计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202111242
60(5): 1136−1150, 2023

基于混合计数布隆过滤器的高效数据名查找方法

许　可[1,2]　李彦彪[2,3]　谢高岗[2,3]　张大方[1]
[1](湖南大学信息科学与工程学院　长沙　410082)
[2](中国科学院计算机网络信息中心　北京　100083)
[3](中国科学院大学　北京　100089)
(permission@hnu.edu.cn)

Efficient Name Lookup Method Based on Hybrid Counting Bloom Filters

Xu Ke[1,2], Li Yanbiao[2,3], Xie Gaogang[2,3], and Zhang Dafang[1]
[1] (*College of Information Science and Engineering*, *Hunan University*, *Changsha* 410082)
[2] (*Computer Network Information Center*, *Chinese Academy of Sciences*, *Beijing* 100083)
[3] (*University of Chinese Academy of Sciences*, *Beijing* 100089)

Abstract　Name lookup is a key operation in fundamental building blocks of information-centric networking, content delivery network, as well as the user plane function of 5G core network. It is required to deal with the longest prefix matching with a large-scale rule table, and thus confronts with serious challenges on lookup speed, update overhead and memory cost. In this paper, we design the hybrid counting Bloom filter (HyCBF) that maintains name prefixes and prefix markers within a single counting Bloom filter, while keeping them logically separated. This offers more guidance information without additional memory cost and time overhead. On this basis, we propose a HyCBF-assisted binary search (HyBS) scheme for efficient name lookup. Further, to mitigate the performance loss caused by backtracking operation during the binary search, we associate each unit of the HyCBF with a feature bitmap so as to reduce its false positive rate. Our extensive evaluations show that HyBS outperforms the state-of-the-art approaches in terms of lookup performance, update speed, as well as memory efficiency. In addition, we integrate HyBS into the vector packet processing (VPP) platform to evaluate its performance in terms of system implementation. The experimental results clearly demonstrate their potential to build a high throughput and scalable name lookup engine.

Key words　name lookup; feature bitmap; counting Bloom filter; binary search; vector packet processing (VPP)

摘　要　数据名查找是信息中心网络、内容分发网络、5G核心网中基础功能组件的关键操作，需要面向大规模规则表进行最长前缀匹配，在查找速度、更新开销和存储开销等方面面临严峻挑战. 首先设计了混合计数布隆过滤器（HyCBF），将数据名前缀和前缀标记维护在同一个计数布隆过滤器中同时保持二者的逻辑独立性.这样可在不增加额外存储开销和时间开销的情况下提供更丰富的指示信息. 基于此，提出HyCBF辅助的二分数据名查找（HyBS）方法以实现高效查找. 进一步，为缓解二分查找过程中因回溯导致的性能损失，为HyCBF中每个条目关联一个特征比特位图以降低其假阳性率. 实验表明，HyBS相比现有方法在查找性能和更新速度方面具有明显优势，存储效率也有一定提升. 此外，将HyBS集成到向量化数据包处理（VPP）框架中进行系统性能评估，结果表明HyBS可用于构建高通量可扩展的数据名查找引擎.

关键词　数据名查找；特征比特位图；计数布隆过滤器；二分搜索；向量化数据包处理

中图法分类号　TP393

收稿日期：2021-12-15；**修回日期**：2022-07-07

基金项目：国家自然科学基金项目(62072430, 61976087)
This work was supported by the National Natural Science Foundation of China (62072430, 61976087).

通信作者：张大方(dfzhang@hnu.edu.cn)

路由查找转发是网络基础功能. 在传统的 TCP/IP 架构中, 路由查找转发的核心操作是 IP 查找[1-7]. 在信息中心网络(information centric networking, ICN)中, 路由查找转发的核心操作是数据名查找[8-10]; 在内容分发网络(content delivery network, CDN)中, 基于内容路由也需要路由查找转发, 其核心操作也是数据名查找[11-14]. 此外, 在 5G 核心网用户平面功能(user plane function, UPF)网元转发数据包的过程中需提取应用层的业务标识(URL)进行策略规则匹配, 其本质也可视为数据名查找. 由此可见, 数据名查找是众多网络功能的核心操作. 其中, 数据名采用层次化结构, 由多个字符串组件组成. 在查找时根据业务需求执行组件级的精确匹配、前缀匹配或者最长前缀匹配. 其中, 最长前缀匹配挑战最大, 且最长前缀匹配方法微调后可直接用于精确匹配和前缀匹配, 因此本文主要研究组件级的最长前缀匹配, 并在后续阐述中以此作为数据包查找的内涵.

现有数据名查找方法大致分为 2 大类, 分别是基于前缀树(trie)的方法和基于 Hash(包括布隆过滤器)的方法. 基于前缀树的方法将长短不一的数据名维护为 1 棵前缀树[15-20], 因大量数据名具有相同前缀, 可共享存储节点提高存储效率, 但更新和查找性能相对较差. 基于 Hash 的方法将数据名按照组件长度分组, 每组维护 1 张 Hash 表[21-24]. 基于哈希表的方法的查找和更新性能较高, 还可进一步借助二分搜索[22]和贪心搜索[23]等优化策略来提速, 但为降低冲突或维护额外指示信息导致的存储开销一般较大. 因此, 现有方法都会结合 Hash 表和布隆过滤器[25-31]来压缩存储, 但却会因布隆过滤器的假阳性问题导致二分 Hash 探测过程中的频繁回溯.

本文提出一种混合计数布隆过滤器以及以此为基础的二分数据名查找方法. 本文的主要贡献有 3 个方面:

1)基于数据名前缀和前缀标记的稀疏分布特点, 提出一种混合计数布隆过滤器(hybrid couting Bloom filter, HyCBF)同时维护真实前缀和前缀标记的数量. 此外, 为每条记录维护 1 个特征比特位图以降低假阳性, 减少二分探测过程中的回溯.

2)基于 HyCBF 提出了一种新型的二分数据名查找方法, 即 HyCBF 辅助的二分搜索(HyCBF-assisted binary search, HyBS), 利用 HyCBF 所提供的查找指示信息, 减少 Hash 表探测. 此外, 还设计了一种自底向上的更新策略显著降低更新开销. 实验结果表明, HyBS 相比已有方法具有更快的查找和更新速度, 存储开销也更低.

3)基于向量化数据包处理(vector packet processing, VPP)平台实现了以 HyBS 构建查找转发插件的数据包转发系统, 并开展了系统性能测试. 结果表明, HyBS 具有构建高通量、可扩展数据名查找引擎的潜力.

1 相关工作

本节将介绍基于二分搜索的数据名查找方法的基本原理, 以及现有方法所面临的关键问题.

数据名规则集通常由若干个组件数量不同的前缀组成, 而对于待查询的数据名, 需要在其中找出能够与之匹配且组件数量最多的前缀. 针对这一问题, 顺序探测法是一种较为直接的解决办法——将所有前缀插入 Hash 表, 而待匹配的数据名则在 Hash 表上进行不断地迭代查询. 初始化时取整个数据名作为查询目标, 后续每次迭代时都减少 1 个末尾组件, 直到发现第 1 个匹配成功的前缀后返回结果, 或最终查找失败.

顺序探测的时间复杂度为 $O(n)$, 其中 n 为组件数量, 而有学者采用二分搜索法将复杂度降为 $O(\log n)$ 级别. 在早期解决 IP 查找中的最长前缀匹配问题时, 已有学者引入了二分搜索法[7]. 而在数据名查找问题中, 亦有学者使用二分搜索法提升查找性能[22-23,29-31]. 然而, 二分搜索虽然可以降低查找的复杂度, 但却可能出现查找错误. 查找程序在二分搜索树(binary search tree, BST)上某个节点匹配成功后会转向右子节点继续查找, 但匹配失败后如果直接转向左子节点, 则可能导致查找结果错误, 因为当前节点匹配失败并不意味着右子树中不存在可匹配的前缀. 因此, 搜索树上除了叶子节点以外的节点, 在保存前缀的同时, 还要为其右子树中保存的每个前缀维护 1 个前缀标记(后文简称标记)[22], 以便查找程序在匹配失败后选择正确的后续节点. 标记在形式上和普通前缀并无差异, 只是不附带任何转发信息, 因此可以将其作为一种特殊前缀插入到搜索树节点的 Hash 表中. 但是, 当大量标记插入到 Hash 表中, 不仅会导致 Hash 表的存储开销增长, 还会降低其查找速度. 考虑到标记并不附带任何转发信息, 查找程序只需检查其存在与否, 因此有学者采用布隆过滤器来保存标记[29-31]. 布隆过滤器作为一种数据成员检查(即检查数据是否在某个集合中)的数据结构, 适用于存储标记, 但其依然存在 2 大问题: 1)布隆过滤器需要多次 Hash 计算, 处理时间较长. 对此, 有的学者通过减少 Hash 计算次数或采用并行计算的方式, 大幅度降低

了布隆过滤器处理时间[31-34]. 2)布隆过滤器最主要的问题是存在假阳性,可能会出现错误的检查结果,因此需要引入回溯操作,但会导致查找性能下降.

综上所述,布隆过滤器虽适用于保存二分搜索法所需的标记,但需要降低假阳性率以避免出现频繁的回溯操作. 第 2 节将详细介绍我们的方法.

2 混合计数布隆过滤器辅助的二分数据名查找

本节将详细介绍混合计数布隆过滤器辅助的二分数据名查找方法,即 HyBS. 本节将首先通过一个具体示例介绍本文工作的研究动机,即布隆过滤器假阳性所带来的回溯操作对查找性能的影响,并引出我们的方法.

2.1 研究动机

基于二分搜索的数据名查找方法需要构建二分搜索树,每个节点包含 1 张 Hash 表,其中保存着特定长度的前缀和标记. 使用布隆过滤器可以优化标记的存储效率和查询性能,如图 1 所示的二分搜索树上的每个非叶子节点都包含 1 张 Hash 表和 1 个布隆过滤器;而叶子节点不保存标记,仅包含 1 张 Hash 表. 所有前缀在插入对应 Hash 表中之后,需要在某些特定节点上的布隆过滤器中插入标记,例如 Hash 表-5 上的前缀(即 /e/t/g/h/i 和 /k/c/t/f/g)需要在 Hash 表-4 所关联的布隆过滤器上插入标记(即 /e/t/g/h 和/k/c/t/f,其对应的布隆过滤器探测位置数值均被设置为 1). 假设待查询数据名为/a/b/c/e/f,根据二分搜索原则,查找程序首先在 Hash 表-4 上进行匹配,发现匹配失败后执行布隆过滤器检查,而检查结果显示为阳性(即方框中所示,/a/b/c/e 所对应的探测位置数值均为 1,而实际上/a/b/c/e 并未被插入到布隆过滤器中,因此结果为假阳性),因此查找程序转向 Hash 表-6. 查找程序在 Hash 表-6 上匹配失败,并且由于 Hash 表-7 并未保存任何前缀规则,因此 Hash 表-6 所关联的布隆过滤器为空,查找程序转向 Hash 表-5. 查找程序在 Hash 表-5 上同样匹配失败,并且 Hash 表-5 已经位于搜索树的叶子节点,此时为保证查找的正确性,需要回溯到 Hash 表-2 上进行匹配. 最后,查找程序在 Hash 表-3 上找到正确的最长前缀/a/b/c. 综上,查找程序在经过 5 次节点匹配(即节点 4,6,5,2,3)后找到了正确结果. 如果 Hash 表-4 所关联的布隆过滤器没有给出假阳性结果,则仅需 3 次节点匹配(即节点 4,2,3). 因此,降低布隆过滤器假阳性率有助于提升二分搜索的整体性能. 为此,我们提出混合计数布隆过滤器 HyCBF,并以此为基础提出新型二分数据名查找方法 HyBS.

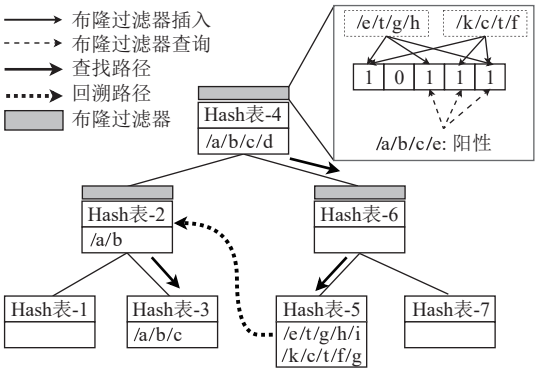


Fig. 1 Backtrack caused by the false positive of Bloom filter

图 1 因布隆过滤器假阳性导致的回溯

2.2 二分搜索整体架构

假设规则集中的数据名长度范围为 1~7 组件,HyBS 的二分搜索整体架构则如图 2 所示. 从二分搜索树的角度看来,所有的叶子节点(即节点 1,3,5,7)保存了 1 张 Hash 表和 1 个前缀计数布隆过滤器(prefix counting Bloom filter, P-CBF),该布隆过滤器用于保存 Hash 表中的前缀,以此避免不必要的 Hash 表查找. 所有的内部节点(即节点 2,4,6)除 P-CBF 和 Hash 表之外,还包含 1 个额外的标记计数布隆过滤器(marker counting Bloom filter, M-CBF),用于保存标记.

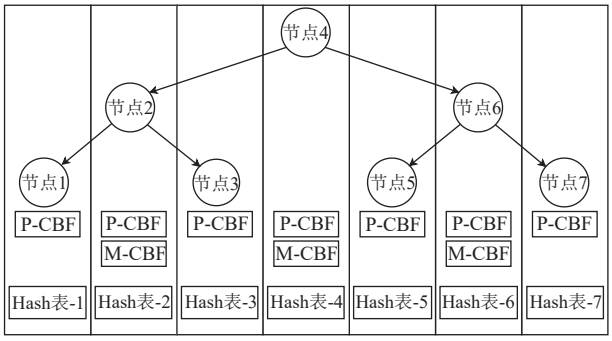


Fig. 2 The overall binary search architecture of HyBS

图 2 HyBS 二分搜索整体架构

在每个内部节点上,我们将前缀和标记分开保存在不同的布隆过滤器中,这样做的好处是:在保留了基本过滤器功能的同时,还获得了更细粒度的检查结果,即可以清楚地获知前缀和标记各自的检查结果. 如果布隆过滤器仅保留标记信息[30-31],那么想要获知是否存在可能匹配的前缀,依然需要查找 Hash 表. 如果布隆过滤器同时保存标记和前缀,当检查结果为阴性时,可以直接避免任何 Hash 查找. 但一旦检查结果为阳性,由于无法区分是前缀阳性还是

标记阳性，需要进一步到 Hash 表中查找[31]，如果标记是阳性而前缀是阴性，这时实际上不必执行 Hash 表查找. 如果前缀和标记分别存储于 2 个不同的布隆过滤器中，我们不仅可以知道检查结果是否为阳性，还可以获得阳性检查结果的具体类型（即，仅前缀阳性、仅标记阳性或者二者皆为阳性）.

2.3 混合计数布隆过滤器

由图 2 可知，所有的叶子节点均不包含标记，因此仅需普通 CBF 即可，在此我们不做讨论. 每个内部节点均关联着 1 个 HyCBF，其结构如图 3 所示，包括 1 个 HyCBF，以及 1 个用于降低布隆过滤器假阳性的特征比特位图（feature bitmap, FB）数组. HyCBF 的设计核心思想是将 P-CBF 和 M-CBF 合并在一起，即 HyCBF 中每个 counter（即计数器）实际上由 2 个 counter 组成. 2 个布隆过滤器逻辑上需要单独执行 Hash 计算获得探测位置，会导致 Hash计算次数翻倍，而实际上可以避免出现这个问题. 具体而言，当 P-CBF 和 M-CBF 的长度分别确定后，取其中的最大值作为 HyCBF 的长度. 对于同一个数据名，其在 P-CBF 和 M-CBF 上 3 个探测位置的索引相同，并不需要额外的 Hash 计算，并且这种方式使得一次内存操作即可把 P-CBF 和 M-CBF 的探测位置数据都载入 CPU 缓存中. 因此，HyCBF 虽然在逻辑上维护了 2 个布隆过滤器，而实际计算时并不需要额外的 Hash 计算和访存操作. 对于如图 2 的二分搜索树结构，某个内部节点上的标记数量等于其所有右子节点上的前缀数量之和，因此 HyCBF 的长度通常取 M-CBF 的长度. 对于布隆过滤器而言，数据总量和探测位置数量不变时，长度越长意味着假阳性率越低，因此在 HyCBF 中，P-CBF 的假阳性率会因为长度增长而出现降低，M-CBF 的假阳性率则保持不变，二者不会出现假阳性率高于设定阈值的情况. 布隆过滤器的长度通常会根据给定的假阳性率阈值和需要插入的数据量计算得出. 假设布隆过滤器的假阳性率阈值为 F，数据集的大小为 n，布隆过滤器探测位置数量（即 Hash 函数数量）为 k，则布隆过滤器所需长度 m 的计算公式为

$$m=\beta n,$$
$$\beta=\left\lceil\frac{k}{-\ln(1-\sqrt[k]{F})}\right\rceil. \tag{1}$$

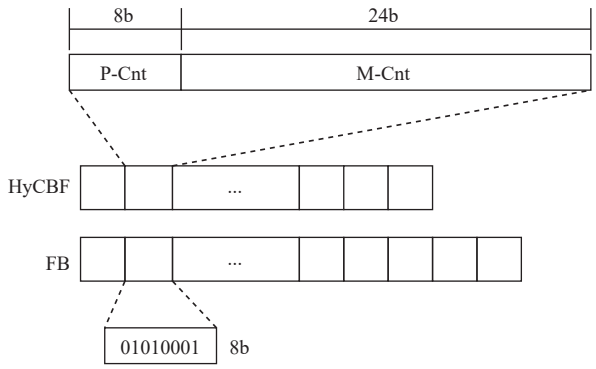


Fig. 3 Structure of HyCBF

图 3 混合计数布隆过滤器结构

HyCBF 中的元素为固定大小的 counter（计数器），为保证不会出现 counter 数值溢出的情况，需要让 counter 维持一定的位宽度（bit width）. 如图 3 所示，P-CBF 和 M-CBF 的 counter 分别为 P-Cnt（prefix counter）和 M-Cnt（marker counter）. 经实验测试，2 个 counter 的长度分别为 8b 和 24b 就足以保证在处理百万规模的规则时不会出现 counter 数值溢出，因此 HyCBF 的 counter 总的位宽度为 32b. 越靠近根节点，HyCBF 所保存的标记就越多，其内存占用也就越大，但是在二分搜索这一特殊场景下，HyCBF 的存储开销实际上可以比理论值更低.

对于某个内节点，由于 M-CBF 中保存着该节点的右子树中所有前缀的标记，会导致 HyCBF 的长度较长，因此存储开销也较大. 但是，这其中实际上存在大量重复标记. 假设规则集中包含 /a/b/c/d/e, /a/b/c/d/f 和 /a/b/c/d/g/h 这样 3 个前缀数据名，在图 2 所示的二分搜索结构中，这 3 个前缀均需要在节点 4 上插入标记/a/b/c/d，我们称之为共享标记. 维护冗余的共享标记是为了确保查找的正确性，假如为所有的标记进行去重处理，将导致 HyCBF 无法支持删除操作. 重复插入标记会导致单节点的标记数量增大，根据式（1），理论上会导致 M-CBF 的长度增长，进而使得 HyCBF 长度增长. 而事实上，面对由共享标记带来的标记数量增长，只要 HyCBF 中的 counter 具有足够的比特位宽就不会出现数值溢出，即使 HyCBF 的长度不变，假阳性率也不会增长. 为证明这一结论，我们将计数布隆过滤器和标准布隆过滤器进行对比分析. 二者的区别在于对更新的支持，在进行数据成员检查时二者的用法没有区别——当某个数据的所有探测位置的数值（即计数布隆过滤器中的单个 counter，以及标准布隆过滤器中的单个比特位）均不等于 0，我们认为该数据存在数据集中. 因此，当执行数据成员检查时，计数布隆过滤器可以等价为标准布隆过滤器，其中计数布隆过滤器中的“非零”counter 和“零”counter 分别对应于标准布隆过滤器中的 1 比特位和 0 比特位. 当计数布隆过滤器插入重复数据时，只要没有发生 counter 数值溢出，这些探测位置的 counter 数值就始终不等于 0，对应的标准布隆

过滤器就不会有任何变化. 综上所述, 当存在重复数据时, 不能采用传统计算方式推算所需的布隆过滤器长度, 而是要剔除其中的重复数据后进行计算, 否则会导致布隆过滤器的长度出现冗余, 使得真实的假阳性率低于我们设定的阈值, 我们将在实验部分验证这一推论.

除了存储开销以外, 传统布隆过滤器需要多次 Hash 计算以获得探测位置的索引, 带来额外的开销. 我们从文献 [32−33] 获得启发, 采用“一次 Hash”方法最大限度地减少 Hash 计算的次数. 此外, 受商过滤器[35]的启发, 仅计算一次 Hash 值, 并将其与布隆过滤器长度进行除法运算, 将所得到的商和余数作为 Hash 种子, 通过迭代计算的方式获得探测位置的索引. 此外, 前缀的 Hash 值等于组成该前缀的每个组件的 Hash 值之和, 已经计算过的组件其 Hash 值可以复用, 这样每个数据名需要执行的 Hash 计算次数至多等于其组件数量. 假设布隆过滤器长度为 M, 每个数据名有 k 个探测位置, 给定数据名的 Hash 值为 h, h 与布隆过滤器长度进行除法运算后, 其商为 $Q(h)$, 余数则为 $R(h)$, 则第 i 个探测位置的索引 $slot_i$ 为

$$slot_i = (Q(h) + i \times R(h)) \bmod M, i \in [0, k). \quad (2)$$

此外, 当经过布隆过滤器处理后, 如果需要执行 Hash 表查找, 我们还可以继续复用布隆过滤器检查过程中的计算结果, 这样可以把单节点上需要进行的 Hash 计算次数降到最低. 假设 Hash 表长度为 H, 那么 Hash 表索引 $index$ 的计算公式为

$$index = (Q(h) + k \times R(h)) \bmod H. \quad (3)$$

2.4 特征比特位图优化

除了存储开销和 Hash 计算时间开销以外, HyCBF 所面临的另一大问题就是假阳性问题, 本节将介绍我们为此而设计的方法——特征比特位图. 正如 2.1 节所述, 较高的布隆过滤器假阳性率会引入频繁的回溯操作, 进而导致查找性能下降. 为此, 我们在设计 HyCBF 时, 为每个 counter 关联了 1 个特征比特位图, 并以此显著降低了布隆过滤器的假阳性率. 假设布隆过滤器在进行数据成员检查时需要探测 3 个位置, 则每个位置需要关联 3 个特征比特位图. 图 4 中 BF(Bloom filter)表示布隆过滤器, FB_0, FB_1, FB_2 则表示 3 个特征比特位图数组, 其中每个数组元素都是 1 个特征比特位图, 并与 BF 对应位置相关联. 当检查某个数据是否存在于数据集中时, 通过 Hash 计算得出 3 个探测位置, 并检查这 3 个探测位置的数值是否均为 1. 图 4 展示的 2 个场景中, $Data_0$ 的探测位置数值均为 1, 但 $Data_0$ 并没有被插入到布隆过滤器中, 因此 2 个场景中 $Data_0$ 的检查结果均为假阳性. 对于图 4(a)所示的场景 1 而言, 布隆过滤器中已插入 2 个数据 $Data_1$ 和 $Data_2$, 二者的探测位置刚好包含了 $Data_0$ 的 3 个探测位置. 因此, $Data_0$ 的检查结果为假阳性. 如图 4(b)所示的场景 2 较为特殊, 因为在该场景下, $Data_0$ 对应的 3 个探测位置数值是被同一个数据置为 1(即 $Data_3$). 这种情况下, $Data_0$ 的检查结果依然是假阳性. 为了更好地理解场景 2 的假阳性, 这里引入探测位置层级的概念. 层级表示探测位置的先后顺序. 假设索引号从 0 开始, 并从左到右递增, 在对 $Data_0$ 进行探测时, 探测位置的先后顺序是 BF[0], BF[1], BF[4], 因此对 $Data_0$ 而言这 3 个探测位置的层级分别是 0,1,2. 而对于 $Data_3$ 而言, 探测位置的先后顺序则为 BF[1], BF[0], BF[4]. 其中, BF[0] 和 BF[1] 在插入 $Data_0$ 和 $Data_3$ 时具有不同的层级, 因此 $Data_0$ 的 3 个探测位置虽然均为 1, 且这 3 个探测位置被同一个数据置 1(即 $Data_3$), 但检查结果依然是假阳性.

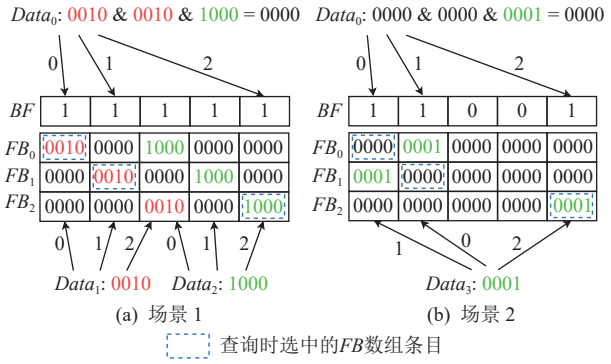


Fig. 4 The feature bitmap

图 4 特征比特位图

布隆过滤器不支持“键-值”查询, 对此有学者提出了一种新型布隆过滤器 NBF[36]. NBF 将数据的编码信息嵌入到对应的探测位置中, 具体方法是在插入数据时, 将数值编码以按位或(OR)运算的形式插入到探测位置. 而查询时, 则将各个探测位置上的数值编码以按位与(AND)运算的方式聚合在一起, 以此过滤掉其他数据嵌入的数值编码(原文称之为“噪声”), 并获得所查询数据的数值编码. 受此启发, 通过在布隆过滤器上每个位置关联 1 个数值编码(即特征比特位图), 可以降低布隆过滤器的假阳性率. 具体而言, 当针对某个数据执行成员检查时, 我们在检查各探测位置数值的同时, 将对应的特征比特位图也取出并进行按位与运算, 通过验证计算结果是否为 0 来判断是否出现了假阳性. 利用特征比特位图, 我们加强了布隆过滤器的检查条件: 不仅要求每个

探测位置的数值非零，还要求每个探测位置上“嵌入”了相同的特征比特位图. 针对某个具体数据，即使其对应的每个探测位置数值均为 1，只要特征比特位图的与运算结果为 0，其检查结果仍然为阴性.

针对图 4 所示的场景 1，$Data_1$ 的特征比特位图是 0010，$Data_2$ 的特征比特位图则是 1000. 这 2 个数据插入布隆过滤器后，各探测位置的特征比特位图也需要嵌入到对应的特征比特位图数组中. 当处理 $Data_0$ 时，在确定探测位置之后，查找程序将各位置上对应层级的特征比特位图（即虚线框选定的特征比特位图）提取出来，分别为 0010,0010,1000. 最后，利用提取出来的特征比特位图执行按位与运算，得出计算结果为 0000. 因此我们得到针对 $Data_0$ 的阴性检查结果，场景 1 中的假阳性问题可以得到规避. 场景 2 不同于场景 1，需要将特征比特位图结合层级信息做出判断. 场景 2 中，$Data_3$ 的特征比特位图是 0001，并且根据每个探测位置的层级，这些特征比特位图将被嵌入到对应的位置. 例如，对 $Data_3$ 而言，BF[0]层级为 1，因此将 0001 嵌入 BF[0]所关联的 1 号特征比特位图 FB_1[0] 中. 在获得 $Data_0$ 的探测位置后，根据探测位置的层级提取出对应的特征比特位图（虚线框选中的特征比特位图），即 0000,0000,0001. 最后，和场景 1 的处理方式相同，将提取出来的所有特征比特位图按位与运算得出的计算结果为 0000，因此针对场景 2 中的假阳性问题，可以利用附带层级信息的特征比特位图来解决.

图 4 所示的特征比特位图从形式上比较直观地描述了其工作原理，但在具体实现时，这种方法会带来较大的存储开销. 假设每个插入布隆过滤器的数据需要访问 n 个探测位置，而布隆过滤器的长度为 m，则图 4 中特征比特位图数量为 $m\times n$ 个. 通过按位或运算将图 4 所示的每个探测位置对应的 3 个特征比特位图合并在一起，这样可以将 n 个特征比特位图数组减少到 1 个. 此外，受 SBF[37] 中移位操作的启发，我们将层级信息通过移位操作的方式嵌入特征比特位图中，避免了合并时丢失掉层级信息. 图 5 展示的特征比特位图结构和图 4 中场景 1 一致，不同之处在于通过引入移位操作将多个特征比特位图数组合并为 1 个. 当确定数据名 *Name* 的 3 个探测位置后，我们在嵌入特征比特位图时，以对应位置的层级数值作为移位的偏移量，将特征比特位图嵌入到偏移后的 FB 数组位置上. 同样地，在对 *Name* 执行成员检查时，得出所有的探测位置后，依然需要将层级数值作为偏移提取出对应位置的比特位图，即图 5 中灰色填充的特征比特位图. 观察图 5 可以发现，如果我们提取出 *Name*（即相当于图 4 场景 1 中的 $Data_0$）所对应的特征比特位图（即 0010,1010,1000）并执行按位与运算，计算结果依然是 0，特征比特位图依然可以发挥作用. 同时，由于偏移操作的引入，特征比特位图附带的层级信息也得到了保存. 相比于图 4 中的结构需要 $m\times n$ 个特征比特位图，图 5 仅需 $m+n-1$ 个特征比特位图.

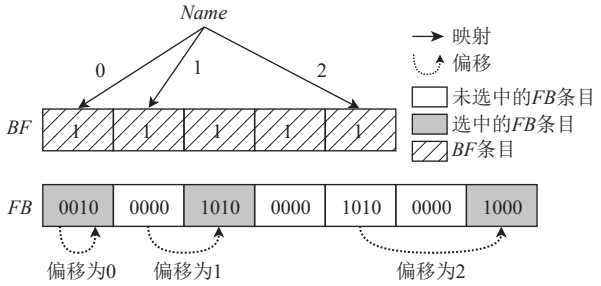


Fig. 5 The implementation of feature bitmap

图 5 特征比特位图的实现

综上所述，特征比特位图可以有效地降低布隆过滤器的假阳性率. 在 HyBS 中，每个数据名的特征比特位图为 8b，并且有且仅有 1 个比特位数值是 1. 因此，形式上看来特征比特位图中大部分比特位数值为 0，因此在检查阶段的按位与运算时，我们能有更大的概率过滤掉由其他数据插入时嵌入的“噪声”（即数值为 1 的比特位），以确保特征比特位图计算的准确性. 在 HyBS 中，我们在计算数据名的特征比特位图时所采用的方式是：将数据名的字符长度除以 1 个常量阈值，然后我们用计算结果作为偏移，将 1 个初始化为全 0 的特征比特位图中的对应比特位翻转为 1，并以此作为该数据名的特征比特位图. 特征比特位图的计算方式是个开放性问题，HyBS 的计算方式则是经实验测试获得.

2.5 查 找

介绍完 HyBS 的核心数据结构 HyCBF 之后，本节将介绍如何在二分搜索中有效地使用它. 在采用 HyCBF 这样一种双布隆过滤器设计之后，二分搜索树上每个节点的处理逻辑如图 6 所示. 由于叶子节点的处理过程和传统的基于过滤器的 Hash 表查找（即，布隆过滤器做初步筛选，Hash 表完成前缀匹配）方法并无差异，在此不做介绍，图 6 仅展示包含了 HyCBF 的内部节点的处理过程.

当处理数据名 *Name* 时，首先在 M-CBF 和 P-CBF 上分别执行标记和前缀检查. 根据检查结果的不同，可分为 4 种情况：

1）P-CBF 阴性，M-CBF 阴性. 这种情况表明该节

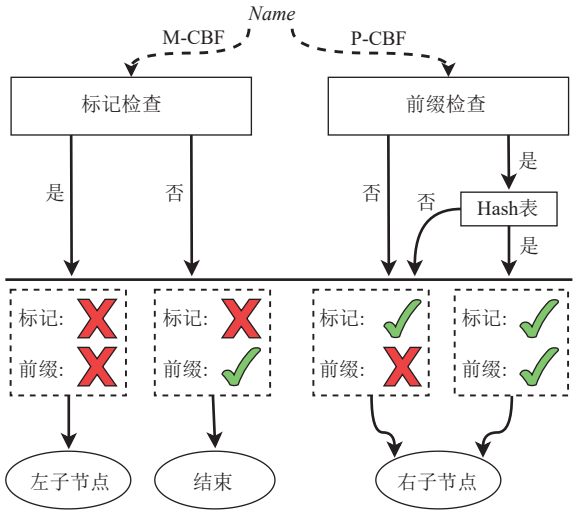


Fig. 6 Process of single node in BST

图 6 二分搜索树单节点处理过程

点没有任何能与 *Name* 匹配的前缀和标记, 因此下一节点为当前节点的左子节点.

2)P-CBF 阴性, M-CBF 阳性. 这种情况说明该节点没有能与 *Name* 匹配的前缀, 因此不必进行 Hash 表查找. 但标记的检查结果为阳性, 表示右子节点中可能保存有 *Name* 的前缀, 因此下一跳节点为当前节点的右子节点.

3)P-CBF 阳性, M-CBF 阴性. 这种情况说明当前节点可能保存着与 *Name* 匹配的前缀, 因此需要进行 Hash 表查找. 同时 M-CBF 为阴性表示 *Name* 在右子节点中不可能存在更长的前缀. 由于布隆过滤器不存在假阴性, 因此如果 Hash 表查找成功, 则查找到的前缀即为 *Name* 的最长前缀, 可以直接返回查找结果. 否则, 下一跳节点设置为当前节点的左子节点.

4)P-CBF 阳性, M-CBF 阳性. 这种情况表示需要进行 Hash 表查找. 同时, 因为标记的检查结果为阳性, 无论查找成功还是失败, 下一跳节点均会被设置为当前节点的右子节点. 当 Hash 查找失败时, 不做任何处理立即转向右子节点; 当 Hash 查找成功时, 我们将查找到的前缀作为当前阶段的最长匹配前缀, 并转向右子节点.

综上所述, 在这 4 种情况中, 情况 3 和情况 4(即当 P-CBF 检查结果为阳性时)需要进行 Hash 表查找. 而如果将前缀和标记不作区分地存入同一个布隆过滤器中, 情况 2 也需要进行 Hash 表查找, 因为这种方式无法区分阳性的检查结果是源自前缀还是标记. 同时, 得益于 HyCBF 的设计, 2 个逻辑上独立的布隆过滤器在实际查找过程中并不会带来额外的时间和存储开销.

2.6 “自底向上”更新

更新性能同样是数据名查找方法的重要性能指标. 通常, 在执行更新之前需要进行查找操作以确定需要更新的具体节点位置. 在最长前缀匹配这一场景下, 二分搜索的目的在于探测可能的前缀长度, 这在查找时必不可少, 因为无法事先确定数据名在该数据集中的最长前缀的长度. 而对于规则的更新操作, 其附带的数据名长度就决定了它最终会被插入到如图 2 所示的查找结构中的哪个节点上. 此外, 前缀所对应的标记也需要更新. 具体而言, 从更新节点到根节点的路径上, 我们需要在所有的右子节点的父节点上更新标记信息, 我们称这些节点为标记节点. 当如图 2 的查找结构确定后, 每个节点的标记节点事实上也已经确定. 例如, 节点 7 的标记节点为节点 4 和节点 6. 因此在每个节点上维护 1 个比特位图, 即标记节点位图, 并以此记录某个节点的标记节点信息. 例如图 7 中节点 7 的标记节点位图为 00101000, 表示节点 7 的标记节点为节点 4 和节点 6.

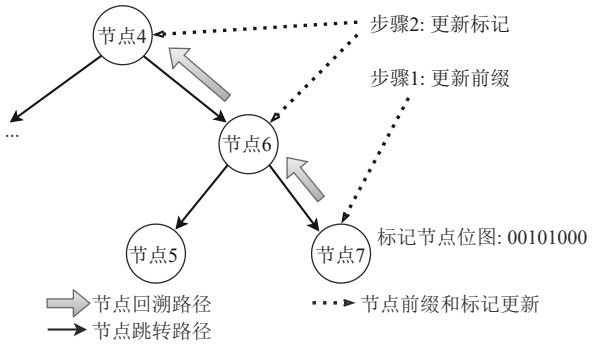


Fig. 7 The "bottom to up" update

图 7 “自底向上”更新

在这样一种设计思路下, HyBS 的更新操作可以分为如图 7 所示的 2 个步骤: 1)根据规则中附带的前缀的长度确定待更新规则位于哪个节点, 并在该节点的 Hash 表中执行相应的更新操作; 2)根据节点附带的标记节点位图确定所有的标记节点, 更新所有标记节点上的标记信息. 因此, 规则更新时至多需要访问 1 个节点的 Hash 表. 同时, 步骤 2 的计算过程中会复用步骤 1 中所得出的计算结果(即组件的 Hash 值), 进一步减少时间开销.

2.7 系统实现和优化

本节将介绍如何将 HyBS 部署到真实的软件包处理平台 VPP 中, 以及结合 VPP 批处理特性所采用的优化设计. 目前主流的软件包处理平台有 OVS(open vswitch)[38] 和 VPP[39]. OVS 采用元组空间搜索(tuple space search, TSS)算法实现流表查询, 并且

目前大部分商用智能网卡都支持 OVS 硬件卸载加速[40]，因此 OVS 被广泛应用于构建高性能网络 I/O 系统. 但是 OVS 为 OpenFlow 协议进行了深度定制设计，因此难以在其基础之上实现基于数据名查找的转发引擎. 而 VPP 作为一种支持用户开发自定义协议的包处理平台，可以满足 HyBS 的部署需求. 在 VPP 中，各个网络功能以节点的形式部署，而各个节点可以通过编程控制进行灵活的编排，以此实现自定义的数据包处理流程. 除了支持自定义配置节点连接关系，VPP 还支持以插件的形式将用户自定义的包处理代码打包成节点嵌入到的包处理节点图中. 因此，我们将 HyBS 以插件的形式嵌入到数据包处理节点图中. 如图 8 所示，整个包处理框架包括 3 个节点：接收节点负责采用数据面开发套件(data plane development kit, DPDK)[41]从网卡接收数据包；转发节点则是嵌入了 HyBS 的包处理节点，负责接收来自接收节点的数据包并执行数据名查找；发送节点则将处理后的数据包采用 DPDK 由指定端口转发出去. VPP 在启用 HyBS 插件时，可以通过命令行接口指定规则文件的路径，后台会进行查找结构的构建，而在后续的数据包处理阶段，所有 CPU 会共享这一份查找结构.

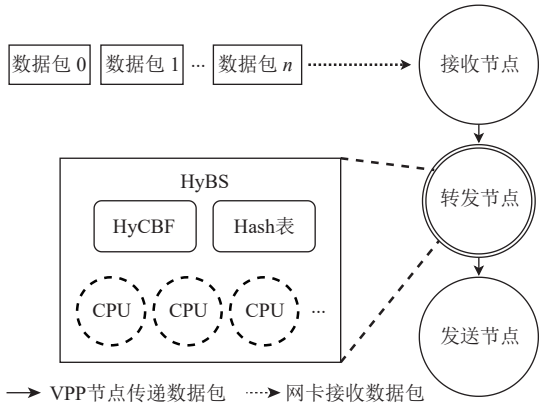


Fig. 8 The system implementation of HyBS in VPP

图 8 VPP 中的 HyBS 系统实现

VPP 利用批处理策略显著提高了 CPU 指令缓存(instruction cache, I-Cache)的命中率，但是却没有针对数据缓存(data cache, D-Cache)做优化设计. 而 HyBS 利用组预读策略，可以提升查找程序对数据缓存的利用效率. 我们以 HyBS 中的布隆过滤器处理为例. 布隆过滤器探测通常可以分为 3 个步骤，即 Hash 计算、内存访问和探测(即数值检查). 当处理某个数据包时，首先提取其附带的数据名，根据 Hash 计算得出布隆过滤器探测位置的索引，然后对该位置发起内存访问请求以获得该位置的计数器数值，最后检查计数器的数值. 图 9 展示了处理 3 个数据包时的布隆过滤器探测阶段的时间分布，其中不同的方块对应着不同操作的时间开销. 为简化展示内容，我们设定布隆过滤器探测位置仅为 1 个. 如图 9 所示，时间线以上的部分展示的是没有采用组预读策略时的时间开销情况，当第 1 个数据名完成 Hash 计算获得探测位置的索引后，查找程序发起对该探测位置的访问，此时该探测位置的数据并没有被载入 CPU 的数据缓存之中，即发生一次缓存未命中，会导致 CPU 出现停顿，直到数据被完整地载入缓存. Hash 计算使得数据包的探测位置具有较大随机性，因此数据局部性较差，极端情况下每个数据名的内存访问请求都会触发 CPU 停顿. 时间线以下的部分则是采用组预读策略时的时间开销情况. 现代 CPU 支持对内存进行预读操作，即当发生缓存未命中时，CPU 依然需要将数据从主存载入缓存中，但不会导致 CPU 停顿，因此 CPU 在数据载入缓存期间可以进行其他计算. 从时间线以下的部分可以看出，当第 1 个数据名完成 Hash 计算并获得探测位置索引之后，查找程序对该位置发起预读请求，这时数据将开始从主存载入缓存中，此时 CPU 无需等待该操作完成，而是继续针对第 2 个数据名执行 Hash 计算，并同样根据计算结果发起预读操作，最后处理第 3 个数据名. 当所有的数据名都完成 Hash 计算之后，再执行对应位置的探测操作. 由于 Hash 计算和缓存加载可以同时进行，这 2 部分的时间开销可以重叠在一起. 虽然组预读并不会使得缓存未命中的次数减少，但通过时间片重叠，数据载入缓存的时间开销被“隐藏”起来.

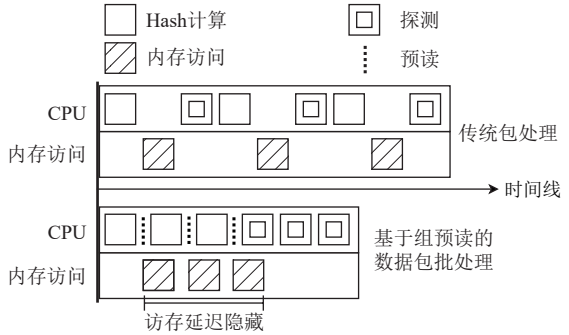


Fig. 9 Group-prefetching strategy

图 9 组预读策略

需要注意的是，图 9 中我们假设 Hash 计算、内存访问和探测的时间开销相同(即三者的时间片长度相同)，但实际情况中访存的时间开销可能会比 Hash 计算大很多. 在极端情况下，当数据载入缓存的

时间开销远大于 Hash 计算的时间开销时，有可能针对 3 个数据名的 Hash 计算全部完成后，依然需要等待较长时间直到最后 1 个探测位置数据载入缓存. 这种情况下，CPU 可能仍然需要停顿较长时间，进而弱化了组预读的优化效果. 因此，不同 CPU 的缓存载入时间和 CPU 计算时间的比值不同，所表现出来的组预读优化效果也不同. 但是，和普通的数据包处理策略相比，基于组预读的批处理策略依然能降低查找的整体时间开销.

3 性能评估

本节通过将 HyBS 与 2 个基于不同数据结构的数据名查找方法进行对比，以评估 HyBS 的综合性能表现. 对比方法为 CBS[29] 和 Nametrie[20]，二者分别基于布隆过滤器和二分搜索(即，和 HyBS 同属一类方法)，以及前缀树. 由于目前不存在真实数据名的数据集，通常使用 URL 数据. 我们采用从 Reddit 论坛评论中收集的 URL 数据[42] 对于每个 URL，我们用其中的特殊符号作为分隔符将 URL 分隔成多个组件，以此将 URL 转换成数据名. 本文生成了不同规模的 6 个规则集，其中每个规则集的规则数量如表 1 所示.

Table 1 Number of Rules of Different Rule Sets

表 1 不同规则集中的规则数量

规则集编号	规则数量
0	50 万
1	100 万
2	150 万
3	200 万
4	250 万
5	300 万

为了验证 HyBS 在不同 CPU 下的性能表现，我们分别在 2 台配备了不同 CPU(即 Intel 处理器和 AMD 处理器)的服务器上进行了性能测试，分别为 $\text{Server}_{\text{Intel}}$ 和 $\text{Server}_{\text{AMD}}$. 2 台服务器的具体配置如表 2 所示. 为了尽量避免受到系统进程调度的干扰，在测试时利用 Linux 的 CPU 核心隔离功能，将查找程序调度到被隔离的 CPU 核心. 考虑到 HyBS 和 CBS 中都需要进行 Hash 计算，统一采用了 MurmurHash[43]，并且均采用链表法来解决 Hash 冲突.

3.1 存储开销

存储开销是一项和 CPU 无关的性能指标，因此该实验不区分 CPU 型号. 我们发现，当规则集规模较大时，对于 HyBS 和 CBS 这样采用布隆过滤器保存标记和前缀的方法，其主要存储开销来源于布隆过滤器而非 Hash 表，因此对布隆过滤器的存储优化对于整体存储开销的降低是有重要意义的.

Table 2 Configuration of Servers

表 2 服务器的配置

配置项	$\text{Server}_{\text{Intel}}$	$\text{Server}_{\text{AMD}}$
CPU 型号	Xeon® Platinum 8160	EPYC 7F52
物理核数量	48 核	32 核
主频/MHz	2 100	3 500
缓存大小/KB	一级：64 二级：1 024 三级：33 792	一级：64 二级：512 三级：16 384
被隔离核心编号	24~47	16~31
内存大小/GB	128	256

如果仅从数据结构设计的角度看，HyBS 中每个 HyCBF 都配备了 1 个长度几乎相等的特征比特位图数组，尽管特征比特位图数组中每个数组元素的大小(即 1B)仅为 HyCBF(即 4B)的 1/4，但这部分的存储开销依然是不可忽略的. 但是，正如 2.3 节所述，HyCBF 中长度通常取决于 M-CBF 的长度，而特征比特位图数组的长度在考虑去除“共享标记”的影响下，事实上相比理论值会小很多，因此 HyBS 的整体存储开销相比已有方法依然是很有优势的. 图 10 展示了 3 种数据名查找方法在采用不同规则集时的存储开销. 如图 10 所示，随着规则集规模的增大，HyBS 的存储开销从 49.5 MB 增长到 279MB，始终小于 CBS 和 Nametrie. 和 CBS 相比，HyBS 能降低 25.8%~30.2% 的内存开销，而相比于 Nametire 则可降低 22.5%~26% 的内存开销.

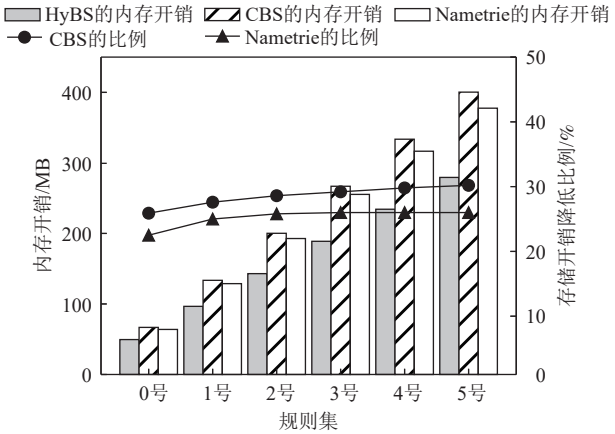


Fig. 10 Memory consumption

图 10 存储开销

3.2 组预读的影响

正如 2.7 节系统实现部分所述，在数据包批处理场景下，使用组预读策略可以做到隐藏内存访问的时间开销，从而降低整体时间开销. 为更加量化地验证组预读策略的有效性，我们对比了 HyBS 在开启和关闭组预读策略时的查找速度，规则集则是采用最小的 Rules_0，实验结果如图 11 所示. 我们分别在 $\text{Server}_{\text{AMD}}$ 和 $\text{Sever}_{\text{Intel}}$ 上进行了该测试，以验证不同 CPU 平台上组预读策略的效果. 柱形图按照不同的内存访问策略分为 2 组，分别对应着开启组预读的测试组，以及关闭组预读的测试组，其中单位为百万数据名每秒（million names per second, MNPS）.

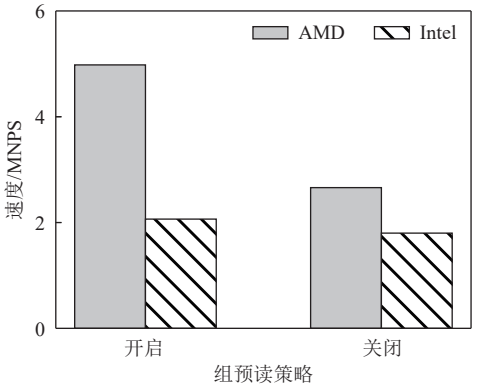


Fig. 11 The effect of group-prefetching

图 11 组预读的影响

$\text{Server}_{\text{AMD}}$ 的测试结果显示，开启组预读可以带来 1.8 倍的性能提升，而 $\text{Server}_{\text{Intel}}$ 的测试结果则显示开启组预读带来的性能提升仅有 15%. 尽管测试结果表明组预读的确可以带来性能提升，但是不同的 CPU 平台上提升程度却差异很大. 同时我们发现，$\text{Sever}_{\text{Intel}}$ 的 CPU 缓存相比 $\text{Server}_{\text{AMD}}$ 的更大，因此问题显然不在于缓存大小. 对此，我们的分析是：正如 2.7 节最后所述，内存访问的时延可能会比 CPU 计算时延（包括 Hash 计算和计数器探测的时延）高很多，甚至会出现 Hash 计算结束时第 1 个数据名的数据依然没有被完全载入 CPU 缓存中，因此 CPU 依然需要等待数据载入操作完成. 如果 2 种时延差异过大，则隐藏的时延无法使整体时延明显降低. 要使得组预读带来明显的性能提升，较为理想的情况是内存访问时延和 CPU 计算时延差异不大. 因此，我们的推论是：$\text{Server}_{\text{AMD}}$ 的内存访问时延和 CPU 计算的时延差异相比于 $\text{Server}_{\text{Intel}}$ 要小很多，因此带来的性能提升也明显很多. 后续的实验中，HyBS 均启用了组预读策略.

3.3 特征比特位图的影响

采用特征比特位图可以大幅度降低 HyCBF 的假阳性率，进而减少因假阳性引发的回溯操作. 为了更加量化地分析特征比特位图的影响，我们比较了 HyBS 在启用和关闭特征比特位图时的回溯操作次数变化情况，并以 CBS 中的回溯次数作为参考对比. 实验所用的测试集以规则集中的数据名前缀为基础生成，以其中最长数据名的组件数量为长度阈值（我们使用的测试集中最长数据名的组件数量为 7）. 当测试集中某个数据名长度小于阈值时，我们在该数据名后面填充足够多的随机组件，使其长度达到阈值设定的长度，在这种极端情况下回溯操作的次数理论上将达到最大值. 我们将布隆过滤器的假阳性率阈值设置为 0.15%，Hash 函数的数量设置为 3，以此计算得出布隆过滤器的长度. 其中，HyBS 在计算时去除了重复的“共享标记”.

实验结果如图 12 所示. 未启用特征比特位图的 HyBS（即 HyBS 对应的测试结果）在不同的数据集上，触发了 762~4 391 次回溯操作，在总查找次数中的占比为 0.143%~0.152%，可以发现这个数值和我们设置的最大假阳性阈值 0.15% 相接近. 这正好验证了 2.3 节中的推论：在计算计数布隆过滤器长度时，虽然有必要重复插入“共享标记”，但在计算时应当避免重复计算它们，这样才能得到准确的计算结果，避免假阳性率出现冗余. CBS 则没有考虑“共享标记”的情况，测试结果显示其触发回溯的查找次数占比在 0.01%~0.026%，相比设置的假阳性阈值低很多. 重复插入“共享标记”并不会导致假阳性率超过阈值，相反，假阳性率会进一步下降，但这样的低假阳性率是冗余的，并且是以更大的存储开销为代价的. 启用了特征比特位图后的 HyBS（即 HyBS-FB 对应的测试结果）触发回溯操作的查找次数占比仅为 0.003%~0.004%，相比未启用特征比特位图的 HyBS 减少了

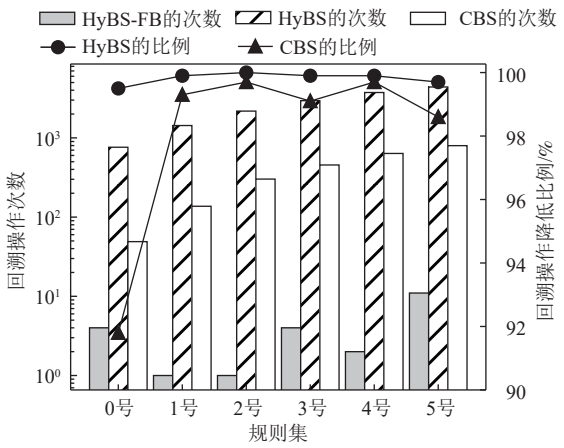


Fig. 12 The amount of backtracks

图 12 回溯次数

99.48%~99.95% 的回溯操作，相比 CBS 则减少了 91.84%~99.69% 的回溯操作. 根据图 12 的实验结果可知，在引入特征比特位图后，即使考虑到“共享标记”对假阳性率的影响，HyBS 的布隆过滤器（即 HyCBF）大小依然有压缩空间. 在后续实验中，HyBS 均启用了特征比特位图.

3.4 查找速度

3.1~3.3 节的测试结果显示，HyBS 所采用的优化技术在压缩存储和提升查找效率方面效果明显. 为了评估 HyBS 的整体查找性能，我们针对理想情况和极端情况在 $\text{Server}_{\text{AMD}}$ 和 $\text{Server}_{\text{Intel}}$ 上进行了查找测试. 首先测试了最理想情况下的性能表现，此时所有测试集中的数据名和对应规则集中的数据名完全一致，因此触发回溯操作的可能性最小. $\text{Server}_{\text{AMD}}$ 的测试结果如图 13(a)所示，HyBS 的查找最高可达 5.29 MNPS，是 CBS 的查找速度的 2.2~3 倍，相比 Nametrie 则可加速 5.1~8 倍. 图 13(b)展示了 $\text{Server}_{\text{Intel}}$ 的测试结果，受限于较低的主频，以及 2.7 节中对组预读策略的分析，Intel 处理器上 HyBS 的性能表现比 AMD 处理器上的差很多，最大能到 2.21 MNPS，相比 CBS 加速比能达到 1.2~1.5 倍，相比 Nametrie 则可达 2.3~3.5 倍.

为了评估极端情况下的性能表现，我们比较了这 3 种方法在采用 3.3 节中所描述的数据名填充方法所构造的测试集上进行的性能测试. $\text{Server}_{\text{AMD}}$ 和 $\text{Server}_{\text{Intel}}$ 的测试结果分别如图 14(a)和图 14(b)所示. 可以看出这 3 种方法均出现了性能下降，其中 HyBS 在 $\text{Server}_{\text{AMD}}$ 和 $\text{Server}_{\text{Intel}}$ 分别出现了 25% 和 27% 的性能下降，但其查找速度相比其他 2 种方法依然更快. 在 AMD 处理器上，HyBS 的查找速度最大可达 3.84 MNPS，相比 CBS 和 Nametrie 速度最大可加速 4.1~5.6 倍. 而在 Intel 处理器上，HyBS 的查找速度可达 1.65 MNPS，相比 CBS 和 Nametrie 速度最大可加速 2.2~2.5 倍.

3.5 更新速度

对于规则更新这一重要性能指标，我们同样在不同 CPU 上进行了性能测试. 图 15(a)和图 15(b)分别展示了这 3 种方法在 $\text{Server}_{\text{AMD}}$ 和 $\text{Server}_{\text{Intel}}$ 上的规

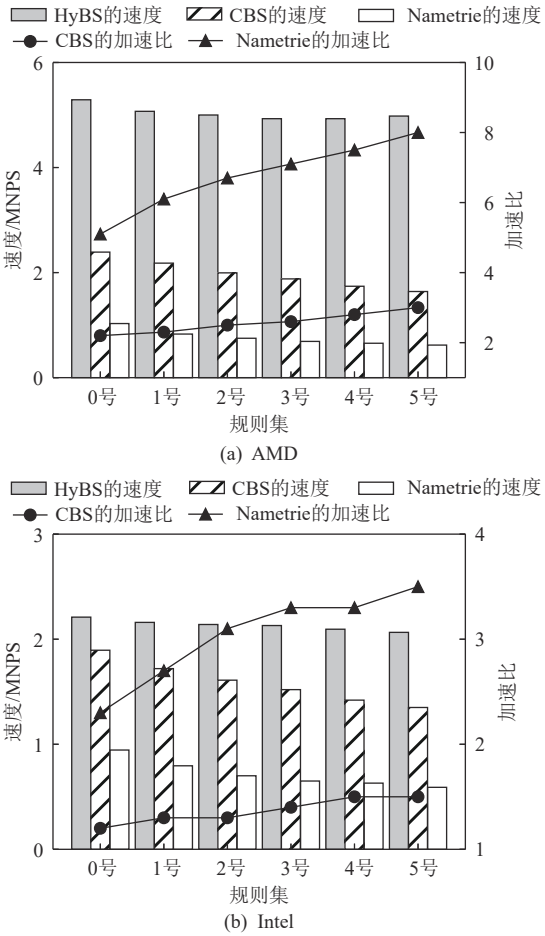


Fig. 13 The lookup speed on ideal case

图 13 理想情况下的查找速度

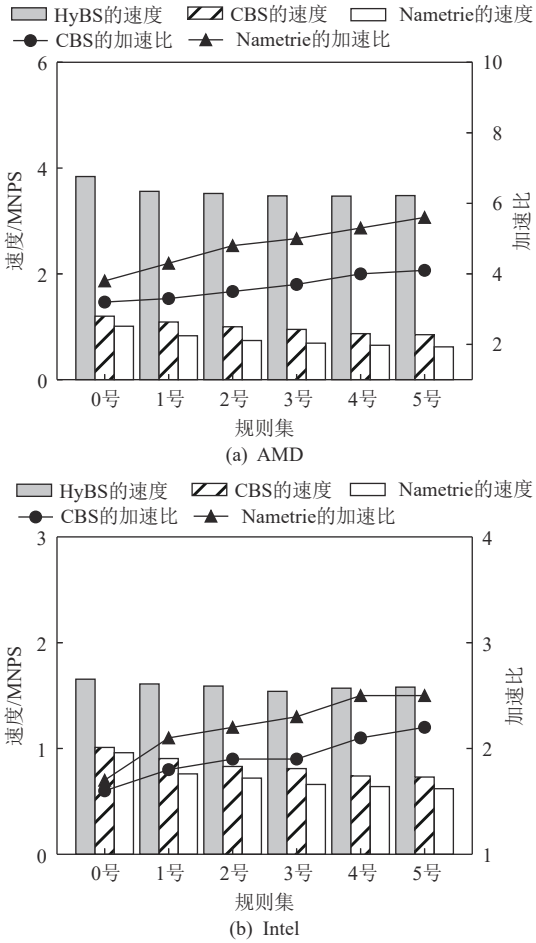


Fig. 14 The lookup speed on extreme case

图 14 极端情况下的查找速度

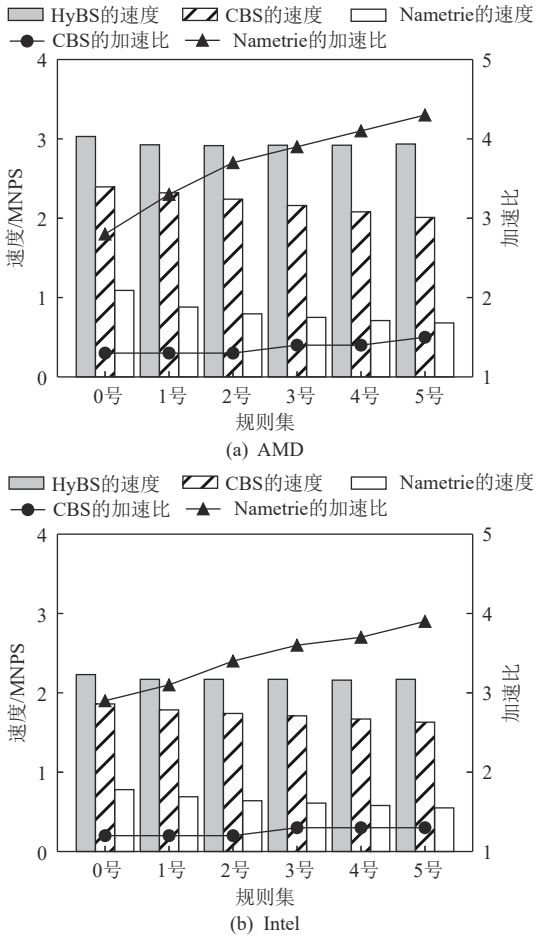


Fig. 15 The inserting speed

图 15 插入速度

则插入性能. 得益于“自底向上”的更新策略, HyBS 相比其他 2 种方法具有更快的更新速度. 在 AMD 处理器上的测试结果表明, HyBS 相比 CBS 可加速 1.3~1.5 倍, 相比 Nametrie 则可加速 2.8~4.3 倍. 而在 Intel 处理器上的测试结果表明, HyBS 相比 CBS 可加速 1.2~1.3 倍, 相比 Nametrie 则可加速 2.9~3.9 倍. 规则删除的速度如图 16(a)和图 16(b)所示. 虽然采用了相同的更新方法, 但插入时需要为冲突的规则分配内存, 而删除则仅需要释放占用的内存, 这使得删除操作的速度远远快于插入操作. $Server_{AMD}$ 的测试结果显示, HyBS 的删除速度可达 6.26 MNPS, 相比 CBS 可加速 2.9~3.2 倍, 相比 Nametrie 则可加速 6.8~10.2 倍. $Server_{Intel}$ 的测试结果则显示 HyBS 的规则删除速度最大可达 3.57 MNPS, 相比 CBS 可加速 2.1~2.3 倍, 相比 Nametrie 则可加速 4. 4~5. 8 倍.

3.6 吞吐量和可扩展性

为了对 HyBS 进行系统级的性能验证, 我们测试了加载了 HyBS 插件的 VPP 的两大性能指标, 包括吞吐量和可扩展性. 此外, 作为对比, 我们也在插件中实现了 CBS 和 Nametrie. 如图 17 所示, 测试环境由 2 台互联的 $Server_{AMD}$ 组成, 网卡则采用了 2 块双 100 Gbps 接口的 Mellanox ConnectX-5 网卡. 为避免 NUMA(non-uniform memory access)架构[44]下远程内存访问带来的额外时延, VPP 所有的 CPU 都位于相同的 NUMA 节点. 此外, 为了最大限度发挥 CPU 的单核性能, 我们关闭了 CPU 超线程特性. 一台 $Server_{AMD}$ 运行加载了数据名查找插件的 VPP, 其中的包处理节点拓扑图则如图 8 所示. 在验证 HyBS 的可扩展性时, 需要启用多个 CPU 以及多个网卡队列. 目前网卡的多队列调度采用的是接收端负载均衡(RSS)技术, 需要 TCP/IP 包头中的五元组信息, 该技术无法支持基于数据名的队列调度. 因此, 我们在附带数据名的数据包前面封装一个填充了随机 IP 地址、MAC 地址和端口信息的 TCP/IP 包头, 以此实现数据包的多队列调度. 另一台服务器则作为发包测试仪, 从网卡其中一个接口(即发送接口)发送, 同时监听另一个接口(即接收接口), 以此分析 VPP 在加载不同的数据名查找方法以及启用不同数量的 CPU 时的性能表现. 为适

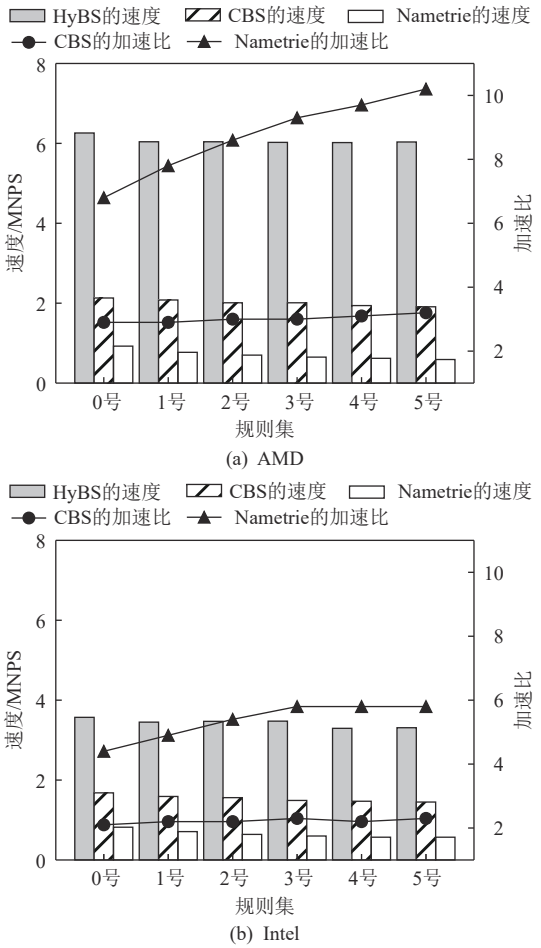


Fig. 16 The removal speed

图 16 删除速度

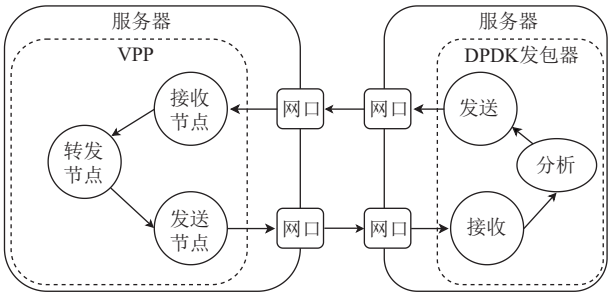


Fig. 17 The test architecture of VPP

图 17 VPP 测试架构

配数据名的长度,我们将数据包大小设置为 512 B.

实验结果如图 18 所示,我们统计了不同数据名查找方法在启用了不同数量的 CPU 时的吞吐量. 为方便对比分析,我们将不做任何查找操作的“空”转发时的吞吐量数据作为参照基准,即图 18 中的 Empty. 在我们的测试环境下,“空”转发的吞吐量为 93.02 Gbps. 当仅启用 1 个 CPU 时,HyBS 的吞吐量可达 12.41 Gbps,而 CBS 和 Nametrie 则分别是 4.44 Gbps 和 3.76 Gbps. 观察数据包大小和吞吐量数据可以发现,这 3 种方法在实际系统中相比于 3.4 节中的本地测试均出现了性能降低. 不同于本地测试,系统测试中查找模块需要和其他模块共享计算和存储资源,因此出现性能下降是符合预期的. 随着 CPU 的数量增长,3 种方法的吞吐量均有所提升,其中 HyBS 的增长速率最快. 当启用 8 个 CPU 时,HyBS 的吞吐量已经和“空”转发一致. 因此,在我们的测试环境下,HyBS 利用 8 个 CPU 核即可实现快速的数据名查找转发,而对于 CBS 和 Nametrie 则分别需要 13 个和 14 个 CPU,相比 HyBS 需要消耗更多的物理资源. 因此,和已有的这 2 种方法相比,HyBS 具有更高的吞吐量以及更好的可扩展性.

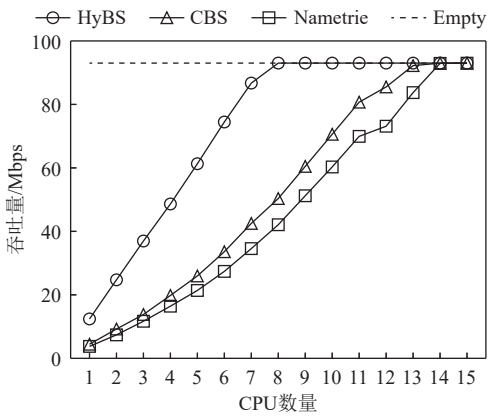


Fig. 18 The throughput and scalability

图 18 吞吐量和可扩展性

4 总 结

本文提出了一种混合计数布隆过滤器 HyCBF 和 HyCBF 辅助的二分数据名查找方法 HyBS. 在 HyCPF 中将数据名前缀布隆过滤器和前缀标记布隆过滤器合二为一,在不增加额外开销的情况下为二分搜索过程提供更丰富的指向信息以实现加速. 此外,针对二分搜索过程中的回溯问题,为 HyCBF 中保存的每条记录构造特征比特位图,进一步减少查找开销. 最后,采用“自底向上”的更新策略加速增量更新. 实验结果表明,HyBS 相比现有方法存储开销更小,回溯次数更少,更新性能更好,在不同 CPU 上的查找速度都更快. 基于 VPP 框架进行系统实现,可进一步通过批处理和组预读提高 HyBS 性能,充分说明了其用于构建高性能数据名查找引擎的潜力!

作者贡献声明:许可负责核心技术点的设计实现以及论文撰写;李彦彪负责系统架构并参与方法设计和论文撰写;谢高岗负责实验方案设计;张大方指导本文工作开展和论文撰写.

参 考 文 献

[1] Yang Tong, Xie Gaogang, Li Yanbiao, et al. Guarantee IP lookup performance with FIB explosion[C]//Proc of the 38th ACM Conf on SIGCOMM. New York: ACM, 2014: 39−50

[2] Asai H, Ohara Y. Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup[J]. ACM SIGCOMM Computer Communication Review, 2015, 45(4): 57−70

[3] Huang Jhihyu, Wang Pichung. TCAM-based IP address lookup using longest suffix split[J]. IEEE/ACM Transactions on Networking, 2018, 26(2): 976−989

[4] Zec M, Rizzo L, Mikuc M. DXR: Towards a billion routing lookups per second in software[J]. ACM SIGCOMM Computer Communication Review, 2012, 42(5): 29−36

[5] Eatherton W, Varghese G, Dittia Z. Tree bitmap: Hardware/software IP lookups with incremental updates[J]. ACM SIGCOMM Computer Communication Review, 2004, 34(2): 97−122

[6] Degermark M, Brodnik A, Carlsson S, et al. Small forwarding tables for fast routing lookups[J]. ACM SIGCOMM Computer Communication Review, 1997, 27(4): 3−14

[7] Waldvogel M, Varghese G, Turner J, et al. Scalable high speed IP routing lookups[C]//Proc of the 21st ACM Conf on SIGCOMM. New York: ACM, 1997: 25−36

[8] Serhane O, Yahyaoui K, Nour B, et al. A survey of ICN content

naming and in-network caching in 5G and beyond networks[J]. IEEE Internet of Things Journal, 2020, 8(6): 4081−4104

[9] Li Jiawei, Luo Hongbin, Jin Mingshuang, et al. Solving selfish routing in route-by-name information-centric network architectures[C]//Proc of the 19th IEEE Global Communications Conf. Piscataway, NJ: IEEE, 2018: 386−392

[10] Guan Yu, Huang Lemei, Zhang Xinggong, et al. Name-based routing with on-path name lookup in information-centric network[C]//Proc of the 24th IEEE Int Conf on Communications. Piscataway, NJ: IEEE, 2018: 1495−1500

[11] Dilley J, Maggs B, Parikh J, et al. Globally distributed content delivery[J]. IEEE Internet Computing, 2002, 6(5): 50−58

[12] Harahap E, Wijekoon J, Tennekoon R, et al. Router-based request redirection management for a next-generation content distribution network[C/OL]//Proc of the 3rd IEEE Globecom Workshops. Piscataway, NJ: IEEE, 2013[2021-11-10]. https://ieeexplore.ieee.org/abstract/document/6825123

[13] Dong Lijun, Zhang Dan, Zhang Yanyong, et al. Performance evaluation of content based routing with in-network caching[C]//Proc of the 20th Annual Wireless and Optical Communications Conf. Piscataway, NJ: IEEE, 2011: 43−48

[14] Abdallah H B H, Louati W. Ftree-CDN: Hybrid CDN and P2P architecture for efficient content distribution[C]//Proc of the 27th Euromicro Int Conf on Parallel, Distributed and Network-Based Processing. Piscataway, NJ: IEEE, 2019: 438−445

[15] Wang Yi, Zu Yuan, Zhang Ting, et al. Wire speed name lookup: A GPU-based approach[C]//Proc of the 10th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2013: 199−212

[16] Li Dagang, Li Junmao, Zheng Du. An improved trie-based name lookup scheme for named data networking[C]//Proc of the 21st IEEE Symp on Computers and Communication. Piscataway, NJ: IEEE, 2016: 1294−1296

[17] Wang Yi, Dai Huichen, Zhang Ting, et al. GPU-accelerated name lookup with component encoding[J]. Computer Networks, 2013, 57(16): 3165−3177

[18] Song Tian, Yuan Haowei, Crowley P, et al. Scalable name-based packet forwarding: From millions to billions[C]//Proc of the 2nd ACM Conf on Information-Centric Networking. New York: ACM, 2015: 19−28

[19] Ghasemi C, Yousefi H, Shin K G, et al. A fast and memory-efficient trie structure for name-based packet forwarding[C]//Proc of the 26th IEEE Int Conf on Network Protocols. Piscataway, NJ: IEEE, 2018: 302−312

[20] Ghasemi C, Yousefi H, Shin K G, et al. On the granularity of TRIE-based data structures for name lookups and updates[J]. IEEE/ACM Transactions on Networking, 2019, 27(2): 777−789

[21] Byun S H, Lee J, Sul D M, et al. Multi-worker NFD: An NFD-compatible high-speed NDN forwarder[C]//Proc of the 7th ACM Conf on Information-Centric Networking. New York: ACM, 2020: 166−168

[22] Yuan Haowei, Crowley P. Reliably scalable name prefix lookup [C]//Proc of the 11th ACM/IEEE Symp on Architectures for Networking and Communications Systems. Piscataway, NJ: IEEE, 2015: 111−121

[23] Wang Yi, Xu Boyang, Tai Dongzhe, et al. Fast name lookup for named data networking[C]//Proc of the 22nd IEEE Int Symp of Quality of Service. Piscataway, NJ: IEEE, 2014: 198−207

[24] Yuan Haowei, Crowley P. Scalable pending interest table design: From principles to practice[C]//Proc of the 33rd IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2014: 2049−2057

[25] Dai Huichen, Lu Jianyuan, Wang Yi, et al. BFAST: High-speed and memory-efficient approach for NDN forwarding engine[J]. IEEE/ACM Transactions on Networking, 2016, 25(2): 1235−1248

[26] Perino D, Varvello M, Linguaglossa L, et al. Caesar: A content router for high-speed forwarding on content names[C]//Proc of the 10th ACM/IEEE Symp on Architectures for Networking and Communications Systems. New York: ACM, 2014: 137−148

[27] Wang Yi, Pan Tian, Mi Zhian, et al. Namefilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters[C]//Proc of the 32nd IEEE Conf on Computer Communications. Piscataway, NJ: IEEE, 2013: 95−99

[28] Quan Wei, Xu Changqiao, Vasilakos A V, et al. TB2F: Tree-bitmap and Bloom-filter for a scalable and efficient name lookup in content-centric networking[C]//Proc of the 14th IFIP Networking Conf. Piscataway, NJ: IEEE, 2014: 395−403

[29] Huang Kun, Wang Zhaohua, Xie Gaogang. Scalable high-speed NDN name lookup[C]//Proc of the 14th Symp on Architectures for Networking and Communications Systems. New York: ACM, 2018: 55−65

[30] He Dacheng, Zhang Dafang, Xu Ke, et al. A fast and memory-efficient approach to NDN name lookup[J]. China Communications, 2017, 14(10): 61−69

[31] Xu Ke, Zhang Dafang, Li Yanbiao. Longest name prefix match on multi-core processor[C]//Proc of the 21st IEEE Int Conf on High Performance Computing and Communications. Piscataway, NJ: IEEE, 2019: 1035−1042

[32] Kirsch A, Mitzenmacher M. Less hashing, same performance: Building a better Bloom filter[C]//Proc of the 14th European Symp on Algorithms. Berlin: Springer, 2006: 456−467

[33] Lu Jianyuan, Yang Tong, Wang Yi, et al. One-hashing Bloom filter[C]//Proc of the 23rd IEEE Int Symp on Quality of Service. Piscataway, NJ: IEEE, 2015: 289−298

[34] Lu Jianyuan, Wan Ying, Li Yang, et al. Ultra-fast Bloom filters using SIMD techniques[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 30(4): 953−964

[35] Bender M A, Farach-Colton M, Johnson R, et al. Don't thrash: How to cache your Hash on flash[J]. arXiv preprint, arXiv: 1208.0290, 2012

[36] Dai Haipeng, Zhong Yuankun, Alex L, et al. Noisy Bloom filters for multi-set membership testing[C]//Proc of the 39th ACM Int Conf on

Measurement and Modeling of Computer Science. New York: ACM, 2016: 139−151

[37] Yang Tong, Alex L, Shahzad M, et al. A shifting framework for set queries[J]. IEEE/ACM Transactions on Networking, 2017, 25(5): 3116−3131

[38] Pfaff B, Pettit J, Koponen T, et al. The design and implementation of open vSwitch[C]//Proc of the 12th Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2015: 117−130

[39] Barach D, Linguaglossa L, Marion D, et al. High-speed software data plane via vectorized packet processing[J]. IEEE Communications Magazine, 2018, 56(12): 97−103

[40] Ma Xiaoxiao, Yang Fan, Wang Zhan, et al. Survey on smart network interface card[J]. Journal of Computer Research and Development, 2022, 59(1): 1−21 (in Chinese)
(马潇潇, 杨帆, 王展, 等. 智能网卡综述[J]. 计算机研究与发展, 2022, 59(1): 1−21)

[41] Zhu Wenjun, Li Peng, Luo Baozhou, et al. Research and implementation of high performance traffic processing based on Intel DPDK[C]//Proc of the 9th Int Symp on Parallel Architectures, Algorithms and Programming. Piscataway, NJ: IEEE, 2018: 62−68

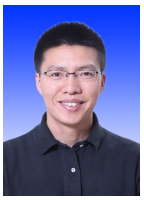
[42] Reddit. Dataset of reddit posts and comments[EB/OL]. (2021-09-08)[2021-11-10]https://www.reddit.com/r/datasets/comments/la0w4n/dataset_of_reddit_posts_and_comments/

[43] Appleby A. MurmurHash[CP/OL].(2011-03-01)[2021-11-10]. https://sites.google.com/site/murmurhash/

[44] Lameter C. NUMA (non-uniform memory access) an overview: NUMA becomes more common because memory controllers get close to execution units on microprocessors[J]. Queue, 2013, 11(7): 40−51

Xu Ke, born in 1992. Master. His main research interests include network system, in-network computation, and data packet computation.
许　可, 1992 年生. 硕士. 主要研究方向为网络系统、在网计算和数据包计算.

Li Yanbiao, born in 1986. PhD. His main research interests include network system, data packet processing algorithms, and routing security.
李彦彪, 1986 年生. 博士. 主要研究方向为网络系统、数据包处理算法、路由安全.



Xie Gaogang, born in 1974. PhD, professor. His main research interests include Internet architecture, data packet processing and forwarding, and Internet measurement.
谢高岗, 1974 年生. 博士, 研究员. 主要研究方向为网络体系结构、数据包处理和转发、网络测量.

Zhang Dafang, born in 1959. PhD, professor. His main research interests include reliable network and system, information security of networking, and next generation of Internet.
张大方, 1959 年生. 博士, 教授. 主要研究方向为可信网络与系统、网络信息安全、下一代互联网.