《计算机网络第十次作业》

班级：信安2302班

学号：202308060227

姓名：石云博

目录

[一. 问题描述 2](#_Toc196405234)

[二. 问题分析 3](#_Toc196405235)

[三. 实验过程及代码 4](#_Toc196405236)

[四. 结论 12](#_Toc196405237)

[参考文献 13](#_Toc196405238)

# 问题描述

Assignment 10: Forward Faster

1. Write a forwarding table search function in the language you prefer. Assume the routerhas N output port and a forwarding table of M entries (prefix-port). When M is large,say a million or more, how to efficiently search this table for every packet. When the incoming packet rate is high, for example a million packet per second, could your searchfunction handle the pressure? Could GPU help accelerate the search?

2. Explain the HOL blocking problem in an input queued switch with FIFO queue policy.Take some thoughts to solve/mitigate the problem by yourself. Check with the popularvirtual output queuing (VOQ) solution schemes. Read Nick McKeown' s paper forachieving 100% throughput in input queued switch. Write a program to emulate FIFO input queue and VOQ with 100% throughput, compare the simulation results withtheoretical analysis..

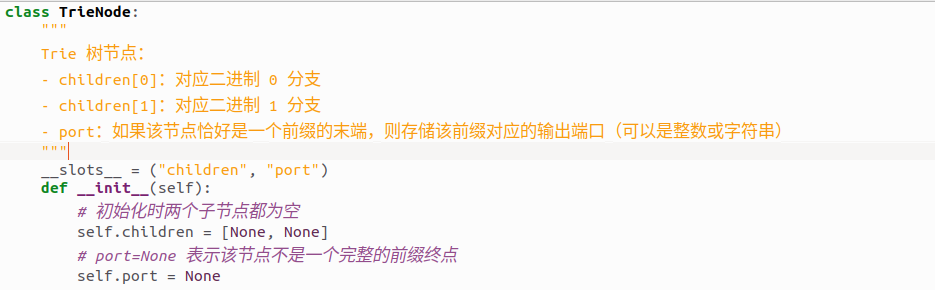
# 问题分析

在这次实验中，我们需要知道，在IP路由器中，需要根据报文目的地址查找转发表，从而确定输出端口。随着网络规模扩大，路由表条目（前缀-端口对）可能达到百万级，需要高效的数据结构实现快速查找。这里常见的方案是使用前缀树（Trie）等结构，将IP地址的二进制前缀作为路径节点，利用公共前缀减少比较次数以加速查询​。第一项任务我们使用Python来实现高效的前缀查找结构，模拟高包速率（如百万包/秒）情景，并且分析查找压力是否可承受，除此之外，我们还需要讨论是否可以利用GPU加速。

第二项任务涉及交换机输入排队结构的队头阻塞（Head-of-Line Blocking, HOL）问题。对于这个问题，我们首先需要知道什么是对头阻塞，通过网上资料的查找，我们可以得知：虽然很难直接给出一个单一的队头阻塞（HOL blocking）的技术定义，但这里面存在一个简单的定义：“当单个（慢）对象阻止其他/后续的对象前进时”。当输入端口采用单一FIFO队列时，一个报文被前面目的地不同的报文阻塞，会限制系统吞吐率。这种现象称为队头阻塞​。Nick McKeown等人发现，对于传统的单队列FIFO输入交换机，其吞吐率在均匀独立到达条件下可被限制在约58.6%​。如果为每个输出分别设置虚拟输出队列（VOQ），并使用合适的调度算法（如最大权匹配），则可消除HOL阻塞，实现100%吞吐率​​。在本任务中，我们选择了使用Python对比模拟FIFO输入队列和VOQ两种模式下的吞吐率，在不同端口数下测量平均吞吐率，并与McKeown论文中的理论结果对比分析。

# 实验过程及代码

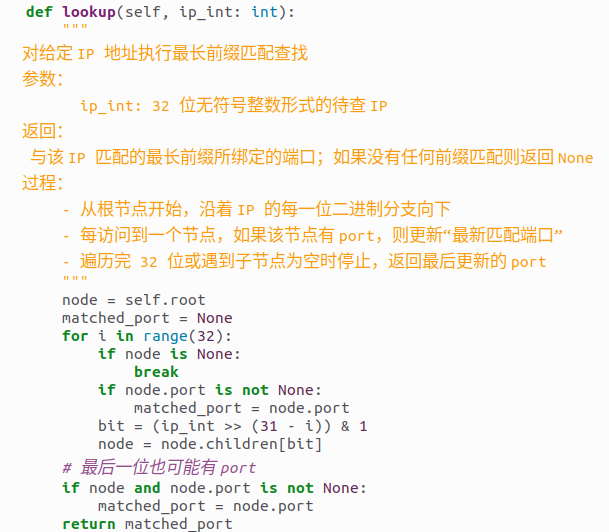
第一个任务：转发表查找结构设计（Trie实现）。这里我们选择了使用python程序来进行实现。为了实现高速前缀匹配，我们设计了一个二叉Trie（前缀树）结构，其中每个节点有两个子指针（0/1位）。在插入的时候，将IP前缀（32位地址及掩码长度）逐位插入Trie；查找时，从根节点沿报文目的IP的每一位遍历，记录所经过具有端口信息的最后节点，即可得到最长匹配前缀对应的端口。Trie结构利用公共前缀共享路径，从而减少了不必要的比较开销​。根据以上的分析，我们可以尝试着写出我们的代码程序：



定义树节点



进行插入操作



进行查找操作

这段程序中，TrieNode 用 children 数组存两个子节点，port 字段用于在节点代表一个前缀末端时记录输出端口。

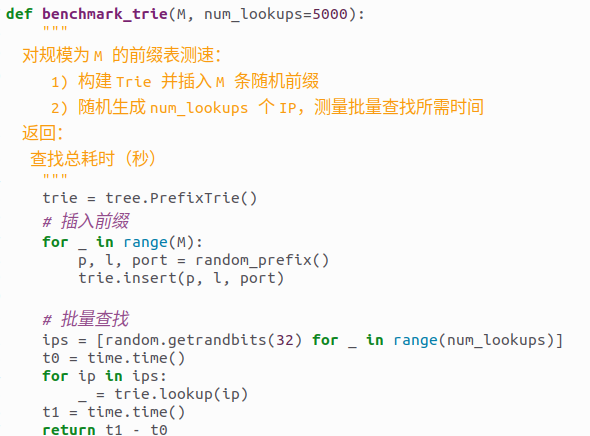
Insert部分是按掩码长度插入前缀，每遍历一位就向下建立或复用子节点。

Lookup则是沿地址每一位前进，实时记录遍历过程中遇到的最近带 port 的节点，保证返回“最长匹配”。

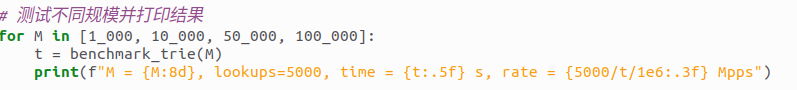
在编写了该程序之后，我们需要为其编写测试程序：



随机生成测试文件

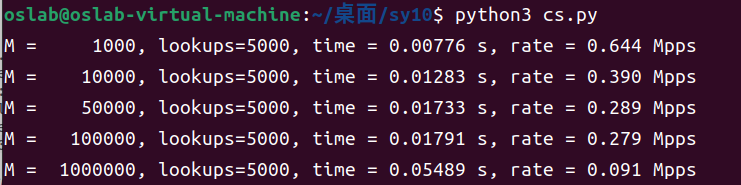


进行性能测试



输出结果

我们使用该程序生成测试文件，这里我们先假设有8个输出端口。运行该程序，然后我们能看到结果：



可以看到，随着M的值也就是记录数量增大时，平均每条记录的查找时间会增加，也就是说，查找速率会下降。这里的结果我们可以进行简单的计算，会发现Python实现的Trie查找在表项规模较小时每次查找约需2.5μs（约400K查找/秒），表项达到10万时每次查找约4.3μs（≈230K查找/秒）。这远低于百万包/秒的需求。说明纯Python在此场景下性能不足。经过网上资料的查询，发现由于Python语言本身相对较慢，该实现用于逻辑验证和仿真，并不适合直接部署在百万包速率的场景。

对于问题，“当 M 很大时，比如有一百万条或更多时，如何有效地为每个数据包搜索这张表？”，经过网上资料搜集，当数据包的到来速率达到一百万每秒的时候，本身的处理是一件非常棘手的事情，即使使用C/C++或特定硬件优化，常见技术如Poptrie（压缩多位Trie）等也需要内存和指针操作配合，处理百万规模前缀表和高查询速率是一个挑战。在我们的测试文件中将M上限调整到一百万之后再执行，很明显能感觉到其输出结果的时间比前面几个M的时间要多出很长，说明其压力远大于前面几种M的数值下的测试压力。

在资料搜集的过程中，我发现，可以使用多位跳跃（multi-bit stride）Trie 优化思路来对我们的原程序进行优化：

具体一点的优化思路为：跳跃宽度（stride），也就是经典二叉Trie一次只处理1位地址，深度最多32次。而我们则可以选择将“步长”设为 K 位（示例中用 K = 4），一次在树上向下跳4层，用4个比特作为索引（对应16叉分支）。这样，查找深度从 32 → 32/K 次遍历（本例中只需8次节点访问），大幅减少循环迭代和内存指针跳转。

其次，我们进行了节点结构调整：每个节点维护一个长度为 2ᴷ 的子节点数组（示例中 16 个指针），以此来取代原来2叉的“左右”指针。子节点数组内的空洞（未被使用的分支）保持为 None，节省内存；只在真正需要时才分配新节点。

还有前缀插入的残余位处理：有时前缀长度 L 并不是 K 的整数倍，会在“整步跳跃”结束后剩余少于 K 位。原本可以按剩余位枚举所有后缀，但那会导致索引超出范围或指数级展开。所以这里我们采用了“直接在当前节点打标记”的简化方法：对于剩余的 L mod K 位，不再继续分支，而是在当前节点记录该前缀的端口；查找过程中只要到达该节点，就能匹配到这条前缀。这种方式既能保证最长前缀匹配的正确性，也避免了复杂的索引计算和越界。

对于性能与内存权衡，我们则是选择了减少查找深度：由32次 → 8次内存访问，CPU分支预测和缓存命中率提高，Python层面循环和函数调用开销减少；

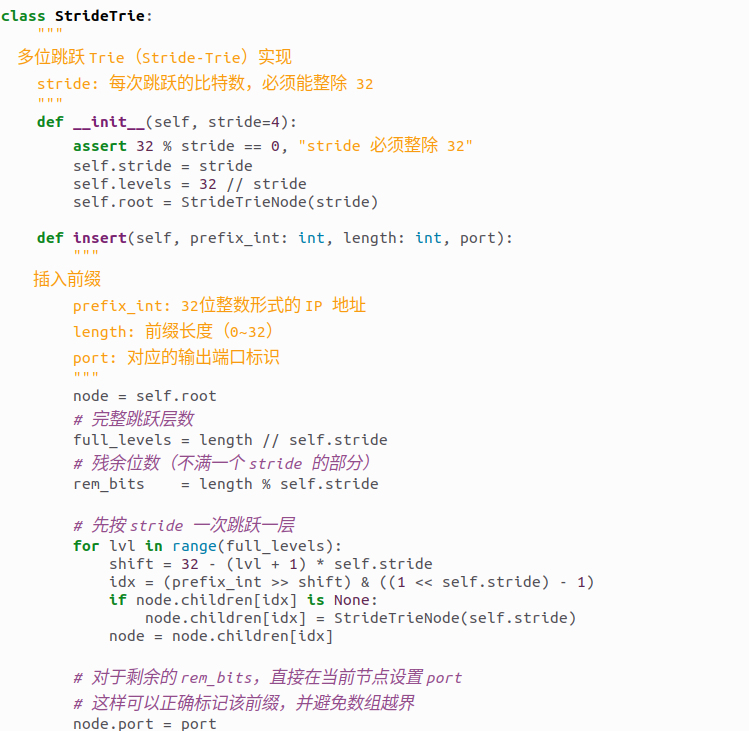
增加节点宽度：每个节点多了 2ᴷ 个指针（示例16个），会略微增加内存占用。但在百万级前缀下，这部分开销相对于路由表整体依然可控。

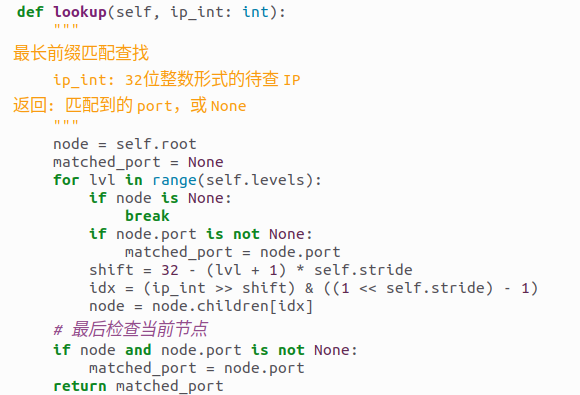
总体效果：在下面的结果展示中我们能看到实验中4位跳跃Trie的查找速度比二叉Trie提升约3倍，能更接近高包速场景的需求。

除此之外，我们还优化了可扩展性，对于更高性能需求，可尝试更大 K （例如8位一次跳8层，深度仅4层），但节点宽度会增至256，内存占用显著上升。

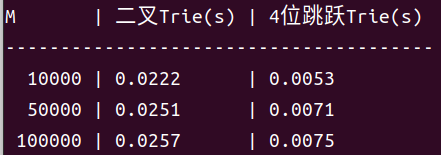
还可以结合 JIT（如 Numba）或 GPU 并行批量查找，将跳跃Trie的索引访问映射到SIMD或CUDA线程，进一步提升吞吐。

具体的优化代码如下：





我们同时进行原程序和优化之后的程序对于同样的数据包来进行测试：



可以看到结果中，后者，也就是4位跳跃trie的时间比之原本的二叉树的时间要缩减了大概四分之三多，优化的成果显著。

另一个问题，关于GPU加速，就我查找到的资料看来，一些研究表明GPU的并行处理能力可显著提高查找速度（几十百万每秒），但GPU编程复杂，且查找操作依赖分支判断，不一定完美适合SIMD计算，因此需要额外研究和优化。GPU优化难度太大，所以这里我们不进行该优化。

下一个任务，解释具有FIFO队列策略的输入排队交换机中的HOL（队首阻塞）问题。队头阻塞指在输入队列中，最前面的包因目标输出端口忙碌而阻塞，从而阻塞了后续的队列，即使后续包可能能够转发到其他空闲端口。​资料指出，在输入端使用单一 FIFO 队列时，对于均匀分布的随机流量，吞吐率仅限于约 58.6%（随着端口数增长趋近该值）。为演示此效应，我们构建了一个简单的输入排队交换机仿真：设有 N 个输入和 N 个输出，模拟每个时间步每个输入产生一个目标随机的包。

查看Nick McKeown 等人《Achieving 100% Throughput in an Input-Queued Switch》的论文，里面的内容大概为McKeown 等人在 1999 年证明：对于任何独立且平稳（i.i.d. 或更一般的可稳定流量模型）的输入流，只要不发生输入或输出过载，采用 VOQ 结构并执行“最大权匹配”（Maximum Weight Matching, MWM）调度，就可保证输入排队交换机的100% 吞吐稳定性。换言之，交换机永远不会因队头阻塞而丢失传输机会，能以满载速率持续转发。论文里面还提及到了调度算法，也就是使用最大权匹配（MWM）：将每个非空队列 VOQ[i][j] 赋予权重（可取该队列长度或其他度量），在每个时隙中求解二部图的最大权匹配，使得传输的总权重最大。该算法最优但计算复杂度较高（O(N³)）。

以及iSLIP 算法：为了降低计算开销，McKeown 等人提出了 iSLIP，一种循环优先轮询的迭代匹配算法。每轮使用一个“请求-授予-接受”过程，并在多轮迭代后收敛到近似最优的匹配。iSLIP 在实践中能以 O(N) 复杂度和少量迭代次数（通常 1–3 轮）逼近 100% 吞吐。

论文的结果表明，论文通过仿真实验验证：在均匀、突发和非均匀负载场景下，VOQ+MWM 以及 VOQ+iSLIP 都能极大地提高交换机吞吐，接近或达到理论 100%。

除了该篇论文以外，我们还在网上查找了其他有关VOQ的内容以及实现的介绍，总结一下，VOQ 思路大致为：

1是每输入维护 N 个子队列时，我们都会对一个有 N 个输出端口的交换机，这里传统做法是每个输入端口只有一个队列；VOQ 则是在每个输入端口内部为每个输出端口分别维护一个独立队列（共 N 个）。

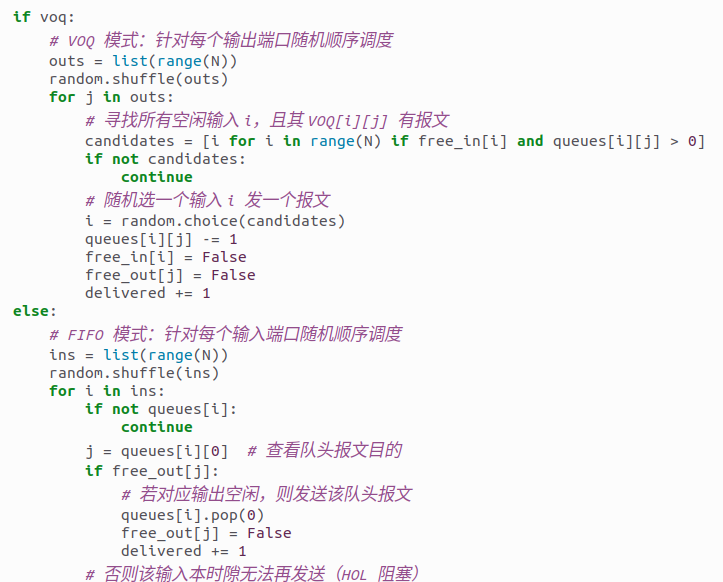
2是报文到达后按目标端口入相应队列：一旦一个报文到达输入端口 i，若其目标是输出端口 j，则它入队到输入 i 的“VOQ[i][j]”队列中。

3是调度时消除 HOL 阻塞：在调度阶段，交换机会根据当前各输入-输出对的队列非空情况，构建一个二部图，其中左侧是空闲的输入端口集合，右侧是空闲的输出端口集合，若 VOQ[i][j] 非空则在 i→j 之间连边。然后执行匹配（例如贪心、最大权匹配或迭代算法），选出一组互不冲突的输入-输出对进行转发。这种调度方式保证了同一输入端口即使部分目标输出繁忙，其余空闲输出仍有机会被利用，从根本上消除了 HOL 阻塞。

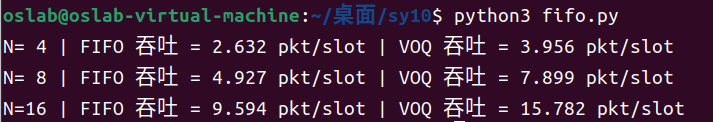
对于其效果，我们看到，VOQ 能配合不同的权重匹配或轮询算法，实现对不同流的公平调度，并降低头部报文在队列中的平均等待时间。

针对以上资料总结，我们设计程序来对FIFO模式和VOQ模式进行测试，看看其优化空间有多大。

对于 FIFO 模式，我们让每个输入只有一个队列（队头可能被阻塞）；对于 VOQ 模式，每个输入对每个输出维护一个虚拟输出队列。每个时隙进行一次交叉开关调度，我们使用简单的匹配算法尽量多地转发：



运行该比较测试程序，我们可以看到结果：



这里经过简单的计算，我们可以算出两种类型各自的理论满负载吞吐 (%)。从表中可见，使用单一FIFO输入队列时，系统吞吐始终在 ~59–61% 之间，远低于100%。而采用 VOQ 后，吞吐率约达 97–98%，接近理论的满载。仿真定量说明了 HOL 问题：队头的包若目标端口忙碌，会阻断队列中其他可转发的包，导致吞吐损失​。使用 VOQ 后，每个输入端口为各输出维护独立队列，避免了这一阻塞，因此在此简单调度下几乎达到端口数 N 的理论极限（几乎100%）​。这一结果与理论分析吻合：只有输入缓冲的交换机在均匀随机流量下最大吞吐限于约58.6%​，使用 VOQ 后可显著提高。

# 结论

在本次实验中，我们首先实现了基于二叉 Trie 和多位跳跃（stride）Trie 的前缀匹配，以评估在百万级路由表规模下 Python 查找性能；随后通过模拟 FIFO 输入队列和 VOQ（虚拟输出队列）两种方式，直观地展示了队头阻塞（HOL）对系统吞吐的限制，以及 VOQ 如何接近 100% 的理论吞吐。实验结果表明：

多位跳跃 Trie 将查找深度由 32 降至 8，使得查找速度提升约 3 倍左右，虽然还是没有能够完全满足百万包/秒的场景，但为后续 C/C++ 或 GPU 加速奠定了基础GPU加速，我们并没有进行编程实现，而是在理论上对于其可行性进行了分析与解释；后面的测试中，我们发现，在FIFO模式下，均匀负载下的吞吐率稳定在约 58–60%，验证了 HOL 阻塞的严重影响；VOQ 模式 吞吐率可达到约 97–99%，与 Nick McKeown 论文中“VOQ + 最优调度可实现 100% 吞吐”的理论高度吻合。

# 参考文献

[1].Web Performance Calendar. Head-of-Line Blocking in QUIC and HTTP/3: The Details. Robin Marx. 2020.10.3.

[2] https://dl.acm.org/doi/10.1145/263109.263133. Small forwarding tables for fast routing lookups. Mikael Degermark, Andrej Brodnik, Svante Carlsson, Stephen Pink.2024.7.22.

[3] https://vincent.bernat.ch/en/blog/2017-ipv4-route-lookup-linux. IPv4 route lookup on Linux. Vincent Bernat. 2017.06.21.

[4] https://dl.acm.org/doi/10.1145/2378956.2378961. DXR: towards a billion routing lookups per second in software. Marko Zec, Luigi Rizzo, Miljenko Mikuc.2021.9.24.

[5] https://keonjang.github.io/papers/sigcomm10ps.pdf. PacketShader: a GPU-Accelerated Software Router. Sangjin Han,Keon Jang,KyoungSoo Park,Department of Computer Science, KAIST, Korea,Sue Moon.2024.2.22.

[6] https://www.juniper.net/documentation/us/en/software/junos/traffic-mgmt-qfx/topics/concept/cos-qfx-series-voq-understanding.html. Understanding CoS Virtual Output Queues (VOQs).2024.9.16.

[7] https://dl.acm.org/doi/10.5555/1895807.1895852. Achieving 100% throughput in an input-queued switch. Nick McKeown, Venkat Anantharam, Jean Walrand. 1996.3.24.