计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(1):61−74 [doi: 10.15888/j.cnki.csa.008849]

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

大规模软硬协同哈希表设计与实现①

杨文韬[1,2], 张士军[1,2], 张　进[1,2], 唐　寅[2], 于洪涛[3]

[1](东南大学 网络空间安全学院, 南京 211100)
[2](紫金山实验室 内生安全研究中心, 南京 211100)
[3](战略支援部队信息工程大学, 郑州 450001)
通信作者: 张　进, E-mail: zhangjin@pmlabs.com.cn

摘　要: 哈希表在网络报文处理, 尤其是带状态的报文处理中发挥着重要作用. 伴随着网络流量的快速增长, 传统软件哈希表难以满足网络性能需求, 而查找是影响哈希表性能的关键之一, 如何提升哈希表的查找速率也一直是一个难点问题. 经研究表明, 现有的网络流量呈现 Pareto 分布特征, 即存在少数的大流量数据——大象流. 基于当前数据中心广泛采用的软硬协同计算模式, 提出了一种基于 DPDK+FPGA 的大规模软硬协同哈希表架构. 根据现有网络流量特征, 将流量分成大象流与背景流. 同时也将哈希表分成硬件表与软件表. 在 FPGA 中构造小规模硬件表, 卸载所有报文的哈希计算, 以及大象流的哈希查找. 在软件中基于 DPDK 构建大规模软件表, 利用 FPGA 卸载哈希计算, 加速背景流的查找. 软件拥有所有流信息, 利用采样法识别大象流并将大象流的键值对信息 (key-value) 更新到 FPGA 的硬件表中, 以加速软件中大规模软件表的查找速率. 采用 Xilinx U200 加速卡和通用服务器作为硬件平台, 实现了软硬协同的大规模哈希表, 并利用测试仪构造了符合当前网络特征的流量数据, 以 DPDK 精确转发为例, 验证了软硬协同哈希表的性能. 结果表明, 在大象流哈希查找完全卸载的情况下, 其性能相较 DPDK 原有的精确转发提升了 64%–75%; 在大象流未卸载的情况下, 其性能提升了 5%–48%.

关键词: 软硬协同; 哈希表; 大象流; DPDK; FPGA

引用格式: 杨文韬,张士军,张进,唐寅,于洪涛.大规模软硬协同哈希表设计与实现.计算机系统应用,2023,32(1):61–74. http://www.c-s-a.org.cn/1003-3254/8849.html

Hardware-software Co-design and Implementation for Large Scale Hash Tables

YANG Wen-Tao[1,2], ZHANG Shi-Jun[1,2], ZHANG Jin[1,2], TANG Yin[2], YU Hong-Tao[3]

[1](School of Cyber Science and Engineering, Southeast University, Nanjing 211100, China)
[2](Endogenous Security Research Center, Purple Mountain Laboratories, Nanjing 211100, China)
[3](PLA Strategic Support Force Information Engineering University, Zhengzhou 450001, China)

Abstract: Hash tables play an important role in network message processing, especially in the processing of messages with states. With the rapid growth of network traffic, the hash tables of traditional software can hardly meet the needs of network performance, and search is one of the key factors affecting the performance of hash tables. In addition, the improvement in the search rate of hash tables has always been a difficult problem. The research reveals that the existing network traffic presents the characteristics of Pareto distribution, namely that there is a small number of massive traffic data—elephant flow. On the basis of the computing mode of software-hardware co-design used in the current data center, a large-scale hash table architecture with software-hardware co-design is proposed on the basis of DPDK+FPGA. According to the characteristics of existing network traffic, this method divides the traffic into elephant flow and background flow, and meanwhile, the hash table is divided into a hardware table and a software table. A small-scale

① 基金项目: 紫金山实验室自立课题; 国家自然科学基金面上项目 (62176264)
收稿时间: 2022-02-21; 修改时间: 2022-03-23, 2022-04-26; 采用时间: 2022-05-28; csa 在线出版时间: 2022-10-28
CNKI 网络首发时间: 2022-11-15

hardware table is constructed in FPGA to unload the hash calculation of all messages and the hash search of elephant flow. In the software, a large-scale software table is constructed on the basis of DPDK, and the hash calculation is unloaded by FPGA to speed up the search of background flow. As the software has all the flow information, the sampling method is used to identify the elephant flow and update the key-value pair of the elephant flow to the hardware table of FPGA, so as to accelerate the search rate of the large-scale software table in the software. The Xilinx U200 accelerator card and general server are employed as the hardware platform to realize the large-scale hash table with software-hardware co-design, and the traffic data in line with the current network characteristics is constructed by the tester. The accurate forwarding of DPDK is used as an example to verify the performance of the hash table with hardware-software co-design. The results reveal that when the hash search of elephant flow is completely unloaded, its performance is 64%–75% higher than the original accurate forwarding of DPDK; when the elephant flow is not unloaded, its performance is improved by 5%–48%.

Key words: hardware-software co-design; hash table; elephant flow; DPDK; FPGA

哈希表, 又称散列表, 是根据关键字 (key) 直接进行访问的数据结构, 一般来说, 一个关键字 (key) 对应一个值 (value), 两者共同组成哈希表的基本单位——键值对 (key-value), 存储键值对的数组就叫做哈希表. 通过将关键字 (key) 映射到表中一个位置, 可以直接访问记录, 以提高查找的速率, 相比较其他的查找结构, 哈希表查找的时间复杂度更低. 其中用于映射的函数称为哈希函数, 哈希函数有多种, 常见的哈希函数包括 CRC32, MD5, SHA 等. 由于哈希表的特殊性质, 其在安全加密, 数据校验, 唯一标识, 负载均衡等场景都有着不可替代的作用. 同时, 哈希表也是网络报文处理的关键模块之一, 许多虚拟网络函数 (VNF) 中数据流的识别基于哈希表, 网络三层转发, ACL 等功能也普遍与哈希表有关. 可以说, 哈希表的性能与网络报文处理的速率直接相关.

哈希表依据其实现方式, 分为软件哈希表与硬件哈希表. 软件哈希表相比较硬件哈希表, 成本低, 设计简单, 空间大, 使用更为普遍. 当前, 一方面, 网络流量持续飞速增长, 另一方面, 摩尔定律失效, 通用 CPU 的处理能力增速放缓. 因此, 单纯基于通用 CPU, 采用软件哈希表已无法满足性能扩展需求. 软硬协同计算是获得持续算力提升的有效手段. 当前, 全球各大顶级数据中心, 如亚马逊、谷歌、微软、百度等, 都大量部署了智能网卡、FPGA 加速卡、面向深度学习等特定领域的 ASIC 芯片等, 通过软硬协同计算, 获得明显高于纯软件方式的算力提升, 同时获得更高的系统效费比. 因此, 软硬协同设计是解决当前软件哈希表性能不足问题的有效方法.

哈希表分为软件哈希表与硬件哈希表. 由于哈希函数散列并不是完全均匀, 当两个不同的键 (key) 被散列到同一个位置时会产生冲突, 这种冲突称为哈希冲突. 软件哈希表解决哈希冲突的方法主要包括线性散列表法[1], 集合关联哈希表法[2], 多函数表法[3], 自适应表法[4], cuckoo 哈希法[5], 以及 hopscotch 哈希法[6]. 线性散列表法对冲突的键 (key) 线性搜寻周围空位置插入, 多函数表法利用多个不同的哈希函数对应多个不同的哈希表, 集合关联表法允许同一位置容纳多个键 (key), 自适应表法根据插入元素的个数不断扩展自身哈希表的大小. Cuckoo 哈希对冲突的表项进行踢出, 判断另一个哈希函数对应的位置是否空, 为空插入, 不为空则继续踢出, 直到满足结束条件或者找到空位置. Hopscotch 哈希法只采用一个哈希函数, 踢出邻居位置. 硬件表处理冲突一般采用的方法是多级散列表[7] 与分段哈希表[8], 多级散列表构建多个哈希表, 对应多个不同的哈希函数, 哈希表大小递减, 前一表冲突的键值对根据不同哈希函数重新计算下一个表的对应位置. 分段哈希表与多级散列表类似, 都构建多个哈希表, 但每个哈希表的大小相同并且只采用一个哈希函数, 对每一个键 (key), 其备选位置的个数与哈希表个数相同. 硬件相比较软件, 复杂的哈希冲突避免设计比如 cuckoo 等难以实现, 通过简化 cuckoo 算法可以一定程度上降低硬件设计的复杂度, 比如 Kirsch 等人[9] 提出硬件表中只进行一次移动可以大大提高空间利用率.

硬件表中键值对的存储方式分成解耦与非解耦,

解耦将键 (key) 与值 (value) 存储在不同的硬件存储空间中, 一般键 (key) 置于高速存储中, 而将值置于大空间的低速缓存中[10], 非解耦方式则相反[11]. 对硬件而言, 其拥有多种不同缓存, 比如 FPGA 中常见的缓存包括片上 BRAM 与片外 DDR SRAM, 片上 BRAM 相较片外 DDR SRAM, 速率快但空间不足. 当 value 大小过大时一般采用解耦方式存储, 避免哈希表空间受到极大限制. 对解耦存储方式而言, 其一般在高速缓存比如 BRAM 中存储 value 的索引与 key, 再根据 value 索引读取低速缓存 DDR SRAM 中的 value 值, 其每次查找与插入时需要经历前后两次顺序的缓存访问, 无法并行化读取. 对非解耦方式而言, key 与 value 存储的在同一片高速片上缓存 BRAM 中, 根据哈希值并行访问同一片缓存内的 key 表与 value 表, 避免前后顺序化的内存访问, 有利于提高查找与插入速率. 在本文提出的软硬协同哈希表中, 考虑最大化硬件哈希表性能与缓存, 本文选取非解耦方式作为键值对的存储方式, 将 key 与 value 存储在高速 BRAM 中.

相较软件 CPU, 硬件专用集成电路并行化程度高, 对功能加以区分, 考虑软硬件的优势, 将适合硬件处理的功能, 比如排序, 计算等卸载到硬件中, 软硬件构成流水线, 硬件负责数据处理, 为软件提供数据支撑. 比如文献 [12] 中利用硬件 SoC, 将示例排序中的分区与排序功能卸载到 FPGA 中, 极大地提高了大型数据中心对数据集排序的速率. 文献 [13] 针对加密中 SHA 哈希计算需求, 构建相应硬件计算架构, 由硬件完成哈希计算, 加速加解密流程. Panait 等人[14] 为加速比特币计算, 基于硬件 SoC 构建数据处理模型处理比特币计算核心——SHA256 的哈希计算并利用 DMA 进行软硬间的数据传输. Fairouz 等人[15] 为了解决链式哈希表查找时间复杂度为 O(m) 的问题, 在 FPGA 中构建硬件哈希表 HT, 接收软件下发的键 (key), 对 HT 并行查找, 成功将时间复杂度降低为 O(1). 由此可见, 软硬协同是提高软件算力的有效方法.

目前关于 DPDK[16] 的软硬协同工作, 主要包括两种, 第 1 种是利用 DPDK 的中的部分功能, 提高上层软件的运行效率, 充分发挥硬件设计的性能. 比如在文献 [17] 中利用 DPDK 的内存池存储数据结构, 提高 NDN 网络的运行速率. 比如文献 [18] 基于 DPDK 实现网络损失仿真器. 第 2 种是将 DPDK 的功能卸载到可编程硬件比如 FPGA 中, 由硬件实现 DPDK[19] 的完整功能, 软件为硬件提供服务, Intel 在 DPDK 峰会上提出[20] 在 FPGA 中布置多个可部分重构的加速单元 (accelerated function unit, AFU), 每个 AFU 单元都可以被软件扫描, 软件经由 DPDK 控制 FPGA, 加载相应功能的 AFU 以及驱动, 满足快速实现的硬件配置需求并且允许多个不同的用户使用不同的 AFU 进行加速, 以提高网络速率的设想. 目前尚没有针对 DPDK 内部功能模块的加速, 在本文的方案中, 软件作为主导, 硬件作为辅助加速 DPDK 内部的哈希表模块.

本文针对上述场景, 以服务器+FPGA 加速卡作为通用硬件平台, 以 Linux+DPDK 作为开源软件平台, 创新性地给出了大规模软硬协同哈希表的设计与实现方案. 已有的研究工作表明, 网络流量服从 Pareto 分布, 例如, 20% 的数据流占据了网络总流量的 80% [20]. 这些少数的大流量数据流也称为“大象流” (elephant flow). 软硬协同哈希表充分利用了网络流量分布不均的特点, 将大象流的哈希查找任务卸载到硬件中, 从而实现在消耗少量硬件资源的前提下, 即可对 DPDK 哈希表的加速. 基于实验比较基于该哈希表的精确匹配 (exact match) 的网络三层转发方法与 DPDK 现有的基于自身哈希表的精确匹配 (exact match) 的网络三层转发方法的转发速率. 实验结果表明, 当大象流全部卸载时, 本文的方法能对 DPDK 三层转发中最高效模式提高 75% 的转发效率.

1 软硬协同哈希表的总体结构

本文提出的软硬协同哈希表架构如图 1 所示. 从整体架构看, 其主要分成软件模块与硬件模块.

硬件由硬件报文处理模块, 硬件表模块, QDMA 模块以及接口模块构成, 硬件报文处理模块主要负责从外界以太网帧中提取数据报文并为数据报文附加硬件表查找信息. 硬件表模块为支持查找和插入的硬件哈希表, 接口模块负责构建 FPGA 与外网以及软件的物理通路. 软件由 RX TX 模块及软件处理模块构成. RX TX 模块与 QDMA 模块相互对应, 构建 FPGA 与软件交互的逻辑通道. 软件报文处理模块主要负责识别大象流, 并将大象流表项下发硬件, 同时查找硬件未能成功卸载的流.

从数据流角度出发, 本方案主要包括两种数据流, 分别为控制报文与数据报文. 数据报文是数据流主体, 来源为外界的 TCP 与 UDP 报文, 对接收到的数据报文, FPGA 先根据硬件表查找, 查找失败的数据报文由

软件根据软件表查找, 所有的数据报文均由软件根据查找结果处理后转回硬件, 硬件最后将数据报文发送外界. 控制报文是软件与硬件交互, 传递消息的报文. 按照功能进行划分, 其主要分成两种, 分别为硬件表更新报文以及硬件表读取报文. 硬件表更新报文由硬件镜像表生成, 软件报文处理模块负责发送, 硬件表负责接收, 由硬件表的插入与删除消息构成. 数据报文与硬件表更新报文都经由 AXI_Stream 总线进行传输. 硬件表读取报文在硬件表更新报文下发完成后由软件生成, 负责对硬件表更新报文的结果进行检查, 比如当硬件表更新报文由于 PCIE 通道拥堵丢失时, 硬件表未能正确插入表项, 此时硬件表读取报文读取到当前表项与硬件表镜像中的表项不符时, 会记录当前表项, 并触发针对该位置的硬件表更新报文重新下发.

从哈希表角度出发, 本方案主要包括 3 种哈希表, 分别为硬件表, 软件表以及硬件镜像表, 软件表基于 DPDK 构造, 其作为总表, 拥有全部键值对信息, 负责查找硬件未匹配的数据报文, 为硬件镜像表提供表中键 (key) 对应的值 (value). 硬件镜像表为硬件表的镜像并控制硬件表, 其负责存储检测完成的大象流键 (key), 从软件表获取的对应值 (value) 以及大象流键对应的速率, 由大象流模块定期触发更新. 硬件表则负责对数据报文中的大象流进行查找, 根据自身表项卸载大象流的查找需求, 其表内容受到软件中的硬件镜像表控制.

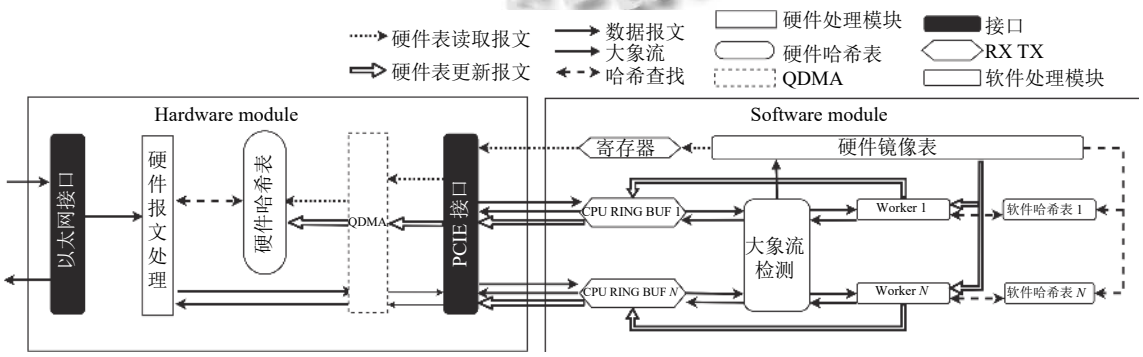

图 1 软硬协同哈希表的总体结构

从数据流的方向出发, 本文将数据流分成两个不同的方向, 从软件向硬件传输的方向本文称为下发方向, 反之则称为上行方向.

从整体工作流程出发, 本方案主要包括两个流程, 两个流程独立运行, 但相互间存在一定的联系. 第 1 个流程为大象流检测与下发的流程, 其目的为从数据报文中检测大象流, 再将大象流信息下发到硬件哈希表中. 第 2 个流程为数据报文软硬环路流程, 该流程为本文软硬协同哈希表的主体流程, 负责对数据报文查找并根据查找结果对数据报文进行对应动作, 比如三层转发的动作为目的 mac 替换, 再将数据报文重新发送回网络环境. 第 1 个流程为第 2 个流程更新硬件哈希表, 第 2 个流程则为第 1 个流程提供数据报文. 以三层转发为例, 在基于哈希表的三层转发中, 键值对中存储的 key 为五元组, 其 value 为目的 mac 与转发端口 port. 在第 1 个流程中, 大象流检测模块基于采样法从数据报文检测大象流信息, 检测完成后将提取出的流标识 (key) 发送到硬件镜像表模块中, 硬件镜像表基于 key 对软件哈希表查找, 将查找结果即目的 mac 与 port 更新到自身镜像表中, 利用硬件表更新报文沿下发方向依次经过 RX TX 模块, PCIE 接口与 QDMA, 最终将 key 和 mac 与 port 组成的键值对信息更新到硬件哈希表中. 在第 2 个流程中, 硬件报文处理模块首先从外界报文中提取出 TCP 与 UDP 报文作为数据报文, 利用硬件哈希表查找, 如果该数据报文属于大象流, 则在硬件哈希表中查找成功, 将查找结果即目的 mac 与 port 与查找成功的标志附着在数据报文中, 如果该数据报文不属于大象流, 则在硬件哈希表中查找失败, 其目的 mac 与 port 全置 0, 将其与查找失败的标志附着在数据报文上. 所有数据报文经 QDMA, PCIE 接口与 RX TX 模块到达 worker 线程. Worker 线程对数据报文加以区分, 对查找失败的数据报文, 利用软件哈希表加以查找目的 mac 与 port. 至此, 所有的数据报文均完成查找. 根据目的 mac 重新封装数据报文, 并从对应 port 转发至硬件中, 硬件最终将数据报文返还外网环境.

本文提出的架构实物图如图 2 所示. 本文将分别从硬件 (第 2 节) 与软件 (第 3 节) 两个角度介绍本方案各个模块的具体实现方法.

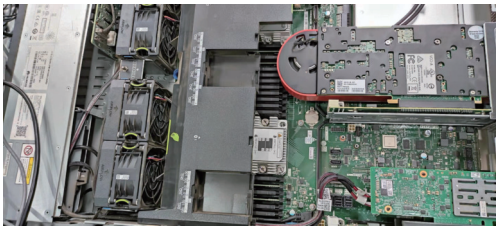

图 2 软硬协同哈希表架构实物图

2 硬件模块设计与实现

本文提出的硬件模块主要架构如图 3 所示, 其核心部分 vivado 电路设计图如图 4 所示, 根据功能的不同, 本文将硬件分成了 3 个部分, 分别为硬件报文处理模块 (第 2.1 节), 哈希表模块 (第 2.2 节), QDMA 模块 (第 2.3 节) 以及接口模块 (第 2.4 节), 下面本文将分别对不同部分进行详细介绍.

2.1 硬件报文处理模块

硬件报文处理模块是硬件功能的主要实现模块, 其主要目的分成两个, 一是对接收的外部以太网帧过滤出数据报文, 并为数据报文附加硬件表查找信息, 第 2 个目的是接收软件下发的数据报文并将其与被过滤的外部以太网帧一同发送外网. 其主要由 packet classifier 模块, packet buffer 模块, packet encapsulation 模块与 scheduler 模块构成.

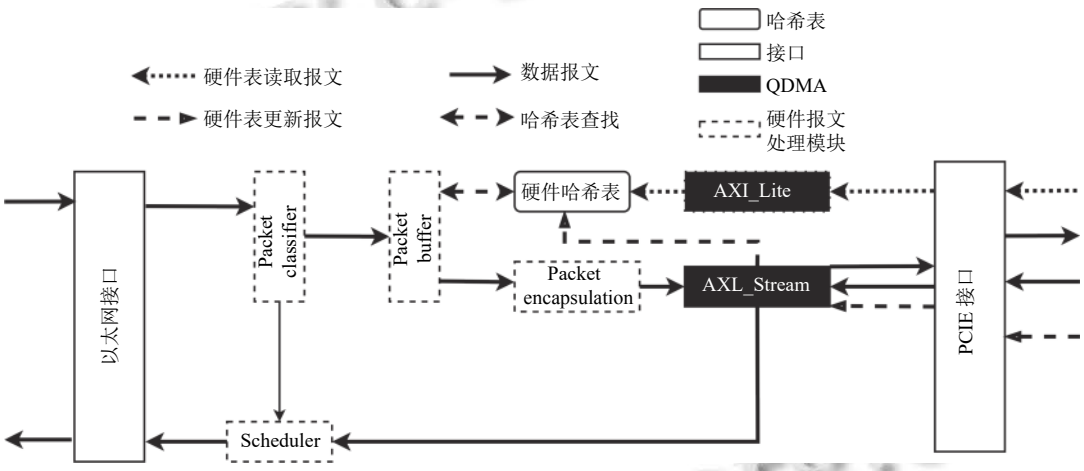

图 3 硬件总体架构图

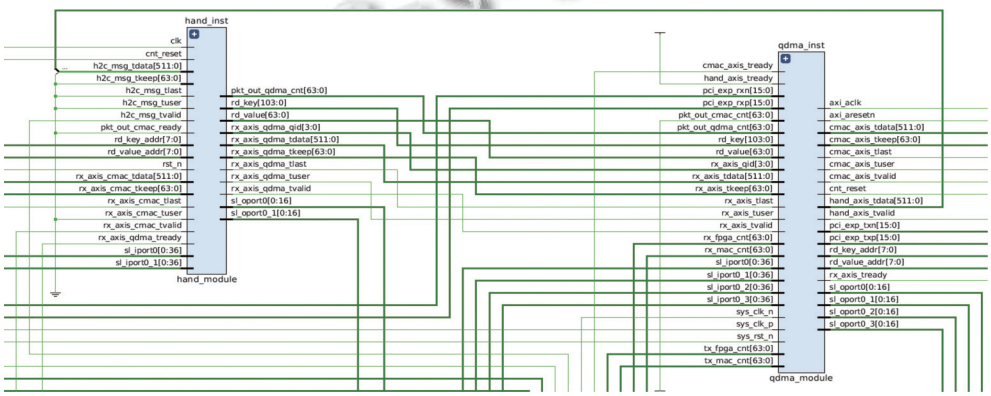

图 4 硬件核心 vivado 设计图

Packet classifier 模块的目的在于对外部数据帧先进行过滤, 剔除非 TCP 与 UDP 报文. 根据文献 [21,22], 本文构建了其结构, 将网络帧中的 TCP 与 UDP 报文暂存 packet buffer 模块, 其余报文发送 scheduler 模块.

Packet buffer 模块目的是暂时存储 TCP 与 UDP 报文, 等待硬件表查找的值 (value) 以及计算得到的哈希值, 将哈希值, 查找结果以及 TCP 与 UDP 报文一一对应组合发送包封装模块.

Packet encapsulation 模块的目的在于给数据报文附加查找结果于哈希值信息, 根据 packet buffer 模块传递的组合, 对二层网络帧封装内部数据报文头, 哈希值与查找结果, 未匹配的报文仅封装内部数据报文头与哈希值, 将封装完成后的报文一一发送给 QDMA 模块的 user_logic.

Scheduler 模块依次轮询下发方向的数据报文队列与在 packet classifier 模块中被过滤掉的报文队列, 将报文发送以太网接口.

2.2 哈希表模块

哈希表模块的功能主要分成两部分, 一部分为满足 packet buffer 模块的哈希查找需求, 另一部分为满足下发方向的哈希表插入和读取需求. 哈希表模块主要由哈希计算以及表两部分构成. 哈希计算采用流水线设计, 构造 4 级流水线, 根据键 key 计算得到哈希表中的位置, 其哈希计算的 vivado 硬件电路图如图 5 所示.

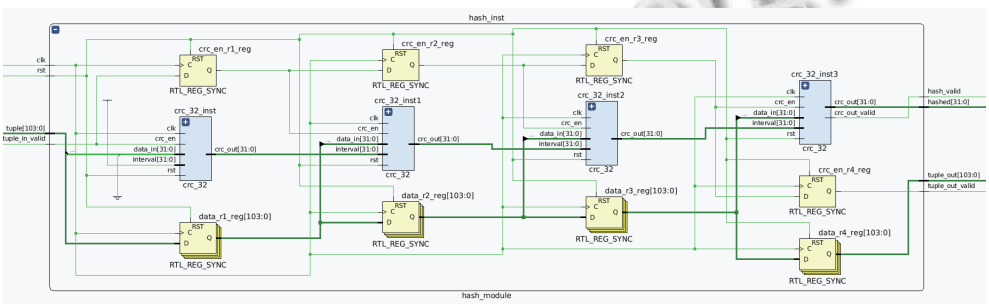

图 5 哈希计算 vivado 设计图

为方便软件根据报文结构一次性提取键 (key), 提高软件提取键的速率 (key), 本文以五元补充数组作为键 (key). 五元补充数组是五元数组的补充, 其具体形式如图 6 所示, 相比较传统的五元组, 其增加了两个全 0 的 pad 部分.

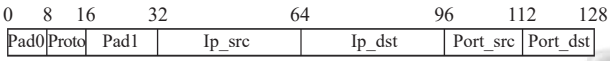

图 6 五元补充数组

针对 packet buffer 模块的哈希查找需求, 硬件哈希表会返回值 (value) 以及计算得到的哈希值. 对下发方向的哈希表插入需求, 硬件哈希表会进行键值对 (key-value) 置换, 当键值对全为 0 时, 硬件哈希表将键值对置 0, 这视为对硬件哈希表的一次删除. 对下发方向的哈希表查找需求, 硬件哈希表仅返回键 (key), 读下发方向的哈希表查找返回 key.

2.3 QDMA 模块

QDMA 模块是硬件与软件 CPU 交互的关键, 其主要目的为构建硬件与软件大规模流量传输的桥梁, 其构造如图 7 所示. QDMA 引擎由 3 个部分构成, 左侧为 user_logic, 中间为总线, 右侧为硬件队列.

User_logic 负责硬件其余部分与 QDMA 模块的数据传输, 其对接收的数据区分, 从上行方向来说, 其只接收硬件报文处理模块的数据报文并交予 AXI_Stream 总线, 从下发方向来说, 其负责区分 3 种不同类型的报文, 并将其交给硬件的不同模块.

QDMA 模块的总线主要包括 AXI_MemoryMap (MM) 总线, AXI_Stream 总线以及 AXI_Lite 总线. MM 总线以及 stream 总线适用于大规模数据传输, 两者共用相同的硬件队列, MM 总线相比较 stream 总线, 在传输之前需要指定传输的地址空间, 适用于批量传输文件, 而 stream 总线则不需要. 根据本设计方案需求, 硬件与软件 CPU 之间是持续传输的数据流, 因此挂空 MM 总线, 启用 stream 总线. AXI_Lite 总线相较其他总线, 传输数据的单元大小受限, 传输速率低. 由于其和另外两个总线使用不同的队列, 因此在未达到 PCIE 通道传输上限的条件下, 可以作为其余总线的补充, 传输部分短数据, 降低其余总线传输数据的种类, 充分利用 PCIE 通道的传输性能. 在本方案中, 本文采用 AXI_Stream 总线作为数据主体 (数据报文与硬件表更新报文) 通道, 持续高速发送与接收数据流, AXI_Lite 总线低频传输硬件表读取报文, 在不影响功能的情况下保

证总体传输的高性能.

2.4 接口模块

本文将接口分成以太网接口与 PCIE 接口, 这两个接口都位于 FPGA 中, 其利用特定协议, 完成 FPGA 内部数据与其他设备之间的数据转换, 具体来说, 以太网接口负责 FPGA 数据链路层的以太网帧与外部物理层数据流的转化, PCIE 接口负责将 FPGA 中的数据交换到软件中.

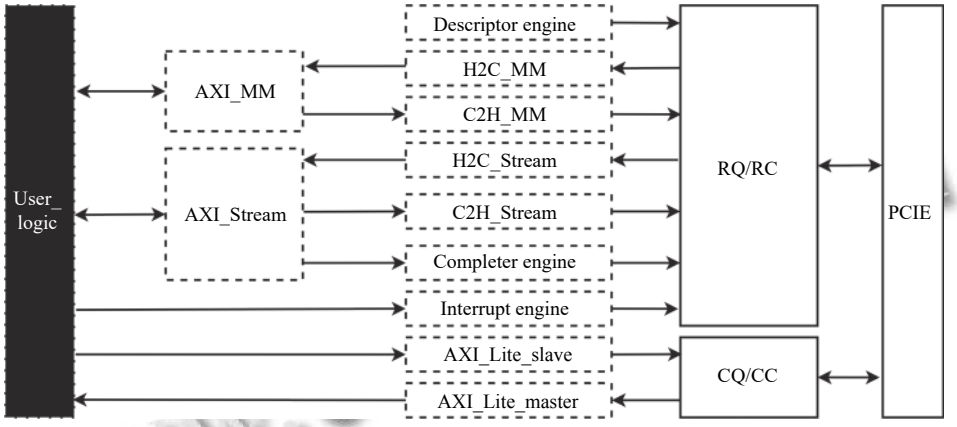

图 7 QDMA 架构图

3 软件模块设计与实现

软件模块的功能主要分成 3 点: 第 1 点为识别大象流, 根据大象流与软件表创建硬件镜像表; 第 2 点根据数据报文附加的查找信息, 对未能卸载的数据报文由软件进行查找; 第 3 点为构建良好的软硬件通信机制, 保证硬件表更新报文, 数据报文, 硬件表读取报文的良好传输. 软件模块的主要结构如图 8 所示, 本文将其分成了两部分, 分别为软件报文处理模块 (第 3.1 节) 以及 RX TX 模块 (第 3.2 节) 两个部分, 最后本文介绍使用的软硬通信协议结构 (第 3.3 节).

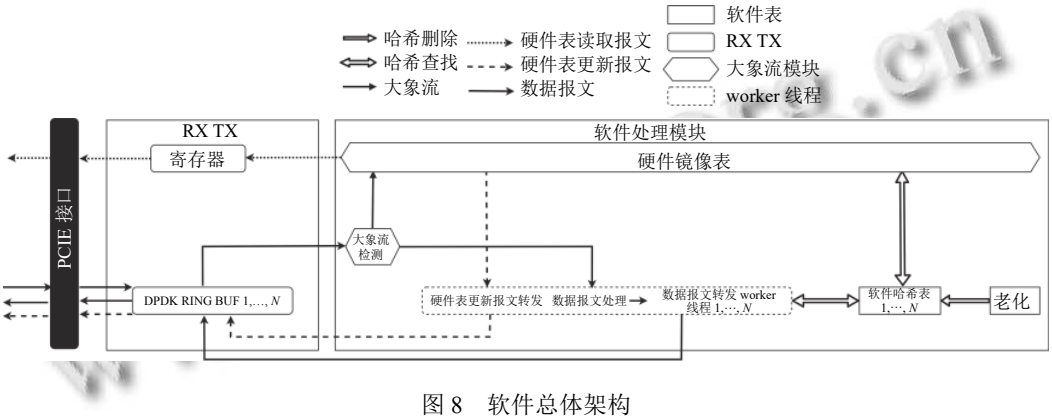

图 8 软件总体架构

3.1 软件报文处理模块

本文将软件处理模块将其分成了 3 个模块, 分别为数据报文处理与报文下发模块 (第 3.1.1 节), 大象流模块 (第 3.1.2 节) 以及软件哈希表模块 (第 3.1.3 节), 数据报文处理与报文下发模块是软件功能的执行体, 负责对硬件上送的数据报文进行处理以及发送数据报文与硬件表更新报文, 大象流模块负责根据数据报文识别大象流, 并实时更新硬件镜像表, 软件表模块负责满足其余模块的查找需求.

3.1.1 数据报文处理与报文下发模块

数据报文处理与报文下发模块由一个或多个相同的 worker 线程构成, 每个 worker 线程的功能都相同.

每一个 worker 线程循环完成 3 步工作, 依次为硬件表更新报文下发, 数据报文处理及数据报文下发. 其具体结构与逻辑如图 9 所示. 从数据流角度来说, worker 线程负责数据报文与硬件表更新报文, 利用处理顺序

为硬件表更新报文赋予高优先级, 保证硬件表更新报文传输的稳定性. 报文的收取与发送均采用批处理形式, 以 64 个报文为一组, 不足 64 个报文也视为一组, 以组为单位进行收发.

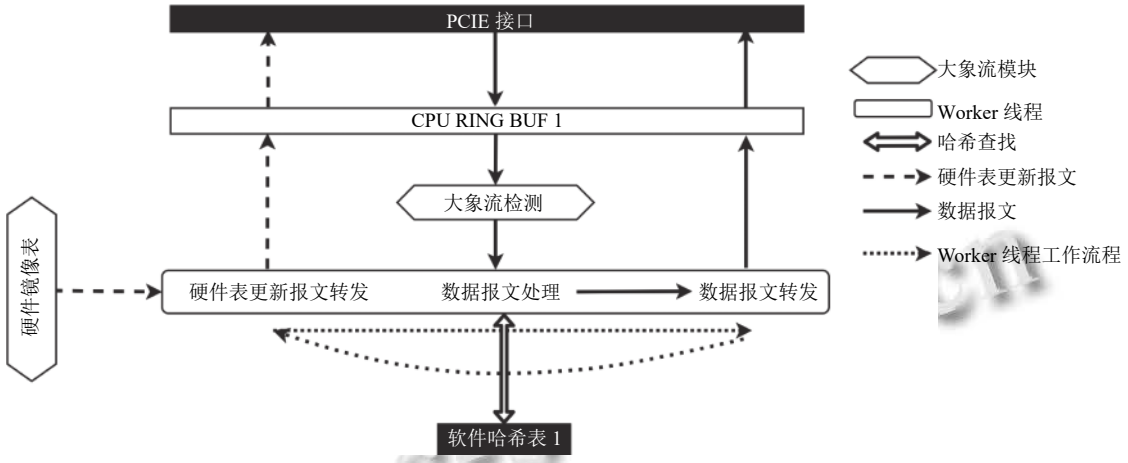

图 9 Worker 线程架构

本文的方案支持多核多队列运行, worker 线程与软件表以及软件队列一一对应, 每一个 worker 线程对应一个软件表, 同时对应一个队列. 数据报文处理部分每次从队列中收取一批数据报文, 根据数据报文报头中的匹配结果进行区分, 对硬件匹配的报文不进行软件表查找, 对未匹配的报文根据报头中的哈希值对软件表查找.

3.1.2 大象流模块

大象流模块由大象流检测以及硬件镜像表构成, 目前大象流检测的方法主要分为计数法[23], LRU (least recently used)[24] 以及采样法[25], 一般来说达到网络流量 0.1% 的流, 本文认为其为大象流, 判别大象流一般先设定阈值[26], 对流量定时 1 s 或者 1 min 进行分析, 判断其是否超出阈值. 在本方案中, 本文使用采样法构建了大象流检测部分, 对数据报文基于概率采样, 按五元组对流量进行统计, 计算采样的每条流在所有流量中的占比, 比值超出阈值的均认为是大象流. 对大象流提取键 (key), 将键与频率按照一一对应发送硬件镜像表.

硬件镜像表与硬件表大小相同, 硬件表处理冲突方法不足并且不便于实现, 本文选择由软件来处理哈希冲突问题. 本文的硬件镜像表采用单哈希的线性表, 本文的冲突解决方法是采用竞争的方法, 对发生冲突的位置, 比较该位置原始流与冲突流之间的流速, 流速大的流拥有该位置, 流速小的流被舍弃, 最终本文的硬件镜像表是大象流的 topn 流速表. 硬件镜像表根据自身键 (key) 对所有软件表按批次查询, 更新自身键 (key) 对应的值. 其后与原有镜像表比对, 对发生替换的键值对 (key-value) 生成硬件表更新报文. 为降低业务线程的硬件表更新报文发送时间, 本文仍然使用批处理的思想, 以 64 个报文为一组对硬件表更新报文均匀分割, 最后不足 64 个报文也视为一组. 以组为单位, 按顺序每次发送一组硬件表更新报文至 worker 线程, 直到所有硬件表更新报文均被发送完毕. 所有线程轮流按组发送硬件表更新报文, 其具体分配方式如图 10 所示. 硬件镜像表需要对硬件表监测, 纠错硬件表更新报文, 本文用硬件表读取报文实现该功能, 利用寄存器按顺序查询硬件表, 对下发失败的表项重新下发硬件表更新报文.

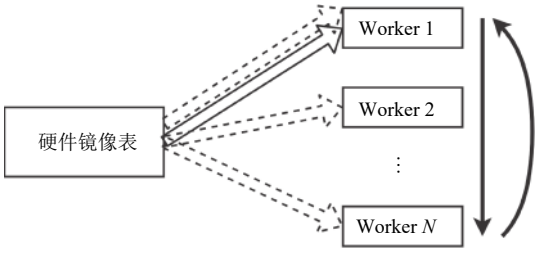

图 10 硬件表更新报文分发

3.1.3 软件表模块

软件表模块服务于 worker 线程与大象流模块. 对 worker 线程而言, 其需要软件表对硬件表查找失败的

数据报文重新进行查找, 对硬件镜像表而言, 其需要为其内部的键 (key) 查找对应的值 (value). 软件表模块由软件哈希表与老化线程构成, 老化线程由定时器触发, 定时对软件哈希表遍历, 剔除超过老化时间的表项, 防止软件哈希表过大. 下面本文将重点介绍软件哈希表的基本构造.

本文使用的软件哈希表基于 DPDK 的哈希库, DPDK 哈希库基于 cuckoo 算法以及集合关联表法两种方法实现, 其具体结构如图 11 所示. 为提高空间利用率, DPDK 选择 cuckoo 算法与集合关联表法共同使用, 整个哈希表由数组构成, 数组每个位置可以存储 8 个键值对. 为降低哈希计算需要的时间, DPDK 哈希表以哈希计算结果 $h(x)$ 作为 prim 位置, 以其 $h(x)^{sig}$ 获取 alt 位置, sig 为哈希值的高 16 位, 一次哈希计算得到两个位置, 以此为基础进行 cuckoo 算法. 为提高查找的效率与 cacheline 的契合性, DPDK 提取哈希值高 16 位字符作为标志位 sig, 每次比较前先比较 sig, 而无需直接进行 key 值的比较, 基于此结构在批处理查找时, 利用 prefetch 可以有效提高查找效率.

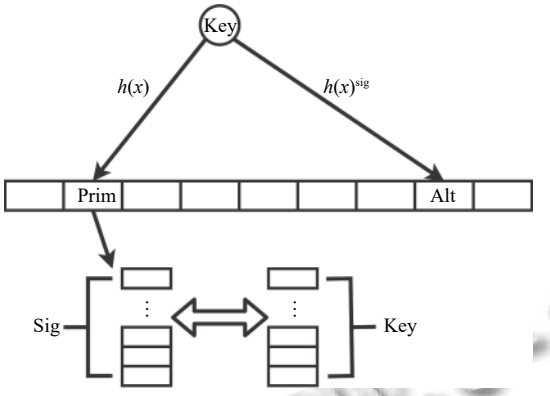

图 11 DPDK 哈希表

软件哈希表作为总表, 其应该尽可能保证稳定, 避免出现因冲突出现重哈希, 在哈希表未满的情况下, 应尽可能避免插入失败, 这要求选择的哈希表空间利用率高. 软件哈希表作为上层应用的基础, 其表项的大小应要满足大规模需求, 同时由于查找性能直接影响业务模块的性能, 无论表的大小与负载率如何, 其查找的时间应该是稳定的. DPDK 的哈希表满足该需求, 因此本文选择 DPDK 哈希库作为软件表.

CRC32 散列函数相比较其他的散列函数, 散列均匀, 冲突率低, 实现简单, 有利于简化硬件电路与软件的计算流程, 因此软件表, 硬件表以及硬件镜像表均采用 CRC32MPEG2 的哈希函数.

3.2 RX TX 模块

RX TX 模块的目的是利用 QDMA 驱动, 构建软件与 FPGA 中的交互通路, 保障双方数据传输与信息交流的稳定性. 其主要包括两个部分, 分别为 DPDK 队列与寄存器. 对 DPDK 来说, 其根据网卡类型, 自动识别网卡, 加载其内部库中为该网卡设计的对应驱动, 网卡接收到的数据流会被该驱动引导到用户态中并由网卡分配到对应队列. 在本文的设计方案中, 本文将 FPGA 视为网卡, 将 QDMA 注册到 DPDK 中, DPDK 自动识别 QDMA 并加载 QDMA 驱动函数, QDMA 驱动为数据封装队列号信息并将数据根据队列号发送到不同软件队列中, 以支持多队列的数据传输, 每一个队列都为双向队列, 支持双向数据传输. 寄存器允许软硬件双向使用. 软硬件可以利用寄存器传递短消息.

3.3 软硬通信机制

本节将从数据流角度详细介绍软件与 FPGA 通信的报文格式以及交流方案. 本文将 CPU 与硬件交互的报文分成 3 类, 分别为数据报文, 硬件表更新报文以及硬件表读取报文, 后两者报文均属于控制报文. 数据报文大小一般不会低于 64 字节, 相比其他报文更长, 硬件表更新报文要求高时效, 能快速送达完成硬件表更新, 硬件表读取报文负责检查硬件表更新报文的结果, 报文长度低, 时延要求低. 针对不同报文的特性, 本文将数据报文与硬件表更新报文作为一体利用 AXI_Stream 总线传输, 硬件表读取报文利用 AXI_Lite 总线传输. 本文在第 3.3.1 节介绍数据报文, 第 3.3.2 节介绍控制报文.

3.3.1 数据报文

数据报文来源为外部以太网帧中的 TCP 协议报文以及 UDP 协议报文, 从数据流向上看, 其是软件与 FPGA 双方向传递的, 本文在原始二层帧上附加本文自定义的报头来传递查找结果以及软硬件信息, 根据数据流向本文分别从下发方向与上行方向介绍本文的报文格式.

(1) 下发方向

下发方向的数据报文格式如图 12 所示, 软件下发的数据报文相比较原始数据帧, 前 8 位标志区分数据报文与硬件表更新报文.

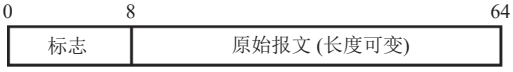

图 12 下发方向的数据报文

(2) 上行方向

上行方向数据报文格式如图 13 所示, 标志位用于指明上传的数据的目的, 匹配结果用于标志当前数据报文是否在硬件表中匹配, 当匹配结果为 0 时, value 全置 0, 哈希值为当前键 (key) 计算得到的哈希值, 为满足软件 cacheline 大小, 剩余 24 位备用以便于未来扩展. 其后为硬件表中的 value 值, 对应硬件卸载的 value 部分, 最后为原始数据帧.

图 13 上行方向的数据报文

3.3.2 控制报文

控制报文是软件与 FPGA 间协同, 消息传递的媒介, 主要针对本方案中的关键模块——硬件表. 根据其功能主要分成硬件表更新报文以及硬件表读取报文.

(1) 硬件表更新报文

硬件表更新报文由硬件镜像表生成, 负责硬件表更新, 从数据流向来说, 其只有软件下发硬件方向, 其具体格式如图 14 所示. 标志用于区分硬件表更新报文与数据报文, 位置指明硬件表插入位置, 五元补充数组对应键 (key), value 为软件希望硬件携带的值信息

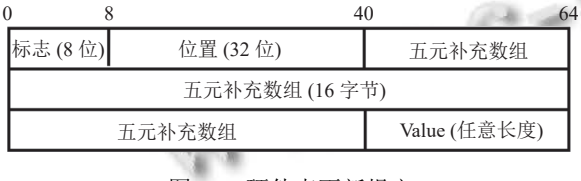

图 14 硬件表更新报文

(2) 硬件表读取报文

硬件表读取报文主要是对硬件下发的键 (key) 确认, 由软件询问, 硬件回复. 如图 15 所示, 其利用寄存器作为沟通媒介, 寄存器 A 由软件写入 FPGA 读取, 寄存器 B 由 FPGA 写入, 软件读取. 寄存器 A 存储软件访问的硬件表位置, 寄存器 B 存储硬件表对应位置的键 (key), AXI_Lite 总线则负责根据寄存器 A 的位置, 提取键 (key) 置于寄存器 B 中.

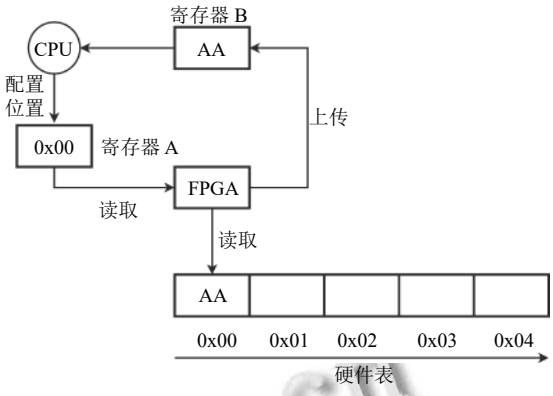

图 15 寄存器通道数据传输

4 实验结果

哈希表的评价指标主要包括插入速率, 查找速率与空间效率, 查找速率指在给定一个 key 的情况下, 返回成功查找与不成功查找的结果所需要的时间, 插入速率指在给定一个 key 的情况下, 插入成功或者失败所需要的时间. 空间效率指出现必须重哈希或者扩容前的最大能插入的 key 与所有尝试进行插入 key 的比例. 这 3 个参数是衡量哈希表性能的关键指标, 对基于哈希表的网络设备而言, 查找速率是最为重要的指标, 其与网络设备的功能直接相关, 是用户最为看重的性能指标, 空间效率应尽可能维持高水平, 尽量避免插入失败带来的重哈希, 最后插入速率也应尽可能保持稳定.

由于本文的方法为利用硬件卸载加速 DPDK 哈希表的查找性能, 因此本文首先比较 DPDK 哈希表算法与其他哈希表算法的查找速率, 其结果如图 16 所示. 本文将 DPDK 的哈希表算法与 hopscotch, cuckoo, 多函数表法, 线性表法, 集合关联表法进行了对比. 可以发现相较其余哈希表算法, DPDK 无论在何种负载下, 其都能维持高速查找, 因此加速 DPDK 哈希表算法的查找性能, 相较加速其余哈希表算法, 具有更大的优势.

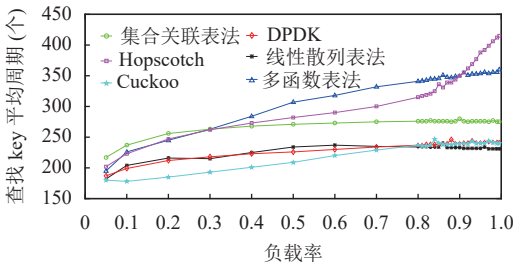

图 16 哈希表算法查找速率对比图

由于本文无法直接测量硬件查找至软件查找需要的时间, 所以本文选择三层转发 (l3fwd) 场景作为基准场景间接测试查找性能. 三层转发是网络转发中的一个重要功能, 其主要包括两种实现方法, 一种基于最长路径匹配, 另一种为基于哈希表的精确匹配. DPDK 也分别实现了对应的函数库, 分别为 l3fwd_lpm (longest prefix match) 库与 l3fwd_em (exact match), 前一种根据最长匹配路径转发, 后一种基于 DPDK 的哈希表, 根据具体五元组进行转发. 为了比较哈希表性能, 本文选择与 DPDK 的 l3fwd_em 方法进行对比, 两种方法的具体步骤如图 17 所示. 两种方法的总体步骤相同, 均为接收网络报文, 对哈希表进行查找, 根据查找结果替换 mac 头, 最后转发, 其转发速率与该 4 个步骤的性能有关.

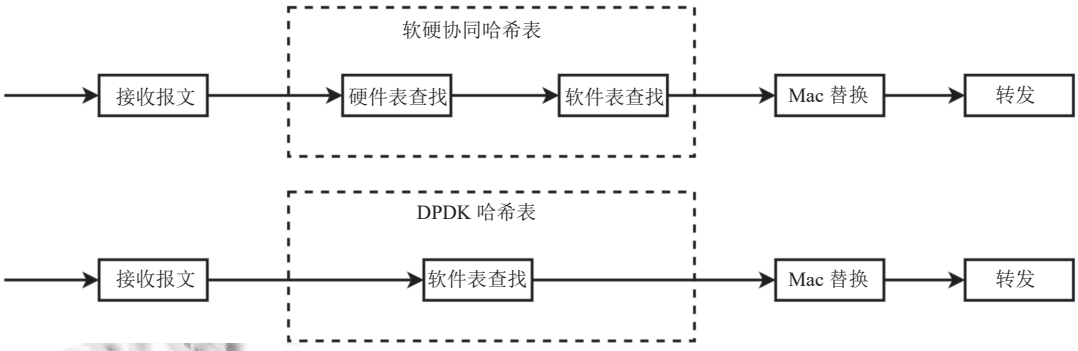

图 17 三层转发对比图

本文使用的设备其报文接收速率与转发速率远超出本文哈希表的处理能力, 并不影响转发性能. 同时所有的 mac 头替换工作均由相同的软件程序完成, 因此转发速率仅与两种哈希表的查找性能有关.

所有的实验的硬件均基于 Xilinx alveo U200 的硬件环境与 DPDK+Linux 的软件环境, 硬件主时钟频率设为 250 MHz, 硬件设计采用 Xilinx Vivado v2019.2. 软件服务器的 CPU 为 Intel(R) Xeon(R) Gold 5218 处理器, CPU 维持最高频率, DPDK 版本为 18.02 版本. 测试用发包仪使用信而泰 bigtao220 测试仪, 测试仪端口提供 0–100 G 的数据流. 本文构造了 10 万条数据流, 其中大象流的数量为 800 条, 剩余的流量均为背景流, 背景流的源 IP 按照递增 1 的方式进行构造, 在 2, 4, 8 线程的情况下所有流能被近似均匀地散列到不同的线程上, 大象流流量总和占总流量的 80%.

由于 DPDK 选择采用 cuckoo 算法与集合关联哈希表法构造哈希表, 因此软件查找需要访问的内存次数与表中键值对 (key-value) 的位置密切相关. 为此本文将本文的查找分成两种情况, 分为乐观查找与悲观查找, 悲观查找指查找的表项在软件表中不存在, 为实现该种情况, 本文将软件表清空, 此时每次查找需要访问两个位置, 比较 16 个 sig. 乐观查找指查找的表项在哈希表, 并且只需要进行一次查找的情况, 在该种情况下, 本文将软件表设置成 100 万大小, 这样情况下每一个键 (key) 基本都位于 prim 位置下的第一个键值对 (key-value) 内, 只需要访问一个位置, 比较 1 个 sig. 这两种方法分别对应了三层转发的最好情况与最坏情况.

DPDK 现有的 l3fwd_em 方法支持两种模式, 第 1 种模式采用哈希表的批查找数据包功能, 本文将其命名为 l3fwd_em_hlm (hash lookup multi) 方法. 第 2 种模式为顺序查找哈希表, 即 l3fwd_em 方法. 在本文的方法中, 软件接收的数据报文是 FPGA 查找完成与未完成混杂的, 一方面提取未查找的数据报文会带来额外的性能开销, 另一方面 DPDK 哈希库并不支持带哈希值的批查找模式, 因此本文的方案基于第 2 种模式, 本文的方案命名为 l3fwd_em_shch (software hardware co-design hash) 方法. 针对本文的方法, 本文将其分成两种情况, 分别为大象流全部卸载情况, 命名为 l3fwd_em_shch_fo (fully offload), 以及大象流全部未卸载情况, 命名 l3fwd_em_shch_no (no offload).

以两种查找方式作为区分, 本文分别测试了两种查找下 l3fwd_em_hlm, l3fwd_em, l3fwd_em_shch_fo 及 l3fwd_em_shch_no 方法的性能. 对每种方法, 本文分别测试了 1, 2, 4, 8 业务线程数下其性能, 测试的包全都采用 128 字节大小的数据包, 所有数据包数量总和为 30 万. 其结果分别如图 18 与图 19 所示.

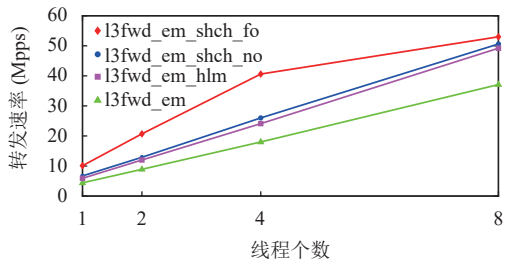

图 18 悲观查找三层转发性能对比图

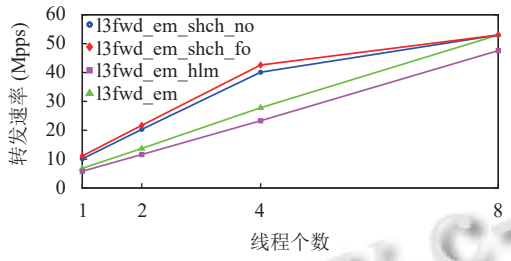

图 19 乐观查找三层转发性能对比图

根据图 18, 在悲观查找下, 随线程个数的增加, 4 种方法的性能都近似随着线程个数的线性增加. 根据 QDMA 官方的多队列测试报告, QDMA 存在着传输上限, 根据本文的测试, 在当前环境下, QDMA 传输上限为 53 Gbps, 与其测试报告上限速率接近, 因此在 8 线程时 l3fwd_em_shch_fo 方法由于达到了传输速率的上限, 因此未呈现线性增长的趋势. 在悲观查找下, l3fwd_em_hlm 方法的查找性能高于 l3fwd_em 方法, 阅读 DPDK 源码可以发现因在悲观查找下, DPDK 哈希表的两个位置都要被访问, hlm 方法利用 prefetch 方法预读取了两个位置, 减少了 cache miss 情况, 因此其效率会高于 l3fwd_em 方法. l3fwd_em_shch_no 方法相比较 l3fwd_em_hlm 方法, 性能提升不大. 原因在于悲观查找下, l3fwd_em_shch_no 内存访问次数为 16 次, 大量的内存访问增加了查找的时间, 但是仅卸载哈希计算仍然能带来 5% 的性能提升. l3fwd_em_shch_fo 方法由于卸载了大象流查找, 因此只有小部分背景流需要进行查找, 同时该背景流也携带有哈希计算结果, 因此性能得到大幅度提升, 相比较 DPDK 内该种情况下自身最高效的 l3fwd_em_hlm 方法, 性能提升接近 75%.

根据图 19, 在乐观查找下, 随线程个数的增加, 4 种方法的性能都近似随着线程个数的线性增加. 与悲观查找相反, 乐观查找 l3fwd_em_hlm 方法的查找性能低于 l3fwd_em 方法, 原因在于 l3fwd_em_hlm 方法主要依赖 prefetch 预取来提高查找性能, 为了预取的需要, 其每次都会预取 prim 位置以及 alt 位置的键值对, 但是在乐观查找下, 其由于不需要访问 alt 位置并且只需要访问 prim 位置的第一个键 (key), 所以 l3fwd_em 方法性能更高. l3fwd_em_shch_no 方法相比较 DPDK 中最高效的 l3fwd_em 方法性能提升了 48%, 原因在于内存访问次数相同的情况下, 哈希计算的卸载能大幅度提高查找的性能. l3fwd_em_shch_fo 方法相较 DPDK 自身最高效的 l3fwd_em 方法, 性能提升接近 64%.

综合图 18 与图 19, 对 l3fwd_em_hlm 方法, 悲观查找的性能高于乐观查找, 原因在于无论悲观查找还是乐观查找, 其预取的操作不发生变化, 但悲观查找比乐观查找少了一次读取键值对 (key-value), 因此其性能略高于乐观查找. 总体来说, 两者性能接近, 其性能稳定. 由于乐观查找带来的内存访问次数的影响, l3fwd_em 方法、l3fwd_em_shch_no 方法以及 l3fwd_em_shch_fo 方法的性能比悲观查找都有所提升.

本文方案仅在软件哈希表插入, 为测试本方案的空间效率与插入速率, 将本文的软硬协同哈希表与常用哈希表算法比较, 其对比图如图 20 与图 21 所示. 如图 20 所示, 本文的软硬协同哈希表相较其他的哈希表算法, 其空间效率具有极大的优势, 可以有效避免重哈希, 提高系统的稳定性.

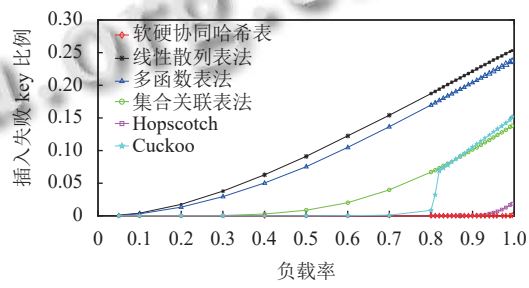

图 20 哈希表算法空间效率对比图

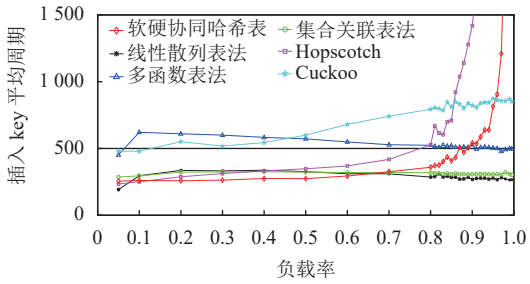

图 21 哈希表插入速率对比图

如图 21 所示, 本文的软硬协同哈希表相较其他哈希表算法, 其插入速率在 80% 负载以下时处于较高水平, 超过 80% 负载时, 其插入速率呈现指数级增长. 低于 90% 负载情况下, 本文的软硬协同哈希表的插入速率处于可接受范围内. 为避免插入带来的最坏时间影响, 应尽量扩大软件哈希表大小以避免负载率超出 90%.

综合来说, 本文的方案无论是乐观查找还是悲观查找, 其性能都能够得到大幅度提升, 与 DPDK 最高效的三层转发方法相比, 在悲观查找下, 大象流表项全部卸载性能能提升接近 75%, 未卸载性能也能提升接近 5%, 在乐观查找下, 大象流表项全部卸载性能能提升接近 64%, 未卸载性能能提升接近 48%. 并且本文的方案相较现有常用的哈希表算法, 空间效率更高, 插入速率在 90% 负载以下时能维持接近稳定水平.

5 结论与展望

本文提出了一种基于软硬协同的大规模哈希表设计方案, 该方案采用软硬结合的思想, 将查找一分为二, 硬件表负责查找大象流, 软件表负责查找剩余的流量以及硬件表的更新, 软硬间以控制报文和在数据报文上附加信息的方式进行协同. 经过测试, 该方案可以大幅度提高哈希表的查询性能, 并且本文的方案空间效率高, 鲁棒性强同时插入在 90% 负载下能保持高性能. 本文目前实现的软硬协同哈希表, 其性能上限受到 QDMA 的限制, 同时 DPDK 哈希库的查找效率仍然存在改进的空间, 因此本文下一步工作主要为对当前的 QDMA 加以优化, 提高软件与硬件之间交互速率的上限并且优化插入算法, 提高高负载下的稳定性.

参考文献

1 Colbourn CJ, Ling ACH. Linear hash families and forbidden configurations. Designs, Codes and Cryptography, 2009, 52(1): 25–55. [doi: 10.1007/s10623-008-9266-7]

2 Panigrahy R. Efficient hashing with lookups in two memory accesses. Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms. Vancouver: Society for Industrial and Applied Mathematics, 2005. 830–839.

3 Le Scouarnec N. Cuckoo++ hash tables: High-performance hash tables for networking applications. Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems. Ithaca: ACM, 2018. 41–54.

4 Tian H, Chen YX, Chang CC, *et al*. Dynamic-hash-table based public auditing for secure cloud storage. IEEE Transactions on Services Computing, 2017, 10(5): 701–714. [doi: 10.1109/TSC.2015.2512589]

5 Devroye L, Morin P. Cuckoo hashing: Further analysis. Information Processing Letters, 2003, 86(4): 215–219. [doi: 10.1016/S0020-0190(02)00500-8]

6 Kelly R, Pearlmutter BA, Maguire P. Lock-free hopscotch hashing. Proceedings of the 1st Symposium on Algorithmic Principles of Computer Systems. Salt Lake City: Society for Industrial and Applied Mathematics, 2020. 45–59.

7 Ahmed DT, Shirmohammadi S. Multi-level hashing for peer-to-peer system in wireless ad hoc environment. Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PerComW 2007). White Plains: IEEE, 2007. 126–131.

8 Kumar S, Crowley P. Segmented hash. Proceedings of the 2005 Symposium on Architectures for Networking and Communications Systems (ANCS). Princeton: IEEE, 2005. 91–103.

9 Kirsch A, Mitzenmacher M. The power of one move: Hashing schemes for hardware. IEEE/ACM Transactions on Networking, 2010, 18(6): 1752–1765. [doi: 10.1109/TNET.2010.2047868]

10 Yang X, Sezer S, McCanny J, *et al*. DDR3 based lookup circuit for high-performance network processing. Proceedings of the 2009 IEEE International SOC Conference. Belfast: IEEE, 2009. 351–354.

11 István Z, Alonso G, Blott M, *et al*. A hash table for line-rate data processing. ACM Transactions on Reconfigurable Technology and Systems, 2015, 8(2): 13.

12 Chen H, Madaminov S, Ferdman M, *et al*. FPGA-accelerated samplesort for large data sets. Proceedings of the 2020 ACM/SIGDA International Symposium on Field-programmable Gate Arrays. Seaside: ACM, 2020. 222–232.

13 Ioannou L, Michail HE, Voyiatzis AG. High performance pipelined FPGA implementation of the SHA-3 hash algorithm. Proceedings of the 2015 4th Mediterranean Conference on Embedded Computing. Budva: IEEE, 2015. 68–71.

14 Panait O, Dumitriu L, Susnea I. Hardware and software architecture for accelerating hash functions based on SoC. Proceedings of the 2019 22nd International Conference on Control Systems and Computer Science (CSCS). Bucharest: IEEE, 2019. 136–139.

15 Fairouz A, Khatri SP. An FPGA-based coprocessor for hash unit acceleration. Proceedings of the 2017 IEEE International

Conference on Computer Design. Boston: IEEE, 2017. 301–304.
16 DPDK. DPDK 21.08 Intel NIC performance report. https://core.dpdk.org/perf-reports/. (2021-10-19)[2022-04-10].
17 Shi J, Pesavento D, Benmohamed L. NDN-DPDK: NDN forwarding at 100 Gbps on commodity hardware. Proceedings of the 7th ACM Conference on Information-Centric Networking. Montreal: ACM, 2020. 30–40.
18 Sasaki K, Hirofuchi T, Yamaguchi S, *et al*. An accurate packet loss emulation on a DPDK-based network emulator. Proceedings of the Asian Internet Engineering Conference. Phuket: ACM, 2019. 1–8.
19 Furukawa M, Matsutani H. A DPDK-based acceleration method for experience sampling of distributed reinforcement learning. arXiv:2110.13506, 2021.
20 Hamdan M, Mohammed B, Humayun U, *et al*. Flow-aware elephant flow detection for software-defined networks. IEEE Access, 2020, 8: 72585–72597. [doi: 10.1109/ACCESS.2020.2987977]
21 刘朝晖, 窦晓光. 基于 FPGA 实现的报文分类智能网卡. 信息安全与技术, 2013, 4(6): 62–65.
22 陈宝远, 张秀芝, 梁状. 基于 FPGA 的高效数据过滤技术研究. 微电子学与计算机, 2017, 34(12): 130–133, 137. [doi: 10.19304/j.cnki.issn1000-7180.2017.12.027]
23 Ben Basat R, Einziger G, Friedman R. Space efficient elephant flow detection. Proceedings of the 11th ACM International Systems and Storage Conference. Haifa: ACM, 2018. 115.
24 严军荣, 叶景畅, 潘鹏. 一种大象流两级识别方法. 电信科学, 2017, 33(3): 36–43.
25 Estan C, Varghese G. New directions in traffic measurement and accounting. ACM SIGCOMM Computer Communication Review, 2002, 32(1): 75. [doi: 10.1145/510726.510749]
26 Qi JH, Li WJ, Yang T, *et al*. Cuckoo counter: A novel framework for accurate per-flow frequency estimation in network measurement. Proceedings of the 2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS). Cambridge: IEEE, 2019. 1–7.

(校对责编: 孙君艳)