实现高效的多类型负载感知——面向网络键值存储的方案

Abstract

分布式键值存储对于高性能在线服务（如社交网络和电子商务）至关重要，但面临着负载不平衡的挑战。为了有效地平衡负载，网络键值存储会基于负载统计信息路由、缓存和复制键值项。然而，现有的网络解决方案要么不可行，要么效率低下，无法计算多类型负载（例如内容、存储和网络），其巨大的关键字空间可能会迅速耗尽内存预算和控制平面带宽。本文旨在实现多类型负载的高效感知，具体地，我们(1)系统地识别和分析可能导致性能瓶颈的负载类型;(2)提出了一个名为Lake的框架，利用随机方差最小化理论实现高效的多类型负载感知;(3)基于负载统计信息，确定了一些基于正确构建的优化技术。最后，基于我们在Tofino交换机上的原型实现，广泛的实验表明，Lake为多类型负载感知提供了更高的准确性（例如F1得分从0.88提高到1），并且具有可接受的硬件消耗（例如哈希调用减少了56.51%）。

1、INTRODUCTION

现代互联网服务，如电子商务和社交网络，严重依赖于分布式键值存储系统，以满足严格的服务级别目标（SLO，例如10-100毫秒的响应延迟[1, 2]）。在实践中，键值存储系统面临的主要挑战是在高度倾斜和动态的工作负载下满足严格的SLO，这可能导致负载不均衡并显著降低系统吞吐量。 为了解决这个挑战，基于可编程交换机的网络键值存储系统[3-5]通过在实时中使用可编程交换机感知和平衡负载。例如，NetCache [3]在ToR交换机上部署了负载平衡缓存。ToR交换机上的缓存由于交换机可见性和高性能的交换机ASIC（即> 10Tbps ASIC [6]）而显著提高了系统吞吐量。 然而，现有的基于可编程交换机的解决方案要么不可行，要么无法高效地全面感知当前可编程交换机上的多类型负载（例如内容、存储和网络）。对于不可行性，它们可能无法同时感知可编程交换机上的多类型负载。对于效率低下，多类型负载感知可能会消耗大量的交换机资源。这个限制根本上源于负载的巨大键空间，它可能迅速消耗交换机资源。具体来说，基于寄存器的解决方案[3-5,7]可以使用寄存器数组感知负载。但是，负载的巨大键空间（对于16B键而言，有2^{128}个负载）可能很快耗尽稀缺的寄存器资源（在我们的基准分析中，NetCache[3]仅支持2个草图）。基于采样的解决方案[5]减少了内存占用，但由于有限的控制面带宽（在我们的基准分析中，Exp#2中的丢包率高达44.54%），可能会丢失大量的负载。因此，系统可能被未知的负载类型阻塞并遭受性能惩罚。负载感知可能无法与基本交换机功能（例如，流表条目）和网络内键值功能（例如，缓存或一致性目录）共存。因此，探索在可编程交换机上高效部署多类型负载感知是否可行，以及如何实现，是本文所解决的关键问题。 本文中，我们确定了三种潜在的负载类型，可能会在网络键值存储系统中引入瓶颈： • 内容负载。来自客户端的请求可能会由于动态和快速变化的负载分布而引入瓶颈，例如写密集型请求可能会引入显着的一致性开销[3，5]。 • 网络负载。交换机中的网络流量可能会由于网络拥塞而导致瓶颈。例如，在Memcached的多对多网络连接模型[9]下，交换机中可能会发生突发流量[8]。 • 存储负载。由于项目分区不均匀，存储服务器可能会过载，例如，负载访问高度不平衡，因此存储负载可能会超过单个服务器的处理能力[5]。 在确定了这些负载类型之后，我们的提供了多种负载的统一抽象，以实现定制化的网络键值存储系统的高效多类型负载感知。具体而言，我们(1)将多类型负载感知转化为子集和估计问题，以提供多类型负载的统一抽象；(2)设计了一个概率框架，称为Lake，基于该抽象实现多类型负载感知并具有理论保证；(3)合成和确定一些已知和新颖的实际优化技术，以基于感知结果进行负载平衡；(4)设计了一种高级语言，以表达用户需求，并进行硬件优化，以便他们可以有效地定制负载感知。在实践中，用户可以通过Lake简单地组装负载字段，并具有理论保证地感知多类型负载。负载统计数据允许用户全面地识别瓶颈并为多类型负载设计负载平衡算法。 在我们的评估中，我们在Barefoot Tofino交换机[10]上进行了广泛的实验。结果显示，Lake可以准确地计算多类型负载(例如，5种负载类型)，并具有更少的资源占用(例如，零负载丢失报告到控制平面)，并且可以启用高效的负载平衡技术(例如，最多4.03倍的MQPS)。

2、BACKGROUND

在本节中，我们首先介绍了工作负载分布的背景知识。接着，我们介绍了新兴的可编程交换机，它启发了一些有前途的网络内键值存储架构。

A. 在键值存储中的倾斜工作负载

当将分布式键值存储扩展到数十亿个客户端时，一个主要的挑战是在高度倾斜和动态的工作负载下满足严格的SLO。具体来说，(1)对于高度倾斜，热门项目接收的请求远远多于其他项目(每天数百万次请求) [11, 12]，(2)对于动态分布，热门项目会动态地、迅速地变得不受欢迎(<10分钟) [13]。热门项目可能突然聚集到少数几个服务器上，导致服务器过载和性能惩罚[5]。实际的键值存储系统的工作负载通常使用Zipf分布进行建模[11, 12, 14, 15]，通常表现出高度倾斜的模式(例如，Zipf分布的α>1 [12, 16])。键值存储系统将项目分区和分发到多个服务器以确保SLO。倾斜的工作负载主要包括倾斜的内容负载、网络负载和存储负载。这些倾斜的负载可能会使少量资源过载，从而成为新的瓶颈，并引入更长的尾延迟。超额配置资源可以减少性能惩罚，但会显著增加总体成本[5]。

B. 可编程交换机

新兴的可编程交换机提供数据平面（DP）上可定制的线速数据包处理能力（>10Tbps [6]）。典型的可重构匹配-动作表（RMT）范式[17]维护了一个阶段的管道，每个阶段都配备了相同的设计、相同数量的资源（例如SRAM和SALU）。每个阶段维护一个匹配表，将数据包头字段与分配的值匹配，并通过动作单元执行简单的指令。交换机资源非常有限。例如，一个典型的商业交换机维护12个管道阶段，拥有10MB的TCAM和SRAM，10个SALU和10个哈希调用[17-19]。用户使用P4语言编写数据平面程序。P4编译器只有在所需资源足够时将程序映射到管道中。

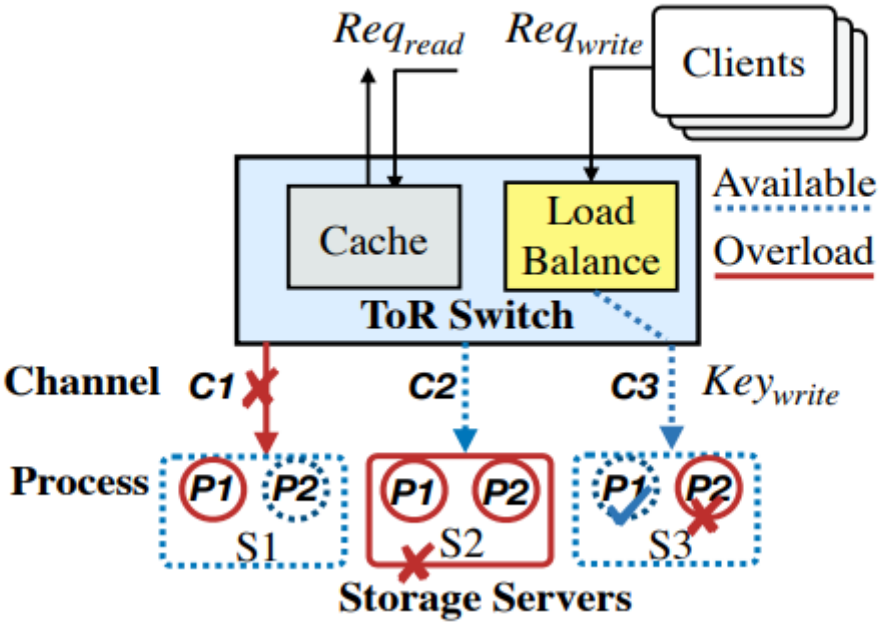
C. 网内键值存储

In-network键值存储系统利用新兴的可编程交换机来路由、缓存和复制流行的数据项[3-5]。通常，这些系统由于交换机可见性（即请求聚合点）而能够感知多类型负载，并由于高性能的交换机ASICs（即>10Tbps [6]）而能够实时平衡负载。因此，这些系统可以为key-value存储提供高吞吐量、低延迟和可扩展性。接下来，我们将详细阐述每个典型的工作流程，分为两个阶段：负载感知和负载均衡。 **网内缓存。**NetCache[3]将ToR（顶层交换机）作为负载平衡缓存（中等缓存命中率<50% [4]），吸收热点项，并使存储服务器上的负载更加均匀。在负载感知阶段，它维护一个CM草图来检测热门未缓存的键和一个Bloom过滤器来在数据平面消除复制的键。在每个时间窗口结束时，将统计信息报告给控制平面，在那里控制器确定缓存集。在负载平衡阶段，数据平面中的缓存集会被新的缓存集更新。缓存的键直接回复读取请求到客户端，而写请求被转发到服务器以确保一致性。 **基于内容的路由。**SwitchKV [4]将存储节点分为慢存储节点和内存缓存节点，并维护每个键的地址，以便可以快速路由请求到存储节点。在负载感知阶段，它为每个后端存储节点保持最近访问键的本地频率计数器。对于频率大于阈值的键，SwitchKV定期将最近的热键和突然变得非常热的键报告给缓存节点。在负载平衡阶段，内存缓存节点添加或驱逐选定的键，并直接将请求路由到相应的存储节点。 **网内复制**是指Pegasus [5]通过将热点数据复制到多个存储服务器来分配负载。它在保证数据一致性和协调性的同时，重新平衡数据副本。交换机维护一个网内一致性目录，跟踪和管理数据副本的位置。在负载感知阶段，它对未复制键的请求进行采样，并将其转发到交换机CPU，在该程序中计算访问频率。每个服务器收集负载统计信息并报告给控制器，在那里，两个独立的寄存器数组跟踪每个副本键的读/写计数。在负载平衡阶段，控制器基于负载统计数据选择副本。

3、MOTIVATION

在本节中，我们详细阐述本论文的动机。

A. 为什么需要多类型负载感知



负载主要包括三种类型：内容、网络和存储。如图1所示，任何负载类型都可能引入性能瓶颈。因此，我们应该全面调度多类型负载以解决这些瓶颈。此外，现有的键值存储系统有动机基于多类型负载平衡负载。例如，Pegasus意识到写入和读取密集的请求，并使用不同的服务器选择策略复制这两种请求。因为写入密集请求的成本很容易抵消负载平衡的好处[5]。此外，现有的系统需要在不同的粒度上平衡负载。例如，该系统[9]在服务器级别或进程级别平衡负载以提高整体系统利用率。然而，现有的系统无法高效地意识到多类型负载。为了分析根本原因，我们定义了一个多类型负载，将现有要求综合在一起：L（例如，184位）= {键频率（key_f，例如，128位键），写键频率（key_w，例如，1位标志），存储服务器的物理地址（$addr_{phy}$，例如，32位IP地址），进程的虚拟地址（$addr_{vir}$，例如，16位端口号），网络流水线的负载（cha_{id}），例如，7位交换机端口}。接下来，我们以L为例分析现有工作的限制和潜在的替代方案。

B. 现存的解决方案和局限性

Table I. The typical systems of in-network key-value stores

In-network System	Load Awareness	Load Balancing
SwitchKV [4]	Hotkeys	Route to storage servers
	Burst hotkeys	Adapt to new workloads
NetCache [3]	Cached keys	Update cache sets
	Uncached keys	Cache new hotkeys
	Remove duplicate	Remove the duplicate keys
Pegasus [5]	Server loads	Balance the loads of servers
	Hotkeys	Select hot replicas
	Write keys	Select hot replicas

如表I所示，我们比较了提供多类型负载感知的不同方法。我们考虑了6个指标：负载感知的准确度、负载感知速度、控制平面（CP）和数据平面（DP）之间的状态一致性、需要感知的类型数量、DP资源占用以及CP和DP之间的带宽使用情况。 **备选方案1（A1）：使用寄存器组进行多类型负载感知。**一种方法是在寄存器组中计算多类

型负载。但是这种方法很昂贵，甚至是不可行的。例如，键空间可能具有2^{128}个16字节的键。然而，硬件资源非常有限（例如，10MB SRAM [17, 19]）。因此，现有的工作[3, 5]只能使用寄存器组计算缓存/复制的键，以减少SRAM的消耗。因此，在工作负载快速变化时，突然变得流行的键可能无法被寄存器组捕获。键值存储可能会丢失热键并遭受性能惩罚。 **备选方案#2（A2）：每种负载类型使用一个sketch。**Sketch是一种高效的近似数据结构。另一种备选方案是使用每个sketch计数每个负载。例如，我们可以部署5个CM sketch，每个sketch都使用寄存器计数L中的一种负载。然而，Tofino交换机最多只支持4个sketch [24]。更糟糕的是，一些键值系统只能部署2个sketch（在我们的基准分析中证明）。此外，必须确保与基本交换机功能（例如防火墙）和网络键值功能（例如缓存或一致性目录）的共存。因此，这种备选方案要么不可行，要么效率低下。 **备选方案#3（A3）：后恢复的全负载感知。**另一个基于草图的备选方案是同时感知所有负载（即L），并通过聚合其他负载来恢复部分负载（例如$addr_{vir}$）。但是相关研究[24,25]已经证明，这种方法可能会导致高估偏差。此外，巨大的键空间可能会带来显著的查询开销（例如，从一个大小为2^{184}的集合L中，聚合2^{64}个地址虚拟化$addr_{vir}$的键值需需要聚合2^{136}个键。 **备选方案#4（A4）：在交换机中进行采样并向控制平面（CP）报告。**一个备选方案是对负载状态进行采样并向CP报告。这样可以减少寄存器的消耗而不存储状态。但是，这也会引入一个新的瓶颈：交换机的PCIe带宽非常有限，可能会丢失大量状态。因此，在低采样率下，替代方案将丢失重要的状态（例如，工作负载分布的突然变化[4]）。在高采样率下，由于CP带宽有限（在Exp#2中证明），状态也可能被CP丢弃。 **备选方案#5（A5）：在服务器中收集并向CP报告。**另一种备选方案是在每个存储服务器中收集负载统计信息并将这些统计信息报告给CP。其主要限制是需要为每个状态维护状态一致性和同步机制。网络和处理能力可能会延迟这些状态的传输。这对于实时受益于交换机键值存储来说是具有挑战性的。此外，服务器很难意识到交换机网络负载，这可能表明网络拥塞。

基准分析。在我们的基准分析中，我们保持NetCahce中相同的配置，即使用4个哈希函数和2^{16}个16位寄存器的CM sketch以及3个寄存器数组的Bloom过滤器，每个数组都有2^{18}个1位寄存器。在Tofino交换机中，我们只能用两个CM sketch成功编译数据平面程序，测量128位键和48位$addr_{vir}$在Exp＃1中详细说明。 **总结。**总之，对多类型负载的认识需求和替代方案（A1-A5）的限制促使我们 更好地了解多类型负载。

4、多类型负载感知

本节中，我们详细阐述了设计目标和挑战。接下来，我们说明了如何转化问题并且提出了一中数据结构来应对这些挑战。

A. 挑战

目标我们旨在在没有复杂一致性机制的情况下高效的感知多类型负载。此外，用户可以灵活的定制负载感知。 **挑战#1（C1）：多功能负载感知。**由于负载类型的异构性，支持高效的多类型负载感知是有挑战性的。具体来说，不同的负载类型（例如：内容、存储和网络）可能具有不同的重要性和表达方式。作为一个框架，我们应该形成多类型负载感知的同构表达式。此外，该表达式在交换机硬件的严格约束下高效运行（在§ IV.B中解决）。 **挑战#2（C2）：重负载的保真度。**重负载的保真度至关重要，因为负载平衡可能会被不准确的结果误导。例如，我们可以清除网内缓存中的热项目，因为写入项目无法从网内缓存中受益。但错误的清除热项目可能会减少系统吞吐量。因此，我们需要保证重负载的保真度。但是，交换机可能无法储存所有的负载去确保无误的感知，或者由于交换机硬件限制而无法预测所有的重负载。因此，这需要我们量身定制的设计来保证重负载的保真度（在§ IV.C中解决）。 **挑战#3（C3）：元数据依赖性。**多类型负载的元数据依赖性可能会消耗重要的流水线阶段。在数据平面，交换机ASIC可以实现高速负载感知。为了从交换机ASIC中受益，不可避免的需要使用寄存器数组进行状态感知。要读取或写入这些数组，我们应该计算一个键的位置。但更新一个元数据的数组必须等待另一个元数据更新完成。否则，多类型负载必须消耗更多的交换机流水线阶段来防止数据冒险，这会导致资源消耗过多，甚至编译失败（在§ IV.F中解决）。

B.多类型负载感知

为了提供一个统一和正式的负载感知的抽象，我们定义了一个称为多类型负载感知的新类，该类可以同时支持多种类型的负载感知，而不需要预定义负载。具体来说，我们将所有需要感知的负载并集起来，形成一个完整的负载，在其中可以恢复所有的部分负载。 **定义1.**（部分负载）一个负载l_P是完整负载l_F的部分负载（用$l_P < l_F$表示），如果存在映射$g(\cdot):l_F \rightarrow l_P$，并且对于任意定义在$l_P$上的键$e∈l_P$，我们有 $f(e) = ∑_{e'∈l_f, g(e') = e}{f(e')}$，其中$f(e)$是e的统计量。 **例子**.对于一个负载$l_F$ = (键值，请求，服务器ID)(例如，$(k1, write, server_{31})$表示带有写请求的key_1被转发到了$server_{31}$)，一个部分负载（例如，$(k1, write,)$）的键k_1的频率等同于其在全流量$∑_{i = 0}^{31}(k_1, write, server_i)$中的频率。需要注意的是，一个部分负载可以是完整负载中的任何子集，例如$(key,,*)$是l_F的一个部分负载。 **定义2.**（多类型负载感知）。给定一个全负载和一个度量函数f， 在每个时间窗口结束时返回任意键$e ∈ l_P$的$f(e)$，其中l_P是l_F的任意一个部分负载。这个定义允许用户只需要感知l_F就可以在每个时间窗口结束时查询任意l_P了。

C. 负载感知数据结构

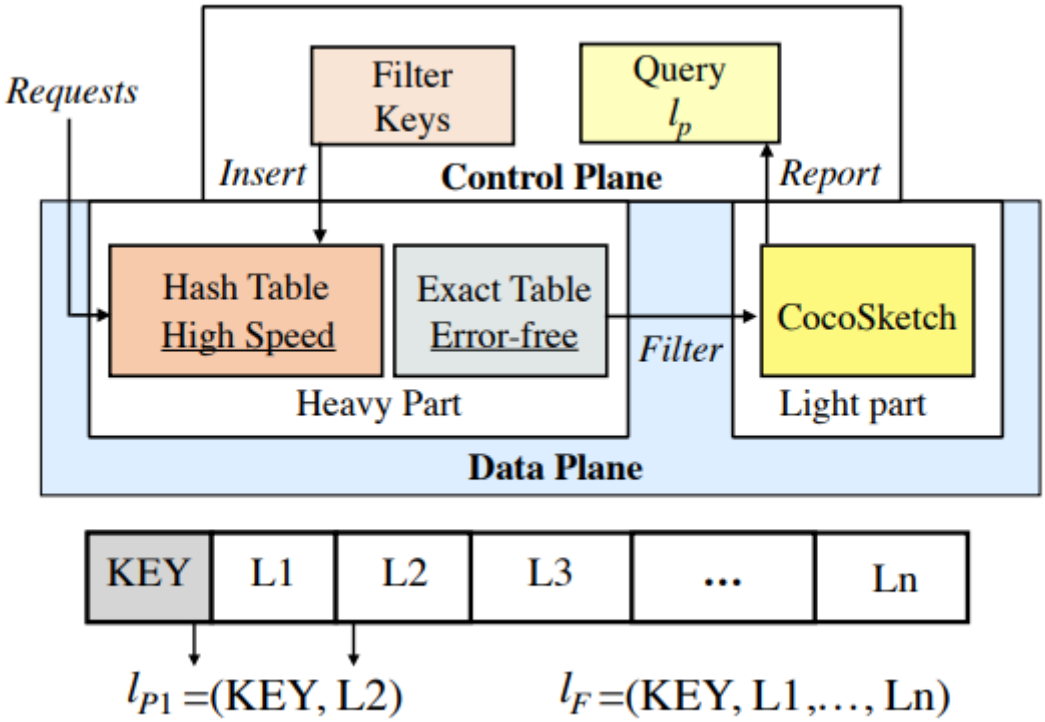
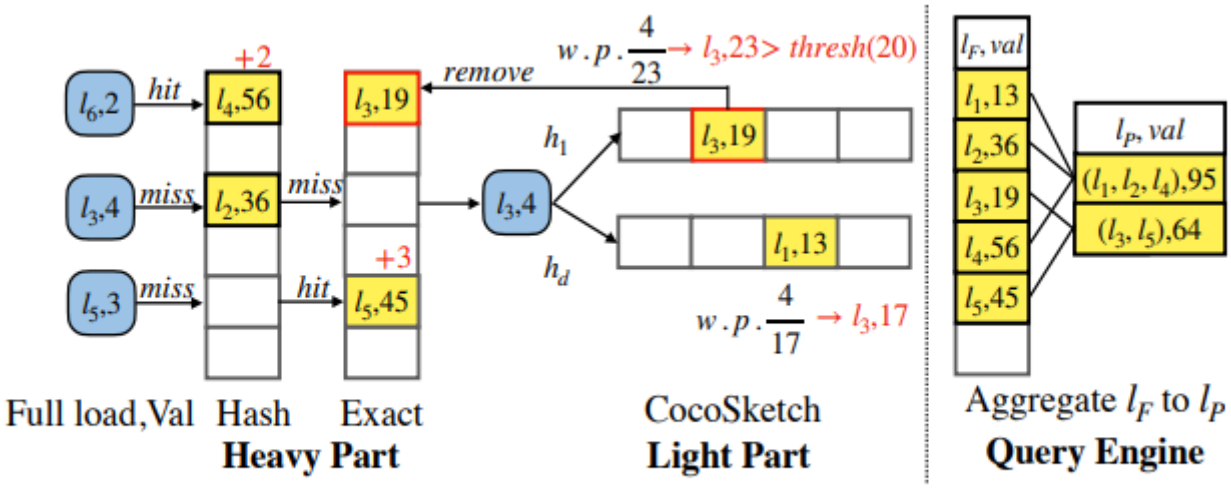


Fig. 4. The full load and a partial load.

概述.如图3所示，Lake的架构包括一个重型部分和一个轻型部分。为了解决C1问题，Lake (1) 从轻型部分筛选出重负载，以确保重负载的保真度，和(2) 替换了Bloom过滤器，以减少重负载的重复报告，以降低控制平面带宽的消耗。在轻型部分中，我们最小化l_F的所有子集和的随机方差，以减少所有l_P的估计误差。具体而言，我们在轻型部分使用CocoSketch，移除和报告重负载(即计数超过阈值的负载)到控制平面。然后，将重载插入重型部分。对于每个负载，重型部分直接在数据平面中增加其计数，绕过轻型部分。因此，重型部分计数重负载的误差可以忽略不计，从而也提高了重负载感知的保真度。此外，轻型部分还减少了哈希碰撞。最后，负载及其统计数据将报告给控制平面以进行进一步的l_P查询。



Step 1: Filter hot loads and measure cold loads in DP **Step 2**: Query l_P in CP

Fig. 5. The detailed process of load awareness using Lake.

数据结构.如图5所示，数据结构由两部分组成：记录热点负载的重部分和记录冷负载的轻部分。重部分H包含一个高速哈希表和一个无误差的精确表。重部分的每个桶记录一个负载的计数：负载键和负载值。轻部分是一个支持多类型负载查询的sketch（例如CocoSketch）。CocoSketch维护d个包含l个键值对的数组。每个数组与一个独立的哈希函数相关联。 **Lake的插入操作**.当需要插入具有负载键和负载值的负载时，Lake会提取负载键并计算哈希函数以定位哈希表中的一个计数器。当负载键与记录的键相同时，Lake直接将负载值增加到计数值。当负载键不同时，Lake会检查精确表中的相应记录键是否相同。如果键相同，则Lake增加相应计数器的值。当重量部分的两个表都未命中时，负载将记录在轻量部分。CocoSketch首先将负载键映射到d个桶中（每个桶来自d个数组之一），并独立地更新每个桶，然后用$\frac{负载值}{总值}$的概率来替换记录的键。当记录值超过阈值时，Lake会将超过阈值的负载移除并将其报告给CP，然后通过CP将负载插入到重部分。 **例子**.如图4所示，Lake对全负载进行计数。对于存储在哈希表中的重负载l_6，Lake直接增加其计数器。对于在哈希表中发生冲突但存储在精确表中的重负载l_5，Lake也增加其计数器。重部分可以快速准确地过滤重负载。对于轻负载l_1和l_3，它们在CocoSketch（硬件版本）中进行计数，同时也受益于随机方差最小化，可以提高l_P意识的准确性。CocoSketch增加所有的值，并用$\frac{值}{总值}$的概率替换负载。在每个时间窗口结束时，完整的负载都会报告给CP来构建一个完整的负载表。在查询部分负载（l_3，l_5）时，Lake聚合全负载l_3和l_5，并生成最终结果64。 **Lake查询**.Lake查询。在CP中，Lake将由重负载部分和轻负载部分记录的负载构建成一个包含两列（负载键，负载值）的完整负载表。在查询时，通过聚合l_F的值来查询部分负载的值。

D. 理论分析

由于将热负载和冷负载分离，Lake在大多数情况下具有搞准确性。在分析之前，我们将要定义如下符号。我们用$l(e)$表示负载e的真实值，用$\widehat{l}(e)$来表示e的估计值，用(e_i, w)来表示每个新的负载和其值。 **随机方差最小化。**我们首先分析负载方差和的最小化。我们的最终目标是最小化负载综合的估计。$$minimize∑_e(l(e) - \widehat{l}(e))^2(1)$$ 在轻量级部分，我们需要考虑每个插入的新方差所导致的负载方差和的逐渐最小化。$$minimize∑_eΔ(l(e) - \widehat{l}(e))^2(2)$$ 我们确保每次插入引起的方差变化最小化，以便Lake可以减少整体估计多类型负载误差。 **引理1。**更新桶(e_j, l_j)的最小方差增量为 $$∑_eΔ(l(e) - \widehat{l}(e))^2 = \left{\begin{matrix}2wl_j, e_i≠e_j\0, e_i = e_j\end{matrix}\right.(3)$$ *证明。*对于重负载，每次在重负载中插入数据不会改变方差和，因为它具有无误差统计数据。因此，可以保证每次Lake的插入都会使方差和增量最小化。 **错误界限。**然后，我们讨论Lake的估计误差。考虑一个$d * l$大小的CocoSketch在轻量级部分。我们使用$R(e)$来表示负载e的相对误差。下面的定义证明的$R(e)$在轻量级部分的误差界。 **引理2。**让$l = 3·ϵ^{-2} and d = O(logδ^{-1})$。对于多类型负载$l_p < l_F$中的任意负载e，我们有$$\mathbb{P}\Bigg[R(e)≥ϵ·\sqrt{\frac{f(\overline{e})}{f(e)}}\Bigg]≤δ(4)$$ *证明。*对于存储在重部分中的热负载，只有在插入到重部分之前在轻部分中发生哈希冲突才会产生错误。由于轻量级部分被冷负载所

主导，哈希冲突较少，因此产生的错误也较少。同时，轻负载中的值占热负载的比例很小，因此热负载的误差可以忽略不计。在最坏的情况下，即重部分中的桶在插入后未更新。通过定理2仍然可以获得R(e)的误差界。

定理1。让$l = 3·ϵ^{-2} and d = O(logδ^{-1})$。对于多类型负载$k_p < k_F$中的任意负载e，我们有$$\mathbb{P}\Bigg[R(e)≥ϵ·\sqrt{\frac{f(\overline{e})}{f(e)}}\Bigg]<δ(5)$$ 总的来说。我们可以我们可以得出结论，Lake的$R(e)$的误差界可以通过定理3得到。

E. 基于查询的负载感知

我们提供了一种前端语言，让用户表达他们对负载感知的要求。该语言被翻译成P4代码，同时受益于硬件优化。查询语言。在控制平面中，我们首先通过查询Lake上记录的完整关键流量来构建一个包含两列（Full Load，Size）的表格。对于重负载部分，我们直接将完整负载及其值放在表格中。对于轻负载部分，我们像原始硬件版本[24]一样将不同数组中的估计大小的中位数作为其最终估计大小。我们定义了SQL语句接口来查询感知结果：

```
SELECT g(l_F), SUM(Value)
FROM Table_light A
INNER JOIN Table_heavy B
ON A.Key = B.Key
GROUP BY g(l_F)
```

其中$g(l_F)$是从l_f到l_p的映射函数。

F. 硬件优化

虽然Lake在使用随机方差最小化和热负载分离的情况下可以准确地计数多种类型的负载，但通过键值存储的方式在硬件平台上实现时可能不够高效或无法成功部署，以解决这一问题，我们去除了元数据依赖性并提供了一种聚合重负载的原则。 **元数据依赖。**在键值存储中，键的长度可能超过寄存器数组的最大位宽。为了存储键，我们将一个键分割成多个片段，并使用多个寄存器数组分别存储每个片段。然而，多个寄存器数组可能依赖于一个元数据来索引数组中键的位置。因此，一个寄存器数组必须等待另一个寄存器数组处理元数据。多个寄存器数组依赖于一个元数据，因此它消耗了大量的流水线级别。例如，为了在轻量级部分中计算一个 184 位的 full load（例如 L），我们使用一个 32 位寄存器数组来计算值，4 个 32 位寄存器数组来存储键，以及一个 1 位、7 位、16 位、32 位寄存器数组来存储 WRITE/READ 类型、通道 ID、端口、IP 地址。此外，在重量级部分（哈希表）中，我们还应使用 9 个寄存器数组来存储完整负载。这些寄存器数组依赖于之前计算键的索引的哈希函数，导致多个寄存器数组依赖于一个元数据并消耗大量的阶段。 **并行化索引计算。**为了消除元数据依赖，我们利用RMT的架构并行计算索引。具体而言，我们使用多个哈希函数并行计算多个不同的索引，使得每个寄存器数组都能够在没有依赖于一个元数据的情况下获得索引。虽然并行性会消耗一些元数据值，但与SRAM块的总预算（例如10MB）相比，这是可以忽略的。我们在评估中进行了演示（实验1）。 **参数配置。**用户可能有不同的资源预算和SLO（服务级别目标），他们可以使用SketchGuide [19]来配置Lake，以在给定的资源预算下最大化SLO。

5、利用Lake实现负载平衡

Table III. The notations of Lake.

Symbol	Description
H	The hash table in the heavy part;
$h(.)$	The hash function of the hash table;
$H[i]$	The i^{th} bucket of hash table;
$H[i].K$	The key field in $H[i]$;
$H[i].V$	The value field in $H[i]$;
E	The exact table in the heavy part;
$E[i]$	The i^{th} bucket of the exact table;
$E[i].K$	The key field in $E[i]$;
$E[i].V$	The value field in $E[i]$;
C	The CocoSketch in the light part;
d	The number of arrays in CocoSketch;
l	The number of buckets in one array;
$h_i(.)$	The hash function of the i^{th} array in the light part;
$C_i[j]$	The j^{th} bucket in the i^{th} array in the light part;
$C_i[j].K$	The key field in $C_i[j]$;
$C_i[j].V$	The value field in $C_i[j]$;
$P()$	Probability substitution function;

Lake提供了高效的多类型负载感知技术，适用于网络中的键值存储。基于负载统计数据，本节中，我们综合并确定了4种优化技术，如表III所示。我们的目标不是提出一种负载均衡的最优算法，而是展示使用Lake进行简单调度的负载均衡的好处。对于三种类型的负载，我们利用这些结果来解决性能瓶颈。此外，我们定义了每种技术的原语，即基本和自动的P4代码。由于空间限制，我们在表IV中列出了技术的适用性。

Table IV. Lake Optimization Comparation

No	Load Type	Load awareness	Load Balancing	Thresh	Primitive	Applicability
O1	Network	Available bandwidth	Schedule the underutilized channels firstly	T_c	`load-bal`	cache/route/replication
O2	Storage	Server loads	Schedule the underutilized servers firstly	T_s	`re-route`	route/replication
O3	Storage	Process loads	Schedule the underutilized process firstly	T_p	`re-route`	route/replication
O4	Content	Write-intensive items	Eliminate write-intensive items in cache	T_w	`filter_key`	cache/route

A. 网络负载

通过网络负载感知，我们计算通道负载和每个项使用的通道。 **优化1（O1）：首先安排未利用的通道以避免网络拥塞。**由于网络拥塞可能会降低系统吞吐量，我们计算每个通道的请求次数，并将请求安排到未利用的通道中以避免拥塞的通道。具体来说，超过阈值Tc的访问次数的通道被认为是过载的。然后，我们将请求安排到那些未利用的通道中（基本操作load-bal）。

B. 存储负载

通过对存储负载的感知，我们计算服务器和进程负载，并重新路由请求以最大化并行性。 **优化2（O2）：首先安排未利用的服务器以平衡存储负载。**我们计算服务器负载并将请求安排到未利用的服务器上。具体来说，被识别为过载的服务器是访问频率超过阈值T_s的服务器。我们重新路由请求到未利用的服务器上（基本操作re-route）。在存储服务器中，键可以有多个副本，O2只修改服务器访问的优先级以确保正确

性。 **优化3（O3）：首先安排未利用的进程以平衡存储负载。**类似于O2，我们计算并安排存储负载，但存储节点是一个进程。具体来说，阈值被记录为T_p，基本原操作为re-route。O3只修改进程访问的优先级以确保正确性。

C. 内容负载

在内容负载感知中，我们计算写入密集的项目。 优化4（O4）：消除缓存中的写入密集型项目，以计数更多的读取密集型项目。网络内部缓存的一个限制是它不能提高写入项的吞吐量。更糟糕的是，写入密集的项可能占用缓存，减少缓存读取密集型项的机会。因此，我们可以计算出频率超过阈值T_w的写入项，并在网络内缓存中过滤它们（基本操作filter_key）。写入密集型项目将触发缓存未命中并直接转发到服务器。这些写入密集型请求在交换机故障时会丢失，但这些请求会直接转发到服务器，不需要机制保证一致性。

D. 将O1-O4一起使用

为了提供硬件优化，我们识别出了基本操作，以便消除重复的代码并减少硬件资源的消耗。例如，我们将 O1 到 O4 放在一起，首先从客户端中提取请求（提取），然后将这些请求进行缓存（缓存）或路由（路由）。然后，我们使用 Lake 进行多类型负载感知（负载感知），并从控制平面的缓存中过滤出写入密集型键（O4，filter_key）。接下来，控制器基于感知结果平衡负载（O1，load_bal）。然后，我们通过将这些项目重新路由到未被充分利用的服务器（O2，re-route）或进程（O3，re-route）来进行调度。

6、实现和评估

我们实现和评估了Lake，表明：• Lake使用可接受的硬件资源占用，在Tofino交换机中计算多类型负载(Exp#1)。• Lake具有可忽略的控制平面通信开销，可以计算多类型负载(Exp#2)。• Lake在相同的资源配置下，随着请求数量的增加，呈现出很高的可扩展性(Exp#3)。• Lake在相同的资源配置下，随着负载数量的增加呈现出更高的可扩展性(Exp#4)。• Lake在相同的资源配置下，比同类最新作品更准确地计算多类型负载(Exp#5)。• Lake在不同的工作负载偏斜情况下，以相同的资源配置，表现出很高的稳健性(Exp#6)。• Lake在不同的写入比率下，以相同的资源配置，表现出很高的稳健性(Exp#7)。• Lake使用可接受的估计误差计算多类型负载并识别瓶颈(Exp#8)。• Lake通过O1到O4改进系统吞吐量和减少资源占用(Exp#9)。

方法

测试环境。我们的测试环境包括两台服务器和一个6.5Tbps Barefoot Tofino交换机。每台服务器配备一个20核CPU（Intel Xeon Silver 4210R）和64 GB总内存。两台机器都配备了一个100G NIC（Mellanox MT27800）。一台机器作为客户端生成键值查询，并将请求发送到另一台作为键值存储服务器的机器上。 **工作负载。**我们使用具有不同偏斜度参数（即0.8、0.85、0.9、0.95、0.99）的Zipf分布的偏斜工作负载，这是测试键值存储的典型工作负载[4,26]，并且得到了实际部署[1,27]的证明。我们在大多数实验中使用Zipf 0.99来证明Lake即使在极端情况下也提供高精度。我们还展示了即使在较少偏斜的工作负载（例如Zipf 0.8）下，Lake仍然提供低估计误差。我们在客户端中使用近似技术来快速生成Zipf分布下的查询[4,28]。我们使用16字节的键和128字节的值。键空间在存储服务器之间进行哈希分区[29]。写入比率从0.1到0.9不等。为了计算多类型负载，我们意识到了section III.A节中相同的L。 **度量标准。**我们使用以下准确性指标来评估Lake的L，包括召回率$RR = \frac{TP}{TP + FN}$，精确率$PR = \frac{TP}{TP + FP}$，F1-score,$F1 = \frac{2·P·R}{P + R}\frac{1}{|Ψ|}$，并且平均相对误差（ARE）$Σ_{e ∈ Ψ}\frac{|f(e) - \widehat{f}(e)|}{f(e)}$。 硬件资源有限。一旦资源占用超过总资源限制，程序将无法成功编译。**• SRAM。使用表格/数组跟踪/计数状态，两者都会消耗SRAM块。• SALU。表格/数组读/写内容或维护哈希调用，两者都会消耗SALU。• Hash call。键被哈希为特定计数器，消耗预先分配的哈希调用资源。• Stages。P4程序在有限的资源上编译和分配到交换机流水线上。**

7、实验测试平台

我们在一台Tofino交换机中实现了Lake [10]，并具有500KB的内存预算。重型部分使用第IV.F节中的技术进行了优化。确切的表格具有1280个条目。剩余的内存预算全部被轻量级部分中的CocoSketch所消耗。 **实验1：硬件资源。**我们展示了Lake能够有效地计数多种类型的负载。我们将NetCache的感知部分作为基准，即CM Sketch和Bloom Filter，并消除其他功能。我们遵循原论文中的相同配置。对于每个负载，我们使用相同配置的CM Sketch（遵循A1）进行增量计数。如表V所示，我们计数两种负载：（1）热键（NC1）和（2）热键和$addr_{phy}$（NC2）。NC2消耗昂贵的91.67%的管线阶段，并无法编译3个负载（key_w）。而Lake在消耗一定硬件资源的同时，计数了所有5个负载。此外，Lake在大多数情况下显著降低了资源消耗，除了SRAM和Meter ALU比NC1更高。但Lake的资源消耗不比NC2更高，同时还测量了3个以上的负载。由于随机变量最小化，Lake实现了资源效率，从而降低了所有负载的误差。

Table V. (Exp#1/#2) Hardware resource utilization of Lake, which count 5 loads and mostly reduces resource consumption than NetCache measuring hotkeys (NC1), NetCache measuring hotkeys and hot processes (NC2). We list the resource reduction with NC1 (R1) and NC2 (R2).

Resource Type	NC1(%)	NC2(%)	LA(%)	R1(%)	R2(%)
Exact Match xbar	9.83	11.39	6.38	35.10	43.99
Hash Bits	5.37	7.93	4.81	10.43	39.34
Hash Dist Unit	20.83	31.94	13.89	33.32	56.51
SRAM	5.21	8.96	5.52	-5.95	38.39
Map RAM	7.81	14.06	7.12	8.83	49.36
TCAM	0.00	0.00	0.00	0.00	0.00
Meter ALU	14.58	22.92	22.92	-57.20	0.00
Stats ALU	0.00	0.00	0.00	0.00	0.00
Stages	58.33	91.67	41.67	28.56	54.54

实验2：通信开销。我们展示了第三节中的A4可能会受到控制面带宽和感知准确性之间的权衡的限制。我们使用Pktgen [30]发送了1M个数据包，速率为10K。Tofino交换机的数据面采样数据包并将其报告给交换机操作系统。图7显示，随着采样率的增加，由于控制平面带宽的限制，交换机操作系统丢弃的数据包数量增加。交换机操作系统接收到的最大数据包约为50%。这意味着只有约50%的数据包会有益于感知结果。Lake保持低丢包率，因为交换机操作系统只读取Lake寄存器，这消耗带宽非常有限。

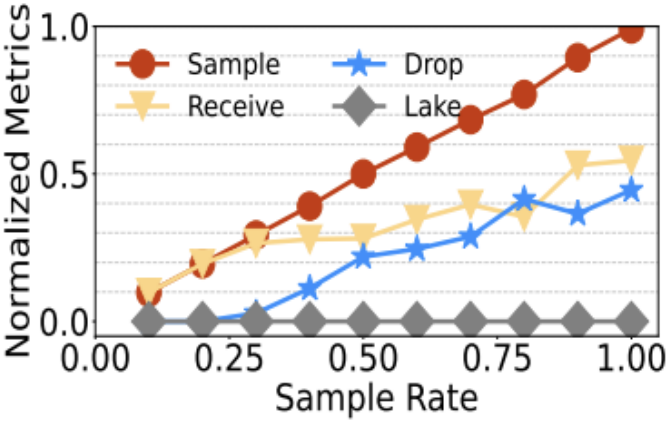


Fig. 7. (Exp#2) Control plane received fewer packets due to the bottlenecks of communication bandwidth.

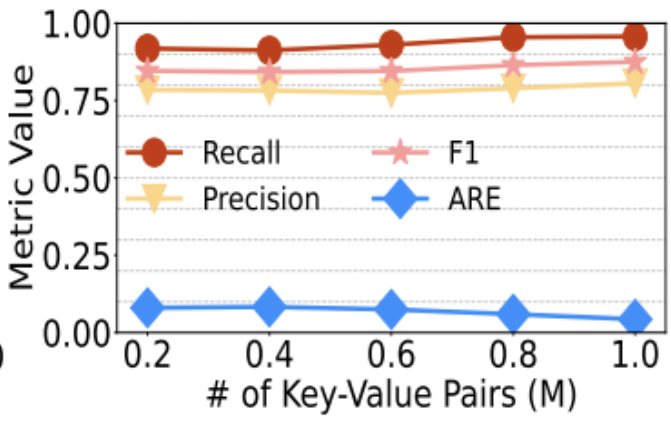


Fig. 8. (Exp#3) Lake maintains low estimation errors under the different number of requests.

A. Lake可扩展性

我们根据不同的内存预算计算不同数量的负载，以展示Lake的可扩展性。我们将CocoSketch[24]、Count-Min Sketch[23]、Count Sketch[31]、Elastic Sketch[21]和UnivMon[22]等一些最先进的sketch作为基线进行比较。我们在基于Python的模拟器中运行Lake和这些sketch，该模拟器在行为上等同于数据平面程序。模拟器允许我们在不同的配置下自由运行许多精度实验。每个sketch都有500KB的内存预算，并保持先前的工作中相同的配置[24]。我们将阈值设置为负载总数的10^{-4}。 **实验3：Lake的可扩展性。**为了测试Lake的可扩展性，我们改变了请求数量。随着请求数量的增加，估计误差保持稳定甚至略微减少。Lake是可扩展的，因为重量部分防止多种类型的负载共享同一个计数器，从而可以准确地计算重载。此外，当键值对的数量增加时，重部分还可以增加准确性，因为当重载的计数增加时，负载可以被过滤到重部分。 **实验#4：多类型负载可扩展性。**如图7所示，我们逐步为每个草图添加新负载。随着负载数量的增加，一些单键草图（即CM、Count sketch、Elastic sketch和UnivMon）会受到性能惩罚。尽管它们的精度比其他草图高，但几乎无法识别热负载（召回率较低）。Lake和CocoSketch在所有准确性指标上都保持高可扩展性。Lake始终具有比CocoSketch更高的性能，因为Lake将重载从轻部分中过滤出来，因此可以准确地识别重负载和轻负载。因此，基于网络的键值存储系统可以由于对热负载的准确识别而做出更精确的决策。 **实验#5：内存可扩展性。**我们改变内存预算，其他设置与实验#3相同。如图9所示，Lake在相同的内存预算下比其他草图更准确。此外，Lake具有比CocoSketch更低的ARE，因为重部分过滤了重负载从轻部分过来的两部分，以确保重负载的准确性。

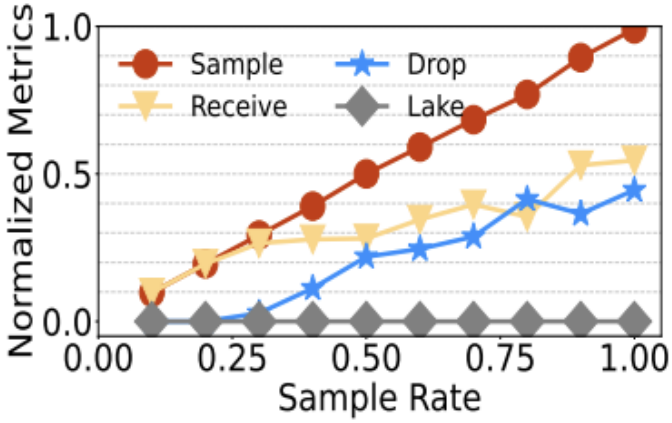


Fig. 7. (Exp#2) Control plane received fewer packets due to the bottlenecks of communication bandwidth.

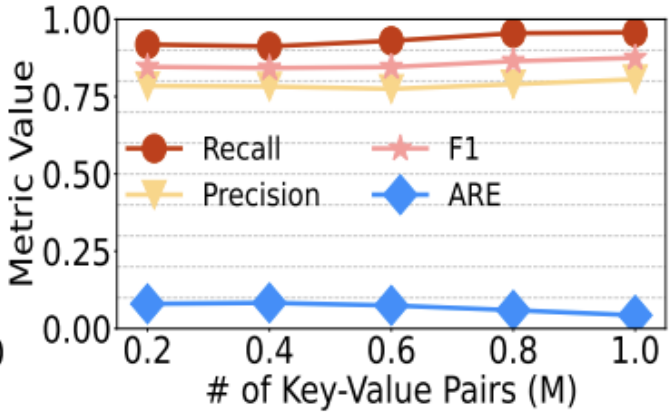


Fig. 8. (Exp#3) Lake maintains low estimation errors under the different number of requests.

C. 负载平衡

实验8：负载感知。我们使用Lake计数5个负载。总体而言，Lake保持高精度，但对于不同的负载表现不同。造成这种差异的原因是，具有较大键空间的负载可能消耗更多的内存占用，并引发更多的哈希冲突。此外，Lake从轻负载中过滤出重负载，从而确保重负载具有高精度。 **实验#9：负载平衡。**我们证明简单的优化技术可以从网络内键值存储的已识别优化技术 (O1-O4) 中受益。我们使用一致性哈希（CH）作为基准，并逐步在系统上添加优化技术（例如，“+O3”表示我们在系统上部署了O1、O2和O3）。通过多类型负载感知，我们可以在多个约束条件下部署多个优化技术。

8、相关工作

A. 网内负载感知

网内键值存储是改善高度不均衡和动态工作负载下负载平衡的有效的方法。代表性的方法包括网络内缓存（如NetCache[3]和DistCache[7]）、路由（如SwitchKV[4]和AppSwitch[32]）和复制（如Pegasus[5]和KVSwitch[33]）。然而，这些方法可能无法高效地计数多类型负载（在第III节中进行了演示），更不用说解决每个负载中的性能瓶颈。Lake能够有效地识别多类型负载，提供了解决这些瓶颈的机会。

B. 负载平衡

传统的方法，如一致性哈希和虚拟节点，在处理变化的工作负载时存在不足[3]。 "二次选择的力量" 和数据迁移策略[9，35]可以平衡动态工作负载，但会引入额外的系统开销或难以处理的高度倾斜的工作负载[3]。Lake通过实现高效的多类型负载感知，使得键值存储系统可以使用简单的技术在网络中高效地优化负载平衡。

9、总结和未来的工作

本文提出了一种名为Lake的框架，以实现高效的多类型负载感知，用于网络内键值存储。Lake消耗可接受的资源，并具有高准确性。在不同的工作负载分布下，Lake具有高度的可扩展性和稳健性。基于负载统计数据，键值存储可以快速识别和解决瓶颈问题。该框架允许用户为负载感知进行定制并自由调度工作负载。未来，我们计划将Lake扩展到多个机架，并为网络内键值存储设计更多的优化技术。

加了一个模块。 可视化一下，给用户函数，让用户可以优化自己的键值存储系统，加具体的例子，加入生活中的例子。