基于可编程交换机的高效键值存储监控系统

设计文档

所在赛道：A

目录

一. 研究内容

1.1 研究背景

键值存储是许多现代互联网服务（如电子商务和社交网络）存储基础设施的基本组成部分。这些服务通常以键值格式存储可访问的项。通常，用户通过三个步骤来获取这些项：（1）用户向服务发送带有项键的请求；（2）这些服务的存储系统根据键寻址项；（3）系统将项的值发送给用户。键值存储已被广泛应用于存储基础设施的生产中，例如 Redis [1]和 Memcached [2]。

实际上，这些在线服务对键值存储的 SLO 即 Service Level Objective（服务水平目标）的缩写，它是一种用于衡量和规定服务质量的指标（例如在 10-100 微秒延迟内为数十亿请求提供服务）有严格的要求。满足 SLO 的一个主要挑战是热点问题，即一小部分受欢迎的项相较于其他项被服务不成比例地访问。例如，类似突发新闻的实时事件可能导致 90%的访问集中在生产中的仅 1%的项上 [5]。在运行时，大量的热点请求涌入少数存储服务器，导致超载和尾部延迟增加 [6-8]。热点问题越来越严重且昂贵：每 100 毫秒的加载延迟会导致亚马逊销售下降 1%，而 Google 搜索每增加 500 毫秒的加载延迟会损失 20%的流量 [9]。

热点监控通过收集工作负载信息并确定热点及其特征。监控为快速和全面的热点缓解提供了必要的信息。本项目研究了现有的工作并确定了监控要求。理想情况下，管理员需要可扩展且多功能的热点监控。就可扩展性而言，监控系统应该在高请求率（例如每秒十亿个请

求）和更大的群集规模（例如数千台机器）时迅速检测热点[10]。对于多功能性，现有的工作对不同的工作负载类型（例如内容[5、6、8、11-23]、操作[5、8、13-15、17、21、23]、大小[5、15-18、23]）和过载位置（例如服务器[11-15、17-25]、信道[16]和交换机[6、8]）进行了热点检测。因此，管理员可以快速监控具有工作负载类型的热点、准确定位过载并采取综合性的缓解措施。

然而，现有的工作未能同时满足这两个要求。首先，基于端主机的解决方案在服务器上监控数据并在控制器上分析数据。它们缺乏对交换机的可见性，因此可能无法准确定位交换机内部的过载，例如网络内部缓存[6]和一致性目录[8]。此外，基于通用 CPU 的控制器在交换机与控制器之间的通道带宽有限，可能会在扩展到更多请求和机器时成为性能瓶颈（例如 PCIe 通道或 TCP 连接）[26-28]。其次，一些工作[6、8]在交换机上监控热点，但由于交换机资源有限，仅能识别有限的工作负载类型和过载位置。更确切地说，巨大的项键空间（例如 16B 键的 2^{128}）和位置地址空间（例如 IP 地址的 2^{32}）可能无法适应交换机资源。基于 sketch 的解决方案通过压缩监控数据来减少资源消耗，但仅支持少量工作负载类型）。尽管基于样本的工作[8]减少了开销，但在有限的控制平面带宽下可能丢失大量信息（在有限的控制平面带宽下的 44.54％丢失率）。

因此，一个基本而实际的研究问题是：能否设计一个系统以支持可扩展且多功能的热点监控？**本项目提出了 SpotMon，这是一个可扩展的系统，可以检测具有工作负载类型和过载位置的热点，旨在快速和全面地进行热点缓解。**SpotMon 利用**可编程交换机**在数据平面收集

和聚合项信息。交换机 ASIC 的高数据包处理速率（即>10Tbps 的 ASIC[29]）允许 SpotMon 实时聚合监控数据。因此，只有极少的监控数据会传输到并由控制器处理。这种线速率的聚合允许在扩展到更多请求和更大集群规模时，仅增加很少的监控数据量。此外，交换机具备天然的机架级可见性，使 SpotMon 能够观察所有传入的请求、工作负载类型以及请求将遍历的位置（例如交换机、通道和服务器）。这种对交换机的可见性为多功能的热点监控提供了独特的机会。

然而，一些可扩展性问题使得设计 SpotMon 变得具有挑战。具体而言，（C1）配备通用 CPU 的控制器可能无法满足键值存储中的 BQPS 项[6]要求。（C2）交换机与控制器之间有限的通道带宽可能会被监控数据淹没。（C3）可编程交换机的有限资源使得多功能性监控具有挑战性。

为了应对这些问题，本项目的核心思想是**多功能热点查询的抽象和基于 sketch 的数据结构**在可编程交换机上的应用。具体而言，SpotMon 在数据平面上完全聚合监控数据。这种设计既减少了通道传输，又减少了控制器的计算量，以解决 C1 和 C2 问题。对于 C3，本项目提出了基于任意部分键查询[30]的多功能热点查询的抽象。该抽象将键、工作负载类型和过载位置连接为一个集合。因此，本项目可以对集合进行计数，并查询其任何子集，例如通过交换机端口传播的写入类型项的计数。更确切地说，为了支持此抽象，本项目设计了一个基于 sketch 的数据结构，该结构受到最小化所有子集的随机变化的理论支持。因此，本项目提高了多功能监控的准确性，同时有效利用了

交换机资源。

SpotMon 还通过一种分层查询算法支持一组易于使用的查询接口。通过这种方式，系统操作员可以表达他们关心的内容：哪些项是热点、热点的工作负载类型是什么，以及热点导致的过载位置在哪里。本项目在 Barefoot Tofino 交换机上进行了广泛的实验。结果显示，SpotMon 能够精确地识别带有五种工作负载类型和三个过载位置的热点，共计十八种查询类型。它消耗微不足道的资源占用（例如，控制器的监控数据丢失为零），并且能够实现有效的负载均衡技术（例如，高达 4.03 倍的 MQPS）。

1.2 研究内容

（1）通过 LaSketch 数据结构，在数据平面上聚合监控数据，提供监控平台。LaSketch 数据结构的设计充分利用了交换机的计算能力，使得监控数据能够在网络中被高效地汇总和分析。这种数据平面聚合的方法不仅大大减少了监控数据传输到控制器的负载，还显著降低了计算控制器上的工作量。使得监控平台的资源消耗得以控制在可接受的范围内，特别是在可编程交换机的资源限制下。这意味着本项目能够在不牺牲准确性的情况下，实现高效的监控和分析，为系统的性能和稳定性提供保障。

（2）通过提供用户接口实现用户实时的热点查询的功能。SpotMon 实现了用户对系统中热点数据的实时查询功能。这个用户接口允许用户根据不同的查询需求，灵活地获取系统中不同类型的热点数据信息。SpotMon 通过其中的数据结构和算法，使用户可以轻松地

监控和了解系统中各个负载类型的热点情况，从而为资源分配、性能优化以及负载均衡等决策提供了有力的支持。

（3）实现四种优化方案实现对热点负载的综合性优化。SpotMon 通过接下来将会提到的四种优化方案，实现了对热点负载的综合性优化。这些优化方案相互协同工作，共同实现了对热点负载的全面优化，提高了分布式键值存储系统的性能和资源利用效率。

1.3 应用场景和价值

（1）应用场景

如 1.1 所述，本项目主要应用于电商、搜索引擎、社交网络等需要高性能在线服务的平台。SpotMon 系统可以检测具有工作负载类型和过载位置的热点，旨在快速和全面地进行热点缓解。

（2）价值

学术价值：1、提出了快速和全面进行热点缓解的系统：SpotMon 系统为解决热点问题提供了一种全新的方法，通过在网络中利用可编程交换机来实现热点监控和缓解，从而实现了快速和全面的热点缓解策略。这一方法不仅创新地结合了网络技术和数据存储技术，还为热点问题的解决提供了一种高效、可扩展的方案。

2、推动了可编程交换机和键值存储系统的研究：SpotMon 的实现基于可编程交换机和键值存储系统，这两种技术都是当前网络领域的研究热点。SpotMon 的成功应用推动了这两种技术在实际网络中的应用和研究，同时也为其他研究者提供了一个很好的思路和方法。

3、为数据中心网络的设计和优化提供了借鉴：SpotMon 是一种新

型的数据中心网络架构，其设计和优化思路都可以为其他数据中心网络的设计和优化提供借鉴。例如，SpotMon 中的负载感知和调度技术可以应用于其他数据中心网络中，以优化网络性能和资源利用率。

社会价值：1、支持了关键业务服务的性能优化：SpotMon 系统的提出为在线服务（如电子商务等）的性能优化提供了一种新的路径。通过快速、准确地识别热点情况，系统管理员可以采取针对性的缓解策略，确保关键业务服务的稳定性和高性能。

2、降低了系统管理和调优的复杂性：SpotMon 系统为系统管理员提供了一种简单易用的查询接口，使其能够轻松地监测和分析不同类型的热点情况。这一功能降低了系统管理和调优的复杂性，帮助管理员更好地理解和优化系统的性能。

3、为未来网络优化提供了新思路：SpotMon 系统的设计理念和方法对于未来网络的优化具有一定的启示意义。通过在网络中引入可编程交换机来实现监控和缓解功能，可以为网络中的其他问题提供创新的解决方案，进一步推动网络技术的发展和进步。

二. 设计思路与方案

2.1 总体设计思路

SpotMon 模型主要由以下几个模块组成，并且每一模块所解决的问题如下。

1. 数据平面模块：这个模块位于可编程交换机的数据平面，实时监控通过网络传输的数据流量。它使用流量采样技术，对数据流进行抽样和聚合，将采样的监控数据传递给控制平面模块。通过在数据平

面进行监控，可以及时捕捉到网络中的热点负载情况。

2.控制平面模块： 控制平面模块负责接收来自数据平面的监控数据，并进行数据处理和负载分析。它使用了 SpotMon 的核心算法 Lake，识别并区分不同类型的负载，包括重负载和轻负载。控制平面模块通过分析监控数据，识别出热点负载、负载类型和负载位置，为后续的查询和优化提供基础。

3.Lake 模块: Lake 是 SpotMon 的核心组件,使用了 Bloom Filter 和 Count-Min Sketch 等数据结构。它负责对监控数据进行处理，将数据流划分为重负载和轻负载，并利用精确统计和估计的方法进行热点负载的监测。Lake 的设计使得系统可以准确地感知热点负载情况，为后续的查询和优化提供了准确的基础。

4.多功能热点查询模块： 这个模块为系统操作员提供了多类型负载感知查询的接口。操作员可以根据需求查询不同类型的热点负载信息，从而实现精细的负载监控和管理。这个模块使得系统操作员可以根据具体情况进行查询，了解不同类型的热点负载的情况从而采取相应的措施进行管理和优化。在其中实现了一种层次化查询算法，用于识别热点负载。算法从粗粒度到细粒度逐步进行查询，同时避免了不必要的查询操作，从而提高了查询的效率和精确度。这个算法使得系统可以快速而准确地识别出热点负载。

5.热点优化模块: SpotMon 采用了四种优化技术来针对热点负载进行综合性的优化。这些技术包括通道平衡、服务器平衡、进程平衡、消除写入项目。通过根据热点负载信息进行系统性能的优化，可以提

高系统的吞吐量和响应时间。

2.2 可编程交换机

可编程交换机是一种网络设备，它能够在数据包转发过程中执行用户定义的程序，以实现网络行为的配置和调整。可编程交换机现在已经被十分广泛的应用在各个方面，如网络流量控制、安全和远程监测等。与以往的交换机相比，可编程交换机具有以下几个特点：

1.灵活性和可编程性：可编程交换机允许用户定义数据包处理逻辑和控制方法，以满足不同的应用需求，并且能够随时修改和更新。

2.高效性：可编程交换机的处理速度要比通用处理器快得多，因为其处理逻辑是集成在硬件中的。

3.可扩展性：可编程交换机可以与其他网络设备组合使用，以扩展其处理能力或添加其他功能。

2.3 热点

当将分布式键值存储扩展到数十亿客户端时，面临的主要挑战是在高度不均衡和动态的工作负载下满足严格的 SLO。具体来说，(1)高度不均衡时，受欢迎的项目即热点接收的请求数量（每天数百万）远远多于其他项目（几乎没有），(2)对于动态分布，热点的集合动态变化，很快变得不受欢迎<10min。热点可能会突然聚集到少数服务器上，导致服务器超载和性能下降。

实际上键值存储系统的工作负载通常使用 Zipfian 分布进行建模，通常表现出高度不均衡的模式（例如，α>1 的 Zipf 分布）。键值存储系统将项目分区并分发到多个服务器以确保 SLO。不均衡的工

作负载主要包括不均衡的内容负载、网络负载和存储负载。这些不均衡的负载可能会超载少部分资源，这可能成为新的瓶颈并引入更长的尾部延迟。过度提供资源可以减少性能惩罚，且显著增加总体成本。

2.4 网内键值存储

网内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。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）而能够实时平衡负载。这些系统可以为键值存储提供高吞吐量、低延迟和可扩展性。接下来，本项目将详细阐述每个典型的工作流程，分为两个阶段：负载感知和负载平衡。

2.4.1 网内缓存。NetCache 将 ToR（顶层交换机）作为负载平衡缓存（中等缓存命中率<50%），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 CMsketch 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。

2.4.2 基于内容的路由。SwitchKV 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。

2.4.3 网内复制是指 Pegasus 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。

2.5 随机方差最小化

由于将热负载和冷负载分离，SpotMon 在大多数情况下具有高准确性。在分析之前，本项目将要定义如下符号。本项目用 $l(e)$ 表示负载 e 的真实值，用 $\hat{l}(e)$ 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。本项目首先分析负载方差和的最小化。本项目的最终目标是最小化负载综合的估计。

$$minimize \sum_{e} (l(e) - \hat{l}(e))^2 (1)$$

在轻量级部分，本项目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。

$$minimize \sum_{e} \Delta (l(e) - \hat{l}(e))^2 (2)$$

本项目确保每次插入引起的方差变化最小化，以便 SpotMon 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

$$\sum_e \Delta(l(e) - \hat{l}(e))^2 = \begin{cases} 2wl_j, e_i \neq e_j \\ 0, e_i = e_j \end{cases} (3)$$

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 SpotMon 的插入都会使方差和增量最小化。

错误界限。然后，本项目讨论 SpotMon 的估计误差。考虑一个 *d***l* 大小的 CocoSketch 在轻量级部分。本项目使用 $R(e)$ 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 $l = 3 \cdot \epsilon^{-2} and d = O(log\delta^{-1})$。对于多类型负载 $l_p < l_F$ 中的任意负载 e，有

$$\mathbb{P}\left[R(e) \geq \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] \leq \delta (4)$$

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 $l = 3 \cdot \epsilon^{-2} and d = O(log\delta^{-1})$。对于多类型负载 *kp*

$<kF$ 中的任意负载 e，有

$$\mathbb{P}\left[R(e) \geq \epsilon \cdot \sqrt{\frac{f(\bar{e})}{f(e)}}\right] < \delta(5)$$

总的来说。可以得出结论，SpotMon 的 $R(e)$ 的误差界可以通过定理 3 得到。

2.6 负载符号定义

l_p	热点候选项的部分负载
l_f	热点负载想的完整负载
H	重负载部分中的哈希表
h(.)	哈希表的哈希函数
H[i]	哈希表的第 i 个桶
H[i].K	H[i]中的键字段
H[i].V	H[i]中的值字段
ε	重负载部分中的精确表
ε[i]	精确表的第 i 个桶
ε[i].K	ε[i]中的键字段
ε[i].V	ε[i]中的值字段
C	轻负载部分的 CocoSketch
d	CocoSketch 中的数组数量
l	一个数组中的桶数量
$h_i(.)$	轻负载部分中第 i 个数组的哈希函数
$C_i[j]$	轻负载部分中第 i 个数字的第 j 个桶
$C_i[j].K$	$C_i[j]$中的键字段
$C_i[j].V$	$C_i[j]$中的值字段
P()	概率代替函数

三. 方案实现

如图是本项目的文件目录树。接下来将会具体解释各个关键模块的功能。

```
.
├── BenchMark.h
├── CMakeLists.txt
├── Common
│   ├── MMap.h
```

```
|       ├── Util.h
|       └── hash.h
├── Multiple
|       ├── LASketch.h
|       ├── MultiAbstract.h
|       ├── MultiAbstract2.h
|       ├── OurHard.h
|       ├── OurSoft.h
|       ├── USS.h
|       └── WASketch.h
├── Single
|       ├── CMHeap.h
|       ├── CountHeap.h
|       ├── Elastic.h
|       ├── SingleAbstract.h
|       ├── SpaceSaving.h
|       └── Univmon.h
├── Struct
|       ├── BitMap.h
|       ├── CMSketch.h
|       ├── CSketch.h
|       ├── CuckooMap.h
|       ├── Heap.h
|       └── StreamSummary.h
├── build
|       ├── CMakeCache.txt
|       ├── CMakeFiles
|       |       ├── 3.17.0
|       |       |       ├── CMakeCCompiler.cmake
|       |       |       ├── CMakeCXXCompiler.cmake
|       |       |       ├── CMakeDetermineCompilerABI_C.bin
|       |       |       ├── CMakeDetermineCompilerABI_CXX.bin
|       |       |       ├── CMakeSystem.cmake
|       |       |       ├── CompilerIdC
|       |       |       |       ├── CMakeCCompilerId.c
|       |       |       |       └── CMakeCCompilerId.o
|       |       |       └── CompilerIdCXX
|       |       |           ├── CMakeCXXCompilerId.cpp
|       |       |           └── CMakeCXXCompilerId.o
|       |       ├── CMakeDirectoryInformation.cmake
|       |       ├── CMakeError.log
|       |       ├── CMakeOutput.log
|       |       ├── CPU.dir
|       |       |       ├── CXX.includecache
```

```
│   │   │   ├── DependInfo.cmake
│   │   │   ├── build.make
│   │   │   ├── cmake_clean.cmake
│   │   │   ├── depend.internal
│   │   │   ├── depend.make
│   │   │   ├── flags.make
│   │   │   ├── link.txt
│   │   │   ├── main.cpp.o
│   │   │   └── progress.make
│   │   ├── Makefile.cmake
│   │   ├── Makefile2
│   │   ├── TargetDirectories.txt
│   │   ├── cmake.check_cache
│   │   └── progress.marks
│   ├── CPU
│   ├── Makefile
│   └── cmake_install.cmake
├── convert.txt
├── generator
│   ├── gen_kv.py
│   ├── gen_query_uniform.py
│   ├── gen_query_zipf.py
│   ├── hot.txt
│   ├── kv.txt
│   ├── query.txt
│   └── query_zipf.txt
├── main.cpp
├── query_zipf.txt
└── run.sh
```

3.1 基础数据结构模块

3.1.1 哈希。SpotMon 使用哈希来存储和管理键值对，其主要运用于两个方面：

（1）将 key 映射到哈希中的一个桶，加速查找过程。

（2）桶中存储的是一个指针数组，该数组中的每个元素指向一条 KV 记录。主要在 Common/hash.h 文件中实现。

3.1.2sketch。SpotMon 中多次使用了 sketch，主要运用了 Cocosketch 和 CMsketch。在 Multiple/LASketch.h 文件和

Struct/CMSketch.h 文件中实现。

（1）Cocosketch。SpotMon 在 light part 使用了 Cocosketch 来实现高效的去重和计数操作。具体来说，SpotMon 将每个数据项映射到 Cocosketch 中，每个位置报错了一个计数器和一个指向该计数器的指针。在插入新数据项中，SpotMon 首先会在其中查找是否已经存在相同的数据项，如果存在，则直接增加该数据项对应的计数器；否则 SpotMon 会尝试将新数据项插入到 Cocosketch 中，并利用 hash 来避免冲突。在删除数据项是，SpotMon 会将响应的计数器减少，当计数器减少到 0 时，则将该数据项从其中删除，通过 Cocosketch，SpotMon 可以以较小的空间开销实现高效的去重和计数操作。详情请见 4.3.2。

（2）CMsketch。在 SpotMon 中，CMsketch 被用来实现频率估计功能，他用来估计每个数据块的出现频率，对于每个数据块，SpotMon 会计算出其哈希值，并将其插入到 CMsketch 中。当需要查询某个数据块是否已经出现在数据集中出现过，SpotMon 会进行查询，如果该计数器值超过了阈值，那么就判断其出现过。核心代码请见附录。

3.1.3Priority Queue。SpotMon 使用 Priority Queue 来维护数据流中出现频率最高的元素。通过 Priority Queue，SpotMon 能够高效的找到频率最高的元素，以及快速更新其元素。代码见附录。

3.1.4Bloom Filter。在 SpotMon 中，Bloom Filter 用于过滤掉不可能匹配的 key，从而加速查询的过程。代码见附录。

3.2 Lake 模块

Lake 的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。其主要在 Multiple/LASketch.h 文件中实现。

3.2.1 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**。

接下来是实现 heavy part 的关键代码，在 hash_tbl 中存储出现次数较多的数据项，而在 exact_tbl 中存储出现次数较少的数据项，他们的出现次数是准确的，即 exact count。因此，不需要再在轻量级计数器中继续进行计数，也不需要再随机采样，直接在 heavy part 中进行统计即可。

代码实现：

```
    if(hash_tbl[hash_position].count != 0 &&
hash_tbl[hash_position].ID == item){
            hash_tbl[hash_position].count += 1;
        return ;
    } //判断当前数据项在 hash_tbl 中是否存在，如果存在，则将该数据项在
hash_tbl 中的计数加 1，并直接返回。
    else if(exact_tbl.find(item) != exact_tbl.end()){
        exact_tbl[item] += 1;
        return ;
    }//判断当前数据项在 exact_tbl 中是否存在，如果存在，则将该数据项在
exact_tbl 中的计数加 1，并直接返回。
```

3.2.2 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。

```
    for(uint32_t i = 0;  i < HASH_SUM; ++i){ //遍历哈希函数的数量
        uint32_t position = hash(item, i) % LENGTH;//计算第 i 个哈希函数的哈希值，然后确定其在计数器中的位置
        conter[i][position].count += 1;
        if(randomGenerator() % counter[i][position].count == 0)
          //该条件以 1/count 的概率选择要更新计数器的位置的 ID 值。这可以确保更新的 ID 与目前已存储的 ID 均匀分布，从而降低估计误差
            counter[i][position].ID = item;//将计数器位置的 ID 设置为当前 item
        if(counter[i][position],count > threshold){//如果计数器超过阈值，那么将其从 light part 移到 heavy part
            if(hash_tbl_size < HASHTBL_SIZE && hash_tbl[hash_position]. count == 0){//如果哈希表还有空间，并且哈希表中该位置未被占有，则将该位置的计数器信息移动到哈希表中
                hash_tbl[hash_position] = counter[i][position];
                //将计数器信息赋值给哈希表中
                hash_tbl_size++;
                //增加哈希表的项数
                memset(&counter[i][position], 0, sizeof(counter));
                //将原来位置的计数器信息清零，以释放内存
            }
            else if(exact_tbl.size() < EXACTBL_SIZE){
            //如果哈希表已满，但精度表还有空间，则将该位置的计数器信息移动到精度表
                exact_tbl[counter[i][position].ID] = counter[i][position].count;
                memset(&counter[i][position], 0, sizeof(counter));
                //将原来位置的计数器信息清零，以释放内存
            }
```

```
            }
    }
```

3.3 多功能检测模块

为了提供多样化监测的统一和正式抽象，本项目定义了一个称为多功能热点查询的新问题类别，支持查询热点的键、工作负载类型、超载位置以及它们的任何组合。具体来说，本项目将每种基本查询类型称为负载。本项目将所有负载连接起来形成一个完整的负载。任何负载或负载组合在正式情况下统称为部分负载。

定义 1：（部分负载）。如果存在一个映射函数 $g(.):l_f \rightarrow l_p$，并且对于任意定义在部分负载 l_p 上属于 l_p

的键 e，本项目有 $f(e) = \sum_{e' \in l_f, g(e')=e} f(e')$，其中 f(e)是 e 的统计量。

注意：例如，l_f = (键, 操作, 服务器 ID)（例如，(k_1, 写入, server31）表示带有写入请求的键 k_1 被转发到 server31）。请注意，l_p 可以是 l_f 的任何子集，例如 (key, *, *) 是 l_f 的一个 l_p，其中“*”是通配符。

定义 2：Versatile Hotspot Query。给定一个 l_f 和一个度量函数 f，返回满足 $l_p < l_f$ 的任意键 $e \in l_p$ 的 f(e)值。

注意：这个定义使得本项目能够支持多样化的热点查询。本项目可以通过仅计数 l_f 来自由查询任何 l_p。本项目通过聚合所有具有相同负载的 l_f（例如，(k_1, write, *) = $\sum_{i=0}^{31}$ (k_1, write, $server_i$)）来获得 l_p 的计数。因此，本项目可以查询项目键的计数（k_i, *, *），项目类型的计数(k_i,write,*)以及超载位置的计数(k_i,*,$server_i$)。

3.4 热点优化模块

SpotMon 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，接下来，本项目综合并确定了 4 种优化技术。本项目的目标不是提出一种负载均衡的最优算法，而是展示使用 SpotMon 进行简单调度的负载均衡的好处。对于三种类型的负载，本项目利用这些结果来解决性能瓶颈。此外，本项目定义了每种技术的原语，即基本和自动的 P4 代码。本项目在表 1 中列出了技术的适用性。

NO	负载类型	负载感知	负载平衡	阈值	原语	适用性
优化1	网络负载	可用的带宽	首先安排利用率低的通道	T_c	load-bal	缓存、路由、复制
优化2	存储负载	服务器负载	首先安排利用率低的服务器	T_S	re-route	路由、复制
优化3	存储负载	进程负载	首先安排利用率低的进程	T_p	re-route	路由
优化4	内容负载	写入密集型项目	清除缓存中的写入密集型项目	T_w	filter_key	缓存、路由

表 1SpotMon 优化

3.4.1 优化 1 通道平衡：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，SpotMon 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，SpotMon 将请求安排到那些未利用的通道中。

3.4.2 优化 2 服务器平衡：首先安排未利用的服务器以平衡存储负载。本项目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 Ts 的服务器。本项目重新路由请求到未利用的服务器上（原语 re-route）。在存储服务器中，键可以有多个副本，服务器平衡只修改服务器访问的优先

级以确保正确性。

3.4.3 优化 3 进程平衡：首先安排未利用的进程以平衡存储负载。类似于上一种优化方式，本项目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，原语为 re-route。进程平衡只修改进程访问的优先级以确保正确性。

3.4.4 优化 4 消除写入项目：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，可以计算出频率超过阈值 Tw 的写入项，并在网络内缓存中过滤它们（原语 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。

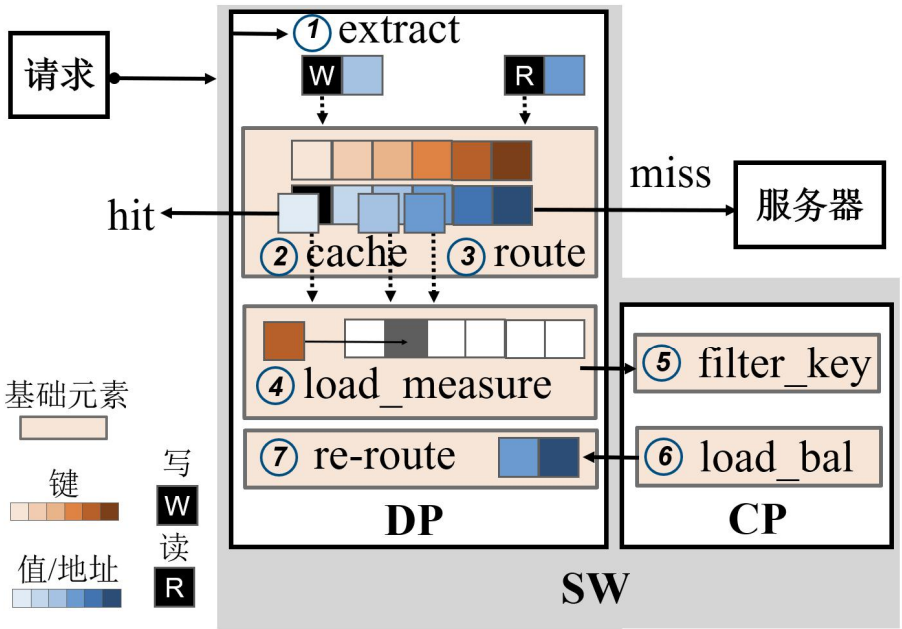
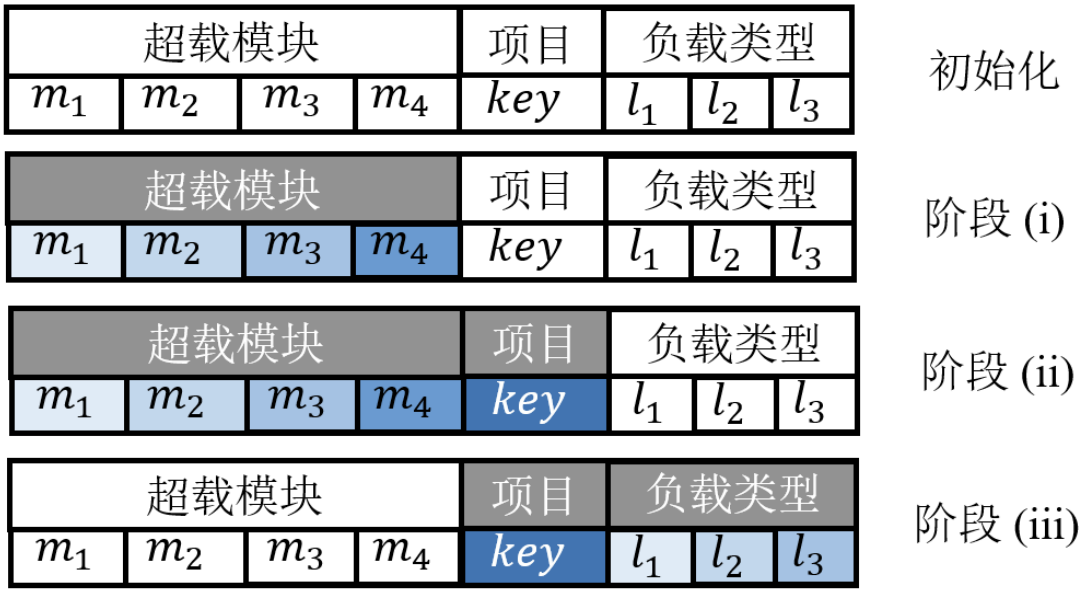


图 3 优化方案

3.4.5 将前四项一起使用。为了提供硬件优化，本项目识别出了基本操作，以便消除重复的代码并减少硬件资源的消耗。例如，本项目将前 4 中优化方式放在一起，首先从客户端中提取请求，然后将这些请求进行缓存或路由。然后，本项目使用 SpotMon 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（04，filter_key）。接下来，控制器基于感知结果平衡负载（01，load_bal）。然后，本项目通过将这些项目重新路由到未被充分利用的服务器（02，re-route）或进程（03，re-route）来进行调度。

3.5 多功能热点查询模块

在该模块中，本项目设计了一个分层查询算法。该算法通过消除不必要的操作来实现快速查询。此外，本项目还提供了一组查询接口。



键分析。在控制平面中，SpotMon 构建了一个包含两列的 lf 表（lf，lf.cnt）。图 4 显示了分层查询算法从粗粒度到细粒度识别热点。当当前阶段没有结果返回时，该过程停止。因此，本项目减少了不必要的查询。具体而言，SpotMon 首先查询超载位置。如

果某个模块超载，SpotMon 会在带有工作负载类型的位置中查询热点。最后，Lake 识别工作负载类型。

超载位置。在第（i）阶段，SpotMon 首先查询超载位置，如交换机端口等。如果 lp 的计数超过阈值，则将 lp 视为超载位置。系统操作员可以估计模块中项目的最大计数并定义阈值。例如，如图 4 所示，Lake 从 m1 到 m4 查询每个系统位置，返回位置的地址。如果没有返回地址，Lake 将停止所有阶段并返回无输出。

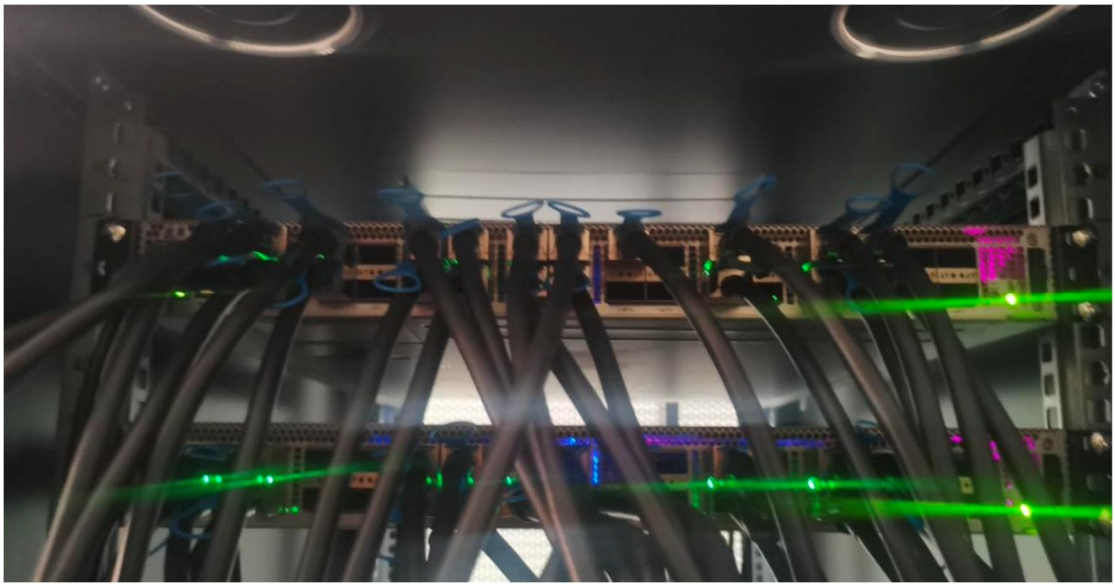
热点查询。在第（ii）阶段，如果存在超载位置，SpotMon 将识别此位置的热点。具体而言，SpotMon 将键和位置组合成 lp。接下来，lp 的计数超过阈值的将被视为热点。

工作负载类型。在第（iii）阶段，SpotMon 查询每个热点的工作负载类型。它将工作负载类型和键都组合成 lp。lp 的计数超过阈值，被认定为具有工作负载类型的热点。

查询接口。SpotMon 提供了查询的统一抽象。系统操作员使用简单的接口来表达查询需求。接口的描述如表 IV 所示。本项目展示了一个查询程序的示例，并因空间有限省略了详细描述。loadInfo

```
.filter(l=>field=(f1, f2))
.map(l=>(f1, f2))
.reduce(keys=(f1, f2), f=sum)
.filter(l.(f1, f2)=>cnt>Th.(f1, f2))
```

四. 运行结果



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。

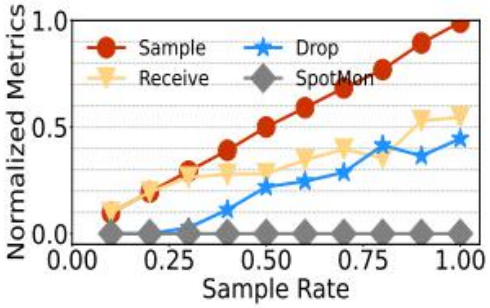
4.1 硬件资源

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

在这里展示了 Lake 的资源效率。本项目将 NetCache 的监控部分（即 CM sketch 和 Bloom 过滤器）作为基准，并消除其他功能。本项目遵循原始论文中的相同配置。对于每个负载，本项目使用相同配置

（遵循 A1）递增一个 CM sketch。如表 V 所示，本项目计算了两个负载：（1）热键（NC1）和（2）热键和 $addr_{phy}$（NC2）。NC2 消耗了昂贵的 91.67%的阶段，并且无法编译 3 个负载（key_w）。Lake 计算了所有 5 个负载，同时消耗了硬件资源的一部分。此外，Lake 大多显著降低了资源消耗，除了 SRAM 和 Meter ALU 比 NC1 要少。但是 Lake 在监控 3 个以上的负载时不会消耗比 NC2 更多的资源。SpotMon 通过随机方差最小化实现了资源效率，从而减少了所有 1_f 的误差。

4.2 通信开销



使用 Pktgen [46]发送了 1 百万个数据包，速率为 10K。Tofino 交换机的数据平面对数据包进行抽样并报告给交换机操作系统。图 5 显示，随着采样率的增加，由于受到有限的控制平面带宽，由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的数据包数量的最大值约为 50%。这意味着只有约 50%的数据包会对感知结果产生影响。SpotMon 保持了低的丢包率，因为交换机操作系统仅读取 SpotMon 的寄存器，这消耗非常有限的带宽。

4.3 可扩展性

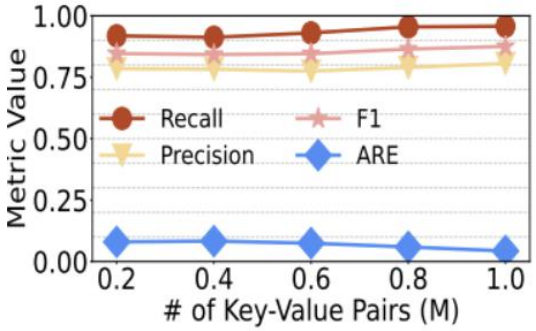


图 5 键值对数量对于 SpotMon 性能的影响

为了测试 SpotMon 的可扩展性，本项目改变了请求数量。随着请求数量的增加，估计误差保持稳定甚至略微减少。SpotMon 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。

4.4 多类型负载可扩展性

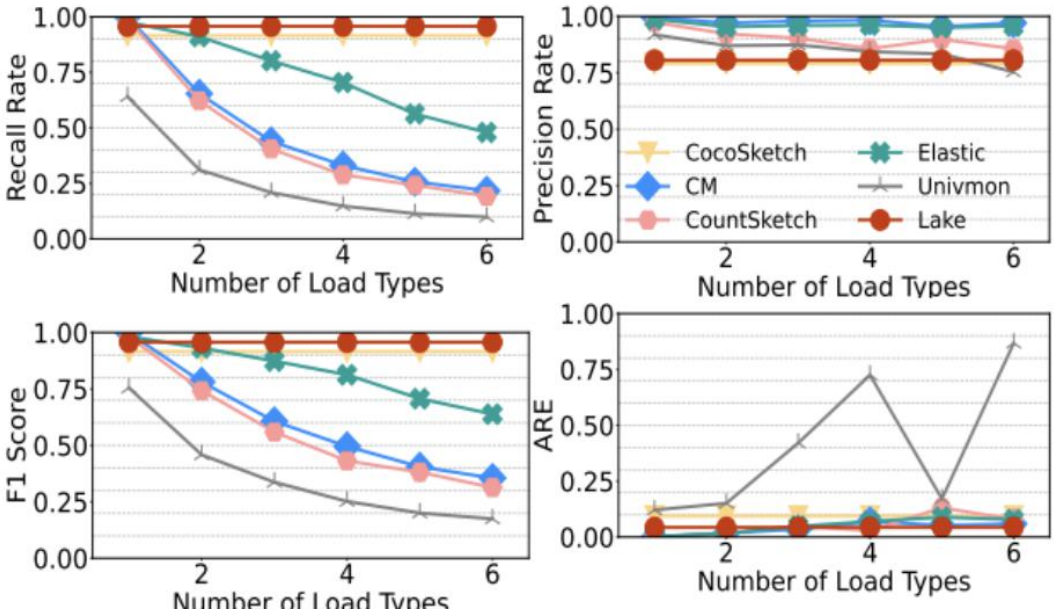


图 5 与其他优化方案比较

本项目逐渐为每个 sketch 添加新的负载。随着负载数量的增加，一些单键 sketch(即,CM、Count sketch、Elastic sketch 和 UnivMon)

会受到性能惩罚。尽管它们的精度率比其他 sketch 高，但它们几乎不会识别出热负载（召回率较低）。Lake 和 CocoSketch 在所有准确性指标上都保持高度可伸缩性。Lake 始终具有比 CocoSketch 更高的性能，因为 Lake 从轻部分中过滤出重要流量，从而既可以准确地识别重负载，也可以准确地识别轻负载。因此，在网络中的键值存储系统可以由于对热负载的非平凡识别而做出更精确的决策。

4.5 内存可扩展性

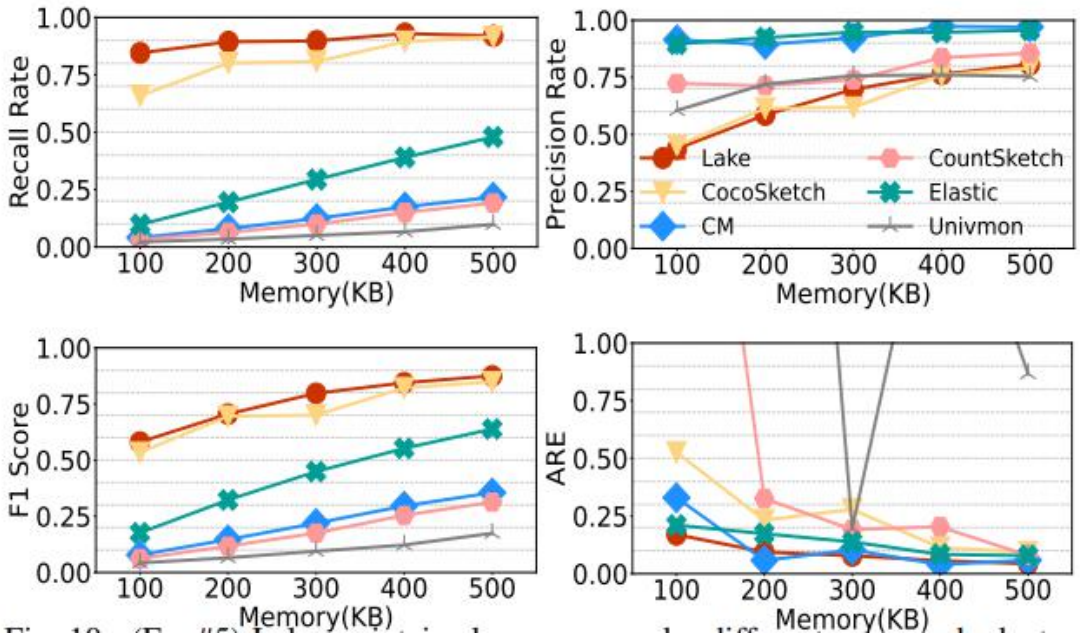


图 5 SpotMon 在不同的内存预算下保持低错误率

本项目改变内存预算，其他设置与实验 3 相同。如图 5 所示，SpotMon 在相同的内存预算下比其他 sketch 更准确。此外，SpotMon 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。

4.6 工作倾斜负载

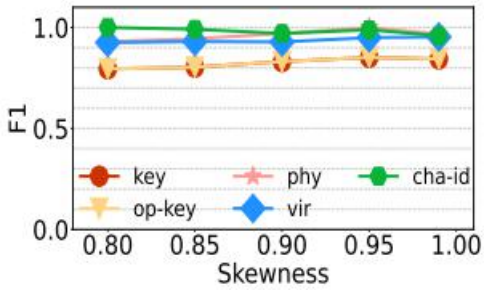


图 6 SpotMon 在不同负载偏度下保持低误差

本项目计算了 5 种不同的负载，以测试当请求的偏斜度从 0.8 变为 1 时，SpotMon 的鲁棒性。F1 保持稳定并略微提高。随着偏斜度的增加，重负载的比例增加并向重负载部分移动。因此，当工作负载分布高度倾斜时，SpotMon 表现出鲁棒性。

4.7 写入比率

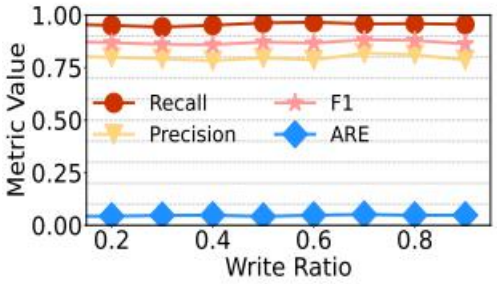
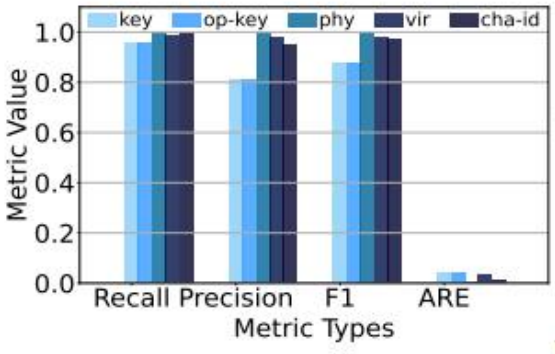


图 7 SpotMon 在不同的写率下保持低错误率

本项目在写入比率从 0.2 到 0.9 时计算每个度量值。每个度量值保持稳定，展现出在读密集型工作负载和写密集型工作负载中的高鲁棒性。当查询写密集型工作负载时，SpotMon 聚合相同操作的部分负载。操作的识别仅为 1 位，但 SpotMon 通过随机方差最小化进行了定制，用于子集和估计，并保持高精度。

4.8 热点检测



在这里使用 SpotMon 计算了 5 种负载。总体而言，SpotMon 保持了较高的准确性，但对不同负载的表现也有所不同。这种差异的原因在于具有较大键空间的负载可能会消耗更多的内存占用，并引起更多的哈希碰撞。此外，SpotMon 从轻部分中过滤出重要负载，从而确保了对重要负载的高准确性。

4.9 负载平衡

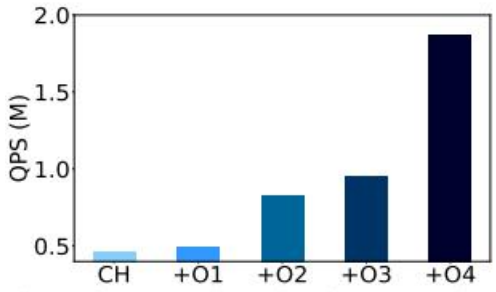
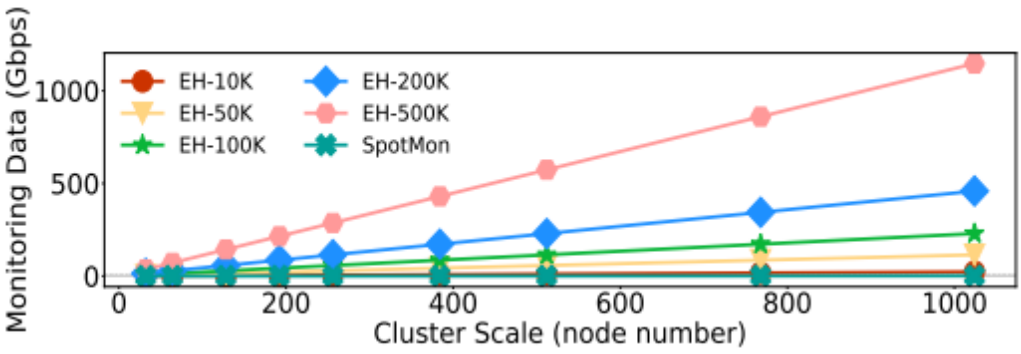


图 9 SpotMon 通过 O1-O4 提高了系统吞吐量

本项目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。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”表示本项目在系统上部署了 O1、O2 和 O3）。通过多类型负载感知，可以在多个约束条件下部署多个优化技术。

4.10 监控的可扩展性



在这里展示了在扩展到更大的集群时，SpotMon 的可扩展性非常高。集群节点数量从 32 个扩展到 1024 个。本项目与基于终端主机和基于抽样的方法进行了比较。监测数据以键-值格式表示，其中包括 128 位键和 32 位值。每个终端主机每次上报的键-值对数量从 10K 到 500K 不等，这表示了抽样率。较少的键值对数量表示较小的抽样率。本项目每隔 100 毫秒报告一次监测数据。随着集群规模的增加，两种替代方法都会报告大量的监测数据。监测数据可能会超负荷地占用控制器通道。因此，运营商必须通过更多的通道带宽来扩展系统。此外，控制器的处理能力也应该进行扩展。而 SpotMon 报告的监测数据非常少，因为监测数据已在交换机中进行了聚合。每个 32 节点的机架只需要一个 SpotMon。只需向交换机控制平面报告一个 SpotMon，从而大部分监测数据都由 SpotMon 进行了聚合。因此，在规模上，Lake 的可扩展性非常高。

五. 创新与特色

5.1 SpotMon 的框架可以实现高效的多类型负载感知，用于网络内键值存储。

SpotMon 框架通过统计网络内键值存储的负载数据，可以感知

不同类型的负载，并根据不同类型的负载选择不同的处理策略。例如，在高并发读写的情况下，SpotMon 可以采用一些高效的缓存策略来减少网络和存储的负担。在写入量较大的情况下，SpotMon 可以采用副本复制等策略来保证数据的可靠性和一致性。同时，SpotMon 框架可以对不同类型的负载进行统计和分析，通过识别和解决瓶颈问题，提高存储系统的性能。因此，SpotMon 的框架可以实现高效的多类型负载感知，为网络内键值存储系统提供了更加灵活和高效的处理方式。

5.2 SpotMon 消耗可接受的资源，并具有高准确性。

SpotMon 框架在实现负载感知时，可以达到高准确性的目标，这是因为它采用了多种负载识别技术和算法，如负载分布统计、聚类分析等方法，可以准确地识别出网络中各种类型的负载，并针对不同的负载类型采取相应的优化措施。同时，SpotMon 框架所消耗的资源是可接受的，它可以在较低的资源消耗下实现高准确性的负载感知，这也使得它在实际应用中更加具有可行性和可靠性。

5.3 在不同的工作负载分布下，SpotMon 具有高度的可扩展性和稳健性。

SpotMon 框架的高度可扩展性和稳健性是指，该框架在不同的工作负载分布下仍然能够保持较高的性能表现，同时具有较好的稳定性和稳健性。SpotMon 框架基于负载统计数据进行工作负载的感知和调度，能够快速适应不同的负载分布。在负载分布发生变化时，SpotMon 框架能够自动调整负载感知和调度策略，以确保系统性能的高效和稳定。具体来说，SpotMon 框架通过动态调整资源分配和任务调度策略，

以适应不同的负载分布。当系统中存在多种类型的负载时，SpotMon 框架可以根据负载类型的不同，调整资源的分配和任务的调度方式，从而实现不同类型负载的高效处理。此外，SpotMon 框架还支持动态伸缩，即根据负载情况自动调整系统的资源规模，以实现更好的性能表现。

5.4 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。

在 SpotMon 框架中，系统会实时监控不同负载类型的性能指标，并根据这些指标进行负载均衡。通过统计数据分析，SpotMon 可以帮助用户快速识别出系统中可能存在的瓶颈问题，并提供针对性的解决方案。例如，当某种类型的工作负载增加时，系统可能会出现性能下降或响应延迟等问题，这时候 SpotMon 会根据负载均衡策略，自动将该负载类型的任务分配到其他节点上，以避免单个节点过载而影响整个系统的性能。通过这种方式，SpotMon 可以保证系统的高可用性和可扩展性，并提高系统的整体性能。

5.5 SpotMon 框架允许用户为负载感知进行定制并自由调度工作负载。

SpotMon 框架允许用户根据具体的需求进行负载感知的定制，并能够自由调度工作负载。用户可以根据自己的需求，设置不同的负载类型和调度策略，以满足其特定的应用场景。例如，用户可以指定不同的负载类型，如读取、写入、删除等，以及每种负载类型的权重和优先级，SpotMon 框架会根据这些指定自动对工作负载进行调度。此

外，用户还可以自定义调度策略，以更好地满足其特定的需求。这种灵活的定制性和自由度，使得 SpotMon 框架可以在不同的应用场景下发挥最佳的性能。

参考

[1] (2017) Redis lab. redis. [Online]. Available: https://https://redis.io/N

[2] (2012) Matthew shafer. memcached. [Online]. Available: https://github.com/memcached/memcached

[3] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, “Attack of the killer microseconds,” Communications of the ACM, 2017.

[4] J. Dean and L. A. Barroso, “The tail at scale,” Communications of the ACM, vol. 56, no. 2, pp. 74–80, 2013.

[5] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, “hotring: A hotspot-aware in-memory key-value store,” in Proc. of FAST’20, 2020.

[6] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, ´ and I. Stoica, “Netcache: Balancing key-value stores with fast in-network caching,” in Proc. of ACM SIGOPS SOSP, 2017.

[7] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, “Distcache: Provable load balancing for large-scale storage systems with distributed caching,” in Proc. of FAST’19.

[8] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, “Pegasus: Tolerating skewed workloads in distributed storage with in - network coherence directories,” in Proc. of OSDI’20, 2020.

[9] (2006) Greg linden. akamai online retail perfor mance report. milliseconds are critical. [Online]. Available: http://glinden.blogspot.com/2006/11/marissa-mayer at-web-20.html

[10] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab et al., “Scaling memcache at

facebook," in Proc. of NSDI 13, 2013.

[11] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in Proc. of NSDI'16, 2016.

[12] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box ap proach to numa-aware persistent memory indexes." in Proc. of OSDI'21, 2021.

[13] V. Gavrielatos and e. Katsarakis, "Scale-out ccnuma: Exploiting skew with strongly consistent caching," in Proc. of EuroSys'18.

[14] Y.-J. Hong and M. Thottethodi, "Understanding and mitigating the impact of load imbalance in the memory caching tier," in Proc. of SoCC'13, 2013.

[15] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in Proc. of European Conference on Computer Systems, 2015.

[16] D. Didona and W. Zwaenepoel, "Size-aware sharding for im proving tail latencies in in-memory key-value stores." in Proc. of NSDI'19, 2019.

[17] H. H. Chan, Y. Li, P. P. Lee, and Y. Xu, "Hashkv: Enabling efficient updates in kv storage via hashing," in Proc. of ATC'18.

[18] J. Yang, Y. Yue, and R. Vinayak, "Segcache: a memory-efficient and scalable in-memory key-value cache for small objects." in Proc. of NSDI'21, 2021.

[19] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Autoscaling tiered cloud storage in anna," Proc. of VLDB, 2019.

[20] J. Wang, Y. Lu, Q. Wang, M. Xie, K. Huang, and J. Shu, "Pacman: An efficient compaction approach for log-structured key-value store on persistent memory," in Proc. of ATC'22.

[21] L. Cui, K. He, Y. Li, P. Li, J. Zhang, G. Wang, and X. Liu, "Swapkv: A hotness aware in-memory key-value store for hybrid memory systems," IEEE Transactions on Knowledge and Data Engineering, 2021.

[22] Y. Jia, Z. Shao, and F. Chen, "Slimcache: An efficient data compression scheme for flash-based key-value caching," ACM Transactions on Storage (TOS), 2020.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "Mica: A holistic approach to fast in-memory key-value storage," in Proc. of NSDI'14, 2014.

[24] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven streaming network telemetry," in Proc. of SIGCOMM'18, 2018.

[25] Y. Li, C. Tian, F. Guo, C. Li, and Y. Xu, "Elasticbf: Elastic bloom filter with hotness awareness for boosting read perfor mance in large key-value stores." in Proc. of ATC'19, 2019.

[26] (2023) An update on the memcached/redis benchmark. [Online]. Available: http://oldblog.antirez.com/post/update-on memcached-redis-benchmark.html

[27] (2023) Opensoc scalability. [Online]. Available: https://goo.gl/CX2jWr.

[28] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in Proc. of SOSP, 2013, pp. 423–438.

[29] (2022) Barefoot tofino 2: Second-generation of world’s fastest p4-programmable ethernet switch asics. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino-2/

[30] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, “Cocosketch: high-performance sketch based measurement over arbitrary partial key query,” in Proc. of ACM SIGCOMM’21, 2021.