区之间有方向的功能转换。它定义了两个时区之间的关系，以及连接各时区信号的实际硬件逻辑。PipeStep 可以使用其源时区中可用的任意信号，也可以在其接收端时区中声明新信号。PipeStep 将流水线级这一明确定义的概念加以泛化；流水线级由组合计算及其对应的流水线寄存器构成。PipeStep 可以描述常见的流水线级，也可以描述纯粹的组合赋值或多周期计算（通常用于对内存访问进行建模）。

流水线以一系列 PipeSteps 的形式，对硬件操作进行显式调度描述。流水线由一组 PipeSteps 及其关联的时区组成，其中 PipeSteps 的接收端时区是下游 PipeSteps 的源时区。流水线可以包含多个分支，如下所示。多个共享同一源时区的 PipeSteps，表示流水线在不同分支之间拆分。反之，多个共享同一接收端时区的 PipeSteps，表示流水线分支的合并。

信号（流水线）流水线的命名元素。
既指与源时区关联的原始（电）信号，也指该信号的所有延迟版本，这些版本用于下游 PipeSteps 和时区。在流水线中，一个信号描述多个电信号，每个电信号都与一个本地时区相关联。各电信号的值序列之间的延迟，与各自时区之间的关系相匹配。在当前模型中，缺失关系只能表示对先前定义的源信号存在歧义的下游引用。具体而言，所有新信号都必须在一个时区中创建，并与周围时区建立显式关系。

同步模型 由时区作为节点、关系作为边组成的有向无环图（DAG）。从流水线描述中可得到一个初始部分图，即包含缺失关系的图。它作为硬件的中间表示，用于获得

完全同步且已实现的电路，这得益于下文介绍的同步解析算法。

3.3 电路解析流程

基于前文的定义，电路同步通过以下三个阶段实现：

1. 模型构建 根据电路的 临时编写的 面向流水线的描述，这第一步会生成前述同步模型。模型构建的详情见第4.2节。**2. 同步解析** 以同步模型作为中间表示进行迭代，此第二步会计算实现电路同步所需的所有额外硬件和连接。各种求解策略及其实现详见第5节。**3. 电路更新** 基于同步解析的结果，最后一步将生成实际且完整的电路描述。

能够清晰界定这些步骤，凸显了同步模型及其相关解析算法与电路具体描述之间的完全独立性。特别是，尽管本文将在下一节介绍基于*Chisel* 实现这一流程，但所有形式化方法和求解策略均独立于实现语言。

4 框架实现

作为同步模型的输入，初始流水线描述旨在仅保留确保同步能够确定性地完成所需的最少实现细节。具体而言，为了提供所需的架构参数化，前文介绍了两类可以省略的实现细节：

- 从信号定义的时区到其使用位置的显式传播，
- 握手机制和协议信号。

在流水线描述中硬编码的实现细节越少，采用恰当的设计方法后，它就越灵活、越易于配置。

4.1 流水线设计方法

关键的实现选择通常由应用场景和可用的硬件目标决定。本节介绍一种设计方法，将流水线描述与其架构配置解耦。图3 展示了我们的方法。该方法围绕上一节定义的同步模型构建，并采用三部分应用程序接口（API）：

用户 API 流水线结构和操作由用户通过一组专用的类和对象进行描述。此 API 定义了语法和用于创建同步模型的原语。

后端API 同步解析算法和协议实现作用于同步模型，以获得一致的电路。此 API 包含第 5 节所述的算法，用户可以对其进行扩展，或将自定义算法开发为额外的库。

配置 API 展开时的配置 *API* 在面向用户的 *API* 与后端 *API*之间的映射中提供精细的配置能力。它驱动根据用户 API 和后端 API 中可用的类和对象生成实际代码。

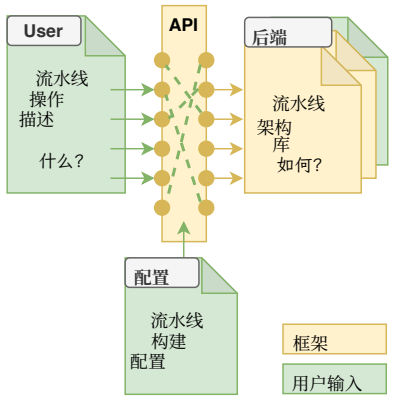


图 3. 解耦流水线描述方法

信号同步模型是用户 API（用于描述信号交互）与后端 API（负责保证信号同步）之间的通用接口。第 5节详细介绍了用于在后端实现信号同步的算法。本节其余部分首先聚焦于用户 *API* 的具体实现，然后给出一些 配置 *API*的实际示例。

4.2 用户 API：模型构建

用户 API 的实际实现是一个 Scala 库，用于定义面向流水线的对象和属性，旨在 Chisel 硬件生成环境中使用。

Pipe 作为显式对象创建 `Pipe(input, name)`后，流水线便是一个容纳流水线分支的容器。它定义了一个默认主分支，其初始时区包含来自`input` 关系的可用信号。流水线的行为类似于分支，但还定义了一个构建 方法，用于根据描述触发同步模型的实际展开。

分支 PipeStep 的连续序列；分支最显著的特点是提供拆分 和合并 方法。**步骤** 定义为匿名函数`(p,n) =>`，PipeSteps 定义了前一个时区`p`与新时区`n`之间的关系和硬件连接。

信号普遍性属性 在同步模型中，信号是一个名称，同时指代源信号及该信号的所有下游使用，并带有适当的延迟补偿。在流水线描述中，任何通过名称对先前定义的信号进行的下游引用，都会以经过适当延迟的原始源信号版本提供给设计人员，无需显式描述延迟。为确保这一属性，流水线采用静态单赋值（SSA）形式。在给定时区内，每个信号只定义和连接一次，之后即可在流水线下游多次使用。由于纯粹的 SSA 形式会与高级电路设计所需的生成结构（循环、函数）相冲突，因此 SSA 形式仅在封闭区域内强制执行。随后，这些区域可以通过按名称显式映射信号彼此连接，方式类似于传统硬件描述中的模块实例化模式。

将此设计模式应用于第3 节的动机示例，详见清单 5。为便于理解，代码背景色对应如下：(1) 橙色表示对流水线对象及其分支执行的显式操作（拆分、合并、构建）；(2) 绿色表示对流水线主（默认）分支执行的操作（PipeSteps）；(3) 黄色表示对显式 乘法分支执行的 PipeSteps。

```
类FlatComputeAddReg(
    输入： FromTZ[XYZone]，
    输出： ToTZ[ZZone]
)继承模块 {
  值管道 = Pipe(输入， “管道”)
  // 显式定义分支 "mul"
  值乘法 = pipe.split("mul")
  乘法.乘法 =(p, n) => {
    n.mulXY = 阶段.寄存器(p.x)
    n.XY乘法： = p.x * p.y
  }
  // 主分支上的 2 个加法器级
  pipe.加法 =(p, n) => {
    n.sumXY = 步.寄存器(p.x)
    n.XY求和： = p.x + p.y
  }
  pipe.addB =(p, n) => {
    n.sum2XY = 步.寄存器(p.sumXY)
    n.XY求和： =p.x +p.XY求和
  }
  // 显式分支合并和最终异或
  pipe.merge(乘法)
  pipe.异或 =(p, n) => {
    n.z =步骤.继续(p.XY求和)
    n.z： = p.XY求和 ^p.XY乘法
  }
  pipe.feed(out)
  pipe.构建
}
```

清单 5。使用基于 Chisel HCL、面向流水线的框架实现的运行示例

```
类 XYZone 继承 Bundle {
  值 x = 无符号整数(5.W)
  值 y =无符号整数(5.W)
}

类ZZone继承Bundle {
  值 z = 无符号整数(5.W)
}
```

清单 6。I/O 时间区

这个面向流水线的版本等价于清单 FlatComputeAddReg中所示的原生 Chisel 版本 4，后者在每个加法器之后插入一个寄存器。在这样一个小型示例中，代码行数（LoCs）显著增加，因为流水线、其分支以及各步骤的具体定义都分别引入了几行额外代码。由于该流水线的实际操作内容相当少（仅包含 4 个组合运算符），所有这些额外代码对于这一任务而言似乎过于冗余。不过，在更复杂的示例中，例如第 6节所介绍的示例，可以观察到代码行数大幅减少。这个初始示例主要旨在说明我们方法的核心概念：

- 流水线和分支操作均被明确表示，步骤也能快速识别，
- 每个新时区的相对调度都由以下内容明确描述：步骤对象，可以是连线或寄存器，

- 信号传播和握手过程被省略，
- 对于用户描述的时区间关系所做的任何本地更改，都应确保所有传播信号保持正确同步。

同步算法将在下一节详细介绍，这些算法可确保满足这一最后的属性。本小节将进一步详述用户 *API* 的语法，随后将基于这一描述展示若干生成示例。

代码清单 5 介绍了用户 API 提供的主函数对象和类的实际用法。具体类 Pipe 是此 API 的核心。第 5 行对其进行实例化时，指定了流水线的源关系（in）

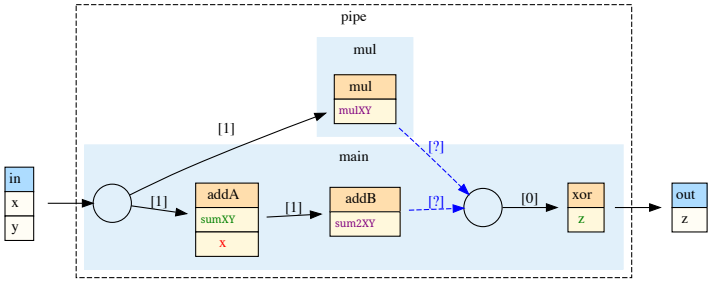
其类型为 `XYZone`，其声明见代码清单 6，其中提供了信号x 和 y。这两个输入信号可供所有下游操作直接使用，如第 10、15 和 19 行所示。

该`Pipe` 对象提供一个默认的（主）分支，操作会依次以显式函数（PipeSteps）的形式记录在该分支上。为实现此示例，主分支上声明了三个这样的 PipeSteps，分别位于第 13–16 行（第一个加法器）、第 17–20 行（第二个加法器）和第 23–26 行（最终异或）。每个 PipeStep 都由一个匿名函数构成，该函数接收两个时区作为参数，此处为p （previous，源端）和n （next，接收端）。在第 14 行的函数体中，`Step.Reg` 函数描述了在时区 n 中创建新信号 `sumXY`，并建立与时区 p 之间具有 1 周期延迟的关系。虽然这种语法与常见的寄存器描述非常相似，但它也定义了同步模型中时区与其信号之间的关系。该信号实际如何连接，即如何将x 和y 信号相加，描述于第 15 行。类似地，在第 24 行最终异或操作的函数体中，`Step.Wire` 函数描述了在时区n中创建新信号z，并建立与时区p 之间具有 0 周期延迟的关系，这相当于常见的连线，也就是组合逻辑阶段。在流水线展开期间，通过依次执行各个 PipeSteps，按顺序创建时区。最新创建的时区将作为下一个 PipeStep 的前一个时区，而该 PipeStep 本身又会创建另一个时区。

`Pipe` 对象还提供面向流水线的显式操作，例如创建新分支、从主分支的当前状态（即最后一个 PipeStep 之后）进行拆分。创建`mul` 分支的拆分操作发生在第 7 行，即流水线的最开始处，并创建了一个完全并行的分支。第 8–11 行在这个新分支上定义了一个 PipeSteps。随后，它在第 22 行与主分支合并。最后一个操作发生在第 23–26 行，此时两个分支的信号均已可用。

为了转发计算结果，在第 27 行，`Pipe` 实例在其当前状态下声明与外部 TimeZone 的连接：此处，在最后一个 PipeStep 中声明的信号 `n.z` `pipe.xor` 连接到接收端关系 `out`。该关系的底层类型 `ZZone` 仅包含信号 `z`，如清单 6所述。

最后，第 28 行触发流水线展开，并实际构建同步模型，
基于此前记录的所有操作，涵盖从初始化、各分支上的操作到最终馈送。实际的时区对象会被创建，并作为参数传递给 PipeStep 函数，以记录同步模型中的信号创建和使用情况。



(a) 与代码清单 5 对应的图表示

信号状态颜色方案	
1	待处理：本地
2	
3	
4	可用：未使用
5	可用：本地使用
6	可用：下游使用（转发）
7	可用：本地和下游使用
8	已声明：未使用
9	已声明：本地使用
10	
11	

(b) 信号配色方案

图 4. 设计意图

在此阶段，流水线描述仅聚焦于设计意图：信号声明、连接以及这些信号之间的操作。根据对目标性能和应用需求的先验了解，已在每个加法器和乘法器之后明确设置了一些寄存器级。然而，文中没有提及与任何操作相关的协议信令，也没有

寄存器补偿，尤其是为了确保 主函数 与 乘法分支之间的适当同步。图 4a 展示了相应的设计意图，其中对流水线的两个分支显式执行了拆分/合并 操作，同时有些关系有意未作指定。边标签展示了相应的关系延迟，可能是未知的 *[?]*，也可能像这里一样明确指定为寄存器 *[1]* 或简单连线 *[0]*。每个 PipeStep 都会产生一条指向某个节点的关系——该节点是其输出n时区——并以其名称作为表头。随后，该节点会列出所有由上游 PipeStep 定义或下游 PipeStep 所需的信号。图 4b中的颜色编码进一步展示了每个时区中各信号的当前状态。这里，信号XY求和由addA PipeStep 定义，并立即被addB PipeStep 使用，因此标为绿色，对应状态 9：已声明：本地使用。信号x也被addB PipeStep 使用，但并非由addA PipeStep 提供，因此标为红色，对应状态 1：待处理：本地。

因此，图 4a 展示了将这一简单设计意图转换为可工作电路所需的基本同步机制：延迟连接信号 x，并平衡分支 主函数 和乘法。

4.3 配置 API：模型参数化

清单 5 中所示的流水线描述是一种硬件生成器，需要进行参数化才能生成电路；此处使用 in 和 out 时区。展开时参数化使设计人员能够基于相同的设计意图生成各种电路，并在此处用于提供两个自由度：（1）协议多态；（2）额外信号传播。协议多态可通过关系定义实现，而额外信号传播则基于扩展周围的 in 和 out 时区以实现额外信号同步的能力。清单 7 展示了以最精简形式生成电路的方式，不包含协议信令（ **RawIO** ）。

```
// 最基本的流水线实例化


new FlatComputeAddReg(
  RawIO。来自(新建 XYZone),
  RawIO。至(新的 ZZone)
)
```

清单7. 不含协议的框架生成

```
特征Extra {
  vale = 无符号整数(64.W)
}
新建 FlatComputeAddReg(
  ReadyValidIO.From(新建 XYZone 带有 Extra),
  ReadyValidIO.To(新建ZZone 带有Extra)
)
```

清单8. 含*ready-valid*协议和额外信号的框架生成

若要生成另一种电路，使各时区之间通过就绪/有效握手进行通信，只需将**RawIO** 替换为**ReadyValidIO**。清单8 展示了这一协议替换，同时还通过扩展 **XYZone** 和**ZZone** 时区并添加信号e，引入了额外信号传播。

图 5 展示了相应的生成意图，其中引入了信号e。输入和输出端都会自动插入协议信号，即*ready* 和*valid*。图表示中有意省略了这些信号，因为它们已隐含在关系和时区中。

这两项更改都非常简洁，却能生成截然不同的电路，而且无需修改原始流水线描述。

同样，流水线描述本身能够适应局部更改，无需像第 3.1 节中寄存器级插入示例所示的那样，在远处手动进行补偿。若要关闭寄存器级插入，可将用于定义值 **Step.Reg** 函数sumXY 和sum2XY，

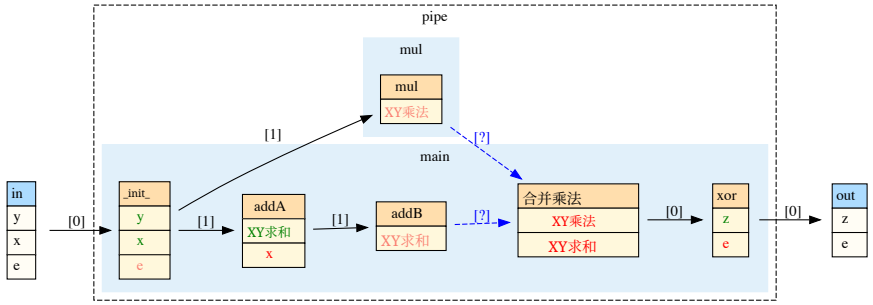


图 5. 带附加信号的生成意图e

分别位于第 14 行和第 18 行，替换为 `Step.Wire` 函数。采用`Step.Reg` 和 `RawIO` 的版本等效于 Chisel 模块`FlatComputeAddReg`，而采用`Step.Wire`的版本等效于 Chisel 模块`FlatComputeAdd`。这些函数为信号定义了恰当的关系，从而支持周边描述的自动时延传播、路径平衡和时延感知。无需针对每种用例进行深入重构，只需少量局部修改即可按协议配置模块，并将流水线级从寄存器转换为连线。

基于对这一细化同步模型的解析，用于实现这些高级生成功能的算法将在下一节详细介绍。

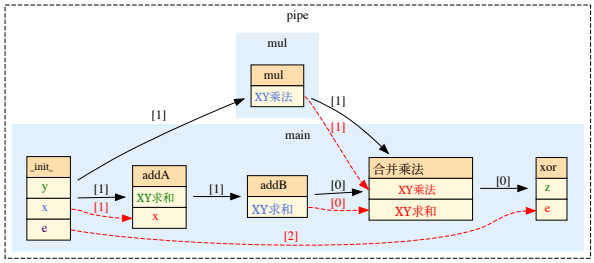
5 模型解析

5.1 基本原则

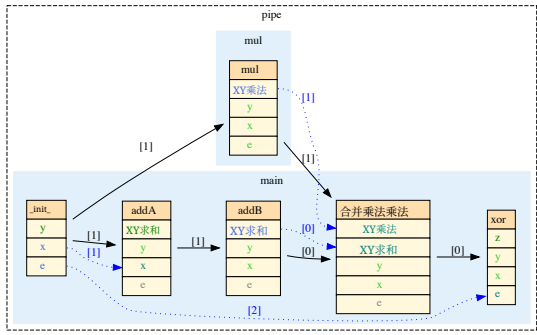
成功的模型解析应生成一个平衡图，且其中不再存在任何未指定的关系。流水线展开后获得的同步模型如图 5所示，可作为解析算法的输入。该模型有意省略了部分内容，以便实现参数化；这意味着解析算法在解析过程中需要考虑某些参数。第一个参数就是算法本身，因为信号传播可以通过多种方式实现。以下各小节将分别详细介绍这些解决方案。不过，我们仍采用如下的通用总体架构：

(1) 首先进行一组分析，计算每条路径的等效时延，并在其合并点处平衡各分支，从而完整确定所有缺失关系（图 6a）；(2) 解析算法在等效关系所在位置生成时延等效硬件，如图 6b、6c和6d所示；(3) 随后，由一个转换过程负责在整个平衡图中统一传播并连接协议信号；(4) 最后进行一次检查，以验证整体同步一致性，并及时向设计人员提供反馈。

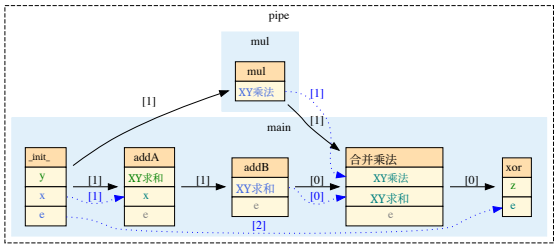
前提条件。 下面介绍的算法假设同步模型图上已经具备一些基本操作。该图应为有向无环图（DAG），即边（关系）具有方向，且图中不存在周期。在我们的应用场景中，这类周期没有意义，因为它们会破坏提供吞吐量保证的能力。因此，节点可以进行排序，同步模型可以提供一条从唯一输入节点到一个或多个输出节点的前向路径（时区）。同样，也可以将该顺序反转，从而提供一条从输出节点到输入节点的有意义的后向路径。实际上，前向



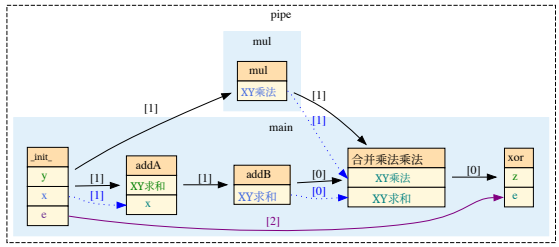
（a）不进行信号传播的合并解析
红色虚线：缺失关系



(b) 穷举式前向传播策略



(c) 点对点传播策略
蓝色圆点：传递实现



(d) 直接传播策略
纯紫色：直接实现

图6. 运行示例的解析步骤和选项

路径通过以下遍历方式建立：从主分支上的输入时区开始，沿着下一个时区继续前进，直到遇到拆分；此时先处理外部分支，然后继续处理最初的主分支。对于我们的运行示例，前向顺序如下：（1）_初始化_，（2）乘法（在分支 乘法上），（3）*addA*，（4）*addB*，（5）合并_乘法_乘法，（6）异或。后向路径顺序就是该列表的反转版本。这类遍历策略确保所有时区在信号声明和使用方面都按照一致的顺序进行处理：前向顺序确保先处理信号声明，再处理任何使用；而后向顺序确保先检查所有使用，再处理声明，这对于单遍信号传播算法十分方便。

图边（关系）可以组合起来形成等价关系。每个时区中的每个信号都保留一个可用性属性，用于表示其用途和连接方式，如图4中相应颜色所示。借助这两个属性，可以定义等价缺失关系：从信号可用的时区指向信号被使用的时区。例如，图 *e* 所示信号的缺失关系 6a 基于从*_init_* 到异或的连续边的组合。更确切地说，图6a 展示了图表示，该图尚未解析，可供后续信号传播算法使用。

5.2 解析算法

本节不仅旨在展示解决同步问题的能力，还介绍了创建任意自定义解析算法的能力，使其能够针对应用需求进行精细定制。

为此，以下段落通过连续三次迭代，介绍一种简单而高效的单遍解析算法的定义。由于具备额外的参数化能力，最终选定该算法用于第 6 节和第 7 节详述的工业实验。为简洁起见，展示解析策略的伪代码以分支而非完整的图作为输入，因此未详细说明拆分与合并管理。

穷举式前向信号传播。第一种朴素方法是将所有已声明和已使用的信号从一个时区转发到另一个时区，不考虑下游的实际使用情况。算法1 详细说明了其逻辑。沿流水线转发的信号数量会随着阶段推进而线性增长，并产生大量未使用的硬件。图6b 展示了这一现象：各个时区中有许多可用但未使用的信号。综合工具或许能在简单情况下部分移除这些死代码，例如未连接的导线和寄存器；但如果无用信号存储在内存中，就不一定能将其剔除，例如使用厂商原语显式实现的先进先出队列。尽管这种朴素方法提供了一种非常简单的方式来确定性地解决同步模型，但其效果不佳，必须加以改进，以消除不必要的硬件开销。

算法 1： 朴素方法：穷举转发

```
1  DeclaredSignals : = []
2  对于 TimeZone，在 Branch 中执行
3      对于 Signal，在 TimeZone 中执行
4          如果 Signal 不在 DeclaredSignals 中，则
5              DeclaredSignals += Signal
6      对 DeclaredSignals 中的每个 Signal 执行
7          如果 Signal 不在 TimeZone 中，则
8              TimeZone.addAndConnect(Signal)
```

算法 2： 反向点对点方法：
向前一个时区请求不可用的信号

```
1  MissingSignals := []
2  对于 TimeZone，按反向顺序遍历 Branch，执行
3      对于 Signal，遍历 MissingSignals，执行
4          如果 Signal 不在 TimeZone 中，则
5              TimeZone.createPlaceHolder(Signal)
6          TimeZone.createConnectionTo(Signal)
7          如果 Signal.isAvailable(TimeZone)则
8              MissingSignals -= Signal
9          else
10             MissingSignals.updateTarget(信号)
11     foreach Signal in TimeZone do
12         如果不是 Signal。isAvailable 则
13             MissingSignals += Signal
```

点对点反向传播。为避免传播不需要的信号，算法必须知道每个信号预期在哪个阶段使用。反向图遍历能够在信号声明之前列出其使用情况。利用这一原理，算法2 详细说明了这种反向解析，其点对点方法与第一个算法有些相似。然而，在这种反向点对点方法中，如果需要不可用的信号，下游时区会向上游时区请求，而不是由上游时区自动推送这些信号。为简化算法说明，*Signal* 对象应理解为按名称引用，即关联到相应时区的全限定硬件信号。若当前时区无法提供某些待处理请求的信号，请求就会转发到最近的上游时区，也就是图的反向遍历中下一个待处理的时区。实际操作中，会在当前时区内创建一个占位信号，将其连接到下游请求时区的信号占位符，最后将请求更新为对新建占位符的实际引用。这正是第*updateTarget* 行函数10的作用。图6c展示了使用此策略解析运行示例的过程。此前缺失的关系（图6a中的红色虚线箭头）通过时区之间已有的关系得以实现，现在显示为蓝色点线箭头。尽管该方法有效，但缺乏自反性，因为缺失关系从未作为对象被直接操作，而只是作为反向传播的副作用得到解决。尤其是，当信号声明缺失时，算法只能报告信号最后一次缺失的时区，而这始终是输入时区。此外，在合并节点中，算法必须向所有上游时区请求该信号；如果多个信号同时定义，可能会导致错误报告含糊不清。要消除这些限制，就必须真正理解预计算的缺失关系。

算法 3： 反向直接传播：
向上游请求不可用的信号

```
1  MissingSignals := []
2  对逆序的 Branch 中的 TimeZone 执行循环
3      对 TimeZone 中的 Signal 执行循环
4          如果 Signal 位于 MissingSignals 中，则
5              TimeZone.createConnectionTo(Signal)
6              如果 Signal.isAvailable则
7                  MissingSignals −= Signal
8              else
9                  MissingSignals.updateTarget(信号)
10         否则，如果 not Signal.isAvailable 则
11             MissingSignals += Signal
```

直接反向传播。算法3详细介绍了一种改进的反向策略：该策略基于缺失关系，直接向缺失信号可用的最近时区请求连接。上游可用性有两种情况：信号声明或信号使用。与之前策略的一个关键区别是，在信号传播中取消连续局部占位机制，改为直接连接。图6d突显了这一差异，因为信号e 不再出现在*addA*、*addB* 和*merged_乘法_乘法* 时区中。此外，*_init_* 与异或 时区之间的缺失关系被一条延迟为 2 的具体关系（紫色实线箭头）所取代。

这一解析算法的第三次也是最后一次迭代不仅解决了上述局限，还开创了更广阔的参数化范式。尽管这三种算法都能实现完全有效的同步，但最后一种算法最能体现该框架的参数化能力。为此，也为简洁起见，第6节工业用例仅采用这一算法。

5.3 解析策略的参数化

硬件原语选择。 如同最新算法中所述，实现两个时区之间的直接关系，会引发一个问题：对于给定的延迟，应选择哪种硬件原语。虽然有些对应关系显而易见，但另一些则具有值得探索的参数化潜力，可根据资源可用性对架构进行精细调整。例如，我们提出一种基于两个整数参数的参数化方案：深度阈值 和宽度阈值，分别指待实现路径的延迟和待传播信号的宽度。随后使用以下决策表来选择要实现的原语：

延迟 l	信号宽度 w	
	w <宽度阈值	w ≥宽度阈值
$l = 0$	Wire	
$l = 1$	寄存器	
l <深度阈值	连续寄存器	
l ≥深度阈值	连续寄存器	FIFO

基于先进先出队列的原语的实际实现将在第 7 节中进一步详述。基于此参数化，我们将之前的 *Direct Backward* 策略拓展如下：

DirectAuto 阈值设为无穷大，并让综合器根据寄存器推断移位寄存器。

DirectForceReg 阈值设为无穷大，并阻止综合器推断移位寄存器 .

DirectForceSRL 阈值设为无穷大，并强制综合器推断移位寄存器。

DirectFIFO:D:W 阈值为 D 和 W，并让综合器推断移位寄存器。

以下结果为 *DirectFIFO* 策略选取了一组阈值（3:3），以展示其如何影响综合器对运行中的流水线示例的处理行为。这种基础方法还可以进一步调整，以更好地匹配给定设计中缺失的关系。在检查复杂设计的关系后，第 7 节将展示该框架在探索阈值方面的灵活性，并提供更广泛的架构参数化范围。

尚未探索的架构参数化机会。 截至目前所述的三种信号传播算法都只关注传播单个信号，但还可以考虑许多其他参数。例如，一种潜在的参数化方式是管理遵循相同关系的多个信号。它们的传播既可以通过拼接成一个位组来实现（例如，一个大型先进先出队列），也可以将信号分别处理（例如，每个信号对应一个先进先出队列）。根据信号深度和/或宽度配置所选方法，将产生不同的综合资源使用量，可用于微调设计，使其适应给定目标。借助图 3 所示的方法，可以利用依赖于目标的策略，为同一流水线描述生成多种不同的实现。

5.4 模型解析算法比较

表 1 展示了架构参数化对清单 5中所述流水线的影响。所选的五种策略，尤其是 *Direct* 策略的四种变体，受到了

表1。综合后的资源使用量：五种分辨率策略的对比，有无就绪/有效握手。*LLUT*：逻辑查找表 / 组合逻辑自适应查找表；*MLUT*：查找表存储器 / 存储器自适应查找表；*FF*：触发器（寄存器）

		Vivado（赛灵思 VU9P）					Quartus（Intel Stratix V）			
协议	策略	∑查找表	逻辑查找表	存储器查找表	移位寄存器查找表	FF	∑查找表	逻辑查找表	MLUT	FF
None (RawIO)	点对点	289	225	0	**64**	563	286	286	0	**640**
	DirectAuto	289	225	0	**64**	563	286	286	0	**640**
	DirectForceSRL	289	225	0	**64**	563	358	294	**64**	456
	DirectForceReg	225	225	0	0	**691**	286	286	0	**640**
	DirectFIFO:3:3	277	237	**40**	0	445	431	367	**64**	563
就绪/有效 (ReadyValidIO)	点对点	290	225	0	**65**	578	286	286	0	**644**
	DirectAuto	290	225	0	**65**	578	286	286	0	**644**
	DirectForceSRL	290	225	0	**65**	578	358	294	**64**	460
	DirectForceReg	225	225	0	0	**708**	286	286	0	**644**
	DirectFIFO:3:3	276	235	**40**	**1**	456	424	360	**64**	553

Vivado（赛灵思）和Quartus（Intel）综合器各自行为的影响。在赛灵思 FPGA 综合中，对于共享同一写使能信号、跨越至少三个寄存器级的信号，通常会推断出移位寄存器查找表（SRL）原语 [29]。相反，*Quartus* 不会自动推断此类 SRL，而 SRL 甚至不是 Stratix 架构的基本原语。因此，为了即使在极简示例中也能突出不同策略之间的差异，这里将异或阶段的延迟设为 2（相当于 2 个连续寄存器），而不是原清单5中规定的 0（组合连线）。由此，电路的总延迟变为 4 个周期，而非 2 个周期，从而能够推断出 SRL。为违背赛灵思综合器的默认行为，我们在`(* shreg_extract = "no" *)` Verilog 属性中采用了 *DirectForceReg* 策略。反之，我们利用显式的*megafunction* `altshift_taps`，强制 Quartus 使用存储器自适应查找表原语实现基于 RAM 的移位寄存器。

第一组行对应于阶段之间没有握手的情况，不会生成并连接额外的协议信号。第二组则实现了沿流水线从输入端传播的有效性信号，以及由输出端驱动的公共反压信号。这里的反压均匀地作用于整个流水线，因为只有一个输出，并且流水线在合并点已静态调度。以下段落将回顾采用这些不同策略和协议所获得的综合结果。

等效资源使用。 在赛灵思方面，对于每种协议实现，点对点、*DirectAuto* 和 *DirectForceSRL* 策略的资源使用量完全相同。这是预期之中的，因为连续寄存器会系统性地推断为移位寄存器查找表，除非像 *Direct-ForceReg* 策略那样明确避免这种情况。使用 *Quartus* 综合器时则呈现出不同的行为：点对点、*DirectAuto* 和 *DirectForceSRL* 策略的资源使用量完全相同。不会推断出移位寄存器，除非像 *DirectForceSRL* 策略那样显式实例化。结果还表明，点对点 和 *DirectAuto* 策略是等效的。由于 *Direct* 策略的可配置性和灵活性高得多，因此工业实验中将只考虑该策略。

架构参数化的影响。 使用存储器（LUTRAM 原语）而非寄存器（触发器）或移位寄存器查找表时，*Direct FIFO* 策略的影响十分明显。读写先进先出队列中的数据始终需要一些控制信号。存在 Ready/Valid 握手信号时，先进先出队列的控制信号很容易驱动。然而，为了实现无协议的先进先出队列策略

必须额外加入一部分硬件，以保证数据通过先进先出队列时延迟恒定。一个初始复位为预期先进先出队列延迟的计数器，会在每个周期递减，直至达到 0 并保持为 0。只有当计数器等于 0 时，才会激活先进先出队列的背压信号，即读使能。与基于寄存器的方案相比，由于先进先出队列和计数器的额外开销，这种机制会显著影响资源使用量。具体而言，这解释了为什么采用 ready/valid协议时，*DirectFIFO* 策略比不采用该协议时所需资源更少；乍看之下，这似乎令人意外。

通过在面向寄存器的上下文中强制使用存储器原语，这种复杂的策略在处理复杂设计中的局部拥塞问题时似乎很有用。局部拥塞确实是由资源使用不均衡造成的，例如某个区域对寄存器的需求极高，而查找表却几乎未被使用。采用这种难以捉摸的策略，可以改用 LUTRAM 来实现完全相同的信号传播，从而缓解寄存器拥塞。第 7 节将进一步探讨此策略的参数化。

性能。 设计时即旨在保证所得流水线的吞吐量和延迟。所得电路严格遵循设计人员表达的设计意图，不对各阶段以及时区之间的显式延迟作任何更改。此处采用的解析流程首先精确平衡所有并发路径——即合并解析——然后以精确的延迟传播所需信号。接着，将协议信号作为全同步电路的一部分实现并连接。遵循这一模式，性能方面无需妥协：先进先出队列始终按照最坏情况进行定尺寸，因为这是高吞吐量网络应用的要求。顺带一提，对于某些以平均性能而非最坏情况性能为目标的应用场景，可以根据预期占空比减小延迟先进先出队列的深度。采用此类策略可以在资源占用与性能之间进行权衡。

总之，此示例展示了尝试多种架构参数化方案的能力：在保留原有功能和性能的同时，可实现截然不同的资源使用量。下一节将进一步介绍如何在更大规模上使用*PAF* 框架。

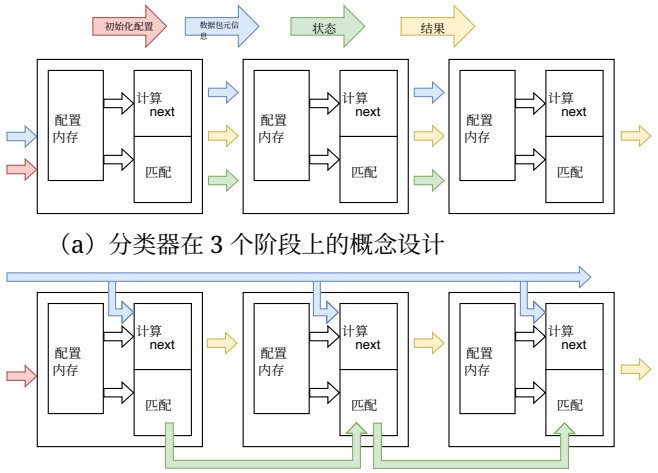
6 工业用例

本节详细介绍*PAF* 在工业应用中的用例，旨在通过以下两项标准，验证其在高速网络应用场景中的适用性：

架构参数化 不同的生成策略会产生各异且总体上可预测的资源使用量。**零成本流水线抽象** 与等效的完整描述相比，使用该框架生成的电路具有相似的资源使用量。由于严格取决于流水线描述，其延迟和吞吐量完全相同。

6.1 应用场景

基于树的分组分类器通常用作网络应用的构建模块，以便按类别对分组进行分组，类别通常取决于其协议、源地址和目的地址。分组分类器使网络设备能够将流定义为一系列具有相同标签的分组。下游处理模块随后会利用这些信息处理分组，例如在必要时丢弃分组。实现分组分类器可采用多种算法及其对应的架构，范围涵盖线性搜索、几何算法和高级启发式算法 [11, 27]。与硬件设计和实现中的其他情况一样，可配置性和性能会带来相当大的资源成本，因此必须在这些参数之间取得平衡，才能提供高效的分组分类器。采用 HyperCuts [26], 中提出的树方法，可以压缩分类规则，使实现



（a）分类器在 3 个阶段上的概念设计

(b) 分类器的实际实现架构 图 7. 分类器设计

节省资源，同时保证每周期处理一个数据包的吞吐量。树结构要求数据包元数据、内部状态和结果之间实现紧密同步，因此它是我们框架的一个有趣应用。图 7a 展示了其基本原理：一系列可配置的处理阶段，每个阶段对应树中的一个层级。此处仅展示了三个阶段，但阶段数量必须完全可配置，并取决于特定应用预期的最大规则数。在每个阶段，系统根据上一阶段的结果，从内存中读取当前配置。随后将该配置与数据包元数据及内部树状态结合，以生成：(1) 更新后的树状态；(2) 分类结果；以及 (3) 下一阶段配置的地址。

在网络环境中，网络数据包元数据通常包含所谓的五元组：源 IP 地址和目的IP地址、协议，以及源端口和目的端口。对于支持 IPv6 的网络设备，IP 地址为 128 位、协议标识符为 8 位、端口为 16 位，因此需要至少 296 位的数据总线，并通过适当的同步将其分发到各个阶段。实际应用中，若采用经实验验证较为合理的 40 个阶段，且每个阶段的延迟为 6 个周期（内存访问占 2 个周期，计算占 4 个周期），那么仅为转发数据并向各阶段提供元数据，就需要超过 7 万个寄存器。在整个设计中实际实现这种信号转发对资源使用量的影响相当显著，因此这类架构选择对于最终满足时序要求至关重要。

该分类器的原始实现采用 SystemVerilog 编写，目前分别面向 Intel 和赛灵思/AMD 的现场可编程门阵列，具体为 *StratixV* 和 *UltrascalePlus* 架构 [15]。在一种将数据与计算一并转发的理想实现中，寄存器数量过多，无法对设计进行高效的布局布线。因此，我们改为利用厂商提供的移位寄存器，但这些寄存器各有其能力和限制。赛灵思提供基于LUT的SRL原语，可管理最长 32 位、宽 1 位的移位寄存器。只要移位寄存器不跨越层次边界，其工具链 Vivado 就能高效地推断出移位寄存器。相反，Intel工具链 Quartus 在大多数情况下会避免推断移位寄存器，因为它会使用 M20K 或 MLAB 存储块，而这类资源在现场可编程门阵列中相当稀缺且宝贵，通常也更难布线。为了强制使用合适的厂商原语，我们不得不手动提取设计中的数据转发部分，并将其直接集成到顶层

层级模块中，这打破了黑盒范式，并要求公开内部流水线的长度信息。

基于图 7a 所示的理论架构，图 7b 描绘了设置这一专用实现所需的显式信号转发策略。三种不同的数据流通过各自的显式转发实现沿着各阶段传递：

数据包元数据 在阶段开始时读取一次，内部不再修改。此处采用抽头移位寄存器非常有效，它可以将元数据沿流水线旁路转发，并在配置检索之后同步地将其提供给各个阶段。

计算结果 在主流水线中转发，因为它们会在各个阶段中持续使用和修改。在此处与计算进行适当拆分较为困难，只能在设计中进行局部拆分，例如与内存访问并行进行。

树状态 仅在计算阶段、配置检索之后修改一次。该修改很简单，可通过组合方式完成，因此能够实现一个独立的、基于移位寄存器的旁路，其深度等于完整的阶段延迟。

最终，这种经过优化的实现极大地限制了原始架构的模块化，使其难以维护和升级。例如，在计算模块中添加或移除寄存器，都需要重新计算延迟的变化，并将其传播到整个流水线，以确保数据保持正确同步。

6.2 使用框架重新实现

为了在该分类器上试用我们的框架，首先使用 *sv2chisel* [4] 将原始 Verilog 版本转换为 Chisel。为避免引入回归问题，首先逐字翻译，并使用现有测试平台对初始转换版本进行验证。经过一些手动调整后，转换后的版本在资源使用量和时序匹配能力方面与原版相近，适用于多个网络应用。从此以后，这个经过验证的版本称为基线，可以放心地使用高级功能进行升级，例如我们的框架所提供的流水线自动化。

清单9 展示了原生 Chisel 版本中原本实现的显式信号传播，以满足同步要求。此代码片段展示了多个流水线级：

输入阶段 第1–4行，为简洁起见，此处仅考虑三个信号
初始组合阶段 第5–8行，（省略）
显式信号传播 第10–19行，用于将输入与计算同步的移位寄存器**中间计算阶段** 第21–24行，（省略）
节点阶段 第26–39行，需要将先前的计算与输入同步**最终阶段** 第40行，（省略）
需要来自前几个阶段的信号，第 40行
此实现中使用的显式信号传播会引发若干问题，具体问题已在代码清单的注释中详细说明。所有这些问题的根本原因在于，开发者必须全权负责管理同步的所有细节，导致描述冗长且容易出错。

为一次性解决所有这些问题，清单 10 展示了相同处理阶段的面向流水线的等效描述。基于显式的 `Pipe` 对象，该实现原生解决了计算阶段之间的同步要求，无需显式的信号传播。基于 PAF 的版本专注于实际的处理操作，省去了那些唯一目的在于确保正确同步的冗余语句。应用于完整模块后，代码行数顺带减少了约 12%。省略的信号传播随后以编程方式实现，从而支持展开时参数化

```
// 输入信号
val in计算 = 布尔值()
val inResult = 布尔值()
值输入配置 = 新 TreeFilterWordT
/*****************************************
 * （省略）若干处 in_cfg 在
 * 组合计算
 *****************************************/

// 使用移位寄存器进行显式传播
// 问题 #1：用户管理的延迟计算
val延迟 =移位器长度 + 1
// 问题 #2：用户管理的信号声明
valsrCompute = 移位寄存器(inCompute, 延迟)
val srResult = 移位寄存器(inResult, 延迟)
值srCfg = 移位寄存器(输入配置, 延迟)

/*****************************************
 * （省略）若干计算阶段
 * 提供 sftValue 信号
 *****************************************/
// 第3期：附加信号声明
值nodeCompute = 寄存器(布尔值())
值nodePrevResult = 寄存器(布尔值())
值nodeCfg = 寄存器(new TreeFilterWordT)

// 需要同步的计算
值nodeResult = 寄存器初始化(false.B)
nodeResult := srCfg.mask & sftValue
// 显式转发未修改的信号
// 实时重命名
nodeCompute := srCompute;
nodePrevResult := srResult;
nodeCfg := srCfg;

/**（省略）使用 node* 信号的最终阶段 */
```

清单 9. 显式传播信号的 VanillaChisel

```
// 使用感知流水线的命名来保证静态单赋值
值 前计算 = 布尔值()
值 前一结果 = 布尔值()
值 配置 =新 TreeFilterWordT
/*****************************************
 * （省略）在_配置中的若干用法
 * 组合计算
 *****************************************/
值 管道 = Pipe(输入, “主函数”)
// 无需显式传播
// 自动延迟计算

// 自动信号传播
// 可在生成时配置

/*****************************************
 * （省略）若干计算阶段
 * 提供 sftValue 信号
 *****************************************/
// 不会产生杂乱的转发信号声明

pipe.node = (p, n) => {
  n.currentResult = 步骤。寄存器初始化(p, false。B)
  n.currentResult := p.cfg.mask & p.sftValue
}
// 无需显式转发信号

/**（省略）使用前一阶段信号的最终阶段 */
```

清单 10。带隐式信号传播的面向流水线的 Chisel 版本

并针对各种同步策略进行实验。与原始实现不同，最终生成的流水线描述与图 7a 中的初始设计仍非常相似，同时保留了其灵活性。因此，探索和优化变得更加容易：修改处理部分时无需考虑数据转发，而改进数据转发部分时则可采用框架中适当的策略。

基于其内部的面向流水线的抽象，该框架还能够提供更丰富的流水线阶段及其信号表示，并着重体现同步要求，如图 6 和 5 中所示。生成的表示对设计人员非常有价值，可用于跟踪信号在整个流水线中的声明和使用情况。

6.3 结果

凭借简洁的组合方式和配置策略，该框架旨在根据流水线的展开时参数化，提供种类丰富的综合结果。本节介绍的结果针对两种不同的 FPGA 架构；这两种架构目前均已部署于 OVHcloud 的生产环境中，且除性能外，实现的是同一器件：

- 赛灵思 *Virtex Ultrascale+*，运行频率为 200 MHz，吞吐量为 400 Gbps。（*Vivado 2018.2*）
- Altera/Intel *Stratix V*，频率为 150 MHz，吞吐量为 40 Gbps。*(Quartus 15.1)*

按照原始 SystemVerilog 和原生 Chisel 实现，此处生成的是面向流水线的版本，不含协议握手信号。

表 2. 赛灵思 Ultrascale 上 52 级分组分类器的资源使用比较+

	∑查找表	逻辑查找表	查找表存储器	SRL	FF	RAMB36	RAMB18	URAM
基线	90,316	59,266	0	31,050	63,988	20	190	92
DirectAuto	90,294	59,199	0	**31,095**	64,128	20	190	92
	-22	-67	0	+45	+140			
	-0.02%	-0.11%	0	+0.14%	+0.22%			
DirectForceReg	57,900	57,842	0	58	**258,860**	20	190	92
DirectFIFO:6:16	86,391	64,216	**15,648**	6,527	42,079	20	190	92

表 2 展示了一个完全真实的 52 级分类器的资源使用量，该分类器由赛灵思 *Virtex Ultrascale+* 现场可编程门阵列综合得出。Xilinx 工具链报告了三类查找表：逻辑查找表 (LLUT)、LUTRAM 和移位寄存器查找表 (SRL)。后两者在内部共享相同的存储器查找表资源。最左列报告的这三种原语之和，是比较资源使用量的一个相关标准。事实上，赛灵思建议将查找表与触发器的比例维持在 1 左右，以获得最佳布局布线性能。就我们的两个标准而言，这些结果完全符合预期。利用 Vivado 的 SRL 推断行为，*DirectAuto* 策略使我们能够获得与原始实现几乎相同的资源使用量，从而验证了 PAF 的零成本抽象 。此外，另外两种策略报告的资源使用量则大不相同。*DirectForceReg* 策略完全基于寄存器（触发器原语），并显著降低了整体查找表使用量。*DirectFIFO:6:16* 策略在 SRL 和 LUTRAM 原语之间平衡了存储器查找表的使用量，同时也减少了寄存器使用量。此处深度和宽度阈值的选择完全是任意的，后续将在第7节进一步探讨。

表 3. Intel Stratix V 上 52 阶段数据包分类器的资源使用比较

	∑查找表	存储器查找表	逻辑查找表	FF	M20k	ALM
基线	73,877	16,090	57,787	22,556	447	51,331.1
DirectForceSRL	73,604	**15,140**	58,464	23,043	447	51,525.2
	-273	-950	+677	+487	=	+194.1
	-0.4%	-5.9%	+1.2%	+2.2%	=	+0.4%
DirectAuto	43,823	80	43,743	73,281	**1,004**	**33,430.9**
DirectFIFO:6:16	74,169	2,080	72,089	45,396	**1,116**	46,770.2

同样，表3展示了在一块*Intel/Altera StratixV* 现场可编程门阵列上，对一个经过略微不同配置的 52 阶段分类器的资源使用情况。表格最右列报告了现场可编程门阵列上实际使用的逻辑单元（ALM）估计数量。数值越低，表示电路占用面积越小，资源打包效率越高。*DirectAuto* 策略将这一数量降至最低，但代价是 M20K 利用率提高。尽管基线实现的 ALM 数量较高，但之所以经过审慎选择，是因为它似乎更易于在完整的网络设备设计中完成布线。*DirectForceSRL* 策略实现了与该基线实现接近的资源使用量，

尤其是在保持所需总 ALM 数量方面，即使查找表和寄存器的比例略有不同。该策略能够防止综合器推断基于 M20K 的移位寄存器逻辑，从而保持与基线完全相同的数量。在此处，这种特性非常理想，因为它可以将这些稀缺资源保留给完整网络设备设计中的其他用途。事实上，*DirectAuto*策略展示了如何使用 M20K（块存储器）原语自动推断大型移位寄存器逻辑。最后，*DirectFIFO:6:16* 策略展示了另一种利用率平衡方式，同样基于 M20K 原语，但与*DirectAuto* 策略相比，用查找表替代了寄存器。

总而言之，综合结果在两个目标平台上、针对两项评估标准都完全符合我们的预期。PAF 提供面向流水线的抽象，且不产生任何资源使用量开销，同时支持广泛的架构参数化。对架构参数化能力的成功验证引出了另一个问题：如何为给定用例找到最佳参数化方案？这个复杂问题没有普适的答案，因为它高度依赖于目标平台以及当前设计周边的集成环境。不过，下一节将详细介绍 PAF 如何提供工具，帮助设计人员在几分钟内审查各种潜在配置。

7 架构参数化分析与自动化工具

前几节详细介绍了 PAF 展示多种架构参数的能力，以及这些参数如何直接影响最终的资源使用量。这一强大特性为设计人员提供了丰富的参数化能力，但也要求他们选择与设计需求相匹配的参数集。做出正确决策需要全面了解设计及其内容。借助同步模型，PAF 能够：（1）报告缺失关系，并按分辨率参数分组（例如深度和宽度）；（2）生成设计的所有参数化版本。随后可将这些设计交由综合工具处理，从而针对给定目标平台和特定上下文，选出最合适的参数集。

本文旨在详细介绍*DirectFIFO*策略在第6节所述工业用例中的细粒度参数化。为了更好地阐释该策略的参数化能力（取决于缺失关系的深度和宽度），我们首先详细介绍其实现。

7.1 基于先进先出队列的恒定延迟信号传播

先进先出队列（队列）专为充当可变延迟路径而设计，因为它们提供两个独立的接口：写入（入队）接口和读取（出队）接口。因此，它们通常用作缓冲区，以补偿或同步其他可变延迟路径，例如，在内存输出端接受反压，而内存本身不接受此类反压。为发挥这一作用，此类可变延迟路径需要通过协议信令彼此同步。对于先进先出队列，写入接口连接上游协议信号，读取接口则按如下方式连接下游协议信号：

上游。ready：=！先进先出队列。满先进先出队列。写入_使能：=上游。valid

下游。valid：=！先进先出队列。空先进先出队列。读取_使能：=下游。ready

从此以后，如果我们尝试移除围绕该先进先出队列的协议信令，并考虑到该电路完全不受反压影响且其数据始终有效，则可以观察到以下连接：

先进先出队列。写入_使能：=真。B 先进先出队列。读取_使能：=真。B

电路一上电[1]，，数据便会持续流经先进先出队列，延迟为 1 周期[2]，与先进先出队列的深度无关。要通过这种先进先出队列创建具有固定 N 周期延迟的路径，我们需要将 先进先出队列。读取_使能的激活延迟 N 个周期。此行为可通过一个寄存器实现：将读取_启动寄存器初始复位为值N，并在每个周期递减，直到达到并保持值 0。然后按如下方式连接读接口：

```
先进先出队列。读取_使能: = 读取_开始 === 0。U
```

这产生的效果等同于一个延迟 N 个周期的移位寄存器，但其描述基于使用存储块原语的先进先出队列架构。

基于这一实现，*DirectFIFO:D:W* 策略引入两个可配置阈值，超过这些阈值后，给定的信号传播将通过先进先出队列而非移位寄存器实现。

(1) D：最小信号传播深度阈值 (2) W：最小信号宽度阈值

采用最小阈值是一种初步的简单方法，其合理性在于先进先出队列的资源成本会随深度和宽度呈非线性变化。事实上，先进先出队列基于块存储器原语，而其配置受限于 FPGA架构所支持的一组*(深度，宽度)* 组合。要充分利用这一策略的参数化特性，首先需要分析设计中的信号传播深度和宽度。

7.2 展示和利用架构参数化

除了提供零成本的面向流水线的抽象和架构参数化之外，PAF 还提供了用于高效分析设计的工具。具体而言，它能够报告待实现的信号传播的深度和宽度概览，以及当前策略对其的实现方式。表 4 详细列出了框架正在实现的所有直接信号传播中相应 （深度，宽度）二元组的出现次数，其中深度大于或等于 3。子模块 *TfCompute*以完全相同的参数化形式出现在分类器的每个阶段中，但只需作为独立流水线展开一次，随后在最终电路中实例化 97 次。

基于此报告，可以通过简单的脚本生成相关的（深度，宽度）最小阈值。随后，这些阈值被用作*DirectFIFO:D:W* 策略的参数化配置，该策略可在几分钟内生成相应的 22 个由 Chisel 生成的 Verilog 层次结构。再编写几个脚本，即可启动相应的综合并报告各自的资源使用量，整个过程无需人工干预，只需数小时即可完成。

表5 展示了在上述赛灵思工具链上完成综合后，这项快速探索研究的结果。结果按LUTRAM数量从低到高排列，LUTRAM的存在对应于先进先出队列。正如预期，无限阈值的资源使用量与此前在表*DirectAuto* 策略中报告的资源使用量相同（见表2）。类似地，*(6, 16)* 这一直接先进先出队列策略的默认参数化配置，其资源使用量与*(6, 62)* 参数化配置相同，因为两者在此设计中匹配到相同的信号传播。有趣的是，该默认参数化配置的LUTRAM数量接近中位数，这证实了为当前设计任意选取这些默认阈值是合理的。该参数化配置还使寄存器和查找表的累计数量最少。就查找表与寄存器的比例而言，原始策略仍最接近 Xilinx指南。一个有趣的替代方案

[1]或者是复位，在这种情况下，对于高电平有效复位，`true.B` 应理解为`!reset`。通过简单的存储器实现，可达到 21 个周期的延迟。在不失一般性的前提下，只要更高的延迟仍低于信号传播深度，就可以增加此延迟。

表 4. PAF 在流水线展开后报告的设计中待实现直接信号传播的分布

规格说明		出现次数		
深度	宽度	TfCompute	TfTop	总计
4	64	1×97		97
5	1	2×97		194
6	1		45	45
6	12		1	1
6	264		1	1
8	6		6	6
8	12		90	90
8	62		96	96
8	132		6	6
8	246		51	51
8	264		45	45
419	1		1	1

表5. 相关阈值参数化的资源使用量

深度	宽度	查找表存储器	SRL	LLUT	∑查找表	FF	查找表+触发器	查找表/触发器	RAMB18	URAM
∞	∞	**0**	31,095	59,199	90,294	64,128	154,424	1.41	190	92
419	1	16	30,828	**59,156**	90,000	64,635	154,635	**1.39**	190	92
8	264	4,288	25,229	61,657	91,174	58,943	150,117	1.55	190	92
6	264	4,384	25,129	61,385	90,898	58,727	149,625	1.55	190	92
8	246	11,488	12,966	62,440	86,894	47,218	134,112	1.84	190	92
6	246	11,584	12,836	62,354	86,774	47,181	133,955	1.84	190	92
8	132	11,824	12,430	62,517	86,771	46,739	133,510	1.86	190	92
6	132	11,920	12,296	62,644	86,860	46,621	133,481	1.86	190	92
8	62	15,552	6,662	64,148	**86,362**	42,184	128,546	2.05	190	92
6	62	15,648	6,527	64,216	86,391	**42,079**	**128,470**	2.05	190	92
4	64	15,720	6,896	66,266	88,882	42,965	131,847	2.07	**0**	92
8	12	16,264	6,130	65,631	88,025	42,822	130,847	2.06	190	92
8	6	16,312	6,094	65,780	88,186	42,854	131,040	2.06	190	92
8	1	16,328	5,835	65,732	87,895	43,380	131,275	2.03	190	92
6	12	16,368	5,984	65,688	88,040	42,698	130,738	2.06	190	92
6	6	16,416	5,948	65,782	88,146	42,728	130,874	2.06	190	92
6	1	16,520	5,891	66,204	88,615	43,161	131,776	2.05	190	92
5	1	16,900	5,752	68,199	90,851	44,997	135,848	2.02	190	92
4	62	19,448	1,130	67,819	88,397	**38,424**	126,821	2.30	**0**	92
4	12	20,168	584	69,281	90,033	39,039	129,072	2.31	**0**	92
4	6	20,216	548	69,410	90,174	39,073	129,247	2.31	**0**	92
4	1	20,700	**106**	71,794	92,600	41,854	134,454	2.21	**0**	92

是强制使用BRAM原语而非LUTRAM来实现先进先出队列，以此减少查找表数量。不过，在当前网络设备中，这并不可行，因为设计的其他部分几乎已经占用了所有BRAM。

最后，表 5 揭示了一个异常现象：对于一个 *(4,64)* 传播结构，其输出驱动一个只读存储器（ROM）。使用先进先出队列而非移位寄存器来传播只读存储器的读地址时，综合器会决定移除BRAM原语，并将其内容纳入传播中的LUTRAM中。这种BRAM推断和简化行为在使用 *Vivado2018.3* 时有所改变，因此，对于我们的一些较大型设计，满足时序要求的能力会出现显著差异。借助框架的探索能力，这种细微差异不仅能得到凸显，框架还提供了相应的补偿手段。在这个具体案例中，可以为这条特定路径使用移位寄存器，或者在FIFO实现中采用赛灵思BRAM原语。

凭借如此广泛的架构参数化，流水线自动化框架展现了自身的能力：既能提供指导，帮助选择合适的参数化方案；又能高效生成不同的流水线实现，并显著影响资源使用量。对各种实现进行综合后，它们的资源使用量在设计空间中分布广泛，为设计人员提供了极大的灵活性。尤其是查找表与寄存器的比例可以调整，这可能有助于在布线阶段实现时序收敛。在将硬件抽象提升到便于使用的面向流水线的建模层次的同时，PAF也增强了设计人员对其设计中最细微架构细节的控制能力。

8 结论与未来工作

本文介绍了一种流水线设计方法，旨在提供更强大的架构参数化能力。该方法采用完全解耦的 API：一方面用于表达流水线级的行为，另一方面用于定义同步策略和协议信令实现。我们通过基于图的同步模型弥合了这两个接口之间的差距，并在展开时为用户提供对生成设计的细粒度控制。我们利用 Chisel HCL 将该方法实现为自动化框架，成功迁移了一条复杂的现有流水线，且未增加任何资源开销，同时大幅减轻了开发者的同步负担。此外，将协议信令和同步相关问题与流水线描述解耦，显著提升了代码可读性和可复用性，同时有效减少了代码量。

我们计划将这一自动化方法扩展到可变延迟传播，以及处理不同抽象层级数据的多级流水线。增加从一个层级到另一个层级的自动转换支持，将大大简化跨层级模块之间的交互，并使架构参数化能力得到扩展，且几乎无需付出重新设计的代价。

参考文献

[1] Mustafa Abbas和Vaughn Betz。2018年。面向现场可编程门阵列的延迟不敏感设计风格。载于*2018*年第二十八届国际现场可编程逻辑与应用会议（*FPL*）。IEEE，360–3607。[2] Jonathan Bachrach、Huy Vo、Brian Richards、Yunsup Lee、Andrew Waterman、Rimas Avižienis、John Wawrzynek和Krste Asanović。2012年。Chisel：使用Scala嵌入式语言构建硬件。载于第*49*届设计自动化会议论文集。IEEE，1212–1221。[3] Pat Bosshart、Dan Daly、Glen Gibb、Martin Izzard、Nick McKeown、Jennifer Rexford、Cole Schlesinger、Dan Talayco、Amin Vahdat、George Varghese等。2014年。P4：用于编程协议无关分组处理器的语言。*ACM SIGCOMM*计算机通信评论 44，3（2014年），87–95。[4] Jean Bruant、Pierre-Henri Horrein、Olivier Muller、Tristan Groleat和Frédéric Pétrot。2020年。（系统）Verilog到Chisel的转换，以加快硬件设计。载于*2020*年快速系统原型国际研讨会（*RSP*）。IEEE，1–7。[5] Andrew Canis、Jongsok Choi、Mark Aldham、Victor Zhang、Ahmed Kammoona、Jason H Anderson、Stephen Brown和Tomasz Czajkowski。2011年。LegUp：面向基于FPGA的处理器/加速器系统的高层次综合。载于第*19*届*ACM/SIGDA*国际现场可编程门阵列研讨会论文集。ACM，33–36。[6] Luca P Carloni、Kenneth L McMillan和Alberto L Sangiovanni-Vincentelli。2001年。延迟不敏感设计理论。*IEEE*集成电路与系统计算机辅助设计汇刊 20，9（2001年），1059–1076。

[7] Jianyi Cheng、John Wickerson和George A. Constantinides。2022年。在动态调度电路中寻找并优化静态岛。载于《2022年ACM/SIGDA国际现场可编程门阵列研讨会论文集》（虚拟活动，美国）（FPGA '22）。计算机协会，纽约，纽约州，美国，89–100。[8] Jongsok Choi、Ruo Long Lian、Stephen Brown和Jason Anderson。2016年。一种用于在FPGA硬件中指定流水线和空间并行性的统一软件方法。载于《专用系统、架构与处理器（ASAP），2016年IEEE第27届国际会议》。IEEE，75–82。[9] Leonard W Cotten。1965年。高速流水线系统的电路实现。载于《1965年11月30日至12月1日秋季联合计算机会议论文集》第一部分。489–504。[10] Marc Galceran-Oms、Jordi Cortadella、Dmitry Bufistov和Mike Kishinevsky。2010年。自动微架构流水线化。载于《2010年欧洲设计、自动化与测试会议论文集》。IEEE，961–964。[11] Pankaj Gupta和Nick McKeown。2001年。数据包分类算法。IEEE Network 15，2（2001年），24–32。[12] Thomas G Hallin和Michael J Flynn。1972年。算术函数的流水线化。IEEE计算机汇刊100，8（1972年），880–886。[13] Ed Harcourt和James Perconti。2014年。用于指定流水线抽象的SystemC库。《微处理器与微系统》38，1（2014年），76–81。[14] Steven F Hoover。2017年。事务级Verilog中的时序抽象电路设计。载于《第35届国际计算机设计会议论文集》。IEEE，525–532。[15] Pierre-henri Horrein和Tristan Groleat。2021年。一种在FPGA上分类数据包字段的方法和系统。美国专利申请17/105,996。[16] Matei Iştoan和Florent de Dinechin。2017年。自动化算术数据通路的流水线。载于《2017年欧洲设计、自动化与测试会议论文集》。IEEE，704–709。[17] Lana Josipović、Radhika Ghosal和Paolo Ienne。2018年。动态调度高级综合。载于《2018年ACM/SIGDA国际现场可编程门阵列研讨会论文集》。ACM，127–136。[18] Lana Josipović、Shabnam Sheikhha、Andrea Guerrieri、Paolo Ienne和Jordi Cortadella。2021年。高性能数据流电路的缓冲区放置与定尺寸。《ACM可重构技术与系统汇刊》15，1，文章4（2021年11月），32 10 1145 3477053 页数。https://doi.org/ 。/ Charles E Leiserson和James B Saxe。[19] 1991 同步电路重定时。《Algorithmica》6，1-6（1991年），5–35。[20] Derek Lockhart、Gary Zibrat和Christopher Batten。2014年。PyMTL：用于纵向集成计算机架构研究的统一框架。载于《第47届IEEE/ACM国际微体系结构研讨会论文集》。IEEE计算机学会，280–292。[21] Maria-Cristina Marinescu和Martin Rinard。2001年。流水线电路的高级规范与高效实现。载于《第25届亚洲及南太平洋设计自动化会议论文集》。IEEE，655–661。[22] Rachit Nigam、Pedro Henrique Azevedo de Amorim和Adrian Sampson。2023年。采用时间线类型的模块化硬件设计。《ACM编程语言论文集》7，PLDI（2023年），343–367。[23] Oron Port和Yoav Etsion。2017年。DFiant：一种数据流硬件描述语言。载于《第27届国际现场可编程逻辑与应用会议论文集》。IEEE，1–4。[24] Karem A Sakallah、Trevor N Mudge、Timothy M Burks和Edward S Davidson。1993年。流水线同步。IEEE集成电路与系统计算机辅助设计汇刊12，8（1993年），1132–1146。[25] Pontarelli Salvatore、Bonola Roberto、Bifulcoand Marco、Cascone Carmelo、Spaziani Marco、Bruschi Valerio、Sanvito Davide、Siracusano Giuseppe、Capone Antonio、Honda Michio、Huici Felipe和Bianchi Giuseppe。2019年。FlowBlaze：硬件中的有状态数据包处理。载于《第16届USENIX网络系统设计与实现研讨会论文集》。[26] Sumeet Singh、Florin Baboescu、George Varghese和Jia Wang。2003年。采用多维切割的数据包分类。载于《2003年计算机通信的应用、技术、架构与协议会议论文集》。213–224。[27] David E Taylor。2005年。数据包分类技术综述与分类体系。《ACM计算综述》（CSUR）37，3（2005年），238–275。[28] Han Wang、Robert Soulé、Huynh Tu Dang、Ki Suh Lee、Vishal Shrivastav、Nate Foster和Hakim Weatherspoon。2017年。P4FPGA：用于P4的快速原型框架。载于《SDN研究研讨会论文集》（美国加利福尼亚州圣克拉拉）（SOSR '17）。计算机协会，纽约，纽约州，美国，122–135。[29] 赛灵思。2018年。《Vivado设计套件用户指南：综合》。赛灵思。https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2018_2/ug901-vivado-synthesis.pdf