计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
[doi: 10.15888/j.cnki.csa.010138] [CSTR: 32024.14.csa.010138]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 FPGA 的高精度带宽限制技术①

周　正, 高新平, 徐伟海

(紫金山实验室, 南京 211111)
通信作者: 高新平, E-mail: gaoxinping@pmlabs.com.cn

摘　要: 针对高吞吐网络环境下的带宽隔离需求, 本文提出一种基于 FPGA 的硬件化令牌桶带宽限制技术. 通过设计“时间驱动”模型, 将传统周期累积式令牌桶转化为发送时间计算机制, 避免令牌逐周期更新带来的开销; 结合整数化等效令牌注入技术, 在消除浮点运算的同时, 实现了 10 Mb/s–100 Gb/s 范围内的精细化速率控制. 系统采用流水线架构与双端口 BRAM 优化, 支持超过 4k 队列的并行限速调度控制. 在 Xilinx Alveo U200 平台上的实验结果表明, 所提方案在单队列调度场景下吞吐率提升近 100%, 资源开销显著降低 (LUT 减少 89%, 寄存器减少 77%, BRAM 减少≥20%), 速率控制误差低于 0.1%. 本技术为高性能网络系统提供了一种具备纳秒级精度和高扩展性的带宽隔离解决方案.

关键词: 智能网卡; 令牌桶; 带宽隔离; FPGA; 硬件加速

引用格式: 周正,高新平,徐伟海.基于 FPGA 的高精度带宽限制技术.计算机系统应用. http://www.c-s-a.org.cn/1003-3254/10138.html

High-precision Bandwidth Limiting Technique Based on FPGA

ZHOU Zheng, GAO Xin-Ping, XU Wei-Hai

(Purple Mountain Laboratories, Nanjing 211111, China)

Abstract: To address the demand for bandwidth isolation in high-throughput network environments, this study proposes a high-precision bandwidth limiting technique based on FPGA, which implements a hardware token bucket mechanism. A novel time-driven model converts the traditional cycle-accumulating token bucket into a transmission-time computation mechanism, thereby avoiding the overhead caused by per-cycle token updates. In addition, an integer-equivalent token injection technique is introduced to eliminate floating-point operations while enabling fine-grained rate control over a wide range from 10 Mb/s to 100 Gb/s. The system adopts a pipelined architecture and dual-port BRAM optimization, supporting parallel rate-limiting scheduling for over 4k queues. Experiments on the Xilinx Alveo U200 platform demonstrate that, in single-queue scheduling scenarios, the proposed technique achieves nearly 100% throughput improvement, with substantial reductions in resource usage (an 89% reduction in LUT, 77% in registers, and ≥20% in BRAM). The rate control error remains below 0.1%. This technique provides a nanosecond-precision and highly scalable bandwidth isolation solution for high-performance network systems.

Key words: smart NIC; token bucket; bandwidth isolation; field programmable gate array (FPGA); hardware acceleration

随着网络架构的持续演进以及虚拟现实 (virtual reality, VR)、工业互联网等新兴应用的广泛部署, 终端侧对带宽资源管理提出了“高吞吐、低时延与强隔离”的更高要求. 传统软件化的令牌桶限速机制受到处理器调度粒度与操作系统抖动的限制, 难以在百 Gb/s 级链路下实现纳秒级精度与稳定控制, 已成为性能演进

① 收稿时间: 2025-09-28; 修改时间: 2025-10-27, 2025-11-21; 采用时间: 2025-12-01; csa 在线出版时间: 2026-03-02

的瓶颈. 与之相对, 现场可编程门阵列 (field programmable gate array, FPGA) 具备高度并行与可定制能力, 可将复杂的速率控制逻辑下沉至数据通路, 减少主机 CPU 干预, 显著提升调度效率与隔离强度, 为高精度带宽限制提供硬件基础支持.

在工程实践中, 将发送端与接收端的带宽控制卸载至 FPGA 智能网卡具有直接且可量化的收益: 其一, 面向安全隔离, 在多租户或多模态并发场景下可抑制异常/恶意流量对其他业务的干扰, 降低 DDoS (distributed denial of service) 风险; 其二, 面向系统效率, 接收侧限速可缓解 DMA 通道拥塞与主机内存带宽压力, 从而提升端到端吞吐; 其三, 面向主机负载, 通过硬件整形降低中断频率与 CPU 占用, 改善极端负载下的系统响应; 其四, 面向虚拟化场景, 硬件化的发送端限速可避免虚拟机绕过主机限速带来的精度与安全隐患. 上述动因共同驱动了对“高精度、强隔离、可扩展”的硬件级带宽限制技术的需求.

尽管已有工作探索了多种基于 FPGA 的限速方案, 但在“精度-规模-资源”之间仍存在突出的权衡: 基于寄存器的逐周期令牌更新可获得较高精度, 然而其资源开销随队列规模近线性增长, 难以支撑海量并发流; 依赖 BRAM 的周期性批量注入具有更好的扩展性, 却往往牺牲限速精度与策略灵活性. 这一矛盾提示需要一种新的建模与实现路径, 以同时兼顾高精度与高扩展.

针对上述挑战, 本文提出一种时间驱动的 FPGA 带宽限制技术: 将传统令牌桶的“周期累积”重构为“可发送时间”的按需计算, 避免逐周期更新造成的资源浪费; 结合整数化等效注入消除浮点/除法器开销, 并采用流水线与双端口 BRAM 优化以支持超过 4k 条队列的并行调度与纳秒级精度控制; 在 10 Mb/s–100 Gb/s 的宽速率范围内实现细粒度限速, 误差低于 0.1%. 在 Xilinx Alveo U200 平台上的实验表明, 所提方案在单队列调度场景下的吞吐率接近翻倍, LUT、寄存器、BRAM 资源分别下降约 89%、77%、≥20%, 相较于既有方案展现出更优的性能-资源-精度综合指标与可工程化落地能力.

本文为构建高性能带宽隔离机制提供了关键支撑, 亦为智能网卡的限速子系统设计提供了参考范式. 具体而言, 本文的主要贡献包括以下几点.

(1) 提出一种时间驱动的令牌桶建模方法: 将传统的周期令牌累积机制转换为基于报文发送时间的计算模型, 从而避免逐周期令牌更新所带来的资源消耗, 实现对高精度限速行为的轻量级建模.

(2) 设计并实现可扩展的硬件限速架构: 基于 FPGA 构建高并行度令牌生成与消耗模块, 采用双端口 BRAM 实现令牌桶状态的并发读写, 支持超过 4k 条队列的并行限速控制, 具备良好的扩展性与可部署性.

(3) 引入整数化的等效注入技术: 通过离散化浮点速率参数与优化速率配置表结构, 避免使用高资源开销的硬件除法器, 有效降低 LUT 与寄存器占用, 提升资源效率.

(4) 在真实平台上进行系统验证: 在 Xilinx Alveo U200 平台上进行实现和测试, 验证了所提方案在调度精度、资源效率和系统吞吐方面的优势.

1 相关工作

网络流量带宽限制 (bandwidth limiting) 是实现多租户隔离与 QoS 保障的关键手段. 传统上, 令牌桶算法被广泛用于实现流量的平均速率控制, 每个流对应一个令牌桶, 周期性加入令牌以允许数据发送[1]. 在软件实现中, 如 Linux 内核的分层令牌桶队列规程, 令牌桶限速提供了灵活的策略配置[2]. 虽然纯软件实现的限速方案如 Linux HTB (hierarchical token bucket) 以及数据平面开发套件 (data plane development kit, DPDK) 用户态限速库等具有开发和配置灵活的优点, 但其瓶颈在于单机 CPU 的处理能力和调度延迟. 在高流量或大量细粒度流的场景下, 软件限速往往难以维持限速, 且定时精度受到操作系统调度影响[3]. 因此, 纯软件方案更适用于中低速、流量规模有限或对精度要求不高的场景. 例如在云数据中心中, 运营方更希望将宝贵的 CPU 资源留给应用, 而非消耗在数据包处理上. 因此, 近年来出现了大量将限速功能卸载到 FPGA 等可编程硬件上的研究, 来利用硬件的并行和定时精度优势, 实现高性能的带宽限制. 微软、亚马逊和阿里云等[4–7]大型云服务提供商已在其数据中心广泛部署 FPGA 来加速网络数据包的处理.

Park 等人[8]早在 2006 年采用 FPGA 设计了高速分组过滤与限速引擎, 用于缓解洪水攻击流量. 随后 Chen 等人[9]提出了一种应用于以太网拥塞管理的同步可调速率控制电路, 实现精细的发送速率调节. He 等人[1]针对多业务场景, 引入了“双向可调”令牌桶机制以同时保证各类流的带宽下限和剩余带宽共享. Zhang 等

人[10]则设计并在 FPGA 上实现了“两速率三色标记”(TRTCM) 的限流方案, 对流量进行彩色标记以区分合规和超额流量. 这些基于令牌桶模型的 FPGA 实现, 为硬件限速奠定了基础. 它们大多针对单一或少数流的限速, 验证了 FPGA 在流量整形方面的高效性和低延迟优势.

随着应用需求扩大, 研究者开始关注如何在 FPGA 上支持大规模、多队列的限速, 同时兼顾精度和资源开销. 寄存器式做法为每个流在 FPGA 上分配独立的计数器和参数寄存器, 可实现精细的速率控制, 但其资源占用随支持的流数呈线性增长, 难以扩展到成千上万的流. 例如文献[9–12]的方案中, 每个令牌桶状态参数 (如令牌计数、上次更新时间等) 都存储在高速寄存器或逻辑单元中, 尽管限速精度高, 但可支持的队列数量受限于 FPGA 上的有限查找表 (look-up table, LUT)/寄存器 (flip flop, FF) 资源. 针对这一问题, 部分研究转向利用片上 Block RAM 或外部存储来存储大量令牌桶状态来提升可扩展性[12]. 例如, Park 等人[8]和 Antichi 等人[13]的工作尝试将令牌桶的信息存放在双端口 RAM 或主存中, 在每个固定时间片为所有桶统一添加令牌. 然而, 由于 BRAM 读写的同步特性, 这类方案通常要求同时以固定周期为所有桶注入令牌, 导致无法根据各流需要单独调整令牌更新频率, 精细控制粒度不足. 纯 BRAM 方案虽然极大增加了可支持的流数, 但可能出现限速精度偏离配置值的情况. 由此可见, 在“寄存器式精细控制”与“存储器式大规模扩展”之间存在一个基本权衡: 前者精度高但规模受限, 后者易扩展但控制粒度较粗.

为了同时兼顾精度和大规模支持, 近期一些研究在令牌桶实现机制上进行了创新. Guo 等人[14]提出了一种面向多租户的大规模限速器设计, 核心思想是采用“报文头调度”和“令牌转时间”的新机制来替代传统的周期令牌注入. 该方法将每个队列的令牌产生过程转化为计算该队列头部数据包的可发送时间: 即根据配置速率, 将令牌积累所需的时间换算为报文的调度时间, 当时间达到时再发送报文. 这样避免了逐个周期更新每个桶令牌计数的高开销操作. 实验结果表明, 该设计在支持 512 个队列的情况下, 速率控制精度误差低于 0.4% (覆盖 100 Kb/s–10 Gb/s 范围), 而 FPGA 资源消耗仅约 1.16% 的查找表和 2.62% 的触发器, 比现有方案大幅降低[12]. 这一结果突出表明, 通过优化令牌桶模型和调度方式, 可以在 FPGA 上实现高精度且大规模的限速服务.

可编程调度与限速耦合. 限速不仅决定“是否可发”, 还影响“何时/谁先发”. 基于 PIFO 的可编程调度提出在硬件中以统一原语表达多类调度算法, 并给出线速实现路径[15]; 其核心把调度分解为顺序 (order) 与时间 (time) 两类决策, 可与限速的“资格”判定自然衔接. 后续 PIEO 工作进一步将“排名 (rank)”与“资格谓词 (eligibility)”整合到 PIEO 原语中, 在 FPGA 原型上展示了比 PIFO 更强的表达力与更高的可扩展性[16]. 在系统工程上, 若执行面采用非流水式原语, 多队列总吞吐受每包周期数约束; 而采用流水线可编程调度 (如 PIFO/PIEO 类实现) 可在保持可编程性的同时提升整体调度吞吐, 但需要在表达能力、硬件代价与限速资格判定的接口上做权衡.

标准化与工程实践. 时间敏感网络 (TSN) 提供了工程上可复用的整形/调度基元: 信用式整形 (CBS, IEEE 802.1Qav) 通过“信用”机制在交换设备上平滑流量[17]; 异步整形 (ATS, IEEE 802.1Qcr) 不要求全网时间同步, 支持逐流出口计量与最大时延保证, 其工程解读可视作对令牌桶的改良以降低硬件复杂度[18,19]; 周期排队与转发 (CQF, IEEE 802.1Qch) 通过周期化入/出队实现零拥塞丢包与确定性时延[20]. 这些机制从标准层面界定了限速/整形与调度耦合的边界条件, 也为评测提供了统一维度 (误差、突发支持、并发规模、功耗/资源等).

此外, 为了提升系统灵活性和速率动态可调性, 已有工作提出软硬件协同的限速框架. 典型如 SENIC 系统[21], 将高速令牌桶部署在 NIC 硬件端, 由软件周期性管理桶与流的映射关系, 实现在有限硬件资源下对大规模流量进行近似独立限速. 虽然此类架构兼顾性能与灵活性, 但其实现复杂度较高, 且在策略更新周期内仍存在调度延迟.

总体而言, 当前研究在高精度带宽限制方面主要围绕以下 3 方向展开. 一是基于 FPGA 的逻辑优化, 如流水线结构和整数化运算替代以降低 LUT 和 FF 使用率; 二是存储结构优化, 如使用 BRAM 支持海量状态存储; 三是调度机制重构, 将传统周期性更新替换为按需计算, 提升速率控制的精度与实时性. 本文提出的“时间驱动”模型正是对上述第 3 类方法的系统化扩展, 并在硬件层面实现高吞吐、低资源消耗和纳秒级控制

精度的统一, 验证了该路径的工程可行性.

2 研究动机与整体架构

2.1 研究动机

在现代高性能网络系统中, 带宽隔离 (或称带宽限制) 技术是实现网络资源公平分配、延迟控制以及提升带宽利用效率的关键机制. 传统的流量调度机制多集中于发送端, 然而, 仅依赖发送端的控制策略难以充分应对复杂网络环境带来的挑战. 因此, 在接收端实施有效的带宽限制, 对于保障系统稳定性和性能具有重要的意义. 将发送端与接收端的带宽控制机制卸载至 FPGA 智能网卡, 可显著提升整体系统的性能与稳定性, 并有效减轻主机系统的处理开销. 其必要性主要体现在以下 4 个方面.

(1) 增强系统安全性, 防御拒绝服务攻击. 缺乏有效的流量限制可能导致某些应用和服务产生的恶意或异常流量迅速耗尽系统资源, 进而干扰其他正常业务的运行. 通过在 FPGA 上实现高精度令牌桶算法, 可对发送端和接收端的流量进行精准管控, 防止单一实体过度占用资源, 从而保障整体网络系统的安全性与稳定性.

(2) 缓解 DMA 带宽压力, 提升系统吞吐能力. 在高速网络环境中, 网卡通过直接内存访问 (DMA) 机制将数据包直接写入主机内存. 若接收端缺乏有效的带宽控制措施, 瞬时突发的海量数据包极易引发 DMA 通道拥塞, 加剧主机内存带宽压力, 并干扰关键任务的执行. 在 FPGA 智能网卡端实施高精度接收端带宽限制, 可有效缓解 DMA 通道的拥塞压力, 进而提升系统整体吞吐能力.

(3) 降低中断频率, 减轻 CPU 负载. 当数据包频繁到达时, 传统的中断驱动机制会频繁触发中断, 导致 CPU 资源浪费并形成性能瓶颈. 尽管 NAPI 等机制通过结合中断与轮询方式对此有所改善, 但在极端高负载条件下, CPU 资源紧张问题依然存在. 通过在 FPGA 上实现精确的流量隔离, 限制接收端数据包的突发速率, 可显著降低中断频率, 有效减轻主机 CPU 负载, 并提升系统响应性能.

(4) 解决发送端软件限速的精度与性能瓶颈. 软件实现的发送端流量限速机制通常存在精度不足、抖动显著以及 CPU 资源消耗较高等问题. 此外, 在虚拟机环境中, 存在绕过主机系统流量限制的安全隐患. 将发送端限速机制同样卸载至 FPGA 智能网卡, 能够实现高精度、低抖动的流量控制, 同时有效规避虚拟机绕过限速机制的风险, 从而提升系统资源利用效率与安全性.

本文的目标是在硬件数据面上提供可配置且可验证的带宽控制与调度能力, 确保在多队列环境下既能遵守速率与突发约束, 又能表达优先级与服务区分; 同时, 设计应保持对上层应用和下游链路行为的可预期性, 并在资源与实现复杂度之间取得合理折中. 模块既可以部署在接收侧也可以部署在发送侧. 位于接收侧时, 它在 NIC→主机内存或 NIC→交换芯片的入口处消减突发, 降低 DMA 流量与中断压力, 避免过载扩散; 位于发送侧时, 它在出向链路上执行整形与调度, 保证发往下游的流量符合对端处理能力与策略约束. 两侧共享同一套参数与语义, 但策略侧重点不同: 接收侧偏防御与抑制, 发送侧偏服务质量与成形.

2.2 整体架构

在基于 FPGA 的智能网卡设计中, AXI4-stream (AXIS) 总线已成为传输数据包的主流接口标准. 网络流入的流量经由网卡的媒体访问控制 (MAC) 层和物理接口 (PHY) 层完成串并转换后, 通过 AXIS 总线传输数据帧. AXIS 总线提供高带宽、低延迟的流式数据传输通道, 尤其适用于网络数据包等连续数据流的处理. 其握手机制通过 TVALID 和 TREADY 信号实现流控, 确保数据仅在发送方与接收方均准备就绪时进行传输, 有效避免数据丢失或拥塞. 此外, TKEEP 信号支持对数据字节进行精细控制, 指示有效字节的位置, 增强了数据传输的灵活性. 这些特性为智能网卡中的模块化设计与高效数据处理提供了有力支持.

本文设计的带宽隔离模块正是基于 AXIS 总线构建的. 该模块可灵活地插入 AXIS 数据路径, 无需对现有其他模块进行结构性修改, 充分利用了 AXIS 总线的标准化接口及其握手机制.

图 1 展示了带宽限制模块的总体架构. 该模块可部署于智能网卡的数据接收和发送路径中, 负责对网络数据包进行调度及带宽整形控制. 整体架构构建于 AXIS 数据通路之上, 具备良好的模块化特性与可插拔性.

该架构的核心思想是将带宽限制的“策略计算”与“调度执行”相分离. 本文的贡献主要集中在图 1 上方红色虚线框标出的调度计算模块, 它负责实现具体的带宽限制策略. 各子模块功能如下.

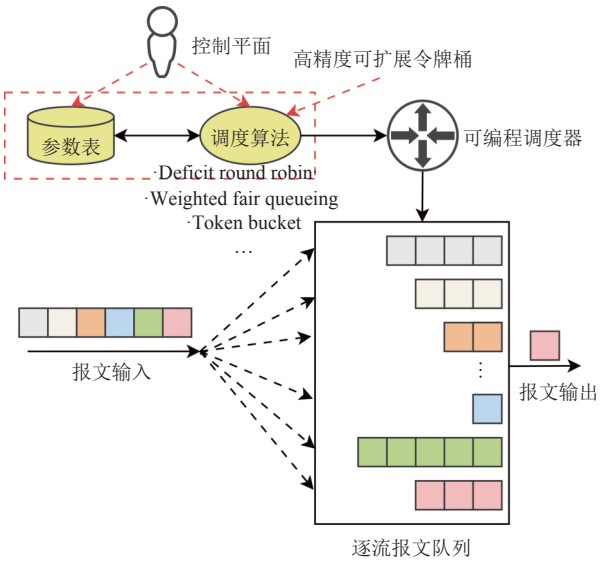

图 1 带宽限制模块总体设计

● 调度计算模块: 模块由参数表和调度算法模块共同组成, 是本文所提高精度令牌桶算法 (即图 1 中的“调度算法”) 的计算单元. 它摒弃了传统令牌桶逐周期累积令牌的方式, 而是采用“时间驱动”模型 (详见第 3 节). “时间驱动”模型正是实现架构高可扩展性的关键. 传统方案为每个队列使用寄存器 (LUT/FF) 存储令牌状态, 导致资源消耗随队列数 (如 4k+) 线性增长, 迅速成为扩展瓶颈. 本设计中, “时间驱动”模型避免了对所有队列的逐周期更新, 使得所有队列的状态 (如 *last_send_time*、*spare_tokens* 等) 可以被高效地集中存储在高密度的“参数表” (基于 BRAM/URAM 实现) 中. BRAM 仅在数据包实际到达该队列时才被“按需访问”一次, 极大降低了存储带宽压力. 这种设计将大规模队列的状态存储开销从逻辑资源 (LUT/FF) 转移至 BRAM, 使得核心逻辑资源消耗几乎与队列数量无关 (见表 1), 从而实现了对 4k 以上队列的高可扩展支持. 该模块根据传入的队列号和报文长度信息, 并读取“参数表”中的队列配置与状态, 为该报文计算出两个关键调度参数: 可发送时间 (*send_time*) 和优先级 (*rank*).

● 可编程调度器: 该模块是架构的执行单元. 在本设计中, 它实现了一种 PIEO (push-in extract-out) 调度原语. PIEO 调度器维护一个全局有序的元素列表, 并根据调度计算模块提供的参数执行两个核心决策: 1) 调度资格判断: 检查当前系统时间 *now* 是否满足报文的 *send_time* (即 *now*≥*send_time*). 2) 调度优先级比较: 在所有满足发送条件的报文中, 选择 *rank* 值最小 (优先级最高) 的报文. 最终, 调度器从多个队列中选出优先级最高且满足发送时间的报文, 送往链路.

● 数据包队列: 按照指定的规则将数据包分类到不同的逻辑队列中, 如按应用或网络模态进行隔离. 同时, 该模块会统计各队列的基本信息 (如队头数据包长度), 并将其发送至调度计算模块, 作为令牌桶算法的输入参数. 在网卡接收侧进行带宽限制时, 需要在 AXIS 总线上对数据包进行分类和缓存, 需要在占用 FPGA 硬件资源缓存数据报文, 也可利用网卡已有的 RSS 模块进行分类和缓存. 在网卡发送侧进行带宽限制时, 可在主机侧进行数据包的分类并将其存储于大容量的主机内存中, FPGA 硬件中只需维护和保存各队列的状态信息即可.

综上所述, 本节提出的整体架构是一个解耦的框架. 调度计算模块通过实现令牌桶算法, 将复杂的速率和突发控制策略转化为 *send_time* 和 *rank* 两个值; 而可编程调度器 (PIEO) 则充当一个通用的执行引擎, 根据这两个值做出最终决策.

第 3 节将详细介绍调度计算模块中高精度令牌桶算法的具体实现, 即如何高效地计算出 *send_time* 和 *rank*, 以实现高精度、可扩展的带宽限制.

3 高精度可扩展的令牌桶设计

3.1 令牌桶算法

在网络流量控制中, 令牌桶算法是一种经典且广泛应用的机制: 系统以恒定速率往桶中加入令牌, 每个数据包到来时需消耗与包长度相当的令牌才能通过, 否则若桶中令牌不足则该包被暂存或丢弃. 尤其在软件实现中因其简单性和灵活性而被广泛采用. 许多操作系统和网络设备通过软件方式实现令牌桶, 以实现基本的带宽限制和流量整形功能. 然而, 软件实现的令牌桶算法存在一些固有的限制, 特别是在高性能网络环境中. 由于操作系统调度的不确定性、上下文切换的开销以及中断处理的延迟, 软件实现的令牌桶难以提供精确的速率控制和低延迟的调度响应. 这些因素可能导致流量控制的精度下降, 影响网络服务质量. 通过在 FPGA 等可编程硬件中实现令牌桶算法进行数据包调度, 可以实现更高的精度、更低的延迟和更好的可预测性. 硬件实现能够在纳秒级别进行精确的时间控制, 适应高速网络的需求. 此外, 硬件实现还可以减轻主机 CPU 的负担, 提高系统的整体性能和稳定性.

为严格控制各数据流的带宽, 并在 FPGA 硬件中实现了 PIEO 调度模型下的令牌桶算法, 以实现更高的精度、更低的延迟和更好的可预测性. 使用令牌桶算法计算的 *rank* 和条件谓词如算法 1 伪代码所示.

算法 1. 令牌桶算法

输入: 数据包信息: *id*, *length*.
输出: 调度信息: *id*, *send_time*, *rank*.

```
1. //计算对应令牌桶令牌累加后的数量
2. tokens[id]+=rate[id]×(now–last_time[id])
3. //判断令牌数量是否溢出, 如果溢出则设置为最大值
4. if (tokens[id]>burst_threshold[id]) then
5.     tokens[id]=burst_threshold[id]
6. end if
7. //计算出该数据包的调度信息 send_time
8. if (length≤tokens[id]) then
9.     send_time=now
10. else
11.    send_time=now+(length–tokens[id])/rate[id]
12. end if
13. //更新桶内令牌数量和上次发送时间, 便于下次令牌累积计算
14. tokens[id]–=length
15. last_time[id]=now
16. //获取该数据包的调度信息 rank
17. rank=priority[id]
```

算法 1 中的一些变量名定义如下.

● *tokens*: 表示当前令牌桶中可用的令牌数量, 是一个按报文类型编号索引的状态变量. 系统周期性地为每个报文类型增加令牌, 用于控制其数据流的发送速率. 单位为字节, 通常与报文长度相对应.

● *rate*: 每个队列对应的令牌注入速率, 单位为“令牌数/时钟周期”, 表示该队列在单位时间内可获得的带宽预算, 用于反映其目标带宽上限. 例如, 当时钟频率为 200 MHz, 且每个时钟周期注入一个令牌时, 则该队列的带宽被限制在 1 600 Mb/s.

● *burst_threshold*: 各令牌桶允许的最大令牌累计值, 即桶的容量上限. 该值限制了突发流量的最大可接收范围, 用于防止系统在空闲后瞬时接入过多数据. 设置该值可以容忍一定程度的带宽瞬时波动, 以此来更好地支持业务和网络的突发特性.

● *now*: 当前系统时间, 通常由一个每个时钟周期自增 1 的硬件计数器标识, 用于计算令牌积累量和判断比较报文的可发送时间.

● *send_time*: 令牌桶算法计算出的报文可发送的最早时间点, 表示该报文满足速率限制条件后具备发送资格的时间. 该值将用于与当前系统时间比较, 判断是否可以立即调度出队发送.

● *last_time*: 每个队列上次成功调度报文时的系统时间戳. 该变量用于记录上一次令牌更新时间, 是令牌累加量 (*now*–*last_time*) 计算的依据, 更新此变量可确保令牌增量计算准确. 当数据包队列数量较多时, 需使用 BRAM/URAM 对 *tokens* 数组进行保存 (使用 LUT 资源进行大规模队列参数存储时时序难以收敛, 扩展性较差), 使用 *last_time* 计算令牌累加量, 可避免每个时钟周期都访问各队列当前 *tokens* 值, 从而减少对 BRAM 的频繁读取操作.

● *priority*: 每个队列的预设优先级值, 用于在 PIEO 调度器中计算该报文的调度等级 (*rank*). 该值越小代表优先级越高, 调度器在调度时将优先选择 *rank* 最低且满足发送条件的报文出队.

可以看出, 基于 PIEO 调度模型下的令牌桶算法为每个队列设置一组参数, 包括发送速率、桶容量和优先级, 硬件维护的一个令牌计数 *tokens* 和上次发送时间 *last_time*, 按照每个周期 *rate* 个令牌增加令牌数且上限为 *burst_threshold*. 具体来说, 每次触发计算时, 桶内令牌数为: $\min\{burst\_threshold, rate\times(last\_time-send\_time)\}$. 当有数据包进入时, 检查该队列的令牌计数是否不少于此包所需令牌. 如果有足够令牌, 则扣减相应数量, 并将该队列头数据包的发送时间 *send_time* 记为系统当前的时间 *now*, 同时更新上次发送时间 *last_time* 为 *send_time*; 如果令牌不够, 则将令牌计数置为 0, 并根据所需令牌的差值 *d* 和速率 *rate* 计算出该队列头数据包的发送时间 *send_time*, 同时更新 *last_time* 为 *send_time*. 主机驱动可在运行时动态配置每个队列的速率上限 (R)、桶容量 (B) 和优先级 (P) 参数, 从而实现带宽限制行为的在线调整与策略优化.

现有的 FPGA 令牌桶限速实现方法中, 大多利用寄存器资源存储队列配置参数和计算相关调度的相关变量, 资源利用率较低, 难以支持大规模队列限速. 利用 BRAM 存储队列配置参数的方法虽然扩展性较好, 但在限速精度上并不占优势. 本文主要使用以下两种机制缓解上述问题以支持低资源消耗下的大规模队列的精确限速.

● 基于时间等效替代的令牌生成: 将数据包长度与令牌桶内累积注入令牌数的关系转化为时间等效模型, 避免在每个时钟周期周期性地注入令牌, 从而使用

BRAM 存储大规模队列的配置信息和状态, 降低逻辑资源消耗.

● 基于整数的等效令牌注入: 将传统的浮点速率计算替换为等效的整数注入策略, 在无需复杂除法或定点运算的情况下, 实现低延迟下的高精度限速, 并大幅节省 FPGA 逻辑资源.

3.2 基于时间等效的令牌计算技术

在大规模队列进行令牌桶调度算法时, 为减少逻辑资源的消耗 (LUT/FF) 和时序收敛/布局布线难度, 需要使用 BRAM/URAM 对各令牌桶的状态信息进行存储. 为避免令牌桶算法定时性向桶中注入令牌, 从而导致 RBAM/URAM 浪费大部分读写周期用于桶内令牌数量的周期性注入, 需要记录状态值 (比如 *last_send_time*) 以便在调度时一次性更新令牌数.

根据令牌桶算法, 可把数据包发送所需的令牌数和令牌桶内可存放的最大令牌数都等效替换为其发送所需的时间, 从而对令牌桶算法进行基于时间序列的抽象建模. 下文中单位时间均为 FPGA 硬件中的一个时钟周期 (CYCLE), 通常为几纳秒.

如图 2 所示为抽象建模后的令牌桶算法, 其中, *now* 为系统当前时间, *last_send_time* 为令牌桶开始注入令牌的时间, *pkt_time* 为系统按照一定速率积累当前传入的数据包所需令牌的时间, *rest_send_time* 为令牌桶已有令牌的注入时长, *MAX_time* 为不同桶深时对应的最大积累时间. 根据不同的桶深 (*MAX_time_1* 和 *MAX_time_2*)、不同的 *rest_send_time* 和不同的数据包长度 (*pkt_time_1* 和 *pkt_time_2*), 数据包的发送时间 (*send_time*) 和状态 (*last_send_time*) 计算方式有所不同.

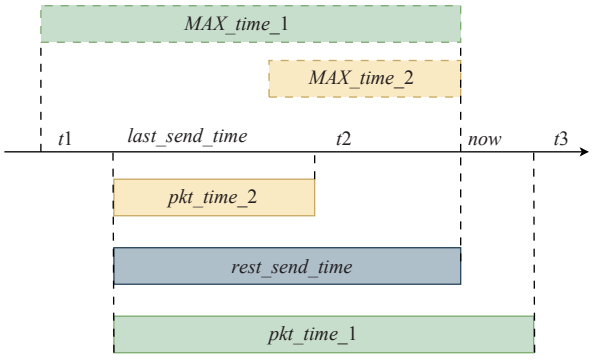

图 2 令牌桶算法模型

此外, 令牌桶算法模型已经隐含以下前提条件.

● *last_send_time*<*now*: 因上一个数据包调度发送之后, 才会选取下一个数据包进入令牌桶计算模块, 所以每次进行计算时, 前一个数据包已经进行了发送, 即 *last_send_time* 一定小于 *now*;

● *MAX_time*≥*pkt_time*: 令牌桶的桶深至少为一个最大传输单元 (MTU) 对应的令牌数量以满足突发需求.

根据以上条件限制, 令牌桶算法可分为桶内令牌未满和已满两种情况.

如图 3 所示, 为令牌桶桶内令牌未溢出的情况, 即 *MAX_time*≥*rest_send_time*, 其中桶深等效为 *MAX_time*, 桶中令牌数等效为 *rest_send_time*. 根据数据包长度的不同, 计算结果可分为立即发送和在未来指定时间后发送, 以达到带宽限制的目的.

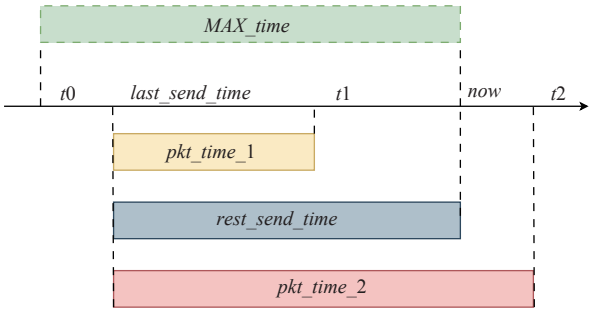

图 3 令牌桶未溢出

● 当桶内令牌数大于等于数据包所需令牌数, 即 *rest_send_time*≥*pkt_time_1* 时, 数据包的发送时间 *send_time* 计算结果为 *now*, 表示调度器可立即进行发送.

● 当桶内令牌数小于数据包所需令牌数, 即 *rest_send_time*<*pkt_time_2* 时, 数据包的发送时间 *send_time* 计算结果为 *t2* (*last_send_time*+*pkt_time_2*), 表示该数据流所需带宽超出了限制, 调度器需至少等待到 *t2* 时间再进行该数据包的发送. 最后将令牌开始注入的时间 *last_send_time* 更新为 *t2* (*last_send_time*+*pkt_time_2*).

如图 4 所示, 为令牌桶桶内令牌已溢出的情况, 即 *rest_send_time*>*MAX_time*. 由于 *rest_send_time* 等效的令牌数已经超出了该令牌桶中令牌的最大值, 需要将数据包开始消耗令牌的时间调整为 *t0* (*now*–*MAX_time*) 时刻. 由于令牌桶的桶深至少为一个最大传输单元, 即此时桶内所含令牌数一定能满足传入数据包所需的令牌数, 所以数据包的发送时间 *send_time* 为 *now* (任意小于 *now* 的, 可以和上述保持一致, *t1* 或者 *last_send_time*+*pkt_time*). 最后, 将令牌开始注入的时间 *last_send_time* 更新为 *t1* (*now*–*MAX_time*+*pkt_time*).

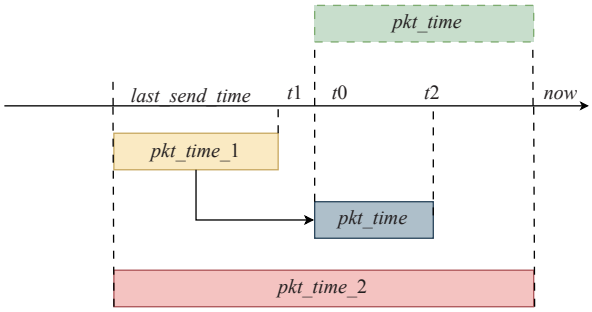

图 4 令牌桶溢出

为清晰地阐明该“时间驱动”模型的核心逻辑, 可将其归纳为算法 2 所示的伪代码, 该算法描述了计算 *send_time* 和更新 *last_send_time* 的完整过程.

算法 2. 时间驱动的令牌桶算法核心逻辑

输入: 当前时间 *now*, 当前数据包所需的等效令牌时间 *pkt_time*.
状态 (读/写): 上次发送时间 *last_send_time*.
配置: 最大累积时间/桶深 *MAX_time*.
输出: 数据包最早可发送时间 *send_time*.

```
1. //步骤 1: 计算自上次发送以来累积的令牌时间
2. // (对应图 3 和图 4 中的 (now–last_send_time), 即 rest_send_time)
3. accumulated_time←now−last_send_time
4. //步骤 2: 根据累积时间, 判断令牌桶是否溢出
5. if accumulated_time>MAX_time then
6. // 情况 A: 令牌桶溢出 (图 4)
7. // 桶已满, 累积的令牌时间被“钳位”在 MAX_time
8. // 必须倒推出令牌开始累积的有效起始时间 (t0)
9.     effective_start_time←now−MAX_time
10.// 由于桶是满的, 且前提条件保证桶深 (MAX_time)>=包大小 (pkt_time)
11.// 所以令牌一定充足, 数据包可以立即发送
12.    send_time←now
13.// 更新 last_send_time, 为下一个包做准备
14.// 下一个包的令牌累积将从这个包的有效消耗时间点 (t1) 开始计算
15.// (t1=effective_start_time+pkt_time)
16.    last_send_time←effective_start_time+pkt_time
17.else
18.// 情况 B: 令牌桶未溢出 (图 3)
19.// 桶未满, accumulated_time 则为桶内的有效令牌时间
20.// 检查令牌是否足够支付当前包 (pkt_time)
21.    if accumulated_time<pkt_time then
22.// 情况 B.1:令牌不足 (图 3 中 pkt_time_2 的情况)
23.// 必须等到令牌累积足够, 即在未来的 t2 时刻
24.// (t2=last_send_time+pkt_time)
25.        send_time←last_send_time+pkt_time
26.// 下一个包的令牌累积, 也必须从未来的 t2 时刻开始
27.        last_send_time←last_send_time+pkt_time
28.    else
29.// 情况 B.2: 令牌充足 (图 3 中 pkt_time_1 的情况)
30.// 令牌足够, 立即发送
31.        send_time←now
32.// 下一个包的令牌累积, 从现在 (now) 开始
33.        last_send_time←now
34.    end if
35.end if
36.// 步骤 3: 返回结果
37.// 最终的 send_time 为 now (立即发送) 或一个未来时间 (推迟发送)
38.// 调度器将使用 (now>=send_time) 来判断是否发送
39.return send_time
```

对于令牌桶算法, PIEO 调度器在调度出队时, 会选择满足调度条件 *now*≥*send_time* 且优先级最高的数据包进行出队. 因此, 如果令牌桶算法的结果是立即发送, 只需要满足输出的 *send_time* 值小于等于 *now* 即可. 如图 3 和图 4 所示, 当数据包应当立即发送时, *send_time* 的输出结果均可以直接使用 *t*1 (*last_time*+*pkt_time*) 表示. 结合令牌桶以上两种情况, *send_time* 和 *last_send_time* 的 RTL 级电路结构如图 5 所示.

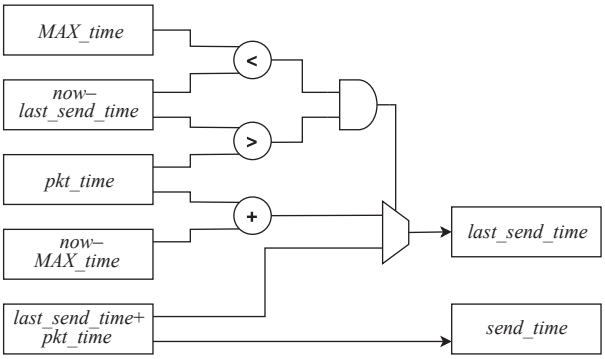

图 5 令牌桶部分硬件电路逻辑 1

一般情况下 (当桶内令牌未溢出时), 上次的发送时间 (*last_send_time*) 更新为本次数据包计算出的发送时间 (*send_time*), 即直接在上次发送的时间基础上加上本次传入数据包的等效时间. 因使用 *last_send_time* 来计算桶内令牌的数量, 且桶内令牌数量受到桶深的限制, 当该队列长时间未发送数据包时 (当桶内令牌溢出时), 需使用 *MAX_time* 作为桶内令牌数来进行后续计算.

3.3 基于整数的等效令牌注入技术

数据包的等效时间 (*pkt_time*) 由令牌桶令牌注入速率 (*rate*) 决定. 一个令牌通常对应一个字节, 使用数据包长度 (*pkt_len*) 除以令牌注入速率 (*rate*) 可得到该数据包的等效时间 (*pkt_time*). 当 *rate* 为 1 时, 代表每个时钟周期向令牌桶内注入一个令牌, 即该队列一个时钟周期发送一个字节. 以 250 MHz 时钟为例, 则对该

队列的带宽限制值为 2 000 Mb/s (不考虑突发情况), 此时, *pkt_len* 和 *pkt_time* 在数值上相等.

但在 FPGA 上直接使用上述方法来计算 *pkt_time* 并不合理, 尤其当需要支持精细的带宽限制时, 这种做法存在以下几个问题.

● 浮点/定点运算开销大: 当带宽限制要求为小数时, 如限制带宽为 3 000 Mb/s, *rate* 的取值为 1.5, 即每个时钟周期需注入 1.5 个令牌. 在软件中, 浮点数或定点数可以方便地处理这样的除法运算, 但在 FPGA 中, 除法操作本身既复杂又资源消耗高, 实现浮点数或定点数除法器, 需要消耗大量的逻辑资源 (LUT)、DSP 甚至 BRAM, 并且会显著增加计算延迟, 严重影响系统的调度精度和吞吐率.

● 整数速率粒度粗: 若限制 *rate* 仅取整数 (如 1、2、10 等), 则可实现的带宽值只能是等间隔的离散点, 例如 2 000 Mb/s、4 000 Mb/s、6 000 Mb/s 等, 无法覆盖中间范围的带宽, 粒度过粗, 难以满足实际需求.

● 定点精度难以权衡: 即使在 FPGA 中采用定点运算模拟小数除法, 也会遇到精度损失的问题. 定点数需要指定小数位数, 但若小数位数不足, 计算出的 *pkt_time* 可能无法精确反映真实的发送时间, 造成周期性积累误差, 影响调度精度; 而过多的小数位数又会占用大量的逻辑资源 (LUT、DSP 等) 和存储资源 (BRAM 等), 影响系统的扩展性和吞吐能力.

为此, 本文使用基于整数的等效注入方法, 用两个正整数参数 *increment* 和 *period* 替代 *rate*, 可准确模拟任意有理速率:

$$rate = \frac{increment}{period}$$

其中, *period* 为令牌注入周期 (时钟周期数), *increment* 为每隔 *period* 个时钟周期注入的令牌数. 例如, *rate*=1.5 可表示为 *increment*=3, *period*=2.

为消除小数截断带来的周期性累积误差, 引入非负整数状态参数 *spare_tokens* (代表桶内可直接使用的令牌数, 即令牌结余). 桶内剩余的总令牌数量由 *spare_tokens* 和第 3.2 节中提到的 *rest_send_time* 共同决定.

当使用每 *period* 个时钟周期注入 *increment* 个令牌来等效表示 *rate* 时, 计算 *pkt_time* 和更新 *spare_tokens* 的 RTL 级电路硬件图如图 6 所示.

假设一个字节对应一个令牌, 其算法逻辑步骤如下.

(1) 消耗结余: 传入的数据包先消耗结余的令牌 *spare_toekns*. 计算还需令牌数: *needed_tokens* = *pkt_len*−*spare_tokens*.

(2) 计算周期数: 使用 *increment* 对 *needed_tokens* 进行整数除法 (该除法器使用 Radix-2 算法实现, 资源消耗极低), 得到商 *quotient* 和余数 *reminder*.

(3) 计算 *pkt_time*: 商 *quotient* 代表了产生 *needed_tokens*−*reminder* 个令牌所需的完整注入周期数. 但数据包还差 *reminder* 个令牌才能被发送, 为了补足这些令牌, 系统必须多等待一个完整的注入周期 (*period*). 因此, 总的等效时间 *pkt_time*=(*quotient*+1)×*period*.

(4) 更新结余: 在第 3 步中, 为了补足令牌, 本文方法多等待了一个周期, 这个周期新注入了 *increment* 个令牌. 数据包消耗掉所需的 *reminder* 个令牌后, 剩余的令牌即新的结余为 *spare_tokens*=*increment*−*reminder*.

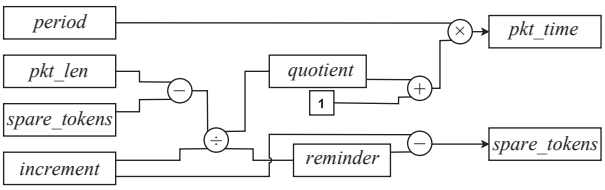

图 6 令牌桶部分硬件电路逻辑 2

即使余数 *reminder* 为 0, 同样可以按照这种方式更新 *spare_tokens*. 由于每次数据包传入时, 首先消耗 *spare_tokens*, 故在任何时刻 *spare_tokens* 的取值范围都不会大于 *increment*.

该方法将高开销的浮点/定点速率运算转换为低延迟、低资源的整数运算, 在保证纳秒级精度的同时, 极大节省了 FPGA 逻辑资源, 是实现高精度限速的关键.

3.4 配置和状态参数

结合第 3.2、3.3 节支持高扩展性和高精度计算机制的讨论, 本文使用下面的参数表保存各队列的配置参数和状态参数, 如图 7 所示. 其中, 增量、注入周期、最大积累时间、优先级为每个令牌桶的配置参数, 上次发送时间和令牌结余为每个令牌桶的状态参数.

Param / Queue id	增量 *increment*	注入周期 *period*	最大积累时间 *MAX_time*	优先级 *priority*	上次发送时间 *last_send_time*	令牌结余 *spare_tokens*
队列 0	30	1	8 192	100	30 000	3
队列 1	15	1	16 384	50	25 678	12
队列 2	15	1	16 384	30	23 456	6
队列 3	2	5	5 120	13	20 000	1

图 7 令牌桶参数表结构

对于令牌桶的配置参数而言, 假设系统工作的时钟频率为 250 MHz, 一个周期为 4 ns. 以队列 0 为例,

增量为 30, 周期为 1, 代表每 1 个时钟周期即每 4 ns 可发送 30 字节, 即对应的带宽限制为 60 Gb/s. 最大积累时间为 8 192 个时钟周期, 等效为桶内最大令牌数量为 240k, 对应桶深为 240 KB; 以队列 3 为例, 增量为 2, 周期为 5, 代表每 5 个时钟周期即每 20 ns 可以发送 2 字节, 即每 1 个时钟周期发送 0.4 字节, 对应的带宽限制值为 800 Mb/s. 最大积累时间为 5 120 个时钟周期, 对应到 1k 个令牌注入周期, 则桶内最大令牌数量为 2k, 对应桶深为 2 KB.

对于令牌桶的状态参数而言, 假设系统当前时间为第 31 000 个时钟周期. 以队列 0 为例, 上次发送时间为第 30 000 个时钟周期, 差值为 1 000 个时钟周期, 未超过 *MAX_time*, 则对应到 1 000 个注入周期, 每个注入周期注入 30 个令牌, 即该令牌桶在这段时间积累了 30 000 个令牌. 再加上结余的 3 个令牌, 该令牌桶内所剩的总令牌为 30 002 个; 以队列 3 为例, 上次发送时间为第 20 000 个时钟周期, 差值为 10 000 个时钟周期, 超过 *MAX_time*, 则对应桶内最大令牌数 2 048 个. 加上结余的 1 个令牌, 该令牌桶内所剩的总令牌为 2 049 个. 虽然此时桶内令牌总数超出了预设的桶深参数, 但由于令牌结余 *spare_tokens* 不大于 *increment*, 且 *increment* 通常与桶内最大令牌数即桶深有数量级的差距, 因此对令牌桶的突发速率造成的影响非常有限, 且不会影响到承诺速率. 对于对突发量或突发速率严格限制的场景, 可先判断令牌积累的时间 (当前时间减去上次发送时间) 是否超过了最大积累时间, 如果超过则将 *spare_tokens* 视为 0 进行后续逻辑运算.

除此之外, *priority* 可用于标识队列中数据包的优先级别, 数值越大代表优先级越高, 该字段可以支持不同队列数据流之间进行带宽抢占. 如图 7 各队列 (不考虑队列 3) 所示, 假设链路总带宽为 100 Gb/s, 分别为队列 0、队列 1 和队列 2 各分配 60 Gb/s、30 Gb/s 和 30 Gb/s. 根据图中优先级字段所示, 虽然各队列分配的带宽总和超出了链路上限, 但调度器仍会优先满足队列 0 和队列 1 的带宽需求; 当链路带宽未被完全占用时, 队列 2 可临时借用队列 0 和队列 1 未使用的带宽, 以达到 30 Gb/s 的带宽. 一旦带宽资源紧张, 队列 0 或队列 1 可以抢占回队列 2 使用的带宽. 该字段在硬件层面直接实现优先级带宽借用, 既支持多级队列的并发调度与服务质量保障, 又能提升链路利用率, 并在高负载或突发流量时保持时序确定性和高稳定性.

4 实验与评估

本节实验基于 Corundum 开源智能网卡框架进行开发, 将设计的带宽限制器集成在 Corundum 的接收路径上. 使用 Xilinx Vivado Design Suite v2022.2 Patch 3[22]进行 RTL 的仿真、综合、实现及比特流生成, 并部署在 Xilinx Alveo U200 数据中心加速器卡 (PCIe Gen3×16) 上. 其采用 UltraScale+架构的 XCU200-FSGD2104-2-E FPGA 芯片, 片上存储资源包括 2 160 块 36 KB 的 BRAM (共约 75.9 MB)、591 840 个可配置为分布式 RAM 的 6 输入 LUT (每 LUT 支持 64 bit, 合计约 36.1 MB 分布式 LUT-RAM) 及 960 块 288 KB 的 UltraRAM (合计 270 MB). 测试带宽限制精度时将板卡安装在 Linux 通用服务器上, 并使用 QSFP 光模块及光纤与信而泰 Bigtao-220 网络测试仪器相连.

4.1 参数表设置

本文实验使用的参数表各字段位宽如图 8 所示, 每个队列共需 116 bit 的存储空间.

	8 bit	8 bit	16 bit	12 bit	64 bit	8 bit
Param / Queue id	增量 *increment*	注入周期 *period*	最大积累时间 *MAX_time*	优先级 *priority*	上次发送时间 *last_send_time*	令牌结余 *spare_tokens*
队列 0	30	1	8 192	100	30 000	3
…	…	…	…	…	…	…
队列 *n*	15	1	16 384	30	23 456	6

图 8 参数表位宽

实验基于 Corundum 智能网卡框架进行开发, 其 AXIS 总线带宽为 512 bit, 即每个时钟周期最多在链路上传输 64 bit, 故增量、注入周期和令牌结余参数的位宽均设置为 8 bit 即可. 对于单个队列, 设置为每 1 个时钟周期注入 64 个令牌时, 即可达到最大链路带宽. 最大积累时间的参数决定了桶深即突发量, 本文采用 16 bit, 即最大可支持 32 MB 的突发量, 若需要增加突发量上限可按需增加位宽. 调度计算模块最多可支持 4 096 及以上个队列, 故优先级字段可设置为 12 bit. 为防止时间计数器溢出, 上次发送时间和系统当前时间字段均设置为 64 bit, 以 250 MHz 时钟频率为例, 该长度字段可支持模块运行 2 000 年以上.

为支持高扩展性, 大规模队列参数需使用 BRAM/URAM 进行存储, 可节省逻辑资源并降低布线难度. 为降低调度计算延迟, 需使用简单双端口 RAM (simple dual port RAM) 存储以满足在同一周期同时进行读取和写入的操作.

4.2 吞吐率

本文设计的带宽限制器可支持 1 024 及以上个队列进行带宽限速. 基于 Corundum 进行开发时, 所有的模块运行在 250 MHz 的时钟频率下, AXIS 总线宽度为 512 bit, 最高可达 128 Gb/s 的链路带宽.

对于调度计算模块, 由第 3 节分析可得, 最长的时间路径是 *pkt_time* 的计算逻辑, 如图 9 中红色加粗的计算路径所示.

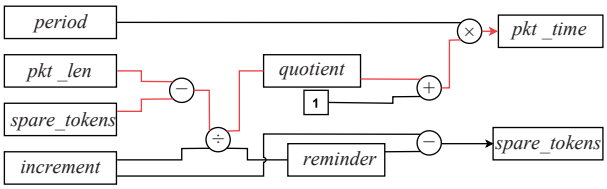

图 9 令牌桶硬件电路逻辑最长路径

在 FPGA 中进行四则运算时, 除法是最慢、最消耗资源的. 即使本文使用了无符号整数的取余除法, 在 250 MHz 时钟频率下, 除法器最少需要构建 3 级流水线, 即操作延迟为 3 个时钟周期. 而加减乘法通常使用组合逻辑即可. 故 *pkt_time* 计算延迟为 3 个时钟周期, 再加上 BRAM 读取的 1 个时钟周期延迟, 一共 4 个时钟周期的延迟.

last_send_time 和 *spare_tokens* 等状态参数的更新需写回 BRAM, 由于同一队列两次有效的调度计算至少要间隔 8 个时钟周期 (PIEO 调度器的出队操作和入队操作各需要 4 个时钟周期), 且简单双端口 RAM 的两个端口支持在同一个时钟周期对不同地址进行读取和写入操作, 故可直接在结果输出后的下个时钟周期进行状态参数的更新写回. 如图 10 所示为可扩展的高精度令牌桶整体调度时序图, 在第 5 个时钟周期, 可同时进行队列 0 的状态参数更新和队列 4 的参数读取, 而队列 0 的下次读取至少要在第 13 周期才会进行. 这是因为调度计算模块采用流水线架构, 多个队列可并行地进行调度计算, 而调度执行模块即 PIEO 调度器并非采用流水线架构, 每次出队操作和入队操作的延迟均为 4 个时钟周期.

对于单个队列调度而言, 从传入队列序号从参数表中读取队列参数信息到结果计算完成需要 4 个时钟周期, 将计算结果传入 PIEO 调度进行入队操作需要 4 个周期完成, 链路空闲时将该数据包从 PIEO 调度器中调度出队操作同样需要 4 个周期. 因此对于单个队列的最大调度吞吐率为 20.83 Mpps, 理想情况下在 MTU 大于 600 字节时即可以 100 Gb/s 线速运行. 对比当前最新的相关工作[14], 其单队列最大调度吞吐率为 10.53 Mpps, 本文的设计提高近 100%, 如图 11 所示.

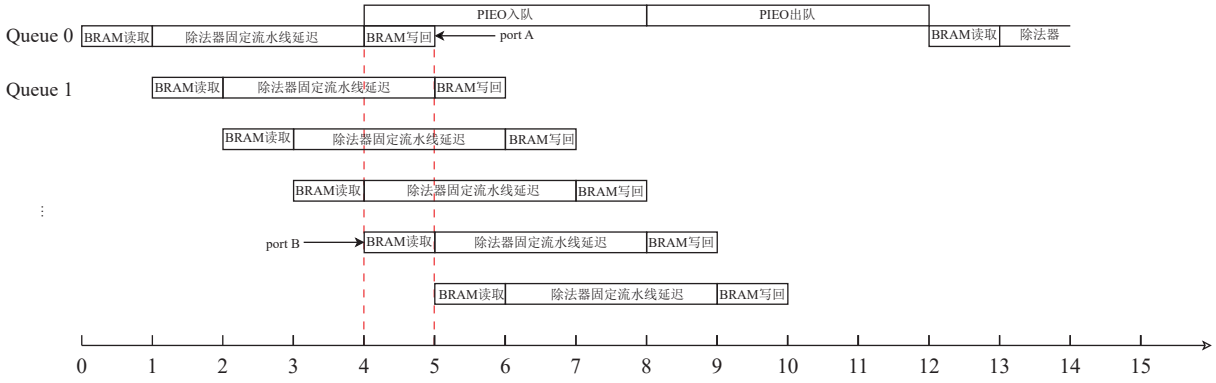

图 10 整体调度时序图

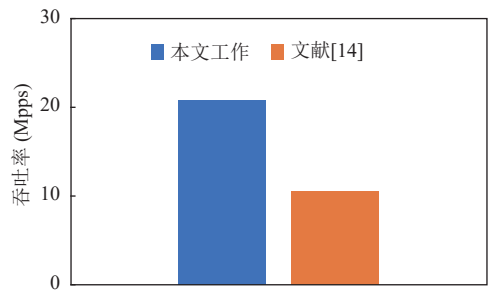

图 11 与文献[14]的单队列最大调度吞吐率对比

对于多个队列调度而言, 由于本文设计的调度计算模块是基于流水线架构设计的, 且同一队列两次有效调度的计算间隔至少为 12 个周期, 所以每个时钟周期都可以传入队列号进行计算, 从而达到 250 Mpps 的调度计算能力. 则对于多队列的调度吞吐率的瓶颈在于调度执行模块, 由于调度执行的模块即 PIEO 调度器并非基于流水线架构进行设计的, 单个数据包发送需要经过入队和出队两次调度原语操作, 共 8 个周期. 因

此, 对于多个队列的最大调度吞吐率为 31.25 Mpps, 理想情况下在 MTU 大于 400 字节时就可以 100 Gb/s 线速运行. 若采用 PIFO[15]等基于流水线架构设计的可编程调度器, 总调度吞吐率可达 250 Mpps, 但会损失调度表达能力, 无法支持复杂的调度算法.

4.3 资源消耗

调度计算模块 (包含参数表) 的资源消耗主要由参数表使用的 BRAM 和除法器使用 LUT (或 BRAM 和 DSP) 组成.

BRAM 是 FPGA 芯片内部专用的片上存储资源, 时序性能优异、功耗低、密度高, 但片上数量受制于硅面积而相对有限, 因此在大规模队列或多端口访问场景下尤显珍贵. 一块标准 36 KB BRAM (位宽为 36 bit、深度为 1 024) 可按需拆分为两个独立的 18 KB BRAM: 既可各自配置为位宽 18 bit、深度 1 024 的单口/简单双端口 RAM, 也可组合为位宽 36 bit、深度 512 的深度优化版本, 从而在位宽与深度之间灵活权衡. 设计者通常利用这一特性, 把存储块切割成若干逻辑上独立、物理上相邻的子 RAM 来承载参数表、环形缓冲区或计数器等数据结构; 与此同时, 借助 BRAM 自带的真双端口 (true dual-port) 访问模式, 可在单时钟周期内完成一读一写或两读操作, 大大简化并行调度通路的数据对齐与更新逻辑.

本文的除法器使用 Divider Generator IP 核进行例化, 其提供了 3 种主要的实现算法: Radix-2 非恢复算法 (Radix-2 non-restoring), 在逻辑资源上最为轻量, 适合 23 位及以下的总操作数 (除数和被除数总位宽); high-Radix 带预缩放算法 (high-Radix with prescaling), 利用 DSP48 器件实现高位进制的快速除法, 适合 25 位以上的大操作数, 但不支持余数除法; LUTMult 算法模式, 通过查找表和乘法器组合实现除法, 在资源和延迟间取得平衡, 可支持中等位宽的除法.

传统以太网 MTU 值为 1 500 字节, 可使用 11 bit 标识数据包长度 *pkt_len*, 即被除数宽度为 11 bit. 由第 4.1 节可知, 除数即增量参数位宽为 8 bit, 故总操作数位宽为 19 bit. 因本文使用的除法为取余除法, 故除法器的实现算法可为 Radix-2 非恢复算法和 LUTMult 算法模式. 在满足 250 MHz 时钟频率时 3 个时钟周期的输出延迟要求下, 采用 Radix-2 非恢复算法和 LUTMult 算法模式实现的资源消耗如表 1 所示. 其中, LUTMult 模式通过把乘加运算映射到 DSP 和 BRAM 上, 实现了对操作数总位宽至多 23 位的扩展, 可支撑更大的 MTU 或更细粒度的带宽整形, 但需要使用额外的 DSP 和 BRAM 资源. 相比之下, Radix-2 非恢复算法充分利用 LUT 进行商、余数的迭代更新, 只需极少量 LUT 即可满足 11 位 MTU 位宽的常规以太网场景, 且完全不依赖 DSP, 易于在不同 FPGA 型号间迁移. 本文选用 Radix-2 非恢复算法模式实现除法器, 并采用简单双端口 RAM 存储参数表以降低布线复杂度和时序压力.

表 1 除法器不同实现算法资源消耗对比

实现算法	被除数与除数位宽	LUT	FF	BRAM	DSP
LUTMult	23	—	63	0.5	2
Radix-2	19	113	71	—	—

硬件调度计算模块的资源消耗如表 2 所示, 并与文献[14]和文献[12]的工作进行了对比.

表 2 调度计算模块资源消耗对比

队列数	硬件资源	本文工作	文献[14]	文献[12]
64	LUTs	235	2 071	23 643
	FFs	293	1 293	6 475
	BRAMs	2	2.5	5
128	LUTs	235	2 077	47 591
	FFs	294	1 297	12 833
	BRAMs	2	2.5	5
256	LUTs	235	2 083	93 306
	FFs	295	1 301	24 969
	BRAMs	2	2.5	5
512	LUTs	235	2 085	186 941
	FFs	296	1 305	49 821
	BRAMs	2	2.5	5
1 024	LUTs	235	2 091	—
	FFs	297	1 309	—
	BRAMs	3.5	5	—
2 048	LUTs	236	2 102	—
	FFs	297	1 313	—
	BRAMs	6.5	10	—
4 096	LUTs	236	—	—
	FFs	298	—	—
	BRAMs	13	—	—

当支持的队列数从 64 增加到 512 时, 调度计算模块的 BRAM 资源占用始终保持 2 块. 原因在于所有队列的静态配置与动态状态均映射到同一组深度为 512 的简单双端口 RAM, 由于一个队列参数的位宽不是 18 bit 的整数倍, 在 Vivado 例化不同深度参数的 BRAM 时, 会进行 BRAM 的灵活拼接, 从而造成 BRAM 的资源消耗并不是严格随队列数的增加 (以 512 为单位) 线性增加. 随着队列规模扩大, 队列索引字段宽度增大, 调度器需暂存的队列号随之增加, 因此 FF 用量呈对数

级缓慢上升, 大约每当队列数翻倍时增加 1 位寄存器; 而 LUT 主要承担状态机控制等逻辑, 其复杂度与队列数量弱相关, 故整体保持近乎恒定.

调度计算模块在支持不同调度队列时资源消耗表明其可支持大规模队列的调度, 具有良好的可扩展性. 对比文献[12]的工作, 其逻辑资源利用率随支持的队列数量线性增长, 在队列数为 512 时消耗了 186 941 个查找表, 对于一些规模不大的 FPGA 芯片来说, 约占其查找表总量的 80%, 这会严重影响其他模块的布局布线和时序收敛. 对比文献[14]的工作, 本文设计的调度计算模块查找表消耗降低约 89%, 寄存器资源消耗降低约 77%, BRAM 资源消耗至少降低 20%. 所使用逻辑资源占 FPGA 芯片总量约 0.02%, 根据队列数量不同 BRAM 的使用量为 0.1%–0.6%.

4.4 精度

由第 4.1 节可知, 增量和注入周期参数均为 8 bit, 最小可支持每 255 个时钟周期只注入 1 个令牌, 即每 1 020 ns 注入 1 个令牌, 此时带宽限制约为 7.84 Mb/s, 占链路目标带宽 100 Gb/s 的 0.007 8%. 若增加增量和注入周期参数的位宽, 可进一步降低限制的最小带宽值并提高带宽限制的粒度.

根据增量和注入周期参数的不同, 可灵活设置各队列的限制带宽速率. 例如当增量和注入周期参数为 64 和 1 时, 限制带宽达到链路最高, 可使用全部目标带宽; 当设置为 1 和 200 时, 代表每 800 ns 注入一个令牌, 此时带宽限制为 10 Mpbs.

测试过程使用信而泰测试仪生成带宽不同的多条数据流发送至服务器, 通过接收路径的带宽限制器后进入主机中. 本文在主机内核中设置队列规则 (queueing discipline, qdis), 对于从该板卡接收到的数据包直接从另一网络接口 (interface) 转发至测试仪, 从而在测试仪侧观察到各数据流经硬件调度后实际的带宽值.

表 3 为单队列下的带宽测试结果, 单次只进行一种数据流的注入, 并确保测试仪注入的带宽值大于设置带宽. 当限制带宽较小时, 误差最大为 0.09%, 当限制带宽较大时, 误差不超过 0.000 1%. 这表明本文设计的带宽限制器可支持高精度的带宽限制.

表 4 为多队列下的带宽测试结果, 同时进行 4 种数据流的注入, 并确保测试仪注入的每种数据流带宽值都大于其设置带宽. 相比于单队列带宽限制, 整体误差变化不大, 这表明本文设计的带宽限制器可支持多种数据流下的高精度带宽限制.

表 3 单队列下不同带宽限制的测量带宽值及误差比例

限制带宽	测量带宽	误差 (%)
10 Mb/s	10.009 3 Mb/s	+0.093
100 Mb/s	100.003 2 Mb/s	+0.032
500 Mb/s	499.994 5 Mb/s	−0.001 1
1 Gb/s	1.000 09 Gb/s	+0.000 9
10 Gb/s	9.999 95 Gb/s	−0.000 5
20 Gb/s	20.000 03 Gb/s	+0.000 03
50 Gb/s	49.999 98 Gb/s	−0.000 02
80 Gb/s	79.999 99 Gb/s	−0.000 01

表 4 多队列下不同带宽限制的测量带宽值及误差比例

限制带宽	测量带宽	误差 (%)
10 Mb/s	10.010 3 Mb/s	+0.103
500 Mb/s	499.992 5 Mb/s	−0.007 5
10 Gb/s	10.000 02 Gb/s	+0.000 2
50 Gb/s	49.999 95 Gb/s	−0.000 05

5 结论与展望

本文围绕基于 FPGA 的高精度带宽限制技术展开, 系统介绍了从整体架构设计到关键模块实现与优化的全过程. 首先, 给出了一种可部署于接收和发送数据路径中的模块化限速框架, 能够与现有 Corundum 架构兼容集成; 其次, 借助 PIEO 原语实现了支持多级抢占与高效资格判断的可编程调度机制; 在速率控制方面, 通过整数化的令牌桶实现, 采用“周期+增量”表示方式与 spare_tokens 补偿机制, 显著降低资源开销同时提升控制精度; 此外, 本文还从时序优化与资源扩展角度出发, 设计了高并发低延迟的调度计算模块, 并支持 4k 以上的队列扩展; 最后, 通过在 Xilinx Alveo U200 FPGA 平台上的实验验证, 证实了本方案在资源利用率、精度、带宽稳定性及能效等方面均优于现有方案.

参考文献

1 He DZ, Zhou WL, Zhang X. A bi-direction adjustable token bucket mechanism for multi-class bandwidth guarantee and sharing. Proceedings of the 2009 IEEE International Conference on Network Infrastructure and Digital Content. Beijing: IEEE, 2009. 65–68.

2 Linux Man-pages Project. qdisc(8): Queueing disciplines manual. https://man7.org/linux/man-pages/man8/tc.8.html. (2024-04-15)[2025-08-10].

3 Kalia A, Kaminsky M, Andersen DG. Design guidelines for

high performance RDMA systems. Proceedings of the 2016 USENIX Annual Technical Conference. Denver: USENIX, 2016. 437–450.

4 李博杰. 基于可编程网卡的高性能数据中心系统 [博士学位论文]. 合肥: 中国科学技术大学, 2019.

5 Cornevaux-Juignet F. Hardware and software co-design toward flexible terabits per second traffic processing [Ph.D. thesis]. Nantes: Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire, 2018.

6 Qiu YM, Xing JR, Hsu KF, *et al*. Automated smartnic offloading insights for network functions. Proceedings of the 28th ACM SIGOPS Symposium on Operating Systems Principles. New York: ACM, 2021. 772–787.

7 Firestone D, Putnam A, Mundkur S, *et al*. Azure accelerated networking: SmartNICs in the public cloud. Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation. Renton: USENIX, 2018. 51–66.

8 Park SK, Oh JT, Jang JS. High-speed attack mitigation engine by packet filtering and rate-limiting using FPGA. Proceedings of the 8th International Conference Advanced Communication Technology. Phoenix Park: IEEE, 2006. 6–10.

9 Chen W, Jin D, Zeng L. Synchronous fine-adjustable rate control circuit for Ethernet congestion management. Electronics Letters, 2008, 44(4): 325–326. [doi: 10.1049/el:20083192]

10 Zhang YL, Hu YT, Li T, *et al*. Design and implementation of two rate three color marker based on FPGA. Proceedings of the 2013 International Conference on Computer, Networks and Communication Engineering. Paris: Atlantis Press, 2013. 642–645.

11 Bianchi G, Bonola M, Bruschi V, *et al*. Implementing a per-flow token bucket using open packet processor. Proceedings of the 28th International Tyrrhenian Workshop on Digital Communications. Palermo: Springer, 2017. 251–262.

12 Benacer I, Boyer FR, Savaria Y. A high-speed traffic manager architecture for flow-based networking. Proceedings of the 15th IEEE International New Circuits and Systems Conference. Strasbourg: IEEE, 2017. 161–164.

13 Antichi G, Shahbaz M, Geng YL, *et al*. OSNT: Open source network tester. IEEE Network, 2014, 28(5): 6–12. [doi: 10.1109/MNET.2014.6915433]

14 Guo YF, Guo ZC, Zhang MT. A multi-tenant rate limiter on FPGA. Electronics, 2025, 14(6): 1155. [doi: 10.3390/electronics14061155]

15 Sivaraman A, Subramanian S, Alizadeh M, *et al*. Programmable packet scheduling at line rate. Proceedings of the 2016 ACM SIGCOMM Conference. Florianopolis: ACM, 2016. 44–57.

16 Shrivastav V. Fast, scalable, and programmable packet scheduler in hardware. Proceedings of the 2019 ACM Special Interest Group on Data Communication. Beijing: ACM, 2019. 367–379.

17 IEEE. IEEE standard for local and metropolitan area networks—Virtual bridged local area networks amendment 12: Forwarding and queuing enhancements for time-sensitive streams (IEEE Std 802.1Qav-2009). https://standards.ieee.org/ieee/802.1Qav/4401/. (2010-01-05)[2025-08-10].

18 IEEE. IEEE standard for local and metropolitan area networks—Bridges and bridged networks amendment 34: Asynchronous traffic shaping (IEEE Std 802.1Qcr-2020). https://1.ieee802.org/tsn/802-1qcr/. (2020-11-06)[2025-08-10].

19 Alon Regev. Asynchronous traffic shaper (802.1Qcr) and its applicability to automotive use-cases (presentation). https://standards.ieee.org/wp-content/uploads/2023/10/4-asynchronous-traffic-shaper.pdf. (2023-09)[2025-08-10].

20 IEEE. IEEE standard for local and metropolitan area networks—Bridges and bridged networks amendment 29: Cyclic queuing and forwarding (IEEE Std 802.1Qch-2017). https://standards.ieee.org/ieee/802.1Qch/6072/. (2017-06-28)[2025-08-10].

21 Radhakrishnan S, Geng YL, Jeyakumar V, *et al*. SENIC: Scalable NIC for end-host rate limiting. Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation. Seattle: USENIX Association, 2014. 475–488.

22 Xilinx. Vivado design suite: Design tools for FPGA and SoC development. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.(2025-02)[2025-08-10].

(校对责编: 李慧鑫)