Optimization of Packet Forwarding in Best-effort Routers

Miguel Ángel Ruiz Sánchez

UNIVERSITE DE NICE-SOPHIA ANTIPOLIS - UFR SCIENCES

Ecole Doctorale STIC

THESE

présentée pour obtenir le titre de
Docteur en SCIENCES
de l'Université de Nice Sophia Antipolis
Spécialité: Informatique
par

Miguel Ángel RUIZ SÁNCHEZ

Optimization of Packet Forwarding in Best-effort Routers

(Optimisation de la réexpédition de paquets dans les routeurs best-effort)

Thèse dirigée par M. Walid DABBOUS
Laboratoire d'accueil:
INRIA Sophia Antipolis
projet Planète

Soutenue publiquement le 5 septembre 2003 devant le jury composé de:

M. Ernst W. BIERSACK	Président	Institut Eurécom
M. Jean-Jacques PANSIOT	Rapporteur	Université Louis Pasteur, Strasbourg
M. Samir TOHMÉ	Rapporteur	ENST Paris
M. Roland SCHUTZ	Examinateur	Thales Communications
M. Laurent TOUTAIN	Examinateur	ENST Bretagne
M. Walid DABBOUS	Directeur de thèse	INRIA Sophia Antipolis

A mis padres y mi esposa

Résumé

La tâche principale d'un routeur est d'acheminer des paquets jusqu'à leur destination finale en passant par les différentes réseaux. Comme chaque paquet est traité individuellement, la performance d'un routeur dépend du temps nécessaire pour traiter chaque paquet. Due à la croissance et à la diversité du trafic dans l'Internet, le traitement nécessaire pour acheminer des paquets doit être optimisé. Cette thèse propose des algorithmes pour optimiser la performance du traitement de paquets lors de leur acheminement dans les routeurs best-effort.

Pour acheminer (réexpédier) des paquets, un routeur doit tout d'abord rechercher l'information de routage correspondant à chaque paquet. La recherche d'information de routage est basée sur l'adresse destination du paquet et elle s'appelle consultation d'adresse. Nous proposons dans cette thèse deux mécanismes pour la mise à jour incrémentale des table de routage basées sur des tries multibit. Tout d'abord, nous déterminons les conditions nécessaires pour supporter des mises à jour incrémentales dans les tries multibit. À partir de ces conditions, nous proposons des algorithmes et des structures de données pour effectuer ces mises à jour incrémentales. En particulier, nous proposons une structure de données que nous appelons le vecteur de bits PN (pour prefix nesting en anglais). Le vecteur de bits PN code un ensemble de préfixes et leurs relations d'inclusion, car cette information est nécessaire pour supporter des mises à jour incrémentales. Nous évaluons la performance de nos mécanismes implémentés en langage C. Nous présentons les performances de nos mécanismes pour les opérations de recherche, insertion et suppression. Nous présentons également les besoins en termes de mémoire.

Une deuxième contribution de cette thèse est l'introduction d'une taxonomie et un cadre de référence pour les algorithmes de consultation rapide d'adresse IP. Notre taxonomie est basée sur l'observation que la difficulté de trouver le plus long préfixe commun avec l'adresse destination est sa double dimension : valeur et longueur. Lorsque nous présentons et classifions les différents mécanismes, l'accent est mis sur le type de transformation que l'on effectue sur l'ensemble de préfixes pour chaque mécanisme. Cette approche unificatrice

que nous proposons nous permet de comprendre et de comparer les compromis des différentes mécanismes. Nous comparons les mécanismes en termes de leur complexité en temps et en espace. Nous comparons aussi leur performance en mesurant le temps de l'opération de recherche. Ces mesures sont réalisées sur une même plateforme et en utilisant une vrai table de routage.

Une troisième contribution de cette thèse est un mécanisme qui optimise l'usage des buffers dans les routeurs pour offrir un haut dégrée d'isolation entre flux. Tout d'abord, nous étudions la fonctionnalité des buffers dans les routeurs et nous déterminons les caractéristiques souhaitables des buffers dans les routeurs. Ensuite nous proposons MuxQ un mécanisme qui fournit un haut degré d'isolation entre flux. MuxQ est basé sur l'idée de protéger la fonction de multiplexage de la fonction d'absorption de rafales d'un buffer. Nous évaluons MuxQ en utilisant le simulateur ns-2. En particulier, nous étudions la capacité de MuxQ pour isoler différent types de flux. Nous comparons les performances de notre mécanisme avec celles des mécanismes Drop-Tail, CSFQ, FRED et DRR. Nous présentons les résultats de simulations avec des conditions de trafic différentes. MuxQ est un mécanisme simple, deployable et qui fournit un haut degré d'isolation de flux, tout en gardant une quantité limitée d'état.

Abstract

The main task of a router is to forward packets through the networks to deliver them to their final destination. Since each packet must be treated individually, the performance of a router depends on the time to process each packet. To keep pace with increasing traffic and wide spectrum of traffic requirements, the packet forwarding capacity of routers need to be optimized. This thesis proposes several algorithms to optimize the performance of the packet forwarding process in best effort routers.

To forward packets, routers must make a forwarding decision. The operation of determining the forwarding information is based on the packet's destination address and it is called address lookup. We propose in this thesis two incremental update mechanisms for address lookup schemes based on the multibit-trie data structure. First, we determine the requirements to support incremental updates in multibit-tries based forwarding databases. Then, we propose algorithms and data structures to support incremental updates. In particular, we propose a data structure called Prefix Nesting bit vector, or PN bit vector for short. The PN bit vector encodes a set of prefixes and their nesting structure, for this information is necessary to support incremental updates. We present performance results of a C-language implementation of our scheme. Performance results are shown in terms of time for the search, insert and delete operations. Memory requirements are also shown.

A second contribution of this thesis is the introduction of a taxonomy and a framework of reference of existing fast address lookup schemes. Our taxonomy is based on the observation that the difficulty of the best prefix matching problem resides in its double dimension: value and length. In our analysis, we emphasize that to improve the performance of the address lookup operation, the different methods make a transformation of the original set of prefixes of the forwarding database. We state the different tradeoffs of the different transformation methods in terms of time and space and we compare the performance of the different schemes. While the most important aspect is the search operation, we also analyze the potential capabilities of the schemes to support incremental updates. We state that to support incremental updates, a mechanism must have additional data structures to keep track of the

prefix transformation process.

A third contribution of this thesis is a mechanism to optimize the use of the buffer in routers to provide flow isolation. First, we study the buffering functionality of IP routers. We find the desired properties of a router buffer system, then we design a mechanism based on these characteristics. We emphasize that buffers in routers have two functions: a multiplexing function and a burst absorbing function. Our mechanism, which we call MuxQ, is based on the idea of protecting the multiplexing function from the burst absorbing function by progressively and dynamically controlling the allocation of buffer space in a FIFO queue. MuxQ is a new queue management mechanism that provides flow isolation by using a very simple algorithm and without using per-flow queuing. We compare the performance of the MuxQ scheme to that of classical Drop-Tail and to that of other proposed schemes, including CSFQ and DRR which provides nearly perfect isolation by using per-flow queuing. By keeping only limited flow-state, our mechanism performs very much better than Drop-Tail. MuxQ achieves performance similar to that of CSFQ but MuxQ does not need modifications to the IP packet header as it is the case for CSFQ. Since MuxQ does not need modifications of the IP packet header and does not expect a special behavior from other routers, MuxQ can be deployed incrementally. We believe that MuxQ is an interesting approach to achieve a high degree of flow isolation with respect to Drop-Tail by using a very simple algorithm.

Remerciements

Bien qu'étant un effort personnel, un travail de thèse ne peut aboutir sans l'aide d'un certain nombre de personnes. Ce sont à elles que j'exprime toute ma gratitude.

Je tiens tout d'abord à exprimer ma profonde reconnaissance pour mon directeur de thèse, Monsieur Walid Dabbous, pour m'avoir laissé la plus grande liberté dans la conduite de mes travaux. Il a mis à ma disposition tous les moyens nécessaires au bon déroulement de ma thèse et il a su me prodiguer de nombreux conseils et encouragements. J'ai largement pu profiter de sa grande acuité scientifique.

Mes plus vifs remerciements vont à Monsieur Jean-Jacques Pansiot, professeur à l'Université Louis Pasteur de Strasbourg, et à Monsieur Samir Tohmé, professeur à l'ENST Paris, pour avoir accepté d'être les rapporteurs de ce travail, malgré leur nombreuses charges. Je veux aussi les remercier pour leurs remarques et suggestions qui m'ont été très précieuses.

Je remercie également, Monsieur Laurent Toutain, maître de conférences à l'ENST Bretagne, et M. Roland Schutz, architecte système à Thales Communications, qui m'ont fait l'honneur de faire partie de mon jury.

J'aimerais également exprimer le plaisir que j'ai eu à travailler avec Monsieur Ernst W. Biersack sur un de mes articles. Je le remercie pour les heures qu'il a investies dans la correction de nombreux manuscrits en anglais. Il a bien voulu me faire profiter de son expertise et a contribué à améliorer ma rédaction en anglais. Je le remercie également d'avoir accepté d'être le président de ce jury.

J'ai eu la chance de faire ma thèse à l'INRIA Sophia Antipolis, un centre de recherche accueillant et vivant où j'ai pu faire des rencontres hautement enrichissantes. Merci à toutes et tous pour avoir partagé ces moments avec moi.

Mes remerciements vont encore à tous mes collègues du projet PLANETE qui m'ont permis de passer des années très agréables et enrichissantes.

J'aimerais aussi remercier du fond du cœur tous mes amis qui en dehors du milieu professionnel m'ont soutenu et aidé durant toutes ces années. Merci à toutes et à tous pour les bons moments passés ensemble.

Mes plus profonds remerciements vont à Francisca et Juan, mes parents. Tout au long de ma vie, ils m'ont toujours soutenu, encouragé et aidé. Ils ont su me donner toutes les chances pour réussir. Qu'ils trouvent, dans la réalisation de ce travail, l'aboutissement de leurs efforts ainsi que l'expression de ma plus affectueuse gratitude. Cette thèse leur est dédiée.

Mes chaleureux remerciements vont à Paty, ma soeur, et à César, mon frère ; j'ai beaucoup apprécié leur soutien constant et leur intérêt dans l'avancement de mon travail. J'adresse une pensée particulière à Oscar, mon frère, qui est décédé. J'aurais tant aimé pouvoir lui montrer ce travail.

Je remercie également ma famille et ma belle-famille pour leur soutien et leur profonde affection.

Mon amour va à Gaby, mon épouse, qui m'a considérablement soutenu tout au long de ce projet, depuis son origine jusqu'à la rédaction du manuscrit final. Je la remercie du fond du cœur pour son soutien, sa compréhension sa patience et son amour, ainsi que pour les nombreuses discussions philosophiques que nous avons eues pendant cette période. Cette thèse lui est dédiée également.

Enfin, je souhaite également faire part de toute ma gratitude au Consejo Nacional de Ciencia y Tecnología (CONACYT) et à l'Universidad Autónoma Metropolitana Iztapalapa pour leur soutien qui m'a permis de mener à bien ce projet.

Merci à tous !

Miguel Ángel Ruiz Sánchez

Présentation des travaux de thèse

Introduction

Dans l'Internet, la communication entre machines hôtes est effectuée en utilisant des paquets d'information. Une fois que les machines hôtes émettent leurs paquets dans le réseau, ce sont les routeurs qui retransmettent ces paquets sur les liaisons des réseaux pour les acheminer vers leur destination finale. C'est à ce processus d'acheminement de paquets dans les routeurs que nous allons nous intéresser dans cette thèse.

Dans l'Internet, chaque paquet est acheminé indépendamment des autres. Ce mode d'opération est connu comme le mode datagramme. Le mode datagramme permet d'offrir un service robuste, car les routeurs peuvent adapter l'acheminement des paquets lors des changements dans la topologie des réseaux. Cependant, le mode datagramme nécessite que les routeurs aient la capacité suffisante pour traiter tous les paquets arrivant à leurs ports d'entrée. Ainsi, avec l'accroissement du trafic, il est nécessaire d'optimiser la performance des routeurs lors de l'acheminement des paquets. Nous proposons dans cette thèse des algorithmes pour optimiser la performance de l'acheminement de paquets dans les routeurs best-effort.

Pour acheminer les paquets, les routeurs doivent accomplir trois tâches principales : Premièrement, les routeurs doivent déterminer où envoyer chaque paquet reçu. Plus spécifiquement, les routeurs doivent déterminer, pour chaque paquet reçu, l'adresse du prochain routeur (ou l'adresse de la destination finale s'il s'agit du dernière relais) et le port de sortie par lequel sera réexpédie le paquet. On appelle l'ensemble de ces deux informations l'information de routage et le fait de déterminer l'information de routage la décision de routage. Pour déterminer l'information de routage, le routeur consulte l'adresse destination du paquet reçu dans une table de routage. Cette opération s'appelle consultation d'adresse (address lookup). Deuxièmement, les routeurs doivent commuter le paquet du port d'entrée au port de sortie approprié. Ensuite, si le lien de sortie est disponible, le paquet sera retransmis sur ce lien ; dans le cas contraire, le routeur doit mémoriser le paquet dans un buffer

en attendant que le lien soit disponible. Ainsi, la troisième tâche à accomplir par le routeur pour acheminer des paquets est de résoudre les possibles contentions pour le lien de sortie. Dans cette thèse, nous nous focalisons sur la première (décision de routage) et la troisième (contention du lien de sortie) tâches. Dans ce qui suit, nous résumons nos travaux avec des renvois sur les sections appropriées pour en connaître les détails.

Mise à jour progressive de tables de routage basées sur des tries multibit

Chaque routeur maintien une table de routage qu'il construit à partir des informations échangées avec d'autres routeurs. Ces échanges d'information sont réalisés par l'intermédiaire de protocoles de routage, tels que RIP, OSPF ou BGP. Cette table de routage contient en général plus d'information que celle strictement nécessaire pour l'acheminement de paquets (e.g., des informations de gestion). Afin de simplifier le processus d'acheminement de paquets, les routeurs maintiennent aussi une autre table ne contenant que l'information absolument nécessaire pour acheminer les paquets. Cette dernière table est appelée la table d'acheminement (en anglais : the forwarding table). Néanmoins, pour ne pas alourdir le texte, nous utiliserons aussi le terme table de routage pour désigner la table d'acheminement, même si elles sont, dans la pratique, différentes.

Pour pouvoir passer à l'échelle, les tables de routage n'ont pas une entrée pour chaque adresse destination, mais une entrée par groupes d'adresses. En particulier, les adresses sont agrégées en utilisant leur préfixe commun. Ainsi, chaque entrée de la table de routage contient l'information de routage correspondant à un groupe d'adresses destination représenté par le préfixe commun de ces adresses. Historiquement, il y avait trois tailles fixes de préfixes (i.e., 8, 16 et 24 bits). Chaque taille fixe de préfixe déterminait une classe différente d'adresses. Mais à l'heure actuelle, la taille des préfixes peut être de 0 à 32 bits dans ce que l'on appelle le routage interdomaines sans classes (CIDR).

Une des opérations essentielles pour acheminer les paquets arrivant à un routeur est la décision de routage, c'est à dire la recherche d'information de routage dans la table de routage. Depuis l'introduction de CIDR (Classless Interdomain Routing), cette recherche d'information de routage consiste à trouver l'entrée de la table de routage ayant le plus long préfixe commun avec l'adresse destination du paquet à acheminer. En effet, plusieurs entrées de la table de routage peuvent avoir un préfixe commun avec l'adresse destination, mais l'entrée avec le plus long préfixe aura l'information de routage la plus spécifique

et donc c'est cette entrée qui doit être utilisée pour effectuer l'acheminement du paquet. Afin de simplifier la lecture, nous utiliserons parfois l'acronyme BMP (Best Matching Prefix) pour designer ce plus long préfixe. Compte tenu que le but principal d'un routeur est d'acheminer des paquets, un facteur capital dans la performance d'un routeur est la vitesse avec laquelle le routeur trouve l'information de routage. Une façon d'optimiser le temps pour trouver le plus long préfixe, et donc de trouver l'information de routage, est d'utiliser une structure de données appelée trie multibit. L'idée de cette approche est de transformer l'ensemble original de préfixes de la table de routage en une autre ensemble équivalent ; équivalent dans le sens où l'on obtient toujours la même information de routage lors des opérations de recherche. La particularité de cet ensemble équivalent est que le nombre de longueur différentes des préfixes est inférieur à celui de l'ensemble original. Avoir moins de longueurs différentes permet de réduire le nombre d'accès à la mémoire et donc de diminuer le temps de recherche. En revanche, en réalisant cette transformation le nombre de préfixes de l'ensemble équivalent est généralement plus grand que celui de l'ensemble original. Plusieurs méthodes basées sur les tries multibit ont été proposées récemment [PZ92], [SV98], [GLM98], [DBCP97], [MS98], [HZ99], [NK99]. Néanmoins, la plus part de ces méthodes ne prennent pas en compte l'aspect de la mise à jour progressive de la table de routage. Or, cet aspect est essentiel car la robustesse du système de routage dépend de la capacité des routeur à s'adapter aux changements dans la topologie du réseau. D'ailleurs, des chercheurs ont constaté que les routeurs de cœur (backbone routers) reçoivent fréquemment des messages de mise à jour [Lab99]. Dans cette thèse nous proposons deux mécanismes permettant la mise à jour progressive dans des tables de routage basées sur des tries multibit. Mais avant de présenter nos mécanismes, nous décrivons brièvement les tries multibit. Les tries multibit sont présenté avec beaucoup plus de détails dans le chapitre 3.

Les routeurs agrègent l'information de routage en utilisant des préfixes. Ainsi, dans une table de routage, chaque entrée contient un préfixe et son information de routage correspondante. Puisqu'un préfixe est une chaîne de bits de longueur variable, ils peuvent être représentés tout naturellement par un trie. Un trie est une structure de données en arbre qui organise ses données en tirant profit du caractère décomposable de ses données. Dans le cas spécifique des préfixes, ce sont les bits des préfixes qui sont utilisés pour déterminer les branches du trie. Par exemple, la figure 3.1 montre un trie binaire (chaque noeud a un maximum de deux fils) qui représente un ensemble de préfixes d'une table de routage. Les préfixes eux mêmes sont représentés par certains noeuds du trie : Chaque feuille du trie représente un préfixe ; mais les noeuds internes peuvent ils aussi représenter des préfixes.

Pour une adresse destination donnée, la recherche du plus long préfixe dans un trie

consiste essentiellement à parcourir le trie à partir de sa racine. On parcourt le trie en utilisant les bits de l'adresse destination pour emprunter les branches correspondantes. Ainsi, à chaque noeud la recherche se poursuivra à droite ou à gauche en fonction de la valeur du bit correspondant. La recherche termine lorsqu'il n'y a plus de chemin à suivre, et le dernier préfixe visité sera le plus long préfixe correspondant à l'adresse destination donnée. La mise à jour d'un trie binaire est relativement facile. Néanmoins, le principal problème avec les tries binaires est que le nombre d'accès à la mémoire lors des recherches est grande. En effet, lors d'une recherche, chaque fois que l'on teste un bit pour décider quelle branche emprunter dans le trie, un accès à la mémoire est nécessaire. C'est à dire que dans le pire des cas, une recherche a besoin de 32 accès à la mémoire dans IPv4. Comme ces accès à la mémoire sont lents, la recherche dans un trie binaire n'est pas appropriée pour des routeurs de haute performance. Une façon de réduire le nombre d'accès à la mémoire nécessaires pour une recherche est l'utilisation de tries multibit. Dans un trie multibit, on ne parcours pas le trie en testant un bit de l'adresse destination à la fois, mais plusieurs bits à la fois. Un exemple de trie multibit est montré dans la figure 3.3. Puisque le parcours dans un trie multibit est effectué par des pas de plusieurs bits, le trie multibit ne peut pas accepter des préfixes de longueur arbitraire. En effet, un trie multibit donné n'accepte que les préfixes de longueur déterminée par la taille des pas du trie multibit. Il est possible cependant d'utiliser un trie multibit pour représenter une table de routage quelconque. Pour ce faire, l'ensemble de préfixes de la table de routage doit être transformé en un autre ensemble de préfixes dont les longueurs soient acceptées par le trie multibit, tout en conservant la même information de routage. Cette transformation est réalisée par une technique appelée expansion de préfixe.

Bien que l'utilisation d'un trie multibit permette de réduire le nombre d'accès à la mémoire lors d'une recherche et donc d'améliorer la performance de la recherche d'information de routage, le fait de transformer l'ensemble original de préfixes rend plus difficile les opérations de mise à jour. Bref, la recherche dans un trie multibit est plus rapide, mais la mise à jour est beaucoup plus compliquée, par rapport au trie binaire. Dans la section 3.4 nous analysons la problématique liée à la mise à jour progressive des tables de routage basées sur le multibit trie. Nous concluons qu'il est nécessaire, entre autres, une structure de données additionnelle pour permettre la mise à jour progressive du trie multibit. Nous y introduisons deux notions qui sont utilisées dans la conception de notre structure de données additionnelle et de nos algorithmes pour la mise à jour progressive. Ces notions sont l'éventail (span) d'un préfixe et le préfixe remplaçant (the coverer). Puis nous proposons dans les sections 3.5 et 3.6 deux mécanismes pour la mise à jour de tables de routage basées sur un trie multibit.

Due à la transformation de préfixes, le trie multibit ne mémorise pas les préfixes origi-

naux de la table de routage, mais les préfixes dérivés. Or les opérations de mise à jour de la table de routage doivent agir directement sur les préfixes originaux.

La recherche du plus long préfixe (BMP) dans un trie multibit est effectuée par approximation successives. Ainsi, l'opération de recherche consiste essentiellement à parcourir à chaque pas un subtrie du niveau suivant. Dans chaque subtrie, on obtient le plus long préfixe (BMP) local commun avec l'adresse destination. À la fin du parcours dans le trie multibit, le dernier BMP local obtenu sera le résultat de la recherche.

Chaque subtrie de degré 2^k est représenté par un tableau à 2^k entrées ; k étant le nombre de bits à tester dans chaque subtrie. Pour que l'opération de recherche marche correctement, il est nécessaire que chaque entrée du tableau soit associée à son BMP local. Le BMP local d'une entrée peut être vide.

Lorsqu'un préfixe est inséré ou supprimé un certain nombre d'entrées d'un tableau doivent être mises à jour ; c'est à dire, on doit mettre à jour leur BMP local. Le nombre d'entrées à mettre à jour est déterminé par l'éventail (span) du préfixe à insérer ou supprimer. Potentiellement, toutes les entrées dans l'éventail du préfixe peuvent être modifiées, mais seulement celles qui n'appartiennent pas à des éventails plus spécifiques seront modifiées. Rappelons que, en général, une même entrée d'un tableau peut être incluse dans des éventails de plusieurs préfixes. La modification des entrées dans un éventail consiste à changer leur BMP local avec un nouveau BMP local. S'il s'agit d'insérer un nouveau préfixe P, alors le nouveau BMP local sera P. S'il s'agit de supprimer un préfixe P alors il faut trouver le nouveau BMP local Q parmi les autres préfixes du subtrie dont il s'agit. En fait, Q est le préfixe dont l'éventail est le plus petit éventail incluant l'éventail de P.

Pour effectuer la mise à jour d'une table de routage basée sur un trie multibit, il est nécessaire de mémoriser les préfixes originaux. Dans notre approche les préfixes originaux sont mémorisés dans des listes chaînées associées aux entrées du tableau. Ces listes chaînées contiennent aussi avec chaque préfixe original l'information de routage correspondante. Les préfixes originaux mémorisés dans les listes chaînées sont utilisés dans la mise à jour pour deux objectifs. Le premier objectif est de décider quels sont les entrées à modifier dans l'éventail d'un préfixe donnée P. Les entrées à ne pas modifier seront déterminées par les préfixes contenus dans l'éventail du préfixe P qui sera inséré ou supprimé. Le deuxième objectif est de trouver le préfixe remplaçant Q lors de la suppression d'un préfixe.

Pour accélérer l'accès aux préfixes, nous proposons deux méthodes. La première méthode, proposée dans la section 3.5, utilise un vecteur de bits par entrée dans chaque tableau. Ainsi, chaque vecteur de bits d'une entrée est associé aux préfixes mémorisés dans la liste chaînée de la même entrée. En d'autres termes, chaque préfixe dans une liste chaînée est

identifié par un bit dans le vecteur de bits correspondant. Les vecteurs de bits permettent d'identifier rapidement la présence ou absence d'un préfixe en testant le bit correspondant. Notre deuxième méthode, proposée dans la section 3.6, n'utilise pas un vecteur de bits par entrée mais un vecteur de bits par tableau. Nous appelons ce vecteur de bits par tableau le vecteur de bits PN. Le vecteur de bits PN est un vecteur de bits qui résulte de la compression de tous les vecteurs de bits du tableau de la première méthode. L'utilisation du vecteur de bits PN optimise l'occupation mémoire ; en revanche, elle requiert le calcul de fonctions pour décoder la position des préfixes dans le vecteur de bits PN. Mais la principal avantage du vecteur de bits PN est qu'il séparé de la structure de données principale, c'est à dire celle qui est utilisée pour la recherche du BMP. Cette séparation implique que la recherche du BMP ne sera pas perturbée par le mécanisme de mise à jour. Nous avons mesuré le temps de recherche du BMP pour nos deux mécanismes et nous l'avons comparé avec celui d'un trie multibit de base, c'est à dire un trie multibit sans la capacité d'effectuer des mises à jour. Nos mesures montrent que nos mécanismes de mise à jour ont un impact quasiment nul sur les performances de l'opération de recherche du BMP.

En conclusion, nous avons proposé deux mécanismes pour effectuer la mise à jour progressive de tries multibit représentant une table de routage. La mise à jour requiert que les préfixes originaux soient mémorisés car le trie multibit ne le fait pas en réalité. Nous utilisons des listes chaînées pour stocker les préfixes originaux. Potentiellement, une liste chaînée est associée à chaque entrée d'un tableau (subtrie). Pour optimiser les opérations de mise à jour, il faut accélérer l'accès et le parcours de ces listes chaînées. Pour ce faire, notre premier mécanisme utilise un vecteur de bits dans chaque entrée de chaque subtrie (tableau). Notre second mécanisme est plus efficace dans l'utilisation de la mémoire car il utilise un seul vecteur de bits par subtrie. En revanche, notre second mécanisme doit calculer des fonctions pour décoder la position des préfixes dans le vecteur de bits PN.

Un cadre de référence et taxonomie pour des algorithmes de consultation d'adresse IP

À part les tries multibit, d'autres approches ont été proposées pour optimiser la recherche de l'information de routage dans les routeurs, c'est à dire pour optimiser la consultation d'adresse IP dans les tables de routage. Nous proposons dans cette thèse une taxonomie de ces méthodes. Notre taxonomie est basée sur l'observation que la difficulté de trouver le plus long préfixe commun avec l'adresse destination est sa double dimension : valeur et longueur.

Ainsi, pour déterminer le BMP, il est nécessaire aussi bien de trouver une correspondance au niveau de la valeur de la séquence binaire, que de trouver la longueur appropriée. Notre taxonomie classifie les algorithmes de recherche en fonction de la dimension principale à chercher et si cette recherche est linéaire ou binaire. Ainsi, nous considérons les quatre cas principaux suivants : 1) recherche linéaire basée sur la dimension de la longueur ; 2) recherche binaire basée sur la dimension de la longueur ; 3) recherche linéaire basée sur la dimension de la valeur ; 4) recherche binaire basée sur la dimension de la valeur.

Lorsque nous présentons et classifions les différents mécanismes, l'accent est mis sur le type de transformation que l'on effectue sur l'ensemble de préfixes pour chaque mécanisme. Cette approche unificatrice que nous proposons nous permet de comprendre et de comparer les compromis des différentes mécanismes. La transformation de l'ensemble original de préfixes consiste, en général, à désagréger de façon contrôlée l'information de routage. Cette transformation vise à optimiser le temps de recherche. Cependant cette désagrégation représente un compromis entre d'une part le temps de recherche de l'information de routage et d'autre part la place mémoire nécessaire et le temps de mise à jour de l'information de routage. Dans ce qui suit nous analysons brièvement les différents cas de notre taxonomie.

Dans le premier et deuxième cas la recherche est basée sur la dimension de la longueur. Les préfixes peuvent être organisés soit en utilisant une table pour chaque longueur différente ; soit en utilisant des tries. Dans le cas de l'utilisation de tables pour chaque longueur, l'approche la plus facile est de faire une recherche linéaire. Ainsi, les tables sont cherchées par ordre descendant de longueur. Dans une table, on cherche si une des entrées a un préfixe commun avec l'adresse destination. Cette opération peut être effectuée par une fonction de hachage. Si cette entrée existe, alors le BMP se trouve à cette entrée, ainsi que l'information de routage correspondante. Si ce n'est pas le cas, alors on continue la recherche dans les autres tables. Dans le pire des cas, toutes les tables devront être cherchées, et donc la complexité du temps de recherche est $O(W)$, W étant la longueur maximale des préfixes. Ceci supposant une fonction de hachage parfaite.

Évidemment, une recherche binaire est préférable mais une recherche binaire sur la longueur des préfixes ne peut être effectuée qu'à condition de transformer l'ensemble de préfixes de la table de routage. Dans une recherche binaire, on réduit l'espace de recherche de moitié à chaque fois. Si la recherche est basée sur la longueur des préfixes, alors on voudrais décider à chaque fois dans quelle moitié des tables se trouve le BMP. Ainsi,on cherche tout d'abord la table correspondant à la longueur de milieu et si l'on trouve une entrée étant préfixe de l'adresse destination, c'est à dire si on obtient un succès partiel, alors on continue la recherche uniquement dans les tables correspondant aux longueurs plus grandes ; et dans

le cas contraire, on continue la recherche dans les tables correspondant aux longueurs plus petites. Mais cette approche ne marche pas. Il est vrai que si l'on obtient un succès partiel, il faut chercher dans la moitié de longueurs plus grandes ; mais s'il n'y a pas de succès partiel, alors le BMP peut se trouver dans n'importe quelle moitié. Pour que la décision basée sur le succès partiel marche, il est nécessaire de transformer l'ensemble de préfixes. Cette transformation consiste à ajouter des préfixes qui serviront à guider la recherche dans le bon sens. Dans la section 4.2.3 nous décrivons cette transformation.

Dans le cas de tries, l'optimisation de la recherche est réalisée en utilisant des tries multibit. Nous avons explique l'utilisation de tries multibit dans le chapitre 3, où nous avons aussi proposé deux mécanismes de mise à jour progressive. Les tries multibit peuvent être utilisés en combinaison avec d'autres techniques. Par exemple, dan la section 4.2.2.2 nous expliquons comment les tries multibit peuvent être utilisés avec des techniques de compression afin de réduire la quantité de mémoire nécessaire.

Dans le troisième et quatrième cas la recherche est basée sur la dimension de la valeur. L'idée de cette approche est de trouver un moyen pour se débarrasser de la dimension de la longueur des préfixes. Pour ce faire, les préfixes doivent être transformés de sa représentation valeur/longueur en une représentation comportant deux valeurs. Ces deux valeurs représentent les bornes de l'intervalle d'adresses défini par le préfixe. Comme ces valeurs n'ont pas une dimension de longueur, il est possible d'utiliser les méthodes classiques de recherche basées sur la comparaison de valeurs. Ainsi, le problème de trouver le BMP revient à déterminer, pour une adresse donnée, un des bornes appropriés ; car ces bornes auront l'information de routage associée au préfixe. En principe, n'importe quel de deux bornes peut être cherché pour une adresse donnée. Par exemple, si l'on choisit la borne supérieur, alors trouver cette borne revient à chercher le successeur de l'adresse en question. Tandis que si l'on choisit la borne inférieure, alors on cherchera le prédécesseur de l'adresse en question. Néanmoins, cette approche ne marche pas si les intervalles contiennent d'autres intervalles ; ce qui est le cas avec CIDR. Le problème est que si des intervalles contiennent d'autres intervalles, chercher l'information de routage appropriée requiert chercher tantôt le prédécesseur tantôt le successeur, en fonction de l'ensemble spécifique de préfixes et de l'adresse en question. Pour que cette approche marche en cherchant soit le prédécesseur ou soit le successeur mais un seul et bien déterminé des deux, il faut transformer les intervalles des préfixes originaux en intervalles disjoints. Avec des intervalles disjoints, le BMP d'une adresse destination est obtenu en cherchant soit son prédécesseur soit son successeur. Le prédécesseur ou le successeur peut être trouvé en utilisant des méthodes de recherche traditionnelles. Dans la section 4.3 nous montrons en détail comment ces méthodes de recherche

traditionnelles sont utilisées pour trouver le BMP. Nous signalons aussi la difficulté de la mise à jour due à la transformation des préfixes en intervalles disjoints. En effet, un seul préfixe peut être transformé en $O(N)$ intervalles disjoints. Nous montrons aussi comment la mise à jour peut être effectuée avec cette approche.

Après avoir analysé les différentes approches pour trouver le BMP, dans la section 4.5.1 nous les comparons au niveau des complexités en temps, en espace et en temps de mise à jour. Puis dans la section 4.5.2 nous comparons les performances de différents approches en mesurant le temps de recherche. Les performances ont été mesurées en utilisant l'information de routage d'une vrai table de routage d'un routeur de cœur de réseau. Nous avons trouvé que l'approche avec les meilleurs performances pour le temps de recherche est l'approche qui utilise un trie multibit compressé. Cependant, cette approche ne permet pas la mise à jour progressive. Dans l'autre extrême, l'approche du trie binaire est la moins performante. Les approches basées sur des tries multibit non compressés offrent de bonnes performances tout en gardant la possibilité d'effectuer de mises à jours progressives.

Optimisation de l'usage des buffers dans les routeurs.

Bien que déterminer l'information de routage soit une tâche essentielle pour acheminer un paquet, pour qu'un routeur accomplisse l'acheminement d'un paquet, il faut en plus le commuter du port d'entrée au port de sortie et puis le transmettre sur le lien de sortie. Or, il se peut que lorsqu'un paquet doit être transmis, le lien de sortie soit déjà occupé. En général, plusieurs paquets provenant d'entrées différentes peuvent simultanément vouloir aller au même lien de sortie. Le routeur doit donc résoudre la contention du lien de sortie pour pouvoir acheminer les paquets. L'utilisation d'un buffer aide à résoudre la contention du lien de sortie en évitant que des paquets soient jetés lorsque le lien de sortie est occupé. Ainsi, l'utilisation d'un buffer est essentielle pour que le trafic des différentes liens d'entrée puissent être multiplexé sur le lien de sortie. Un buffer a donc une fonction de multiplexage. En général, cette fonction de multiplexage d'un buffer permet que les différents flux puissent partager le même lien de sortie. Mais cette fonction de multiplexage peut être perturbée par le trafic de certains utilisateurs, car les buffers sont aussi utilisées pour absorber les rafales des flux individuels. Il est nécessaire donc pour protéger la fonction de multiplexage d'un buffer de contrôler le trafic. Traditionnellement, le contrôle du trafic dans l'Internet est effectué par les sources. En particulier, les sources utilisent TCP pour adapter leur trafic en fonction de la charge du réseau. Néanmoins, en général, les utilisateurs ont le choix d'être ou ne pas être coopératifs. Ainsi, en général, les sources coopératives sont pénalisées de

manière intentionnelle ou non par les sources qui ne contrôle pas leur trafic. Pour résoudre ce problème, c'est à dire pour isoler les flux, il est nécessaire que les routeurs protègent la fonction de multiplexage des buffers. Dans cette thèse nous proposons un mécanisme appelé MuxQ qui optimise l'usage du buffer de façon à protéger sa fonction de multiplexage, et fournir ainsi un haut dégrée d'isolement entre les flux qui partagent le même lien de sortie.

Dans la section 5.1 nous analysons les fonctions des buffers dans les routeurs. En particulier, nous concluons que pour protéger la fonction de multiplexage des buffers deux conditions sont nécessaires. La première est d'avoir espace libre dans le buffer pour absorber des surcharges passagères, soit sous la forme de flux nouveaux soit sous la forme de rafales passagères des flux individuels. La deuxième condition concerne le choix de paquets à jeter. Ce choix doit être effectué en fonction du niveau d'occupation de chaque flux dans le buffer.

Dans la section 5.3 nous proposons MuxQ un mécanisme qui fournit un haut degré d'isolation entre flux. MuxQ est basé sur l'idée de protéger la fonction de multiplexage de la fonction d'absorption de rafales d'un buffer. Pour remplir la première condition (c'est à dire avoir de la place libre dans le buffer pour les surcharges passagères) MuxQ contrôle la longueur de la queue. Quand à la deuxième condition, MuxQ prends la décision d'accepter ou jeter un paquet en fonction de l'information d'état d'un nombre limité de flux : les flux qui ont des paquets dans le buffer à ce moment-là. L'algorithme de MuxQ consiste à décider pour chaque paquet s'il sera accepté ou pas dans le buffer. Cette décision est basée sur l'information du nombre de paquets que le flux a déjà dans le buffer ainsi que le nombre de flux actifs, c'est à dire le flux qui ont des paquets dans le buffer à ce moment-là. Notons que l'information d'état à garder est limitée ce qui permet à notre mécanisme de passer à l'échelle. Par ailleurs, de façon à ce qu'un paquet appartenant à un flux nouveau puisse être accepté la taille de la queue est contrôlée par un paramètre que nous dénotons par *ltqlen* (pour long-term queue length). La valeur de ce paramètre détermine la longueur de la queue à long terme lorsque le nombre de flux actifs reste constant et le trafic de ces flux surchargent le lien de sortie. Pour qu'un paquet appartenant à un flux nouveau soit accepté ce paramètre doit être inférieur à la taille du buffer.

Lorsqu'un paquet ne peut pas être transmis sur le lien de sortie et doit donc être stocké dans le buffer, le routeur vérifie s'il s'agit d'un flux actif et dans ce cas combien de paquets il a déjà dans le buffer. Cette vérification est effectuée en utilisant une opération de hachage sur une table qui garde l'état des flux actifs. S'il s'agit d'un paquet d'un nouveau flux, le paquet est accepté. S'il s'agit d'un flux actif le routeur vérifie si le nombre de ses paquets dans le buffer est inférieur à $maxpkts = \frac{ltqlen}{n}$, où n est le nombre de flux actifs. S'il est

vrai le paquet est accepté, dans le cas contraire le paquet est jeté. Notons que $maxpkts$ est le nombre maximum de paquets qu'un flux peut avoir dans le buffer en même temps. Cette valeur est dynamique car elle change en fonction du nombre de flux actifs. Notons aussi que la longueur de la queue peut être plus grande que $ltqlen$. Ceci arrive lorsque la longueur de la queue est déjà $ltqlen$ est des paquets de nouveaux flux arrivent. Notons également que puisque $maxpkts$ varie en fonction du nombre de flux actifs, un flux peut bien avoir plus de paquets que la valeur $maxpkts$. Cependant, cette situation sera passagère car les paquets suivants de ce flux ne seront pas acceptés (tant que le nombre de ses paquets reste supérieur ou égal à $maxpkts$, bien sûr). La section 5.3.3 présente de façon détaillée les algorithmes de notre mécanisme MuxQ.

Dans la section 5.4 nous évaluons notre mécanisme en utilisant le simulateur ns-2. En particulier, nous étudions la capacité de MuxQ pour isoler différent types de flux. Nous comparons les performances de notre mécanisme avec celles des mécanismes Drop-Tail, CSFQ, FRED et DRR. Nous présentons les résultats de simulations avec des conditions de trafic différentes. Nous avons simulé les scénarios suivants : uniquement des flux coopératifs (TCP) ; des flux coopératifs avec des flux non coopératifs ; des flux de courte durée type web avec des flux de long durée (coopératifs et non coopératifs) ; uniquement des flux non coopératifs mais à différents débits ; Les résultats de ces simulations montrent que MuxQ fournit un haut degré d'isolation dans tous ces cas. Bien que DRR et CSFQ offrent aussi un haut degré d'isolation de flux, MuxQ a les avantages suivantes : MuxQ n'a pas besoin de maintenir une queue par flux, comme c'est le cas pour DRR. MuxQ ne maintien qu'une quantité limitée d'état, l'information d'état concernant les flux qui ont des paquets dans le buffer. Une autre avantage de MuxQ est que à la différence de CSFQ, MuxQ n'a pas besoin de modifications dans l'entête du paquet IP. En outre, MuxQ n'attend pas une coopération spécifique vis à vis des autres routeurs, comme c'est le cas de CSFQ. En effet CSFQ a besoin que les routeurs de bordure mesure le débit des flux et que cet information soit enregistrée dans l'entête des paquets IP. Ce qui veut dire qu'à la différence de CSFQ, MuxQ peut être déployé de façon incrémentale. MuxQ est donc un mécanisme simple, deployable et qui fournit un haut degré d'isolation de flux, tout en gardant une quantité limitée d'état.

Conclusion

Une caractéristique clé de l'Internet est sa capacité d'adaptation. L'Internet a été capable de s'adapter au nombre croissant d'utilisateurs et de volume de trafic. Ce type d'adaptation est souvent connue comme la capacité de passage à l'échelle. L'Internet a été aussi capable

de s'adapter aux changements de topologie et de charge. Cette adaptation consiste à dégrader graduellement le service offert par le réseau et elle détermine la robustesse de l'Internet. Cette thèse a été motivée par l'idée de maintenir la robustesse et la capacité de passage à l'échelle de l'Internet.

Dans le paradigme de datagramme de l"internet, les routeurs jouent un rôle très important, car c'est eux qui doivent traiter chaque paquet. Comme chaque paquet doit être traité de façon indépendante, la performance de l'Internet dépend directement de la capacité de traitement des routeurs. Dans cette thèse, nous avons proposé des algorithmes pour améliorer l'acheminement de paquets dans les routeurs best-effort. Plus spécifiquement, des mécanismes permettant la mise à jour progressive de tables de routage basées sur les tries multibit, tout en optimisant l'opération de recherche d'information de routage. Nous avons également proposé un cadre de référence et une taxonomie des méthodes existants pour optimiser la recherche d'information de routage, quand elle nécessite la recherche du plus long préfixe commun avec l'adresse destination du paquet. Finalement nous avons proposé MuxQ, un mécanisme pour fournir un haut degré d'isolation de flux sans avoir besoin que les sources soient coopératives.

Bien que les algorithmes que nous avons proposés dans cette thèse contribuent à améliorer la performance de l'acheminement de paquets dans les routeurs, le trafic de l'Internet ne cesse pas de croître. Nos travaux futurs visent à continuer à améliorer d'avantage la performance de l'acheminement de paquets dans les routeurs best-effort. Dans le cas de l'optimisation de la recherche d'information de routage, nous envisageons de nouvelles méthodes qui puissent effectuer l'opération de recherche avec un seul accès à la mémoire. Nous considérons deux approches possibles. La première consiste à paralléliser le traitement lors de l'utilisation de tries multibit. Ici les problèmes à résoudre sont notamment la distribution de la mémoire dans les différentes étapes parallèles. Un autre problème est comment supporter la mise à jour dans ce type d'approche sans dégrader l'opération de recherche. La seconde approche consiste à utiliser des TCAMs. Les TCAMs permettent de chercher l'information de routage avec un seul accès à la mémoire, mais les TCAMs nécessitent que les préfixes soient ordonnés par longueur. Dans ce cas, la difficulté est de trouver de méthodes efficaces pour maintenir les préfixes ordonnés tout en permettant de mises à jour progressives de la table de routage.

En ce qui concerne l'isolation de flux, nous avons supposé que les buffers sont placés dans les port de sortie des routeurs. Cependant les routeurs peuvent avoir aussi des buffers juste avant et après le commutateur interne du routeur. Nous envisageons des mécanismes pour l'isolation de flux qui exploitent cette caractéristique.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Internet allows host computers to communicate by using packets of bits. To this end, packets are transfered by routers which are interconnected by links. In the Internet, routers treat each packet independently of the others. This mode of operation is known as the datagram paradigm. The datagram paradigm allows the Internet to provide a robust service because routers can make different forwarding decisions for each packet according to changes in the topology of the network. While the datagram paradigm provides a robust service, to keep pace with increasing traffic and wide spectrum of traffic requirements, the packet forwarding capacity of routers need to be optimized. This thesis proposes several algorithms to optimize the performance of the packet forwarding process in best effort routers.

To forward packets, routers must complete three critical tasks: First, routers must make a forwarding decision; that is, find out the next hop to which the packet has to be sent as well as the output port through which the packet should be sent. The operation of determining the forwarding information is based on the packet's destination address and it is called address lookup. Second, routers must transfer packets from the input port to the appropriate output port, in an operation called switching. After switching, the router can transmit the packet on the outgoing link. However, due to the statistical nature of packet multiplexing, it is possible that packets from different inputs need to be forwarded through the same output link. Hence, a third critical task in the packet forwarding process is to resolve possible contentions for packets that need to be forwarded through the same output link. We concentrate in this thesis on the first (address lookup) and third (output link contention) critical tasks of the packet forwarding process. In the next section we summarize the main contributions of this thesis.

1.1 Contributions

Incremental updates for multibit-tries based forwarding databases

Routers make their forwarding decisions based on routing information gathered by routing protocols. Routers maintain simplified routing information in a forwarding database, also called a forwarding table. To cope with scalability issues, the forwarding database does not contain an entry for each possible destination address; instead, the destination addresses are grouped; and each group is represented in the forwarding database by an address prefix.

One of the main tasks of the packet forwarding process in IP routers is the address lookup operation. With CIDR, the address lookup operation consists in finding the best (longest) prefix that matches the destination address of the incoming packet. Since the performance of routers depends on its forwarding capacity, high performance routers must provide fast address lookups. One way to provide fast address lookups is the use of a data structure called multibit trie. The idea is to transform the set of original prefixes of a forwarding database into a different set of prefixes with less different lengths but with the same forwarding information. While several schemes has been proposed to provide fast address lookups with multibit tries, most of these schemes diminishes the issue of providing incremental updates. Since one of the key aspects of the Internet is its robustness in the form of adaptation to topological changes, providing incremental updates is a requirement. In fact, it is observed that routing information does change frequently in backbone routers. We propose in this thesis two incremental update mechanisms for address lookup schemes based on the multibit-trie data structure. First, we determine the requirements to support incremental updates in multibit-tries based forwarding databases. Then we propose algorithms and data structures to support incremental updates. In particular, we propose a data structure called Prefix Nesting bit vector, or PN bit vector for short. The PN bit vector encodes a set of prefixes and their nesting structure, for this information is necessary to support incremental updates. The PN bit vector is used as an additional data structure to the main multibit trie data structure. The use of the PN bit vector does not affect the Best-Matching-Prefix lookup operation because our scheme effectively separates the PN bit vector from the main data structure used for doing efficient lookup operations. We present performance results of a C-language implementation of our scheme. Performance results are shown in terms of time for the search, insert and delete operations. Memory requirements are also shown.

Part of this work is based on our previous publication [RSD00].

A Framework and a Taxonomy of IP Address Lookup Algorithms

Multibit tries are not the only way to provide fast address lookups. Another contribution of this thesis is the introduction of a taxonomy and a framework of reference of existing fast address lookup schemes. Our taxonomy is based on the observation that the difficulty of the best prefix matching problem resides in its double dimension: value and length. As a result, determining the best matching prefix involves not only comparing the bit pattern itself (i.e., finding a match), but also finding the appropriate length (i.e., the longest one). Our taxonomy classifies the address lookup schemes according to these two dimensions and also if a linear or a binary search is performed. We analyze the next four cases: 1) Linear search based on values; 2) Binary search based on values; 3) Linear search based on lengths; 4) Binary search based on lengths.

Routers aggregate forwarding information by the use of prefixes. In our analysis we emphasize that to improve the performance of the address lookup operation, the different methods make a transformation of the original set of prefixes of the forwarding database. We state the different tradeoffs of the different transformation methods in terms of time and space and we compare the performance of the different schemes. While the most important aspect is the search operation, we also analyze the potential capabilities of the schemes to support incremental updates. We state that to support incremental updates, a mechanism must have additional data structures to keep track of the prefix transformation process.

Part of this work is based on our previous publication [RSBD01].

Optimizing the use of buffers for flow isolation

While performing an address lookup operation allows routers to decide where to send a packet next, this task is only a part of the process to achieve the actual relaying of a packet. Once the router has decided through which output port the packet will be forwarded, the router must multiplex all the packets that need to be forwarded through the same output port. Furthermore, routers must resolve possible contention for a given output port because it is possible that several packets from different inputs need to be forwarded through the same output port at the same time. Usually, routers use buffers to address the problem of output-port contention. Buffering allows routers to retain packets while one of the contending packets is transmitted. However, buffering alleviates the output-port contention only to some extent because, in case of sustained overload, the buffer will eventually overflow and packets will be dropped. Hence, buffers can help to resolve contention only in the case of transient overload. Unfortunately, in the Internet sustained overload is possible and to make

effective use of buffers one needs mechanisms to control the traffic. The control of traffic can be made in two places: at end systems or/and at routers. Traditionally, the traffic control in the Internet has been done by end systems. The sources use algorithms to try to discover the available resources in the network and so adapt their traffic pattern in a dynamic manner. In particular, if congestion occurs the sources should respond by reducing their traffic. The classical algorithm to control traffic of end systems is the TCP congestion control protocol. Nevertheless, in today's Internet responding to congestion is rather a user's choice and in general there are responsive as well as unresponsive users. As a result, buffers in routers are not always used effectively. When the buffers are not used effectively, the network service is degraded in the form of packet losses and/or packet delay. Furthermore, adaptive sources are penalized because unresponsive sources, intentionally or unintentionally, abuse the cooperative nature of responsive traffic. To address this problem, routers need to provide flow isolation. Providing flow isolation is important because with flow isolation the performance perceived by users does not depend on the good behavior of other users. This thesis proposes also a mechanism to optimize the use of the buffer in routers to provide flow isolation. First, we study the buffering functionality of IP routers. We find the desired properties of a router buffer system then we design a mechanism based on these characteristics. We emphasize that buffers in routers have two functions: a multiplexing function and a burst absorbing function. Our mechanism, which we call MuxQ, is based on the idea of protecting the multiplexing function from the burst absorbing function by progressively and dynamically controlling the allocation of buffer space in a FIFO queue. MuxQ is a new queue management mechanism that provides flow isolation by using a very simple algorithm and without using per-flow queuing.

We compare the performance of the MuxQ scheme to that of classical Drop-Tail and to that of other proposed schemes, including CSFQ and DRR which provides nearly perfect isolation by using per-flow queuing. By keeping only limited flow-state, our mechanism performs very much better than Drop-Tail. MuxQ achieves performance similar to that of CSFQ but MuxQ does not need modifications to the IP packet header as it is the case for CSFQ.

One of the important characteristic of a new router mechanism is its incremental deployability. MuxQ does not need modifications of the IP packet header. Moreover, since MuxQ does not expect a special behavior from other routers, MuxQ routers can interact without problem with classical Drop-Tail routers and thus MuxQ can be deployed incrementally. We believe that MuxQ is an interesting approach to achieve a high degree of flow isolation with respect to Drop-Tail by using a very simple algorithm.

Part of this work is based on our previous publication [RSD03].

1.2 Thesis Overview

The rest of this thesis is organized as follows: Chapter 2 gives the relevant background for the per-packet processing in best effort routers, along with short reviews of work in the related areas. Chapter 3 presents our first contribution: a set of data structures and algorithms to provide incremental updates for BMP lookup schemes based on multibit tries. Chapter 4 presents our second contribution: a taxonomy and a reference framework to analyze and compare fast BMP lookup algorithms. Chapter 5 presents our third contribution: a scheme to optimize the use of buffers in best effort routers towards providing flow isolation. Finally chapter 6 gives conclusions and discusses directions for future work.

Chapter 2

Background and Problem Definition

The networks that make up the Internet are composed of end systems, communication links and routers. End systems communicate by using IP, the Internet Protocol [Pos81]. IP provides a basic communication service based on the datagram paradigm. With the datagram paradigm, the information to be transmitted is partitioned into independent packets with a header containing routing directive information. These packets are transmitted through the communication links with the help of routers which forward packets towards their final destination. Each packet is independent because in the datagram paradigm the routers do not keep state of the on-going connections. The end systems use the Internet on a demand basis; that is, the network resources are not reserved; instead, the Internet allows end systems to share the network resources by using packet switching.

The datagram paradigm allows end systems ready access to the network because users can send data at any time without the need to request permission before transmitting; end systems simply send packets whenever they have data to send; and routers multiplex packets in an on-demand order. In other words, routers employs statistical multiplexing. Moreover, statistical multiplexing is particularly adequate to the bursty characteristics of the data traffic. The datagram paradigm provides also robustness because each packet is independent and routers can adapt to changes in the topology of the network. While datagram packet switching provides robustness and leads to a more efficient use of network resources, it requires expensive per-packet processing. As a result the performance of the Internet depends directly on the capacity of routers to process packets. Improving per-packet processing has become even more important because of the increasing traffic load that faces the Internet. We describe in this chapter some background to understand the problems addressed in this thesis. In particular, we will see in the next section the main tasks that a router must perform to process packets.

2.1 The Packet Forwarding Function of Routers

Routers transfer packets between networks and, ultimately from senders to receivers. This transfer of packets, called packet forwarding is the main task of a router. A generic router architecture is shown in figure 2.1. A router consists basically of some input ports, where packets are received; some output ports, through which packets are forwarded; a switching fabric that transfers packets from the input ports to the output ports; and a routing processor that executes the routing protocols (not shown in the figure).

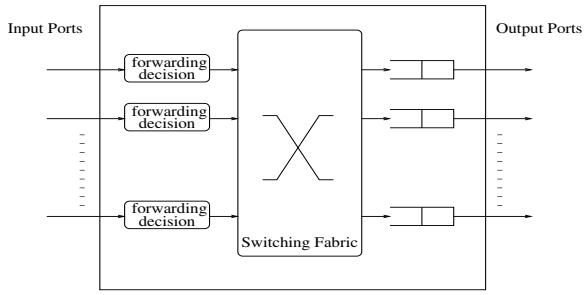

Figure 2.1: A basic IP router

When a router receives a packet in an incoming link, the router should forward the packet to the next router or to the final destination. More precisely, to forward a packet, the router requires to perform three critical tasks[1]:

- Make a forwarding decision; that is find out the next hop to which the packet has to be sent.
- Switch the packet from the input port to the appropriate output port.
- Resolve possible contention for packets that need to be forwarded through the same output link at the same time.

While all three of these tasks present challenges of their own, in this thesis we are interested only in the first and third tasks of the packet forwarding process of IP routers.

For the first task, routers make forwarding decisions based on the destination address of the packet and the contents of a forwarding table. Since the operation of determining the forwarding information is based on the packet's destination address, it is called address lookup. The address lookup operation is discussed in section 2.2. In particular, we will

[1]Other less time consuming tasks are also performed in the packet forwarding process. The interested reader can refer to [Bak95] [Awe00] [PCB+98] for the details.

see the relation between the address lookup operation and evolution of the IP addressing structure. Later in section 2.3, we describe more precisely what the address lookup operation consist of in the current Internet addressing scheme.

Concerning the second task, we do not address in this thesis the topic of how packets are switched internally in routers. The next reference can be consulted for details on this topic [Min01].

For the third task, routers generally resolve output link contention by the use of a buffer. However, the use of buffers to resolve output link contention is only effective under conditions of transient overload. In the general case , when sustained overload can exist, to obtain effective resolution of output link contention, routers need to make effective use of buffers by controlling the input traffic. In section 2.4, we discuss the relation between the output link contention resolution and the problem of flow isolation.

2.2 The IP Address Lookup Operation

The primary role of routers is to forward packets towards their final destination. To this purpose, a router must decide for each incoming packet where to send it next. More exactly, the forwarding decision consists in finding both the address of the next hop (router or final destination) and the egress port through which the packet should be sent. This forwarding information is stored in a forwarding table that the router computes based on the information gathered by routing protocols [Hui00]. The router searches for the appropriate entry in the forwarding table, based on the destination address of the packet; this operation is called **address lookup**. To cope with scalability, routers do not maintain forwarding information at the granularity level of individual destination addresses; instead, entries in the forwarding tables represent aggregates of addresses. The way the forwarding information is aggregated has followed the evolution of the Internet addressing architecture. Hence, before discussing in more detail the address lookup operation of IP routers, we trace in the next section the evolution of the IP addressing architecture[2].

2.2.1 Evolution of the Internet Addressing Architecture

2.2.1.1 The Classful Addressing Scheme

In IP version 4, IP addresses are 32-bit binary numbers and, when broken up into 4 groups of 8 bits, are normally represented as four decimal numbers separated by dots. For example,

[2]For additional information of the IP addressing the reader is referred to [Sem01].

the address 10000010_01010110_00010000_01000010 corresponds in the dotted-decimal notation to 130.86.16.66.

One of the fundamental objectives of the Internet Protocol is to interconnect networks; so routing on a network basis was a natural choice (rather than routing on a host basis). Thus, the IP address scheme initially used a simple two-level hierarchy, with networks at the top level and hosts at the bottom level. This hierarchy is reflected in the fact that an IP address consists of two parts, a network part and a host part. The network part identifies the network to which a host is attached and thus all hosts attached to the same network agree in the network part of their IP addresses.

Since the network part corresponds to the first bits of the IP address we will refer to this network part also as the **address prefix**, or simply the prefix, for short. We will write prefixes as bit strings of up to 32 bits in IPv4 followed by a "*". For example, the prefix 1000001001010110* represents all the 2^{16} addresses that begin with the bit pattern 1000001001010110. Alternatively, prefixes can be indicated using the dotted-decimal notation, so the same prefix can be written as 130.86/16, where the number after the slash indicates the length of the prefix.

With a two-level hierarchy, IP routers forwarded packets based only on the network part, until packets reached the destination network. As a result, a forwarding table only needed to store a single entry to forward packets to all the hosts attached to the same network. This technique is called **address aggregation** and allows routers to represent with a single prefix a group of addresses with the same forwarding information. In other words, each entry in a forwarding table contains a prefix and its corresponding forwarding information, as can be seen in Table 2.1. To find the forwarding information for a given destination address, a router search for the prefix in the forwarding table that matches the corresponding bits of the destination address.

Destination Address Prefix	Next-hop	Output interface
24.40.32/20	192.41.177.148	2
130.86/16	192.41.177.181	6
208.12.16/20	192.41.177.241	4
208.12.21/24	192.41.177.196	1
167.24.103/24	192.41.177.3	4

Table 2.1: A forwarding table

To forward packets based on the network part, the IP address space must be allocated by partitioning it into networks. The addressing architecture specifies how the allocation of addresses is performed, that is it defines how to partition the total IP address space of 2^{32} addresses. Specifically, how many network addresses will be allowed and of what size each of them should be. When the Internet addressing was initially designed, a rather simple address allocation scheme was defined, which is known today as the **classful addressing scheme**. Basically, three different sizes of networks were defined in this scheme, identified by a class name: class A, B, and C (see figure 2.2). Size of networks was determined by the number of bits used to represent the network part and the host part. Thus networks of class A, B or C consisted in an 8, 16 or 24-bit network part and a corresponding 24, 16 or 8-bit host part.

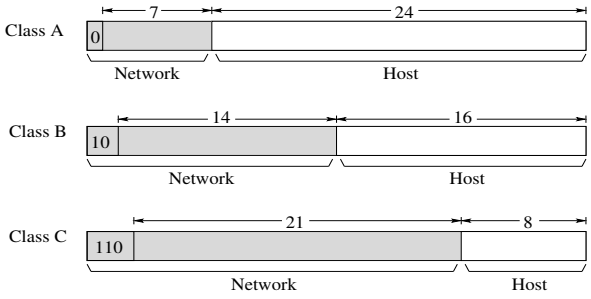

Figure 2.2: Classful Addresses

With this scheme there were very few class A networks and their addressing space represented 50% of the total IPv4 address space (2^{31} addresses out of a total of 2^{32}). There were 16,384 (2^{14}) class B networks with a maximum of 65,534 hosts per network and 2,097,152 (2^{21}) class C networks with up to 256 hosts. This allocation scheme worked well in the early days of the Internet. However, the continuous growth of the number of hosts and networks have made apparent two problems with the classful addressing architecture. First, with only three different network sizes to choose, the address space was not used efficiently and the IP address space was getting exhausted very rapidly, even though only a small fraction of the addresses allocated were actually in use. Second, although the state information stored in the forwarding tables did not grow in proportion to the number of hosts, it still grew in proportion to the number of networks. This was especially important in the backbone routers, which must maintain an entry in the forwarding table for every allocated network address. As a result, the forwarding tables in the backbone routers were growing very rapidly [Hus01][Hus]. The growth of the forwarding tables resulted in higher lookup times and higher memory requirements in the routers and threatened to impact their forwarding

capacity.

2.2.1.2 The CIDR Addressing Scheme

To allow for a more efficient use of the IP address space and to slow down the growth of the backbone forwarding tables, a new scheme called Classless Inter-domain Routing or CIDR [FLYV93] [RL93]was introduced.

Remember, that in the classful address scheme, only 3 different prefix lengths are allowed: 8,16 and 24 corresponding to the classes A, B and C, respectively (see figure 2.2). CIDR makes more efficient use of the IP address space by allowing a finer granularity in the prefix lengths. With CIDR, prefixes can be of arbitrary length rather than constraining them to be 8, 16 or 24 bits long.

To address the problem of forwarding table explosion, CIDR allows address aggregation at several levels. The idea is that the allocation of addresses has a topological significance. Then, we can recursively aggregate addresses at various points within the hierarchy of the Internet's topology. As a result, backbone routers maintain forwarding information not at the network level but at the level of arbitrary aggregates of networks. Thus, recursive address aggregation reduces the number of entries in the forwarding table of backbone routers.

To understand how this works, consider the networks represented by the network numbers from 208.12.16/24 through 208.12.31/24 (see figures 2.3 and 2.4). Suppose that in a router all these network addresses are reachable through the same service provider. From the binary representation we can see that the leftmost 20 bits of all the addresses in this range are the same (11010000 00001100 0001). Thus, we can aggregate these 16 networks into one "supernetwork" represented by the 20-bit prefix, which in decimal notation gives 208.12.16/20. Note that indicating the prefix length is necessary in decimal notation, because the same value may be associated with prefixes of different lengths; for instance, the prefix *208.12.16/20* (11010000 00001100 0001*) is different from the prefix *208.12.16/22* (11010000 00001100 000100*).

While a great deal of aggregation can be achieved if addresses are carefully assigned, in some situations, a few networks can interfere with the process of aggregation. For example, suppose now that customer owing the network 208.12.21/24 changes its service provider and does not want to renumber its network. Now, all the networks from 208.12.16/24 through 208.12.31/24 can be reached through the same service provider, except for the network 208.12.21/24 (see figure 2.4). We cannot perform aggregation as before, and instead of only one entry, 16 entries need to be stored in the forwarding table. One solution that can be used in this situation is aggregating in spite of the exception networks and additionally

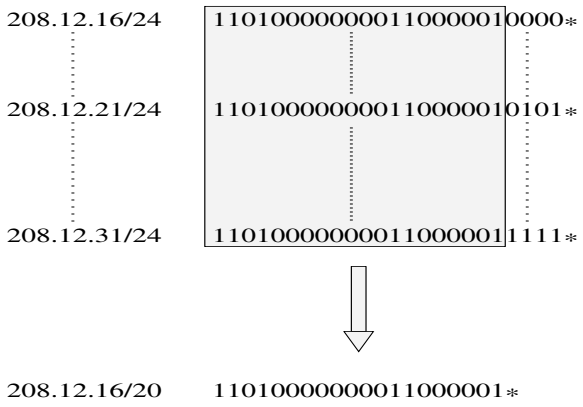

Figure 2.3: Prefix aggregation

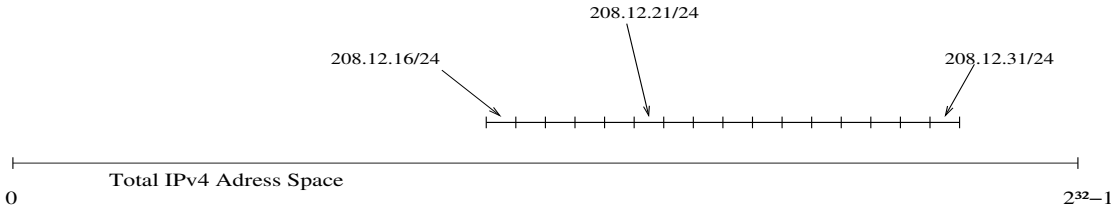

Figure 2.4: Prefix Ranges

storing entries for the exception networks. In our example, this will result in only two entries in the forwarding table: 208.12.16/20 and 208.12.21/24, see figure 2.5 and table 2.1. Note however, that now some addresses will match both entries because prefixes overlap. In order to always make the correct forwarding decision, routers need to do more than to search for a prefix that matches. Since exceptions in the aggregations may exist, a router must find the most specific match, and the most specific match is the **longest matching prefix**. In summary, the address lookup problem in routers now requires to search the forwarding table for the longest prefix that matches the destination address of a packet.

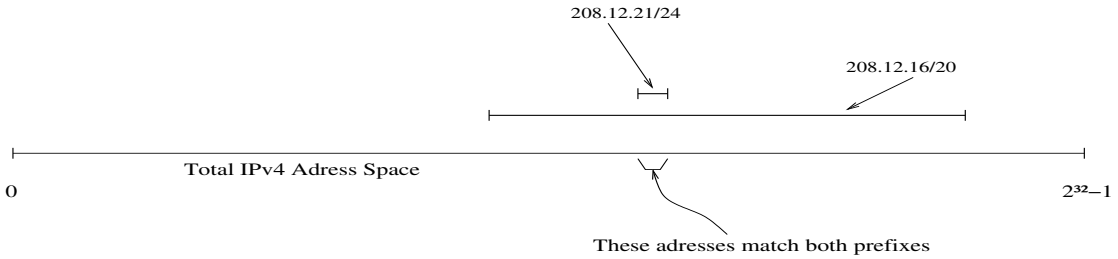

Figure 2.5: Exception prefix

2.3 The Best (Longest) Matching Prefix Search

In the classful addressing architecture, the length of the prefixes was coded in the most significant bits of an IP address (see figure 2.2), and the address lookup was a relatively simple operation: Prefixes in the forwarding table were organized in three separate tables, one for each of the three allowed lengths. The lookup operation amounted to find an exact prefix match in the appropriate table. The search for an exact match could be performed using standard algorithms based on hashing or binary search.

While the use of CIDR reduces the size of the forwarding tables, the address lookup operation becomes more complex. The use of CIDR complicates the address lookup operation because the prefixes in the forwarding tables have arbitrary lengths and no longer correspond to the network part since they are the result of an arbitrary number of network aggregations. Therefore, when using CIDR, the search in a forwarding table can no longer be performed by exact matching because the length of the prefix cannot be derived from the address itself. As a result, determining the longest matching prefix involves not only comparing the bit pattern itself, but also finding the appropriate length. Therefore, we talk about searching in two dimensions: value and length. In what follows we will use N to denote the **number of prefixes** in a forwarding table and W to indicate the **maximum length of prefixes**, which is typically also the length of the IP addresses.

2.3.1 Requirements and Performance Metrics for the BMP Lookup Schemes

Obviously, the main aspect of the performance of BMP lookup scheme is the search time, but it is also important to take into account that to provide robustness, routers must be able to adapt to changes in the network topology. In other words, routers must be able to update dynamically their forwarding databases. In fact, instabilities in the backbone routing protocols can change fairly frequently the entries in a forwarding table. Labovitz [Lab99] found that backbone routers may receive bursts of route changes at rates exceeding several hundred prefix updates per second. He also found that, in average, route changes occur one hundred times per second. Thus, update operations must be performed in 10 msec or less. In summary, a BMP lookup scheme must provide not only fast searches but also incremental updates.

Performance is measured in both time and space requirements. In general, the time to access memory dominates the performance of search schemes. Hence the search time is

generally measured in terms of the number of memory accesses required.

2.3.2 Related BMP Lookup Work

In chapter 4, where we propose a taxonomy and a reference framework for the BMP lookup algorithms, we will classify and describe in more detail the different methods that have been proposed for fast BMP lookups. However, in this section we give a general overview of these proposed schemes for BMP lookups to motivate our work.

A forwarding database of prefixes can be naturally implemented with a binary trie [Sed97] data structure. Several BMP lookup schemes based on variants of binary tries have been proposed [Skl91] [DKN96]. The most commonly implementation is available in the 4.4BSD operating system kernel [Skl91][WS95] (referred to as the BSD trie). Binary tries organizes prefixes in such a way that we can do a sequential search on the prefix length dimension. That is, we can check at step i whether a prefix of length i matches the given address. While binary tries or its variants allow for straightforward algorithms to search, insert and delete prefixes, their main problem is that a search needs to do many memory accesses: $O(W)$; i.e., 32 in the worst case for IPv4 addresses. To improve the search performance of binary tries, a number of schemes use the multibit trie data structure [DBCP97][SV98][GLM98][NK98]. Multibit tries still do linear search on length, but improves search by a constant factor because they allow for inspection of several bits simultaneously at each step. While search performance is improved with multibit tries, insertion and deletion of prefixes are no longer straightforward. Although incremental updates with multibit tries are possible, we need additional data structures to support incremental updates.

Another way to improve the performance of the BMP lookup operation is to use binary search on the prefix length dimension. While binary search on length is not straightforward for best prefix matching, Waldvogel et al. [WVTP97] proposed the use of additional prefixes to guide the binary search. At each step the algorithm checks for a match by using hashing. While this scheme provides $O(logW)$ search performance (assuming perfect hashing), update operations are complex and expensive due to the additional prefixes.

Lampson et al. [LSV98] has proposed to transform a set prefixes into a set of disjoint intervals. Then, BMP lookup can be achieved by using traditional binary search on the endpoints of the disjoint intervals. While the BMP lookup takes $O(logN)$ time (N being the number of prefixes), with this scheme update is very expensive because insertion or deletion of a single prefix may need to update $O(N)$ disjoint intervals.

While intensive research has been conducted to improve the address lookup performance

of routers, most of the proposed schemes addresses the static case; that is, they do not provide for incremental updates. Nevertheless, to provide robustness, routers must be able to adapt to changes in the network topology; and hence, routers must be able to update dynamically their forwarding databases.

We will propose, in chapter 3, our first contribution which allows incremental updates for multibit tries based forwarding databases. More specifically, we will show the details of the additional data structures and algorithms that we have designed to support incremental updates for BMP lookup schemes based on multibit tries.

2.4 Optimizing the Use of Buffers for Flow Isolation

While determining where to send a packet next is essential to forward a packet, this task is only a part of the process to achieve the actual relaying of a packet. Once the router has decided through which output port the packet will be forwarded, the router must multiplex all the packets that need to be forwarded through the same output port. Furthermore, routers must resolve possible contention for a given output port because it is possible that several packets from different inputs need to be forwarded through the same output port at the same time. Usually, routers use buffers to address the problem of output bandwidth contention. Buffering allows routers to retain packets while one of the contending packets is transmitted. However, buffering alleviates the output-port contention only to some extent because in case of sustained overload, the buffer will eventually overflow. This situation is referred to as congestion. In fact, buffers are not only used to resolve the output bandwidth contention, but also to absorb bursts of packets of individual flows. While absorbing bursts of individual flows can help to increase the bandwidth use, it is important to protect the multiplexing function of buffers. Without protecting the multiplexing function, flows can suffer high service degradation in case of congestion. Traditionally, to control congestion in the Internet, sources use the TCP algorithms [Jac88] to discover the available resources and adapt their traffic pattern dynamically.

While control of traffic with TCP allows for **flow isolation** to some extent, this scheme requires that all users cooperate and respond to congestion signals. Nevertheless, in today's Internet responding to congestion is rather a user's choice and in general there are responsive as well as unresponsive users. Furthermore, adaptive sources are penalized because unresponsive sources, intentionally or unintentionally, abuse the cooperative nature of responsive traffic. To address this problem, routers need to provide flow isolation as part of the best effort service. Providing flow isolation is important because with flow isolation the

performance perceived by users does not depend on the good behavior of other users. We propose in chapter 5 a mechanism to optimize the use of the buffer in routers to provide flow isolation.

2.4.1 Related work

To provide flow isolation the router need to control the incoming traffic in times of overload. To provide flow isolation routers need to protect the multiplexing function of buffers. With a FIFO scheduling scheme the router can protect the multiplexing function by selectively discarding packets. One way to protect the multiplexing function of buffers and hence provide flow isolation is to use a separate queue for each flow [Nag87]. With this approach the selection of packets to drop is automatically performed, but it is not scalable and it is complex to implement. RED also uses packet dropping to provide flow isolation to some extent, but it is designed for responsive sources only. RED uses randomization to select the packet to drop. A different way to allow routers to select packets to drop is by including some rate flow information. Schemes that use this approach are [SSZ98][CWZ00][CD01]. While these approaches provide flow isolation for responsive and non responsive flows, to some extent, they require to insert additional information in packets and more importantly they require that all edge routers (or end-hosts) in the system agree on a single scheme to consistently label packets. Our approach, which is proposed in chapter 5 does not need modifications of the IP packet header and achieves a high degree of flow isolation by using a very simple algorithm and without using per-flow queuing.

2.5 Summary

This chapter has provided the necessary background for the rest of this thesis. We have also stated the specific problems that this thesis will address: incremental updates in multibit-tries-based forwarding databases, and optimization of the buffer usage for flow isolation. In the next chapters we provide details of our contributions to these problems.

Chapter 3

Incremental Updates for Multibit-tries based Forwarding Databases

One of the main tasks of the packet forwarding process in IP routers is the address lookup operation. With CIDR, the address lookup operation consists of finding the best (longest) prefix that matches the destination address of the incoming packet. Since the performance of the routers depends on its forwarding capacity, high performance routers must provide fast address lookups. One way to provide fast address lookups is the use of a data structure called multibit trie. While several schemes has been proposed to provide fast address lookups with multibit tries [PZ92], [SV98], [GLM98], [DBCP97], [MS98], [HZ99], [NK99], most of these schemes ignore or diminishes the issue of providing incremental updates. Since one of the key aspects of the Internet is its robustness in the form of adaptation to topological changes, providing incremental updates is a requirement. Furthermore, instabilities in the backbone routing protocols can change fairly frequently the entries in a forwarding table [Lab99]. We propose in this chapter two incremental update mechanisms for address lookup schemes based on the multibit-trie data structure. We start by explaining the multibit trie data structure. Then we state the requirements to support incremental updates in multibit tries. In particular, we introduce two new notions that help to address the problem of providing incremental updates in multibit tries. Finally we develop additional data structures and algorithms that we designed to allow the multibit tries to be updated incrementally.

3.1 The Classical Binary Trie

Routers aggregate forwarding information by using address prefixes; recall that a prefix represents a consecutive interval of IP addresses. As a result, a forwarding table consists of prefixes and their corresponding forwarding information. Prefixes are bit strings of variable length and a natural way to represent prefixes is using a trie data structure [Fre60][Knu98]. A trie is tree-based data structure that organizes prefixes on a digital basis. More specifically, a trie uses the bits of prefixes to direct the branching of its structure. For example, Figure 3.1 shows a binary trie (each node has at most two children) representing a set of prefixes of a forwarding table.

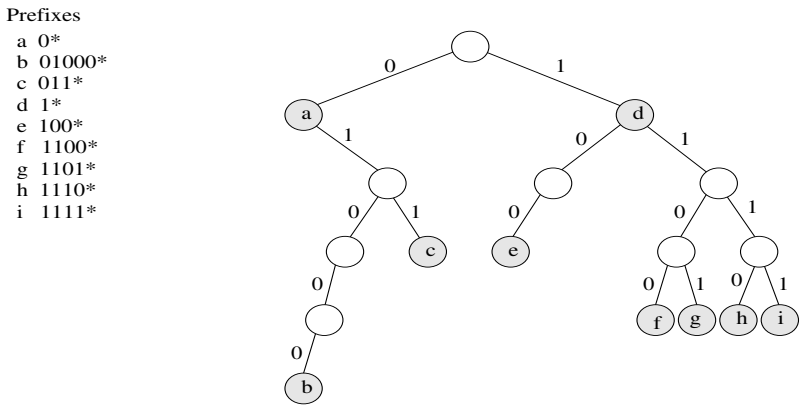

Figure 3.1: Binary trie for a set of prefixes.

Note that in a trie, the bit sequence of a prefix corresponds to the string of bits labeling the path from the root to a node. Note also that a node on level l corresponds to a prefix of length l. For example, node c in figure 3.1 is at level 3 and corresponds to a prefix of length 3. More specifically, node c corresponds to the prefix that represents all addresses beginning with the sequence *011*. In figure 3.1 the nodes that correspond to prefixes are shown in a darker shade; these nodes will contain the forwarding information or a pointer to it. Note also that prefixes are not only located at leaves but also at some internal nodes. This situation arises because of exceptions in the aggregation process (see section 2.2.1.2). For example, in figure 3.1 the prefixes b and c represent exceptions to prefix a. Figure 3.2 illustrates this situation better. The trie shows the total address space, assuming 5-bit long addresses. Each leaf represents one possible address. We can see that the address intervals covered by prefixes b and c are included in the address interval covered by prefix a. Thus, prefixes b and c represent exceptions to prefix a and refer to specific subintervals of the address interval covered by prefix a. In the trie in figure 3.1, this is reflected by the fact

that prefixes *b* and *c* are descendants of prefix *a*; or in other words, prefixes *b* and *c* have themselves *a* as a common prefix. As a result, some addresses will match several prefixes. For example, addresses beginning with *011* will match both prefixes *c* and *a*. Nevertheless, prefix *c* must be preferred because it is more specific (longest match rule).

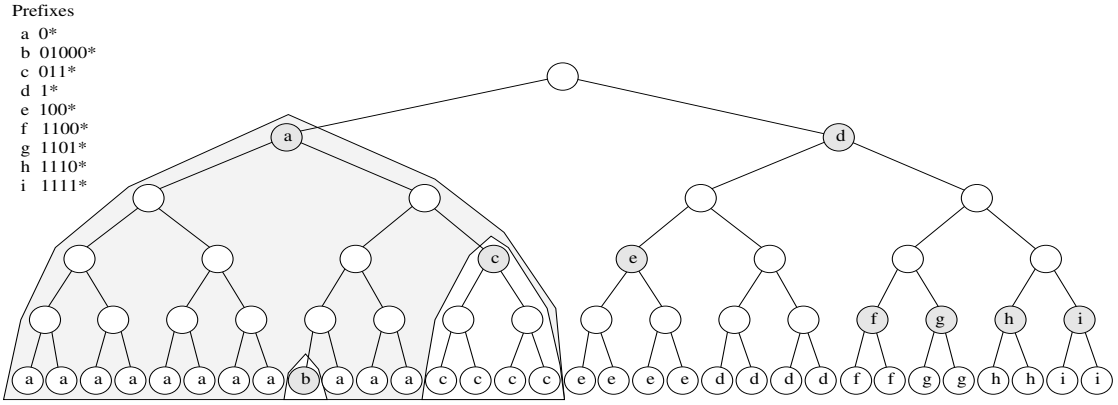

Figure 3.2: Address space

Tries allow finding, in a straightforward way, the longest prefix that matches a given destination address. The search in a trie is guided by the bits of the destination address. At each node, the search proceeds to the left or right according to the sequential inspection of the address bits. While traversing the trie, every time we visit a node marked as prefix (i.e., a dark node) we remember this prefix as the longest match found so far. The search ends when there are no more branches to take, and the longest or best matching prefix will be the last prefix remembered. For instance, if we search the best matching prefix (BMP) for an address beginning with the bit pattern *10110* we start at the root in figure 3.1. Since the first bit of the address is *1* we move to the right, to the node marked as prefix *d* and we remember *d* as the BMP found so far. Then we move to the left since the second address bit is *0*; this time the node is not marked as a prefix, so *d* is still the BMP found so far. Next, the third address bit is *1*, but at this point there is no branch labeled *1,* so the search ends and the last remembered BMP (i.e., prefix *d*) is the longest matching prefix.

In fact, what we are doing is a sequential prefix search by length, trying at each step to find a better match. We begin by looking in the set of length-1 prefixes, which are located at the first level in the trie, then in the set of length-2, located at the second level, and so on. Moreover, using a trie has the advantage that while stepping through the trie, the search space is reduced hierarchically. At each step, the set of potential prefixes is reduced, and the search ends when this set is reduced to one.

Update operations are also straightforward to implement in binary tries. Inserting a

prefix begins by doing a search. When arriving at a node with no branch to take, we can insert the necessary nodes. Deleting a prefix starts again by a search, unmarking the node as prefix and, if necessary deleting unused nodes (i.e., leave nodes not marked as prefixes). Note finally that since the bit strings of prefixes are represented by the structure of the trie, the nodes marked as prefixes do not need to store the bit strings themselves.

3.2 Prefix Transformation

Forwarding information is specified with prefixes that represent intervals of addresses. Although the set of prefixes to use is usually determined by the information gathered by the routing protocols, the same forwarding information can be expressed with different sets of prefixes. Various transformations are possible according to special needs, but one of the most common prefix transformation techniques is **prefix expansion**. Expanding a prefix means transforming one prefix into several longer and thus more specific prefixes but that together cover the same interval of addresses as the original prefix. These new prefixes will be called the **derived prefixes** of the original prefix. As an example, the interval of addresses covered by prefix *1** can also be specified with the two derived prefixes: *10**, *11**; or also, with the four derived prefixes: *100**, *101**, *110**, *111**. In general, to obtain derived prefixes of length *h* from an original prefix of length *l*, all we need is to append, at each time, one of all the possible bit patterns of length *h-l* to the original prefix. Since there are 2^{h-l} distinct binary patterns of length *h-l,* the expansion of an original prefix will result in 2^{h-l} derived-prefixes.

If we do prefix expansion appropriately, we can get a set of derived-prefixes that has fewer different lengths, which can be used to make a faster search, as we will show next.

3.3 Multibit tries

3.3.1 Basic Scheme

Binary tries provide an easy way to handle arbitrary length prefixes. Lookup and update operations are straightforward. Nevertheless, the search in a binary trie can be rather slow because we inspect one bit at a time and in the worst case 32 memory accesses are needed for an IPv4 address.

One way to speedup the search operation is to inspect not just one bit a time but *several bits simultaneously*. For instance, if we inspect 4 bits at a time we need only 8 memory

accesses in the worst case for an IPv4 address. The number of bits to be inspected per step is called **stride** and can be constant or variable. A trie data structure that allows the inspection of bits in strides of several bits is called **multibit** trie. Thus, a multibit trie is a trie where each node has 2^k children, where k is the stride.

While multibit tries allow the data structure to be traversed in strides of several bits at a time, for the same reason, they cannot support arbitrary prefix lengths. To use a given multibit trie, the prefix set must be transformed into an equivalent set with the prefix lengths allowed by the multibit strides, while preserving the same forwarding information. That is, the set of prefixes must be fitted to the strides allowed by the multibit trie. To this end, the original prefixes need to be expanded up to the nearest length allowed in the multibit trie. These new stride-fitted prefixes, i.e the derived prefixes, can now be readily stored in the multibit trie. For instance, a multibit trie corresponding to our example from figure 3.1 is shown in figure 3.3. Since the first stride of the multibit trie is two, prefixes of length one are not allowed, and we need to expand prefixes *a* and *d* to length 2. Expansion of prefix *a* results in the two derived prefixes: 00* and 01*. Similarly, the derived prefixes of the expansion of *d* are: 10* and 11*. Since the derived prefixes represent together the same interval of addresses as their corresponding original prefix, the derived prefixes must have the same forwarding information as their original prefix; in figure 3.3, this is indicated by labeling the derived prefixes with the name of their original prefix. In the same figure it is shown how prefix *c* has been expanded to length 4 (0110* and 0111*). Note that the other prefixes has not been expanded because they fit exactly into the strides of the multibit trie. Note also that the height of the trie has decreased, and so has the number of memory accesses when doing a search.

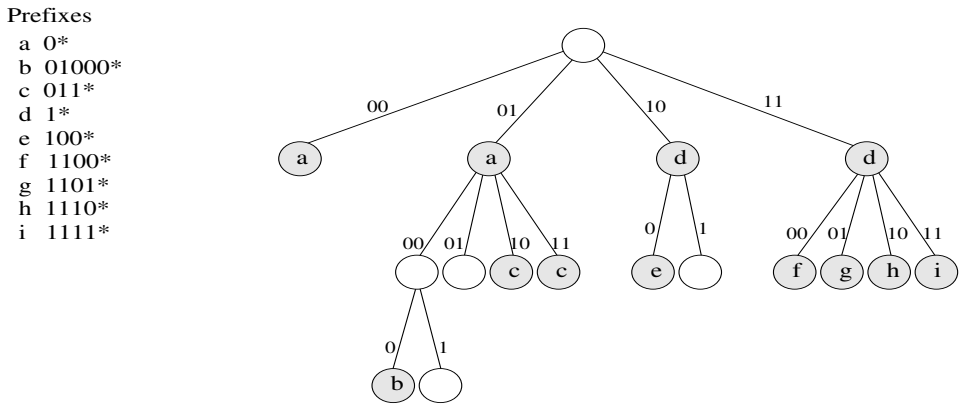

Figure 3.3: An example of multibit trie with the same forwarding information as the binary trie of figure 3.1

Figure 3.4 shows a different multibit trie for our example. We can see again that prefixes *a* and *d* have been expanded, but now to length three. However, two of the derived prefixes produced by expansion already exist (prefixes *c* and *e*). We must preserve the forwarding information of prefixes *c* and *e,* since their forwarding information is more specific than that of the expanded prefixes (i.e., *a* and *d*). Thus, for prefix *a* only three of its four derived prefixes are associated with the forwarding information of *a*. The same applies for prefix *d*.

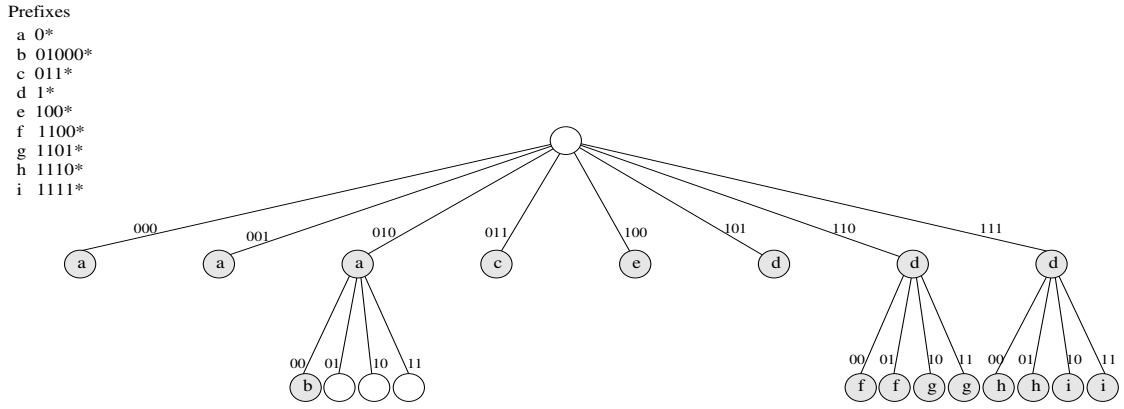

Figure 3.4: Another example of a multibit trie

Figure 3.5 shows the multibit trie in figure 3.4 after inserting a new prefix *j*=11*. Note that this prefix needs to be expanded to length 3, and that its derived prefixes are 110* and 111*. Note also that these same derived prefixes resulted also from the expansion of the prefix *d*=1*. To respect the rule of the longest match, the forwarding information of these derived prefixes must be that of prefix *j*=11* and not that of *d*=1*. This is indicated in figure 3.5 by relabeling the corresponding nodes with *j* instead of *d*.

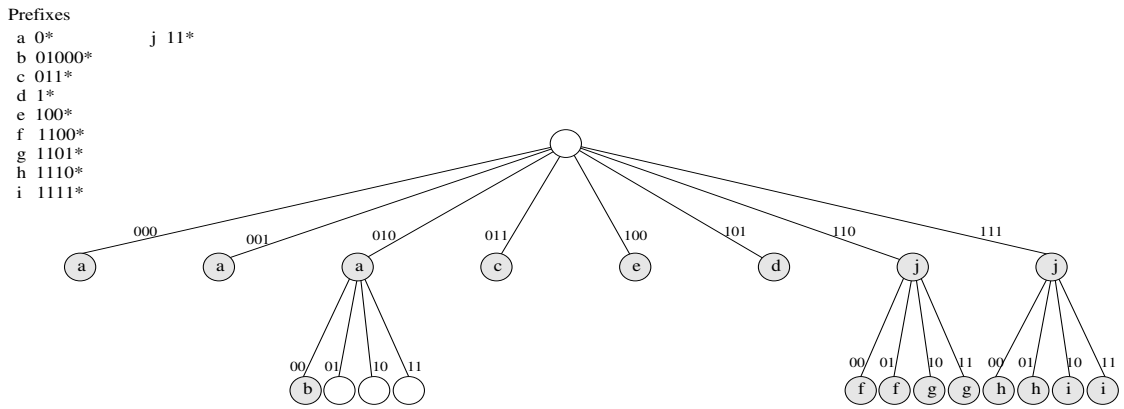

Figure 3.5: The multibit trie of figure 3.4 after inserting prefix *j*=11*

In general, the same node in a multibit trie can be the derived prefix of different original prefixes, when they are expanded. To preserve the forwarding information, we must respect the longest matching rule; hence, we always assign to this node the forwarding information of the longest of these original prefixes. Note that if we insert the prefixes in increasing order of length, we do not need to care whether a node is the derived prefix of several original prefixes. Thus, if we insert the prefixes in increasing order of length, we simply assign to each and all of the derived prefixes of the expansion of prefix P the forwarding information of P; nodes will be automatically relabeled when appropriate. While the approach of inserting prefixes in increasing order of length simplifies the process of expansion of prefixes, clearly, this approach does not allow incremental updates. We will see in section 3, how to avoid the restriction of inserting prefixes in increasing order of length, while still preserving the original forwarding information.

Searching in a multibit trie is essentially the same as in a binary trie. To find the BMP of a given address consists of successively looking for longer prefixes that match. The multibit trie is traversed, and each time a prefix is found at a node, this prefix is remembered as the new BMP seen so far. At the end, the last BMP found is the correct BMP for the given address. Multibit tries still do linear search on lengths as do binary tries, but the search is faster because the trie is traversed using larger strides.

In a multibit trie, if all nodes at the same level have the same stride size we say that it is a **fixed** stride, otherwise it is a **variable** stride. We can choose multibit tries with fixed or variable strides. Fixed strides are simpler to implement than variable strides, but in general waste more memory. Figure 3.4 is an example of a fixed-stride multibit trie, figure 3.3 a variable-stride multibit trie.

3.3.2 Choice of Strides

Choosing the strides requires a tradeoff between search speed and memory consumption. In the extreme case, we could make a trie with a single level; that is, a one-level trie with a 32-bit stride for IPv4. Search would take in this case just one access, but we would need a huge amount of memory to store 2^{32} entries.

One natural way to choose strides and control the memory consumption is to let the structure of the binary trie determine this choice. For example, if we look at figure 3.1, we observe that the subtrie with its root the right child of node *d* is a full subtrie of two levels (a full binary subtrie is a subtrie where each level has the maximum number of nodes). We can replace this full binary subtrie with a one-level multibit subtrie. The stride of the

multibit subtrie is simply the number of levels of the substituted full binary subtrie, two in our example. In fact, this transformation was already made in figure 3.3. This transformation is straightforward, but since it is the only transformation we can do in figure 3.1, it has a limited benefit. We will see later how to replace, in a controlled way, binary subtries that are not necessarily full subtries . The height of the multibit trie will be reduced while controlling memory consumption. We will also see how optimization techniques can be used to choose the strides.

While multibit tries allows for fast BMP lookups, incremental update of the forwarding database is no longer straightforward with multibit tries, as it was the case with binary tries. Incremental updates in multibit tries are possible, and we will see in the next section what are the requirements to support incremental updates in forwarding databases based on multibit tries, whatever the strides they use.

3.4 Requirements to Support Incremental Updates in Multibit Tries

We have seen how to transform a set of prefixes to fit it into a given multibit trie. We have seen that by inserting the prefixes in increasing order of length, we can easily expand the prefixes while preserving the original forwarding information. Unfortunately, this method does not allow incremental updates. In this section we analyze the requirements to support incremental updates in multibit tries.

To better understand the update problem, let's see the multibit trie data structure from a slightly different point of view. We will assume a multibit trie where some prefixes are already inserted, and new prefixes of arbitrary length can be inserted or existing prefixes deleted. Suppose we have a multibit trie of two levels as it is shown in figure 3.5. This multibit trie can also be viewed as a tree of subtries, where every subtrie has only one level. This is best illustrated in figure 3.6; the multibit trie has one subtrie at its first level and three subtries at its second level. Note that a subtrie of stride k has 2^k child nodes. In general, each of these child nodes is associated with an original prefix (we will see shortly why some nodes are not). The original prefix associated with each child node is, in fact, its BMP. For example, the child node whose bit string (i.e., its path) is 11001 is associated with the original prefix f because f=1100* is its BMP. Since the BMP of the child nodes depends on the specific set of prefixes of the forwarding database, when a prefix is inserted or deleted we need to update appropriately the BMP of the child nodes.

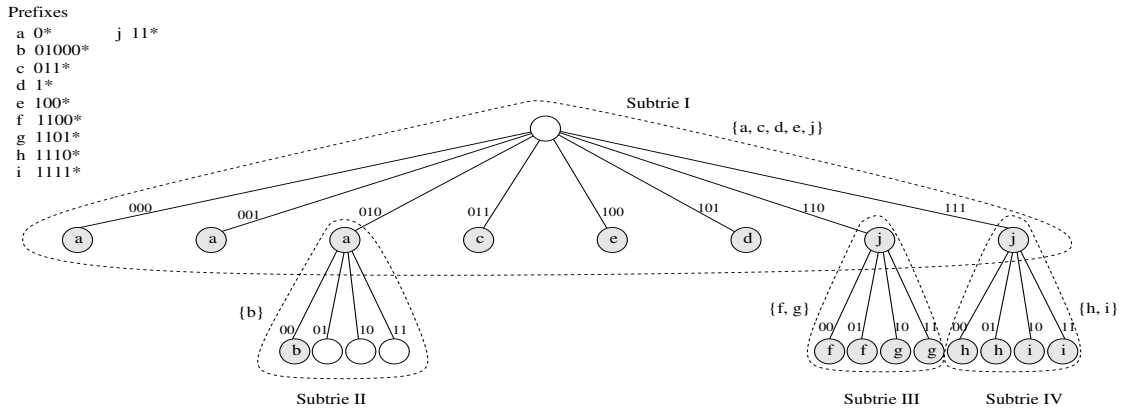

Figure 3.6: Subtries in a multibit trie

While the BMP associated with each child node must be one of the original prefixes, to select the BMP of a given child node, we do not need to consider the whole set of original prefixes. For instance, in figure 3.6, to find the BMP of each child node in the subtrie at the first level (subtrie I), we need to consider only the subset of prefixes $\{a, c, d, e, j\}$. In the subtrie II, the BMP for each child node will be selected from the subset $\{b\}$. In the subtrie III, the BMP for each child node is selected among the prefixes in the subset $\{f, g\}$, and subtrie IV is concerned only with prefixes in the subset $\{h,i\}$. In general, the subset of prefixes, to be considered when selecting the BMP for a child node, consists of only the original prefixes that, due to their length and value, fit into the stride of the concerned subtrie. Since the subsets of prefixes for the different subtries are disjoint, the BMPs computed at each subtrie are entirely independent of the BMPs computed at other subtries. As a result, multibit tries divide the problem of finding the BMP into small problems in which **local BMPs** are selected among a well defined subset of prefixes.

Finally note that in figure 3.6, subtrie II has some child nodes without BMP. In fact, the absence of BMP in a child node indicates simply that there is no local BMP for this node; i.e., among the subset of prefixes corresponding to this subtrie. Of course, when looking for the BMP of a given address, we can always obtain the BMP of this address by traversing the tree of subtries and remembering the last local BMP as we go through it.

Since the subsets of original prefixes for the subtries are disjoint, each original prefix is associated with only one subtrie. As a result, inserting or deleting an original prefix needs to update only one of the subtries; prefix update is completely local. But, which nodes need to be updated in a subtrie when a prefix is inserted or deleted?

When a prefix is inserted or deleted in the forwarding database, we need to update the multibit trie by changing the BMP of some of the child nodes of the appropriate subtrie.

More precisely, when a prefix P is inserted or deleted, the candidate nodes to be updated are the child nodes that correspond to the derived prefixes of the expansion of P in the subtrie. For example, in figure 3.6, if we want to delete the prefix *j*=11*, the candidate nodes to be updated are 110 and 111 because these nodes correspond to the derived prefixes of the expansion of *j*. Similarly, if we want to insert a new prefix t=0100*, then the candidate nodes to be updated are 01000 and 01001 because these nodes correspond to the derived prefixes of the expansion of t.

We will refer to the set of derived prefixes resulting from the expansion of a prefix P in a subtrie as the **span** of P. For example, figure 3.7 shows the spans of the prefixes *d*, *e* and *j*. Thus, when a prefix P is inserted or deleted, we need to update only the nodes (in the subtrie) that are included in the span of the prefix P. Moreover, while all the nodes in the span of a prefix need to be potentially updated, only the nodes not included in the span of longer prefixes than P are actually updated. The spans of longer prefixes than P are not updated to respect the longest matching rule. For example, in figure 3.7 if we delete the prefix *d*, the only node to be updated is actually the node 101, because all the other nodes in the span of *d* are also included in the span of longer prefixes; that is, they correspond to prefixes with more specific forwarding information.

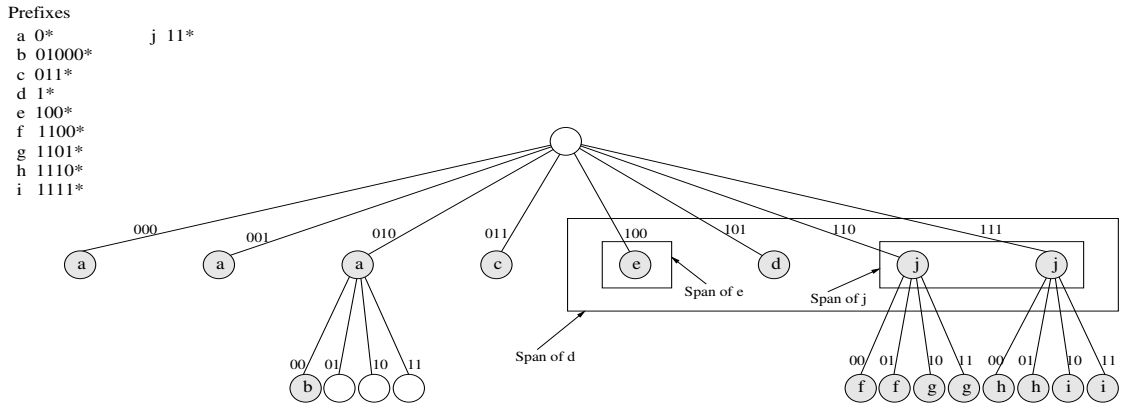

Figure 3.7: Spans of some of the prefixes in the multibit trie

Thus, to support dynamic updates we need a way to identify the spans of (longer) prefixes already inserted.

In the case of insertion of a new prefix P, the operation consists in finding the appropriate subtrie, expanding the original prefix to get the derived prefixes and changing the BMP of the nodes in the span of P not included in spans of longer prefixes than P. That is, for each node in the span of P, it must be decided whether the node keeps its current BMP or its BMP must be substituted with the new prefix P.

In the case of the deletion operation the process is a little more complicated. Suppose that we want to delete the original prefix P. First, we need to find the target subtrie, then we must locate the span of the prefix P. Then again, we must decide for each node in the span of P whether its BMP must be changed: For each node in the span of P, its current BMP must be changed only if it is equal to P. Note that if the current BMP is not P, then the current BMP is necessarily longer than P, and thus the current BMP must not change. But, what is the new BMP in the case that the node's BMP needs to be updated? For example, in figure 3.6, if we delete the prefix *j*=11*, the nodes to be updated are those in the span of *j*; that is, nodes 110 and 111. From the figure 3.7, we can see that when the prefix *j* is deleted, the smallest span that includes these nodes is that of prefix *d*. In other words, when the prefix *j* is deleted, the new BMP for the nodes in the span of *j* is the prefix *d*, because after *j* the next BMP for these nodes is *d*. In fact, these nodes were labeled with *d* before the insertion of prefix *j*, as can be seen in figure 3.4. Note that *d* is the BMP of *j*; and in general, the new BMP for the nodes in the span of a prefix P to be deleted can be obtained by finding the BMP of the prefix P. We will refer to the BMP of an original prefix P as the **coverer** of P. To find the coverer of a prefix we need an additional data structure besides the multibit trie itself. This additional data structure is needed because, in general, a multibit trie does not store the original prefixes of the forwarding database themselves; instead, a multibit trie stores the stride-fitted prefixes derived from the expansion of these original prefixes. Actually, the original prefixes appear only as the associated BMP of each derived prefix (except when the original prefix fits exactly into the stride of the subtrie). To see this better, suppose we insert the prefix *m=10**, in the multibit trie in figure 3.7. Clearly, prefix *d* will disappear in the multibit trie, while it still exists in the forwarding database. Hence, to support update operations we need an additional data structure that keeps the original prefixes of the forwarding database. Obviously, to find the coverer of a prefix, this additional data structure must also keep the covering relationship among the prefixes.

Thus, in summary, incremental updates in multibit tries are possible because insertion or deletion of a prefix needs modifications in only one subtrie. More specifically, if a prefix is inserted or deleted, in a subtrie with a stride of k bits, the update needs to modify the local BMP of at most 2^{k-1}nodes (the span of a prefix in a subtrie has at most half of the child nodes in a subtrie). Thus, choosing appropriate stride values allows the update time to be bounded. While incremental updates in multibit tries are possible, to actually provide incremental updates, we need an additional data structure with two requirements:

1. Identify the spans of prefixes already inserted in a subtrie.

2. Find the coverer of an original prefix in a subtrie.

We present in the next sections the details of our data structures and algorithms that we designed to provide incremental updates for multibit tries. We start with the basic scheme that provides the basic BMP search operation. Then we refine progressively our scheme to provide insertions and deletions of prefixes. Later we propose an optimized scheme based in a data structure that we call Prefix Nesting bit vector, which is introduced in section 3.6.

3.5 A Scheme to support incremental updates in Multibit Tries

In the previous section we have stated the requirements to support incremental updates in multibit-tries-based address lookup schemes. These requirements are based on the notions of span and coverer of a prefix, that we introduced also in the previous section. In this section we propose a first mechanism to support incremental updates in multibit-tries-based address lookup schemes. More specifically, we present the details of the additional data structures and algorithms that we have designed to allow for incremental updates in multibit trie based forwarding databases. Later, in section 3.6, we will propose a second mechanism for incremental updates in multibit tries. This second mechanism uses memory more efficiently than our first mechanism by using a new data structure that we call the Prefix Nesting bit vector.

3.5.1 Implementation of the Basic Multibit Trie

The data structure to implement the multibit trie is very simple. Each *k*-stride subtrie of the multibit trie is implemented as an array with 2^k entries. The entries in an array correspond to the child nodes in the subtrie. Each array entry has a pointer to a next possible subtrie. Also, each array entry has a pointer to its current local BMP. In the case of a variable stride multibit trie, each array entry has an additional field to store the stride of the next possible subtrie. Figure 3.8 shows the array structure corresponding to the fixed-stride multibit trie of the figure 3.6.

To find the BMP of a destination address A, we proceed as follows: We follow a path in the multibit trie guided by the bit string of the destination address A. More specifically, we partition this bit string into segments according to the strides of the subtries that we follow in the path. These segments are used as indexes into the arrays. At each step, the

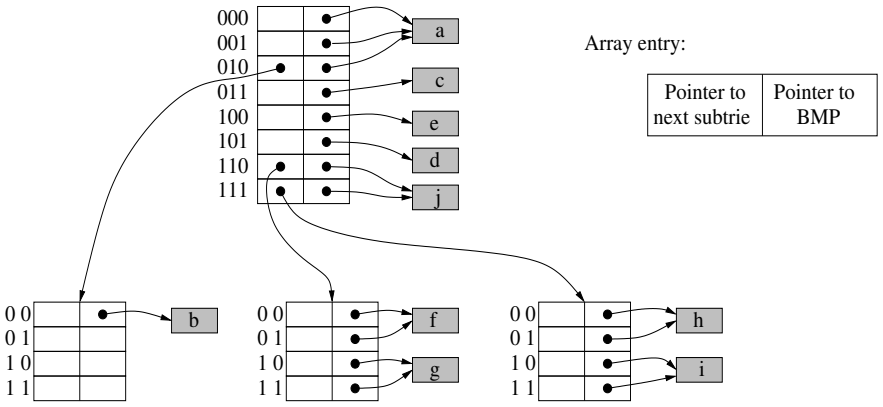

Figure 3.8: The array implementation of the multibit trie

corresponding entry in the array will contain the pointer to the next array. Also, we keep the BMP of this entry, if any, as the BMP found so far. Eventually, an entry will contain no next subtrie pointer to follow. At this moment, the last BMP remembered will be the BMP of the destination address A.

3.5.2 Locating the target subtrie

We have seen that we need to modify only one subtrie when a prefix P is inserted or deleted. Hence, the algorithm must first find the appropriate subtrie for the prefix P; we will refer to this subtrie as the **target subtrie**. To locate the target subtrie of prefix P, we follow a path in the multibit trie guided by the bit string of P. More specifically, we partition this bit string into segments according to the strides of the subtries that we follow in the path. These segments are used as indexes into the arrays (as in the search operation). At each step, the corresponding entry in the array will contain the pointer to the next array. Eventually, the last segment of the prefix will be shorter or equal to the stride of the corresponding subtrie. This subtrie is the target subtrie. Of course, this supposes that all the required subtries already exist. If this is not the case, we simply create arrays when we encounter no more pointers to follow until the last segment of the prefix. Fig 3.9 shows the target subtrie for the prefix P=1100011*, when the subtries have 3-bit strides. Let p_1, ..., p_n be the segments of the prefix P, and let K_i be the stride of the *i*-th subtrie in the path, then the length of the last segment is smaller or equal than the stride of the target subtrie, that is $\mid p_n \mid \leq K_n$.

Actually, not all the entries in the target subtrie need to be updated when a prefix is inserted or deleted, only the entries corresponding to the span of the prefix need to be updated. Remember that the span of a prefix P is the set of derived-prefixes resulting from

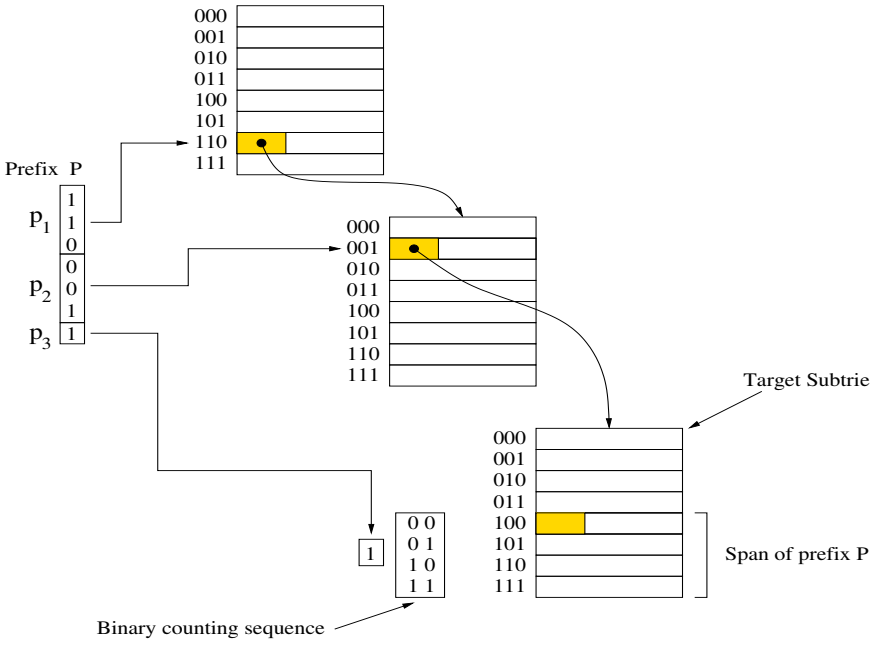

Figure 3.9: The target subtrie and span of a prefix P

the expansion of P, to fit the stride of the target subtrie. The span of a prefix corresponds to a block of consecutive entries in the array. For example, figure 3.9 shows the span of the prefix 1100011*. We can index the entries of the span by using the last segment of the prefix, p_n, and a binary counting sequence. This binary counting sequence is the set of all the bit patterns of length *w*, where *w* is equal to the number of necessary bits to fit the original prefix to the stride of the target subtrie. Thus, the length of these bit patterns is given by K_n-$\mid p_n \mid$; and the initial and final values of the binary counting sequence are 0 and $2^{K_n - \mid p_n \mid} - 1$, respectively. For example, in figure 3.9, the binary counting sequence for prefix P has 4 different bit patterns of length 2 because K_3-$\mid p_3 \mid$= 3-1=2.

3.5.3 The additional data structure to support incremental updates

We have seen that the multibit trie actually do not store the original prefixes of the forwarding database; instead, each array entry has a variable current BMP, selected among the original prefixes.

To support incremental updates we need an additional data structure to keep track of the original prefixes in the forwarding database. This additional data structure must also keep the covering relationship among the original prefixes to meet the two requirements defined in section 3.4. We explain progressively how we implement this additional data structure.

Figure 3.10 shows the updated entries in the target subtrie once the prefix P has been inserted in the example shown in figure 3.9. Apparently, the prefix P itself is stored in our data structure, but this is not true. Suppose that we insert prefixes Q=11000110* and R=11000111*. Figure 3.11 shows the insertion of these prefixes. Note that now there is no way to further find prefix P. This is a problem because if later prefix Q or R are deleted, the corresponding entries must be updated back again with prefix P. The problem lies in the fact that the pointers point, in fact, to the current BMP, which can change when new prefixes are inserted.

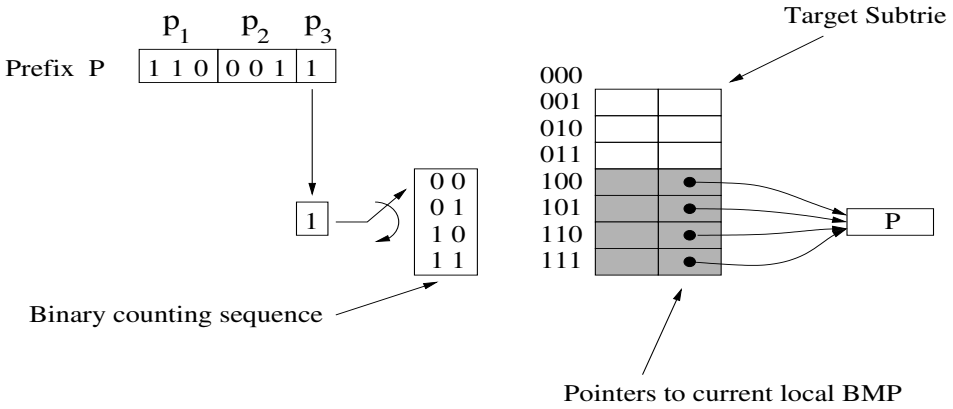

Figure 3.10: The target subtrie after inserting prefix P

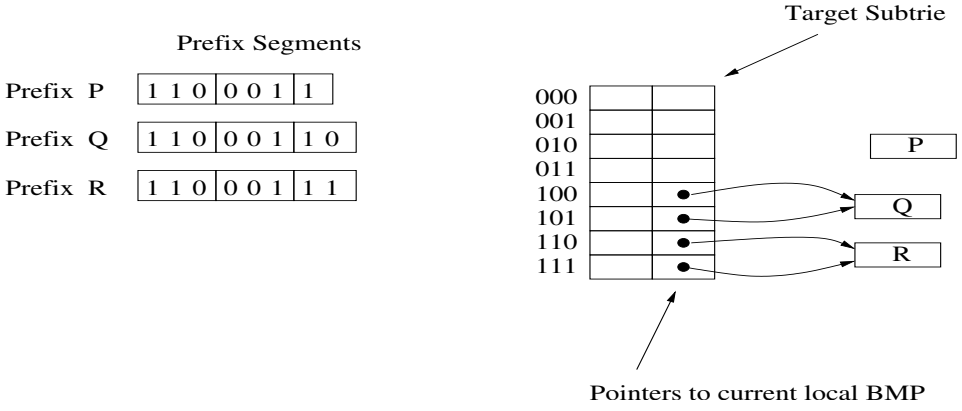

Figure 3.11: The target subtrie after inserting prefixes P,Q and R

One solution is to use an additional pointer per prefix. This additional pointer will always point to the original prefix, even if no array entry has it as its current BMP. That is, a requirement is that once a prefix is inserted, this pointer will exist until the prefix is actually deleted from the forwarding database. The question now is where these pointers should be

stored. We use the following simple rule to store the original prefixes:

> **Prefix storing rule**: Store the pointer to the original prefix P in the entry of the target subtrie where the span of P starts.

The span of prefix P starts at the entry whose index is the last segment of the prefix followed by K_n-| p_n | bits "0". We will refer to this array entry as the **start** of the span of P. Note, however, that by using this prefix storing rule, the same array entry may need to point to several original prefixes. More specifically, the prefixes with the same value but with different length (in the same target subtrie). To point to several prefixes in the same entry, the algorithm uses, in fact, a linked list of prefixes. Figure 3.12 shows this data structure. The array entry with index 100 has a linked list of the prefixes P and Q, because P and Q start their span at this entry.

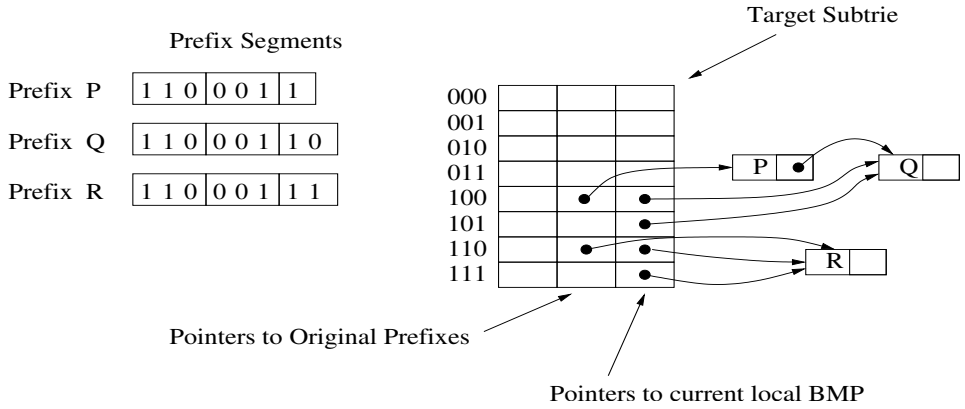

Figure 3.12: Using both a pointer to the current BMP and a pointer to the linked list of original prefixes

Since each entry in the array can potentially be the start of the span of some prefix, one could think that each array entry should store two pointers, as figure 3.12 suggests: One for the current local BMP and one for the linked list of original prefixes. Actually, each array entry needs only one pointer, because if an entry has a linked list, the current local BMP for this entry is, in fact, included in its linked list. All that is needed is a way to identify the current local BMP among the prefixes in the list. To make the BMP search operation efficient, we always keep the current local BMP at the front of the list. In other words, the first prefix is the longest in the list. Figure 3.13 shows the simplified data structure. Note that the pointers at entries with indexes 101 and 111 conceptually do not point to a linked list but only to their current BMP. Figures 3.14 and 3.15 show, respectively, the algorithms to insert and delete the original prefixes in a linked list.

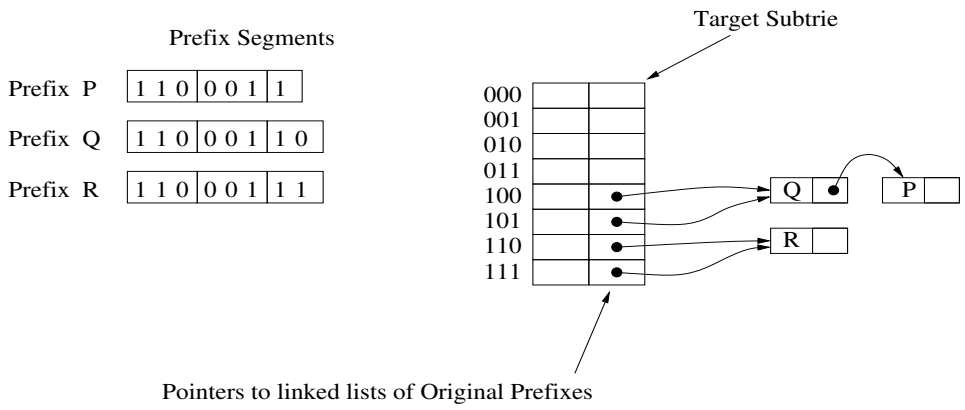

Figure 3.13: Using only one pointer to point to both: the linked list and the BMP

```
input     : prefix to be inserted in the linked list; the pointer to this list is stored in
            the entry of the target subtrie indexed by entryindex
procedure insertoriginalprefix(prefix, targetsubtrie, entryindex, k)
begin
    listpointer = targetsubtrie[entryindex].BMP ;
    if (listpointer = Null) then
        targetsubtrie[entryindex].BMP = memoryaddress(prefix);
    else
        sp = startspan(listpointer->value, k);
        if (sp ≠ entryindex) then
            /* i.e. no list at this entry, only the current BMP */
            targetsubtrie[entryindex].BMP = address(prefix);
        else if (listpointer−>length < length(prefix)) then
            /* insert prefix in front of the list */
            prefix.next= listpointer ;
            targetsubtrie[entryindex].BMP = address(prefix);
        else
            prefix.next= listpointer−>next ;
            listpointer−>next = address(prefix);
        end
    end
end
```

Figure 3.14: Algorithm to insert a prefix in the appropriate linked list

```
input    : prefix to be deleted from the linked list; the pointer to this list is stored in
           the entry of the target subtrie indexed by entryindex
procedure deleteoriginalprefix(prefix, targetsubtrie, entryindex)
begin
    listpointer = targetsubtrie[entryindex].BMP ;
    while listpointer->length ≠ length(prefix) do
        prevptr = listpointer;
        listpointer = listpointer->next;
    end
    prevptr->next = listpointer->next ;
    freememory(listpointer);
end
```

Figure 3.15: Algorithm to delete a prefix from the appropriate linked list

3.5.4 Getting the coverer of a prefix

In section 3.4 we have seen that to support incremental updates two requirements are necessary: Find the coverer of a prefix and find the longer prefixes already inserted in the span of the concerned prefix. In this section we propose an algorithm to meet the first requirement.

We have seen that to delete or to insert a prefix we need to update only the array entries in the span of the prefix. While both operations, insertion and deletion of a prefix need to update the span of the prefix, the update of the span of a prefix to be deleted is more complicated. When we insert a prefix P, P is used as the new BMP for the entries that need to be updated. In contrast, in the case of deletion of a prefix, we need first to find the new BMP to be used to update the entries. In the case of deletion of a prefix P, the new BMP to update the span of P is the coverer of P; a concept that we introduce in section 3.4.

From the above discussion we know that to support deletion of prefixes we need to implement a function that finds the coverer of a given original prefix. To this end, we need a data structure that stores the original prefixes and that keeps the covering relationship among these prefixes.

To locate the coverer of the prefix P, we have to find the longest prefix of P among the original prefixes stored in the target subtrie. If the length of P is *l* then we look successively for the prefix of P of length *l*-1, *l*-2, ... *l*-$\mid p_n \mid$+1. The first prefix found will be the coverer. First, we search the coverer in the same linked list where the prefix to be deleted is located. If the coverer of P is not in this list then we need to search this coverer in other lists. From the rule definition of where to store the prefixes, it follows that to go from one list to another, we start with the current entry index, and proceeding from right to left, we toggle the first

bit “1” we encounter to “0”. The result will be the index of the entry where the next linked list to search is located. With this procedure, we can go from linked list to linked list, to search the coverer of P.

For example, in figure 3.16, if we want to find the coverer of prefix U (110001111), we proceed as follows: We start at the entry where the prefix U is located, that is the entry whose index is 111. Since the list at this entry has no other prefix than U, we need to go to the next linked list to search for the possible prefix 11000111*. To this end, we toggle the last bit “1” to “0” in the index value 111; we obtain 110 as the entry index for the next linked list. Since the prefix 11000111* is not included in the list at this entry, then we continue the search in the next list, but now to search for the prefix 1100011*. Again we toggle the last bit “1” to “0” in the index value 110; we obtain 100 as the entry index for the next linked list. The linked list at this entry does have the prefix 1100011*. Hence, the coverer of U is the prefix P=1100011*.

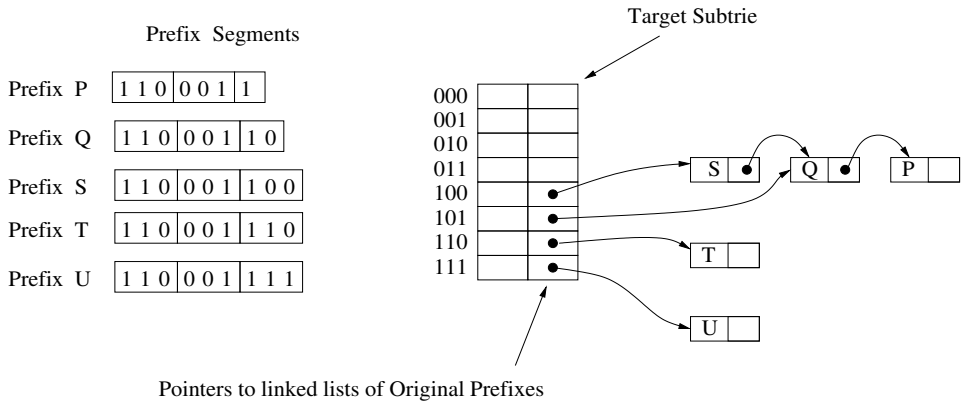

Figure 3.16: Example with a target subtrie after inserting prefixes P, Q, S, T, and U

Now, we propose an optimization to make the search for the coverer faster. Instead of search explicitly at each linked list, we store at each entry a **bit vector** of size equal to the stride. The idea is that the bits in the bit vector indicate whether the prefixes of the corresponding length are in the linked list. With these bit vectors, we do not need to search explicitly in the linked lists, except at the very last list where we search explicitly through the linked list to retrieve the coverer. Actually, the bits in the bit vector indicate only the length of the last segment of the prefix, because the length of the previous segments is the same for all the prefixes in the same target subtrie. That is, prefixes in a subtrie can be uniquely identified by their last segment. Figure 3.17 shows the same example in figure 3.16 but using bit vectors. The complete algorithm to find the coverer of a prefix with the use of bit vectors is shown in figure 3.18.

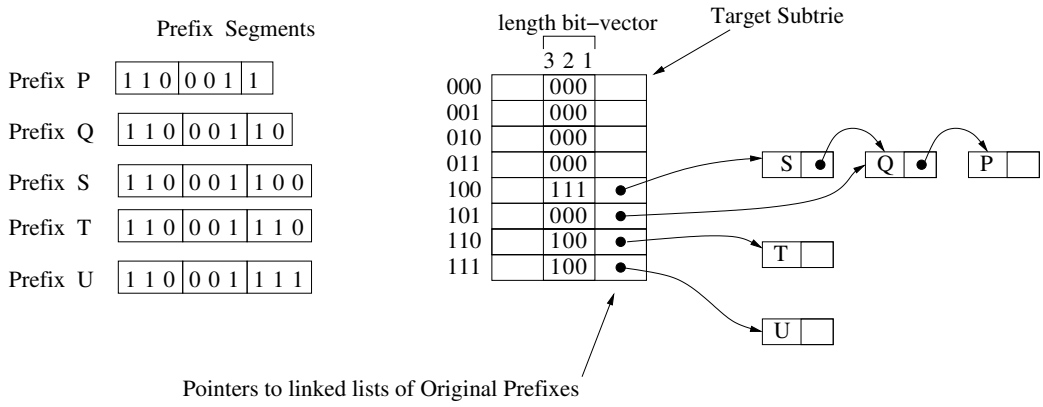

Figure 3.17: Use of a length bit-vector to efficiently search in lists

```
Data     : prefix P of length l
procedure findcoverer(P, targetsubtrie, k)
begin
    lastsegmentP = last segment of prefix P;
    startspanP = startspan(lastsegmentP, k) ;
    entryindex = startspanP ;
    targetlength= length(lastsegmentP);
    partiallen= length(P)-length(lastsegmentP);
    bp = k-targetlength;
    while targetlength>0 and NOT(targetsubtrie[entryindex].bitlengths & 2^bp) do
        targetlength−−;
        bp = k-targetlength;
        /* clear the last bp bits of entryindex */
        entryindex= (entryindex>>bp)<<bp;
    end
    if targetlength>0 then
        listpointer = targetsubtrie[entryindex].BMP;
        covererlen= partiallen + targetlength;
        while listpointer−>length ≠ covererlen do
            listpointer = listpointer−>next;
        end
        return listpointer;
    else
        return Null;
    end
end
```

Figure 3.18: First Algorithm to find the coverer of a prefix P. startspan is function defined in figure 3.24 that returns the array index where the span of P starts. & is the bitwise and operator. $>>$ and $<<$ are respectively the right and left bit shift operators.

3.5.5 Updating the span of a prefix

Both operations, insertion and deletion of a prefix need to update the span of the concerned prefix. The only difference is that for insertion the new BMP is the prefix itself, while in the case of deletion the new BMP is to be searched among the remaining prefixes. In this section we detail the process of updating the span of a prefix with a new BMP.

Remember that the span of a prefix P is the set of derived-prefixes resulting from the expansion of P. In the array, the span of P corresponds to a block of consecutive entries. For example, figure 3.19 shows the spans of the prefixes Q, S, and U. Note that the span of the prefix S is included in the span of the shorter prefix Q. Thus, the span of S is a subspan of Q; and we say that the prefix S is covered by the shorter prefix Q; or equivalently that the prefix Q covers the longer prefix S.

Since the span of a prefix P can include several subspans (i.e., smaller spans of longer prefixes), when a prefix P is inserted or deleted, not all the entries in the span of P need to be updated; only the entries not included in subspans need to change their BMP. This is illustrated in figure 3.20, which shows the example of figure 3.19 after inserting the prefix P. We can see that the only entry not included in subspans is the entry with index 110; thus, only this entry is updated with P as BMP.

We have seen that the subspans of a prefix P are the spans of longer prefixes covered by the prefix P. Note that, by the prefix storing rule definition, the prefixes covered by prefix P are necessarily stored in the entries in the span of P. Hence, update of the span of a prefix P consists of scanning the entries of its span to check the presence of longer prefixes. When such a longer prefix is found the algorithm skips the span of this longer prefix, to respect the more specific forwarding information of this span. In the other case, the entry is updated with the new BMP. The detailed algorithm to update the span of a prefix is shown in figure 3.21. Note that the algorithm skips spans efficiently, because the algorithm always skips the largest possible subspan, when the span of P contains nested subspans. For example, in figure 3.20, when prefix P is inserted, the algorithm skips the span of prefix Q, which includes the span of prefix S. Note that by scanning the entries in the span of P from lowest to greatest index and with the help of the bit vector, the algorithm always skips the largest possible span.

3.5.6 Inserting a prefix

The algorithm to insert a prefix P consists basically in the following steps:

- Find the target subtrie.

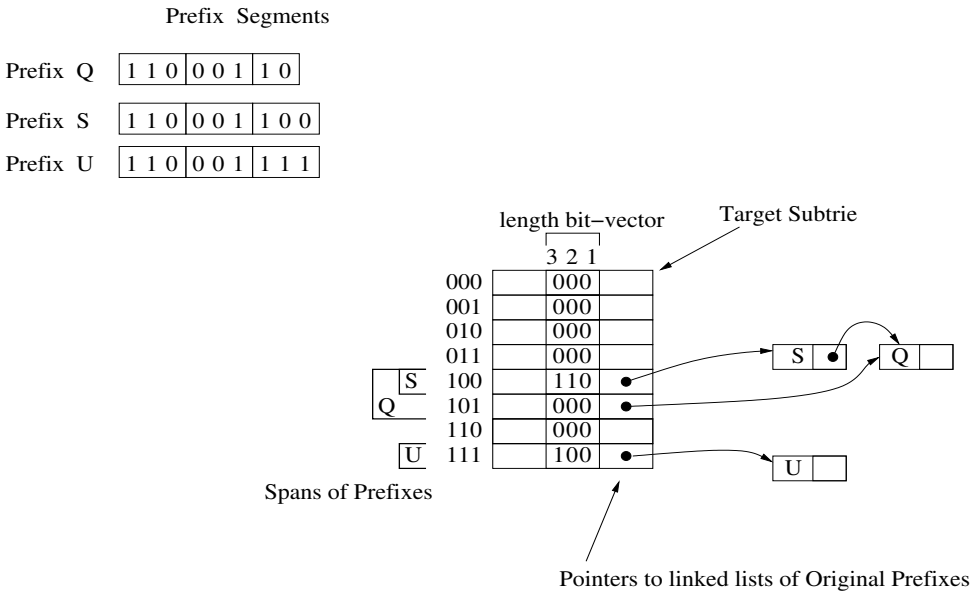

Figure 3.19: Spans of prefixes Q, S, and U

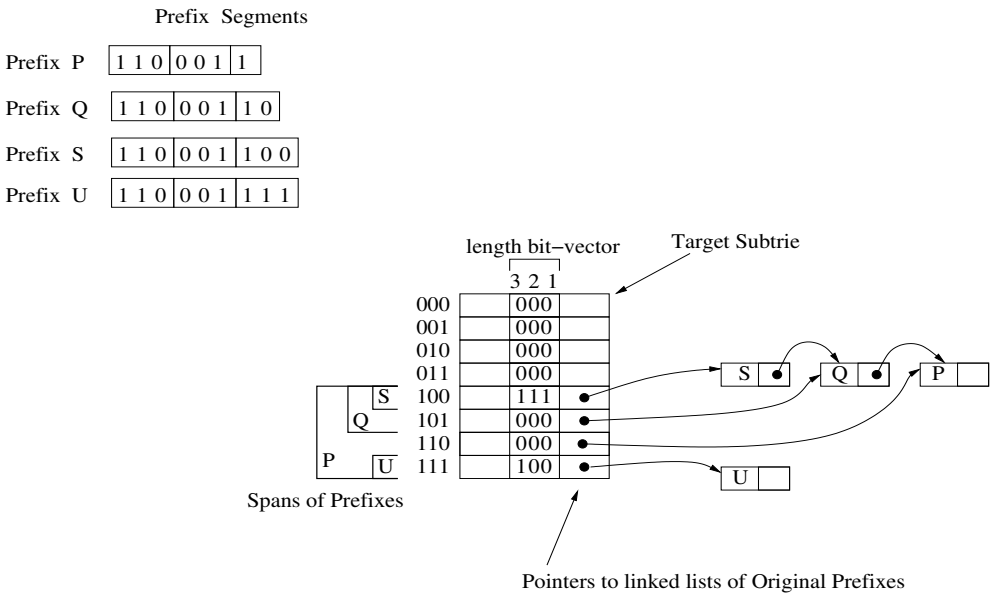

Figure 3.20: Spans of prefixes P, Q, S, and U

```
Data      : prefix P of length l
procedure updatespan(P, targetsubtrie, k, newBMP)
begin
    lastsegmentP = last segment of prefix P;
    startspanP = startspan(lastsegmentP, k) ;
    endspanP = endspan(lastsegmentP, k) ;
    entryindex = startspanP ;
    while entryindex ≤ endspanP do
        targetlength= length(lastsegmentP);
        /* Is there a prefix longer than P? */
        if targetsubtrie[entryindex].lengthbits ≥ 2^targetlength then
            /* Find the exact length of the shortest prefix longer than P */
            while NOT(targetsubtrie[entryindex].lengthbits & 2^targetlength) do
                targetlength++;
            end
            skipentries = 2^(k−targetlength−1);
        else
            /*update current BMP of targetsubtrie[entryindex] with new BMP */
            targetsubtrie[entryindex].BMP = memoryaddress(newBMP);
            skipentries = 1 ;
        end
        /* skip the entries in the span of the longer prefix Q */
        entryindex = entryindex + skipentries
    end
end

.
```

Figure 3.21: First Algorithm to update the span of a prefix P with a new BMP. startspan and endspan are functions defined in figure 3.24 that return the array indexes delimiting the span of P. & is the bitwise and operator.

- Check whether the prefix P is already inserted.
- Expand the prefix P; that is, update the span of P using P as the new BMP.
- Set the bit in the corresponding bit vector.
- Insert prefix P in the appropriate linked list.

The complete algorithm is shown in figure 3.22.

```
Data      : prefix P of length l
procedure InsertPrefix(P)
begin
    Locate the target subtrie ;
    k = the stride of the target subtrie ;
    lastsegmentP = last segment of prefix P;
    len=length(lastsegmentP);
    startspanP = startspan(lastsegmentP, k) ;
    /* Is the length position of P in the corresponding bit-vector set?*/
    if targetsubtrie[startspanP].bitlengths & 2^(len-1) then
        print(" prefix P already exists");
        return "error";
    else
        updatespan(P,targetsubtrie, k, P);
        /* set bit corresponding to P, in the bit vector */
        targetsubtrie[startspanP].bitlengths | = 2^(len-1);
        insertoriginalprefix(P, targetsubtrie, startspanP, k);
    end
end
```

Figure 3.22: First Algorithm to insert a prefix in a multibit trie. startspan is function defined in figure 3.24 that returns the array index where the span of P starts. & is the bitwise and operator. | is the bitwise or operator.

3.5.7 Deleting a prefix

The algorithm to delete a prefix P consists basically in the following steps:

- Find the target subtrie.
- Check whether the prefix P does really exists.
- Retrieve prefix Q, which is the coverer of P.

- Update the span of P using Q as the new BMP.
- Clear the bit, which corresponds to P in the appropriate bit vector.
- Delete prefix P in the appropriate linked list.

Figure 3.23 shows the complete algorithm for the deletion of a prefix in a multibit trie. Note that, in the insertion algorithm the subroutine updatespan is called with P as both, the prefix and the new BMP, while the deletion algorithm calls updatespan with P as prefix and Q as new BMP; where Q is the prefix obtained by the function findcoverer.

```
Data      : prefix P of length l
procedure DeletePrefix(P)
begin
    Locate the target subtrie ;
    k = the stride of the target subtrie ;
    lastsegmentP = last segment of prefix P;
    len=length(lastsegmentP);
    startspanP = startspan(lastsegmentP, k) ;
    if NOT(targetsubtrie[startspanP].bitlengths & 2^(len-1)) then
        print(" prefix P does not exist");
        return "error";
    else
        Q = findcoverer(P, targetsubtrie, k);
        updatespan(P,targetsubtrie, k, Q);
        targetsubtrie[startspanP].bitlengths &= ~(2^(len-1));
        deleteoriginalprefix(P, targetsubtrie, startspanP);
    end
end
```

Figure 3.23: First Algorithm to delete a prefix from a multibit trie. startspan is function defined in figure 3.24 that returns the array index where the span of P starts. & is the bitwise and operator. $\sim$ is the bitwise not operator.

3.6 Optimized scheme using a single bit vector per subtrie: The PN bit vector

While the use of a length bit-vector per array entry allows the linked lists to be searched fast, the memory usage is incremented. In this section we propose a method to use memory

```
input    : bitstring is the last segment of the prefix; and k is the stride of the target
           subtrie (i.e., the array)
output   : the array index where the span of the prefix starts
procedure  startspan (bitstring, k)
begin
    segmentlength = lengthstring(bitstring);
    /* padding with "0"s to fit the stride */
    startspan = bitstring << (k - segmentlength);
    return startspan;
end

input    : bitstring is the last segment of the prefix; and k is the stride of the target
           subtrie (i.e., the array)
output   : the array index where the span of the prefix ends
procedure  endspan (bitstring, k)
begin
    segmentlength = lengthstring(bitstring);
    paddingbits = k - segmentlength ;
    /* padding with "1"s to fit the stride */
    endspan = (bitstring << paddingbits) | (2^paddingbits − 1);
    return endspan;
end
```

Figure 3.24: Algorithm to calculate the start and end of the span of a prefix in a target subtrie of stride k. $<<$ is the left bit shift operator. $|$ is the bitwise or operator.

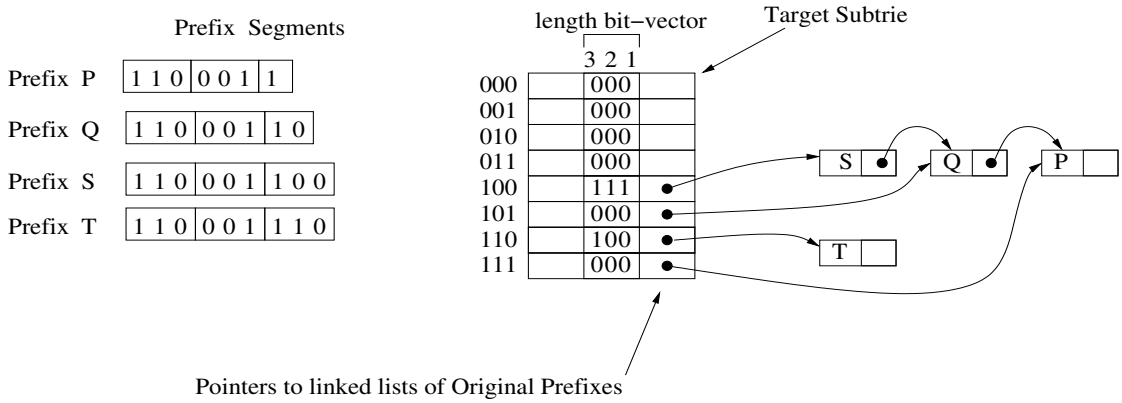

Figure 3.25: The target subtrie after deleting prefix U

efficiently while keeping the advantages of the length bit-vector. Our second mechanism to support incremental updates in multibit tries is based on a new data structure called the Prefix Nesting (PN) bit vector which we will introduce next. To distinguish **the PN-bit-vector mechanism** proposed in this section from the first mechanism proposed in the previous section, we will refer to the mechanism proposed in the previous section as the **bit-vector-array mechanism**.

A key observation is that while every entry in the array can potentially have a linked list of original prefixes, the maximum number of potential prefixes in each linked list is different. In other words, for each array entry, not all the prefix lengths in its bit-vector are possible. For example, in figure 3.20, the bit vector at the array entry whose index is 111 can have set only the bit corresponding to length 3 (prefix U). This bit vector will never have set the bits corresponding to lengths 1 and 2, because the corresponding prefixes are stored in other linked lists (according to the prefix storing rule). As a result, some bit positions in the length bit-vectors are useless. To use the memory efficiently, our idea is to use one single length bit-vector per subtrie, without the useless bits, instead of one length bit-vector per entry with some useless bits. Note also that for small strides, the gain in memory lies in the fact that the minimum length bit vector would be a byte, i.e., without the optimization. Another advantage is that we can keep this unique length bit-vector separated from the array data structure, which optimize the BMP search operation (search of the BMP of a given address). We call this single length bit vector the Prefix Nesting bit vector or **PN bit vector**, for short. As we will see shortly, the PN bit vector not only encodes the length of prefixes but also their nesting structure or covering relationship.

To obtain the single PN bit-vector we conceptually concatenate the length bit-vectors in the array entries, from the smallest index to the greatest index, but without including the useless bits. (We joint the individual bit-vectors of array entries into a single bit vector but without the useless bits). For example, figure 3.26 shows the single PN bit-vector for the example in figure 3.17.

3.6.1 Mapping prefixes to bit positions in the PN bit vector

To support update operations we need to be able to find the coverer of a prefix and also the longer prefixes already inserted in the span of the concerned prefix. To support these tasks with the PN bit vector we need a way to map prefixes to bit positions in the PN bit vector. We introduce in this section these mapping operations.

For a given prefix P, we can obtain its corresponding position in the PN bit-vector

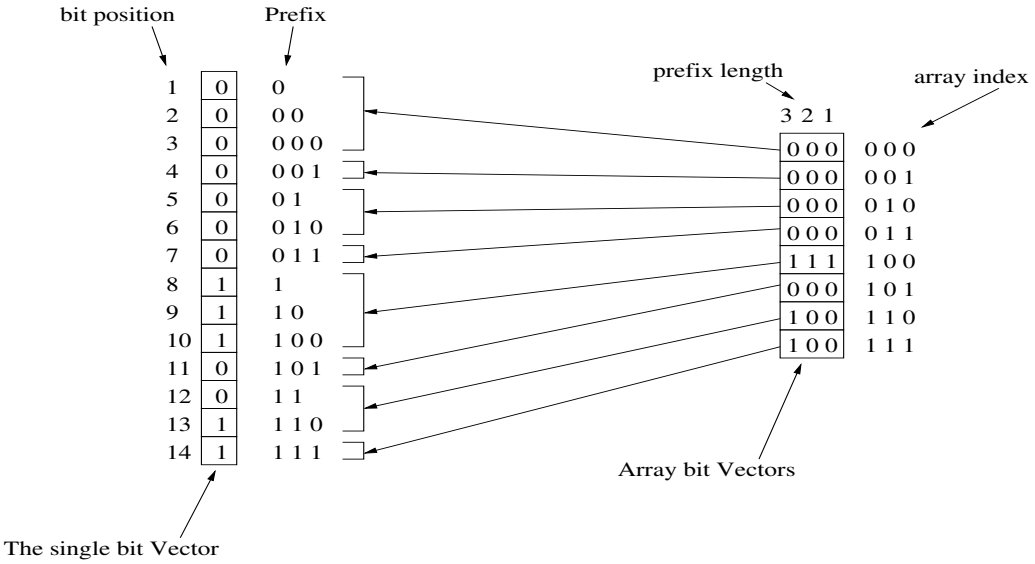

Figure 3.26: The PN bit Vector

as follows: Let $b_m b_{m-1} ... b_2 b_1$ be the bit string of the last segment p_n of prefix P, then its bit position (*bp*) in the PN bit vector is defined by the following function: $bp(P) = (2^{k-m}) \sum_{i=1}^{m} b_i 2^i + \sum_{i=1}^{m} \overline{b_i}$, where *k* is the stride of the target subtrie and $\overline{b_i}$ is the bit complement of b_i . Intuitively, this function takes into account the value as well as the length of the prefix to uniquely identify a given prefix. The term 2^{k-m} is to take into account the number of padding bits to fit the stride. The first summation takes into account the bits "1", while the second summation corresponds to the bits "0". Since this function follows the order of the entries in the array, the nesting structure among the prefixes is also encoded. Figure 3.27 shows the algorithm to compute the bit position of a prefix in the PN bit vector. The inverse operation can be readily computed. Figure 3.28 shows the algorithm to reconstruct the binary string of the last segment of a prefix, given its bit position in the PN bit vector.

3.6.2 Getting the coverer of a prefix with the PN bit vector

Remember that the coverer of a prefix P is the longest prefix of P among the original prefixes stored in the target subtrie. Thus if the length of P is *l* then we look successively for the prefix of P of length *l*-1, *l*-2, ... *l*-| p_n |+1; and the first prefix found will be the coverer. In the case of the bit-vector-array mechanism, to check each of the bit vectors we needed to access the corresponding entry in the array. With the single bit vector we do all the process in the single bit vector and we access only the array entry where the coverer is actually stored, at

```
input    : bitstring is the last segment of the prefix; and stride is the stride of the
           target subtrie
output   : the bitposition of the prefix, in the bit-vector of the target subtrie
procedure getbitposition (bitstring, k)
begin
    bitposition = 0 ;
    length = lengthstring(bitstring);
    paddingbits = k - length ;
    for i=1 to length do
        bit = (bitstring & 1); /* get the rightmost bit */
        if bit is set then
            bitposition = bitposition + 2^(i+paddingbits) ;
        else
            bitposition = bitposition + 1 ;
        end
        bitstring= bitstring >> 1 ; /* to check next bit */
    end
    return bitposition;
end
```

Figure 3.27: Algorithm to calculate the bit position of a prefix in a bit-vector

end of the process. Note that checking for the next shorter possible prefix of P is readily done by computing at each step the corresponding bit position. The complete algorithm is shown in figure 3.29.

3.6.3 Updating the span of a prefix with the PN bit vector

The algorithm to update the span of a prefix is essentially the same as in the bit-vector-array mechanism; except that the longer prefixes covered by the prefix are not searched in the array but in the single bit vector instead. Note that we can delimit exactly where the prefixes covered by P are in the PN bit vector. These prefixes are located in the PN bit vector from the bit position where the prefix P is located to the bit position where the last prefix in the span of P is located. For example, for prefix 0* its end of span is 011, and between the bit position corresponding to prefix 0* (i.e., 1) and the bit position of prefix 011 (i.e., 7) are all the possible prefixes covered by 0*.

Hence, when a covered prefix is found its bit string is computed and its corresponding span is skipped in the array. Since the PN bit vector keeps the nesting structure of the covered prefixes, the algorithm skips efficiently the subspans, when there are nested subspans. The

```
input    : bitposition of a prefix in the bit-vector; stride is the stride of the target
           subtrie
output   : a string, which is equal to the last segment of the prefix
procedure  computebitstring (bitposition, k)
begin
    lastsegment = "";
    length = 0;
    value = bitposition;
    i = k ;
    while value > 0  do
        if  value >= 2^i then
            value = value - 2^i ;
            /* Concatenate a bit "1" at the end of lastsegment */
            lastsegment=lastsegment<<1 ;
            lastsegment++ ;
        else
            value = value - 1 ;
            /* Concatenate a bit "0" at the end of lastsegment */
            lastsegment=lastsegment<<1 ;
        end
        length++ ;
        i-- ;
    end
    return lastsegment;
end
```

Figure 3.28: Algorithm to reconstruct the last segment of a prefix, given its bit position in a bit-vector

```
input    : bitstring is the last segment of the prefix; k is the stride of the target subtrie;
           and PN is the PN bit-vector for the target subtrie
output   : the pointer to the prefix that is the coverer of the input prefix
procedure findcoverer (P, bitposition, targetsubtrie, k, PN)
begin
    lastsegmentP = last segment of prefix P;
    len = length(lastsegmentP);
    partiallen = length(P)-len;
    paddingbits = k - len ;
    bitstring = lastsegmentP;
    repeat
        bit = (bitstring & 1); /* get the rightmost bit */
        if bit is set then
            bitposition = bitposition - 2^(k-len+1) ;
        else
            bitposition = bitposition - 1 ;
        end
        bitstring= bitstring >> 1 ; /* to check next bit */
        len--;
    until (PN[bitposition] is SET) or (len ≤ 0) ;
    if len > 0 then
        startspanQ = startspan(bitstring,k);
        listpointer = targetsubtrie[startspanQ].BMP;
        covererlen = partiallen+len;
        while listpointer->length ≠ covererlen do
            listpointer = listpointer->next;
        end
        return listpointer;
    else
        return Null;
    end
end
```

Figure 3.29: Algorithm to find the coverer of a prefix P in a PN bit-vector

complete algorithm to update the span of a prefix is shown in figure 3.30.

3.6.4 Inserting a Prefix

The algorithm to insert a prefix P consists basically in the following steps:

- Find the target subtrie.
- Check whether the prefix P is already inserted.
- Expand the prefix P; that is, update the span of P using P as the new BMP.
- Set the bit corresponding to prefix P in the PN bit vector.
- Insert prefix P in the appropriate linked list.

The complete algorithm is shown in figure 3.31.

3.6.5 Deleting a Prefix

The algorithm to delete a prefix P consists basically in the following steps:

- Find the target subtrie.
- Check whether the prefix P does really exists.
- Retrieve prefix Q, which is the coverer of P.
- Update the span of P using Q as the new BMP.
- Clear the bit, which corresponds to P in the PN bit vector.
- Delete prefix P in the appropriate linked list.

Figure 3.32 shows the complete algorithm for the deletion of a prefix in a multibit trie. Note that, in the insertion algorithm the subroutine updatespan is called with P as both, the prefix and the new BMP, while the deletion algorithm calls updatespan with P as prefix and Q as new BMP; where Q is the prefix obtained by the function findcoverer.

```
Data     : prefix P. bp is the bit position of P in the PN bit-vector of the target subtrie
procedure updatespan(P, bp, targetsubtrie, k, PN, newBMP)
begin
    lastsegmentP = last segment of prefix P;
    startspanP = startspan(lastsegmentP, k) ;
    endspanP = endspan(lastsegmentP, k) ;
    endbp = getbitposition(endspanP, k) ;
    bp++;
    entryindex = startspanP ;
    while entryindex ≤ endspanP do
        while PN[bp] is NOT SET and bp ≤ endbp  do
            bp++;
        end
        if  bp ≤ endbp then
            /* compute the last segment of the prefix corresponding to PN[bp], re-
            ferred as Q */
            lastsegmentQ = computebitstring (bp, k) ;
            y = startspan(lastsegmentQ, k) ;
            endspanQ = endspan(lastsegmentQ, k) ;
            bp = getbitposition (endspanQ, k) + 1;
            skipentries = endspanQ - y + 1;
        else
            y = endspanP + 1;
            skipentries = 0 ;
        end
        while  entryindex < y  do
            /*update current BMP of targetsubtrie[entryindex] with new BMP *
            targetsubtrie[entryindex].BMP = memoryaddress(newBMP);
            entryindex++ ;
        end
        /* skip the entries in the span of prefix Q just found*/
        entryindex = entryindex + skipentries
    end
end
```

Figure 3.30: Algorithm to update the span of a prefix P with a new BMP

```
Data      : prefix P of length l
procedure InsertPrefix(P)
begin
    Locate the target subtrie ;
    k = the stride of the target subtrie ;
    PN = the PN bit-vector of the target subtrie;
    lastsegmentP = last segment of prefix P;
    startspanP = startspan(lastsegmentP, k) ;
    /* calculate bit position of P in the length bit-vector V
    bp = getbitposition(lastsegmentP, k) ;
    if PN[bp] is SET then
        print(" prefix P already exists");
        return "error";
    else
        updatespan(P,bp,targetsubtrie,k,PN, P);
        setbit(PN[bp]);
        insertoriginalprefix(P, targetsubtrie, startspanP, k);
    end
end
```

Figure 3.31: Algorithm to insert a prefix in a multibit trie

3.7 Performance evaluation

We have proposed two mechanisms to support incremental updates in multibit-trie-based forwarding databases: the bit-vector-array mechanism and the PN-bit-vector mechanism. While the bit-vector-array mechanism does not need to compute mapping functions, it uses memory inefficiently. The PN-bit-vector mechanism uses memory more efficiently than the bit-vector array mechanism. The PN bit vector mechanism obtains efficient memory usage by using our PN bit vector data structure and computing the corresponding mapping functions. In other words, the PN bit vector mechanism trades computation for efficient memory usage. Moreover, Since the PN bit vector mechanism separates the additional data structure needed to support incremental updates from the main data structure used for doing BMP lookup operations, the performance of the BMP lookup operation is not affected.

We present in this section the performance results of our two mechanisms: the bit-vector-array mechanism, which uses several bit vectors per subtrie and the optimized mechanism, which uses only one vector per subtrie, that is the PN bit vector mechanism.

The forwarding database used was extracted from a typical backbone router table [Tel03]. In particular, we used a forwarding table with 143168 prefixes. Our programs were coded

```
Data      : prefix P of length l
procedure DeletePrefix(P)
begin
    Locate the target subtrie ;
    k = the stride of the target subtrie ;
    PN = the PN bit-vector of the target subtrie;
    lastsegmentP = last segment of prefix P;
    startspanP = startspan(lastsegmentP, k) ;
    /* calculate bit position of P in the length bit-vector V
    bp = getbitposition(lastsegmentP, k) ;
    if PN[bp] is NOT SET then
        print(" prefix P does not exist");
        return "error";
    else
        Q = findcoverer(P, bp, k, PN);
        updatespan(P,bp,targetsubtrie,k,PN, Q);
        clearbit(PN[bp]);
        deleteoriginalprefix(P, targetsubtrie, startspanP);
    end
end
```

Figure 3.32: Algorithm to delete a prefix from a multibit trie

in C and were executed in the user space under the Linux operating system in a Pentium-III-based computer with a clock speed of 935 MHz.

We have used a multibit trie with the next fixed strides: 16, 8, 8 at the first,second ant third level respectively. While we have used a fixed stride multibit trie in our emulations, our mechanisms can be used with any multibit trie.

3.7.1 Time performance

In this emulation the forwarding database was randomized. Prefixes were inserted in random order to reduce the effects of cache locality. Once all the prefixes were inserted, the prefixes were deleted in the same random order.

Figure 3.33 shows the cumulative distribution of the prefix insertion times for our two mechanisms. Note that the performance of both mechanisms is very similar, which means that the computing overhead of the mapping functions in the PN bit vector mechanism has no impact on the insertion times.

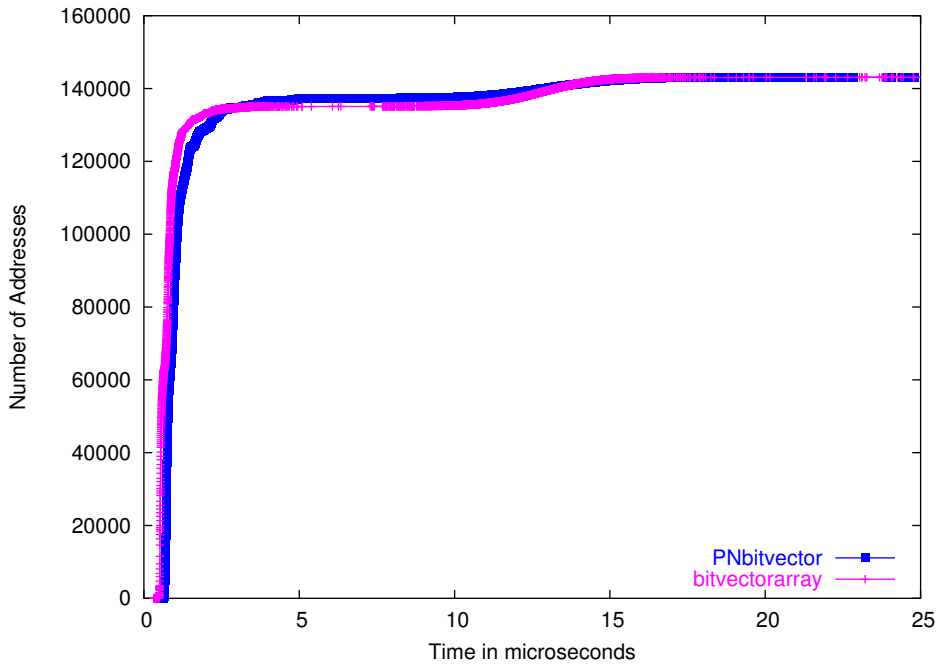

Figure 3.33: The cumulative distribution of the prefix insertion times

Figure 3.34 shows the cumulative distribution of the prefix delete times for our two mechanisms. Again, the performance of both is very similar.

To investigate the impact of the bit vectors on the BMP search operation, we measured

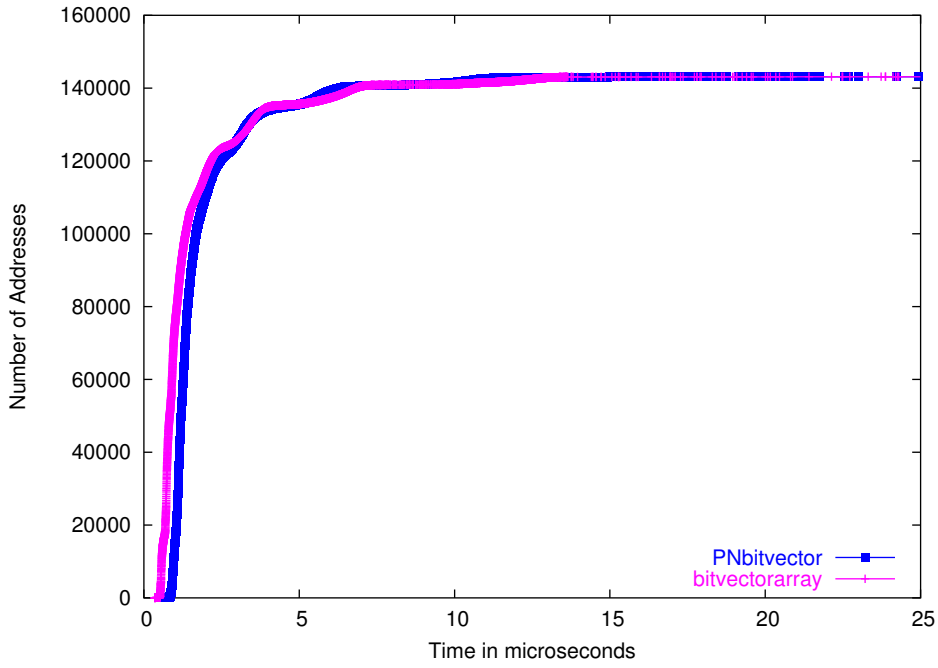

Figure 3.34: The cumulative distribution of the prefix deletion times

the performance of the BMP lookup operation in our two mechanisms and we compared the performance of our mechanisms with that of a "basic" multibit trie mechanism; that is, a multibit trie mechanism without update capabilities. Since traffic statistics depend on the location of the router, what we have done to measure the performance of the lookup operation is to consider that every prefix in the forwarding database has the same probability of being accessed. In other words, we suppose that the traffic per prefix is the same for all prefixes. With this approach we can measure the lookup times inherent to the forwarding database. Indeed, a better knowledge of the specific traffic statistics would allow only a better evaluation of the average lookup time.

Figure 3.35 shows the cumulative distribution of the BMP lookup times for our two mechanisms compared to that of a "basic" multibit trie. Note that the performance of the PN bit vector mechanism is almost equal to that of the "basic" multibit trie. The performance of the bit-vector-array mechanism is slightly lower than that of the "basic" multibit trie. This slight reduction in the performance of the bit-vector-array mechanism is due to the fact that this mechanism increments the memory size of the "main" data structure of the multibit trie (each entry of the arrays must contain one bit vector). In contrast, since in the PN bit vector mechanism the bit vector is maintained separated from the "main" data structure of the multibit trie, the performance of the BMP lookup operation is not affected.

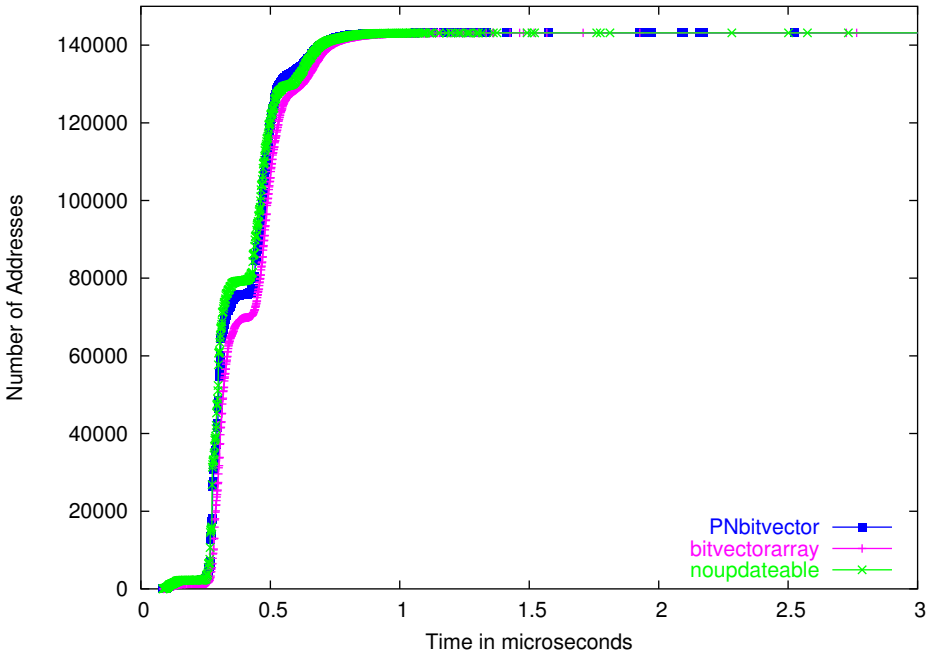

Figure 3.35: The cumulative distribution for the BMP lookup operation

3.7.2 Memory requirements

Table 3.1 shows the number of subtries at each level of the multibit trie and the number of entries for the subtries. The number of entries per subtrie is equals to 2^k where k is the stride of the subtrie. The last row of the table shows the total number of subtries and entries for the multibit trie. Note that these statistics are for the main data structure, that is the data structure needed to do BMP lookups; and they are valid for our two mechanisms. The last column shows the number of bytes required for the entries. Each entry in a subtrie requires to store two pointers[1]; thus, 8 bytes are needed per entry.

Multibit-trie Level	Number of subtries	Entries per subtrie	Total of entries	Total of bytes
1	1	2^{16}	65 536	524 288
2	3 642	2^8	932 352	7 458 816
3	65	2^8	16 640	133 120
Total	3 708		1 014 528	8 116 224

Table 3.1: Subtries per level

[1] one for the current BMP and one for a next possible subtrie

In the bit-vector-array mechanism, each entry in a subtrie has a bit vector. Thus, this mechanism needs additional memory for 1 014 528 bit vectors. The size of each bit vector depends on the stride size of the subtrie. For example, if the maximum stride size is 16, the bit vector needs a minimum of two bytes (i.e., 16 bits). Table 3.2 shows the additional memory required for the bit vectors in the bit-vector-array mechanism. Note that the stride for the first level of the multibit trie is 16, and the strides for the second and third level are 8.

Multibit-trie Level	Number of bit vectors	bytes per bit vector	Total of bytes
1	65 536	2	131 072
2	932 352	1	932 352
3	16 640	1	16 640
Total			1 080 064

Table 3.2: Additional memory for the bit-vector-array mechanism

For the PN bit vector mechanism, only one bit vector per subtrie is required. The size of a PN bit vector in bytes is given by the expression 2^{k-2}, where *k* is the stride of the subtrie. The table 3.3 shows the additional memory required for the PN bit vectors.

Multibit-trie Level	Number of bit vectors	bytes per bit vector	Total of bytes
1	1	2^{14}	16 384
2	3 642	2^{6}	233 088
3	65	2^{6}	4160
Total			253 632

Table 3.3: Additional memory for the PN bit vector mechanism

The memory consumption for the two mechanisms is summarized in table 3.4. The table shows also the fraction of memory for the bit vectors with respect to the total memory for each mechanism.

For the bit-vector-array mechanism the total of memory is obtained by adding the memory size of the main data structure (table 3.1) and the memory size of the bit vectors (table 3.2).

For the PN bit vector mechanism the total memory is again obtained by adding the memory size of the main data structure and the memory size of the PN bit vectors (3.3); but we need to addition also one extra entry per subtrie. This extra entry is needed to point to the PN bit vector. Since there are 3708 subtries, the additional entries contributes with

29664 (3708 x 8) bytes. Note that although each entry in a subtrie has two pointers we use only one of the pointers in the extra entry.

Mechanism	Total of memory bytes	bit vectors bytes	bit vectors % of total memory
bit-vector-array	9 196 288	1 080 064	11.74 %
PN bit vector	8 399 520	253 632	3.02 %

Table 3.4: Comparison of memory consumption of our two mechanisms

3.8 Related work

While several schemes use multibit tries for fast BMP lookups [PZ92], [SV98], [GLM98], [DBCP97], [MS98], [HZ99], [NK99], most of them do not allow for incremental updates. We discuss in this section the schemes that treat the incremental update aspect.

Srinivasan et al. [SV99a] suggests the use of an additional binary trie in each subtrie to store the original prefixes and to find the next best match (the coverer[2]) of an original prefix. Also, they use an extra field in each array entry to store the length of its current BMP. While not all the details of the update algorithms are given (e.g., the deletion algorithm), the idea seems to work. Nevertheless, the use of the extra field per array entry to keep the length of its current BMP can have an impact on the search operation. While the authors claim that this extra field can be stored in auxiliary storage and used only by the update routines, it is not clear how this can be achieved. Also, in their approach, to update the BMP of array entries, they do not skip subspans efficiently, instead they check each entry of the array to decide whether the BMP of the entry must be updated. This is in fact because their approach does not have the notion of spans, which we introduced in section 3.4.

Hong-Yi Tzeng et al. [TP99] use linked lists to store the original prefixes as in our approach. Additionally, they use an extra field per array entry to keep the length of its current BMP, as in the scheme of Srinivasan. Our mechanism differs in that we optimize the search from linked list to linked list by the use of a bit vector. Also we do not need the extra length field which can degrade the search performance.

Hariguchi [Har] scheme also supports incremental updates. However, his approach differs to our approach in that it uses a second pointer instead of our PN bit vector. As a result,

[2]While the notion of coverer is implicitly used in the Srinivasan's approach, it is not clearly defined as we do in our approach.

the arrays need to store three pointers per entry, instead of two as in our approach. While this allows the router to find the coverer with only one access, the memory consumption is incremented, which can impact the search operation performance.

Another approach that supports incremental updates is the Tree bitmap scheme of Eatherton [Eat99]. The Tree bitmap approach reduces the memory requirements by using two bit vectors. One for the pointers to subtries and one for the prefixes in the subtrie. The disadvantage is that subtries need to be stored contiguously. This implies that when a new subtrie is created the array of subtries need to be reallocated and thus rebuild. It is worth to note that Eatherton [Eat99] also uses a bit vector to encode prefixes in a subtrie. Nevertheless, our approach differs from Eatherton approach in several points. First, Eatherton approach uses the bit vector as part of the main multibit data structures. In his scheme the bit vector is used in part for the BMP lookup operation, so its use impacts directly the search operation. The second and more important difference is that our PN bit vector not only encodes existence of prefixes but also their nesting structure. Encoding the nesting structure of prefixes facilitates the updating process in multibit tries.

3.9 Summary

Incremental updates for multibit-trie-based forwarding databases is challenging because it requires to keep track of the prefix transformation process. The main contribution of this chapter is a set of data structures and algorithms to support incremental updates in BMP lookup schemes based on multibit tries. In particular, we have proposed two mechanisms to support incremental updates in multibit-trie-based BMP lookup schemes: the bit-vector-array mechanism and the PN-bit-vector mechanism. While the bit-vector-array mechanism does not need to compute mapping functions, it uses memory inefficiently. The PN-bit-vector mechanism uses memory more efficiently than the bit-vector array mechanism. The PN bit vector mechanism obtains efficient memory usage by using our PN bit vector data structure and computing the corresponding mapping functions. In other words, the PN bit vector mechanism trades computation for efficient memory usage. Moreover, Since the PN bit vector mechanism separates the additional data structure needed to support incremental updates from the main data structure used for doing BMP lookup operations, the performance of the BMP lookup operation is not affected. In our approach we have introduced two key concepts in the framework of incremental updates for multibit tries: the notions of span and coverer.

While the use of multibit tries provides fast BMP lookups, alternative approaches for

doing BMP lookups have been presented by the research community. However, few efforts have been made to provide a framework of reference to compare the different schemes and contrast the tradeoffs of these schemes. In the next chapter, we propose a taxonomy and a framework of reference to analyse and compare the different schemes for doing fast BMP lookups.

Chapter 4

A Framework and a Taxonomy for IP Address Lookup Algorithms

While intensive research has been conducted in the area of IP address lookup, few efforts have been proposed to provide a framework of reference to compare the different schemes and contrast tradeoffs of these schemes. Previous works focuses only on surveying the different methods [TP99][SV99b]. To our knowledge, this is the first time that a taxonomy based on the double dimension of the BMP search is proposed. We propose in this chapter a taxonomy and an effort to use consistent terminology. Moreover, we provide a framework to analyze and compare the different methods to perform fast address lookups when the address lookup operation needs a search for the longest matching prefix.

In chapter 3 we focused on the approach based on the multibit trie data structure and we proposed two mechanisms to support incremental updates. In this chapter we place this approach in the general context of IP address lookup algorithms.

4.1 A Taxonomy of Address Lookup Algorithms

Remember from chapter 2 that, with CIDR, the address lookup operation requires a search for the best matching prefix (BMP). In this section, we propose a taxonomy of the main algorithms to provide fast address lookups; but before and to motivate our taxonomy, we describe the naive algorithm to search for the BMP.

4.1.1 The naive algorithm for the BMP lookup

The simplest method to find the longest matching prefix is a sequential search of all the prefixes. The data structure needed is just an array with unordered prefixes. Each array entry contains the bit string and the length of the prefix. The search algorithm is very simple. It goes through all the entries comparing the prefix with the corresponding bits of a given address. When a match is found, we keep the longest match so far and continue. At the end, the last prefix remembered is the BMP. The problem with this approach is that the search space is reduced only by one prefix at each step. Clearly the search complexity in time for this scheme is a function of the number of prefixes $O(N)$, and hence the scheme is not scalable. With this approach, we do exhaustive search; that is, the search terminates only when all the prefixes has been checked. The only advantage of this approach is that it uses memory efficiently; that is, no extra storage requirements are needed except for the prefixes themselves. Since prefixes are not ordered, insertion is straightforward but deletion needs a search operation.

In the next section we present methods to do more efficient search of the BMP.

4.1.2 Optimized methods

Our taxonomy is based on the observation that the difficulty of the best matching prefix search resides in its double dimension: value and length. Determining the best matching prefix involves not only comparing the bit pattern itself (finding a match), but also finding the appropriate length (the longest one). Our taxonomy classifies the address lookup schemes according to these two dimensions and also if a linear or a binary search is performed. As a result, we have 4 cases:

- Linear search based on lengths
- Binary search based on lengths
- Linear search based on values
- Binary search based on values

We explain each of these categories in the next sections. While the most important aspect is the search operation, it should be taken into account also that the forwarding database is dynamic; that is, it is subject to frequent insertions and deletions. Hence, we analyse the update aspect when we explain the different approaches.

4.2 BMP search based on lengths

In this case the main dimension to search is the length of the prefix. Thus, we organize the prefixes by length, and for a given length, we search for an exact match.

4.2.1 Linear search based on lengths using hash tables

To organize the prefixes by length, we can use different tables for every different length. In each table, we can search for a match by using hashing. Since we need to find the longest matching prefix, the simplest approach is to perform sequential search from the longest length to the shortest one. If we assume a perfect hashing function then the complexity of the search time is $O(W)$; where W is the maximum possible prefix length. Clearly, insertions and deletions are straightforward. Storage requirements are minimal.

4.2.2 Linear search based on lengths using multibit tries

4.2.2.1 The classical binary tries

Another way to organize prefixes by length is by using tries. In fact, as we have seen in chapter 3, a trie organizes prefixes by using their bits to direct the branching. As a result, all the prefixes of the same length are located at the same level of the trie. Hence, when we follow a search path in a trie, we can check at each step *i* the prefixes of length *i*. Moreover, by the way a trie organizes its data, we can check for a match at each length (i.e., level) with a constant time cost (the memory access time). Hence, with tries, we can perform linear search on lengths with cost $O(W)$, without the need to assume a perfect hashing function. The simplest trie, i.e., the binary trie, supports arbitrary prefix lengths, and allows for straightforward insertions and deletions.

While binary tries allow the representation of arbitrary length prefixes, they have the characteristic that long sequences of one-child nodes may exist (see prefix *b* in figure 4.1). Since these bits need to be inspected, even though no actual branching decision is made, search time can be longer than necessary in some cases. Also, one-child nodes consume additional memory. In an attempt to improve time and space performance, a technique called **path-compression** can be used. Path-compression consists of collapsing one-way branch nodes. When one-way branch nodes are removed from a trie, additional information must be kept in remaining nodes so that the search operation can be performed correctly.

There are many ways to exploit the path-compression technique; perhaps the simplest to explain is illustrated in figure 4.2, corresponding to the binary trie in figure 4.1. Note that

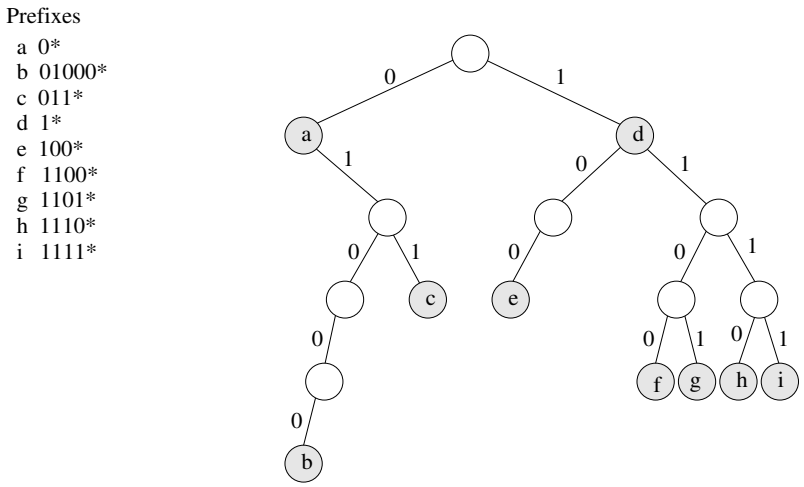

Figure 4.1: Binary trie for a set of prefixes.

the two nodes preceding *b* now have been removed. Note also that since prefix *a* was located at a one-child node, it has been moved to the nearest descendant not being a one-child node. Since in a path to be compressed several one-child nodes may contain prefixes, in general, a list of prefixes must be maintained in some of the nodes. Because one-way branch nodes are now removed, we can jump directly to the bit where a significant decision is to be made, bypassing the bit inspection of some bits. As a result, a bit number field must be kept now to indicate which bit is the next bit to inspect. In figure 4.2 these bit numbers are shown next to the nodes. Moreover, the bit strings of prefixes must be explicitly stored. A search in this kind of path-compressed tries is as follows: The algorithm performs, as usual, a descent in the trie under the guidance of the address bits, but this time only inspecting bit positions indicated by the bit-number field in the nodes traversed. When a node marked as prefix is encountered, a comparison with the actual prefix value is performed. This is necessary since during the descent in the trie we may skip some bits. If a match is found, we proceed traversing the trie and keep the prefix as the BMP so far. The search ends when a leaf is encountered or a mismatch found. As usual, the BMP will be the last matching prefix encountered. For instance, if we look for the BMP of an address beginning with the bit pattern *010110* in the path compressed trie shown in figure 4.2, we proceed as follows. We start at the root node and, since its bit number is 1, we inspect the first bit of the address. The first bit is *0,* so we go to the left. Since the node is marked as a prefix, we compare prefix *a* with the corresponding part of the address (*0*). Since they match, we proceed and keep *a* as the BMP so far. Since the node's bit number is 3, we skip the second bit of the address and inspect the third one. This bit is *0*, so we go to the left. Again, we check whether the

prefix *b* matches the corresponding part of the address (*01011*). Since they do not match, the search stops, and the last remembered BMP (prefix *a*) is the correct BMP.

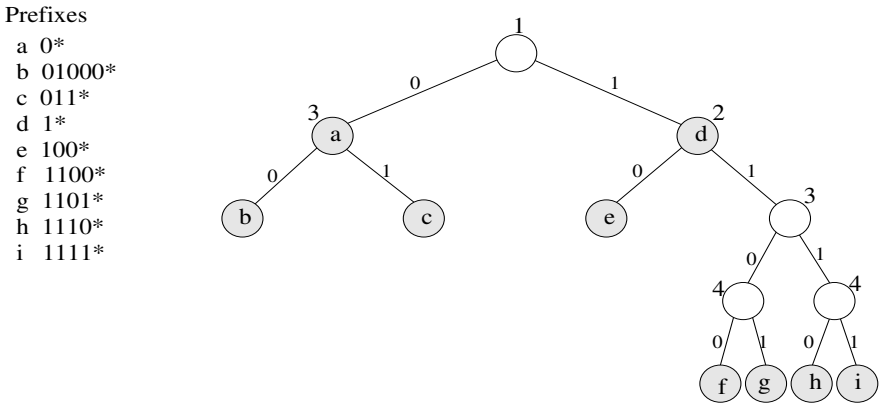

Figure 4.2: A path-compressed trie

Path-compression was first proposed in a scheme called PATRICIA [Mor68], but this scheme does not support longest prefix matching. Sklower proposed a scheme with modifications for longest prefix matching in [Skl91]. In fact, this variant was originally designed to support not only prefixes but also more general noncontiguous masks. Since this feature was really never used, current implementations differ somewhat from Sklower's original scheme. For example, the BSD version of the path-compressed trie (referred to as BSD trie) is essentially the same as that just described. The basic difference is that in the BSD scheme, the trie is first traversed without checking the prefixes at internal nodes. Once at a leaf, the traversed path is backtracked in search of the longest matching prefix. At each node with a prefix or list of prefixes, a comparison is performed to check for a match. The search ends when a match is found. Comparison operations are not made on the downward path in the hope that not many exception prefixes exist. Note that with this scheme, in the worst case the path is completely traversed two times. In the case of Sklower's original scheme, the backtrack phase also needs to do recursive descents of the trie because noncontiguous masks are allowed.

4.2.2.2 Multibit tries

We have seen in chapter 3 that we can improve the search performance by using multibit tries. While multibit tries improve search performance, insertion and deletion are not straightforward because prefixes need a transformation. We have seen in chapter 3 how to provide efficient insertions and deletions while not degrading the search performance.

Linear search of hashing tables provides search, insertion and deletion with similar cost. By organizing prefixes with binary tries, the storage requirement increases but we do not need a perfect hashing function to provide the same $O(W)$ time cost for search, insertion and deletion operations. While multibit tries need increased memory requirements, they allow for a tuning tradeoff between the search time cost and the insertion/deletion time cost. This tradeoff is tuned by choosing the strides and levels of the multibit trie. Multibit tries are usually implemented by using one array per subtrie. Hence, search in a multibit trie can be performed by successively indexing the arrays with the corresponding bits of the packet's destination address.

Choosing the strides requires a trade-off between search speed and memory consumption. In the extreme case, we could make a trie with a single level (i.e., a one-level multibit trie with a 32-bit stride for IPv4). Search would take in this case just one access, but we would need a huge amount of memory to store 2^{32} entries.

One natural way to choose strides and control memory consumption is to let the structure of the binary trie determine this choice. For example, if we look at Fig. 4.1, we observe that the subtrie with its root the right child of node d is a full subtrie of two levels (a full binary subtrie is a subtrie where each level has the maximum number of nodes). We can replace this full binary subtrie with a one-level multibit subtrie. The stride of the multibit subtrie is simply the number of levels of the substituted full binary subtrie, two in our example. Figure 4.3 shows the result of this transformation. This transformation is straightforward, but since it is the only transformation we can do in Fig. 4.1, it has a limited benefit. We will see later how to replace, in a controlled way, binary subtries that are not necessarily full subtries. The height of the multibit trie will be reduced while controlling memory consumption. We will also see how optimization techniques can be used to choose the strides.

Multibit tries with the path compression technique

Nilsson et al. [NK99] uses the idea of replacing full binary subtries with multibit subtries. More specifically, the authors recursively transform the binary trie representation of a forwarding database into a multibit trie with variable strides. Starting at the root, they replace the largest full binary subtrie with a corresponding one-level multibit subtrie. This process is repeated recursively with the children of the multibit subtrie obtained. Additionally, one-child paths are compressed. Since we replace at each step a binary subtrie of several levels with a multibit trie of one level, the process can be viewed as a compression of the levels of the original binary trie. Level-compressed (LC) is the name given by Nilsson to these multibit tries. Figure 4.4 shows the resulting multibit trie (i.e., the LC trie) for the binary

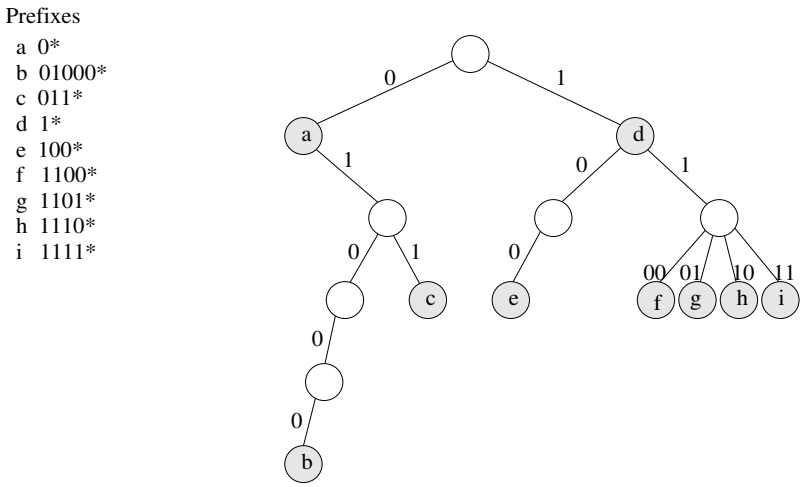

Figure 4.3: Replacing a full binary subtrie with a multibit subtrie.

trie in figure 4.1. Note that prefixes *a* and *d* are not shown in the multibit trie, this is because in a LC trie the internal nodes cannot contain prefixes. Instead, each leaf has a linear list with prefixes; the prefixes that should be in the path to this leaf (i.e., less specific prefixes). As a result, a search in an LC trie proceeds as follows. The LC trie is traversed as in a basic multibit trie. Nevertheless, since path compression is used, an explicit comparison must be performed when arriving at a leaf. In case of mismatch, a search of the list of prefixes must be performed (less specific prefixes, i.e., prefixes in internal nodes in the original binary trie). To achieve a space efficient implementation of the multibit trie, Nilsson stores all the nodes of the LC trie in a single array: first the root, then all the nodes at the second level, then all the nodes at the third level, and so on.

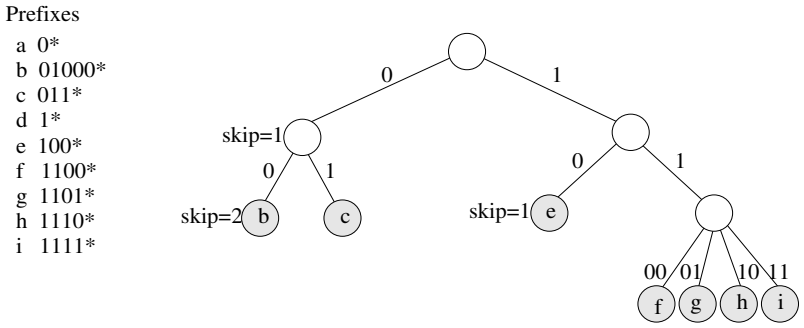

Figure 4.4: The LC multibit trie for the last binary trie. Since the LC trie cannot store prefixes at internal nodes, prefixes a and d are maintained in an additional data structure, which is reached by pointers in the leaves.

While letting the structure of the binary trie strictly determine the choice of strides is a straightforward and simple strategy, this strategy does not allow for control of the height of the resulting multibit trie. One way to further reduce the height of the multibit trie is to let the structure of the trie only guide, not determine, the choice of strides. In other words, we will replace nearly full binary subtries (i.e., binary subtries where only few nodes are missing) with a multibit subtrie. Nilsson proposes replacing a nearly full binary subtrie with a multibit subtrie of stride k if the nearly full binary subtrie has a sufficient fraction of the 2^k nodes at level *k*, where a sufficient fraction of nodes is defined using a single parameter called fill factor x, with $0 < x \leq 1$. For instance, in Fig. 4.1, if the fill factor is 0.5, the fraction of nodes at the fourth level is not enough to choose a stride of 4, since only 5 of the 16 possible nodes are present. Instead, there are enough nodes at the third level (5 of the 8 possible nodes) for a multibit subtrie of stride 3.

Although this optimization allows for a better control of the height of the multibit trie, this strategy does not allow to control the worst case lookup time. Moreover, since the LC trie is implemented using a single array of consecutive memory locations and a list of prefixes must be maintained at leaves, incremental updates are very difficult. Updates will need that almost every array entry to be moved to make space for new nodes. Also, since location of the children of a node is coded with array offsets, insertions or deletions cause the need to update these array offsets.

The additional table maintaining the information about the prefixes that should be at internal nodes is also implemented as a single array using array offsets to relate its elements. As a result, updates will need also array entries to be moved and array offsets to be updated.

Multibit tries and optimization techniques

One easy way to bound worst-case search times is to define fixed strides that yield a well-defined height for the multibit trie. The problem is that, in general, memory consumption will be large, as seen earlier. On the other hand, we can minimize the memory consumption by letting the prefix distribution strictly determine the choice of strides. Unfortunately, the height of the resulting multibit trie cannot be controlled and depends exclusively on the specific prefix distribution. We saw in the previous section that Nilsson uses the fill factor as a parameter to control the influence of the prefix distribution on stride choice, and so influences somewhat the height of the resulting multibit trie. Since prefix distribution still guides stride choice, memory consumption is still controlled. Nevertheless, the use of the fill factor is simply a reasonable heuristic and, more important, does not allow a guarantee on worst-case height. Srinivasan et al. [SV98] use dynamic programming to determine,

for a given prefix distribution, the optimal strides that minimize memory consumption and guarantee a worst-case number of memory accesses. The authors give a method to find the optimal strides for the two types of multibit tries: fixed stride and variable stride. Another way to minimize lookup time is to take into account, on one hand, the hierarchical structure of the memory in a system and, on the other, the probability distribution of the usage of prefixes (which is traffic-dependent). Cheung et al. [CM99] give methods to minimize the average lookup time per prefix for this case. They suppose a system having three types of hierarchical memories with different access times and sizes. Using optimization techniques makes sense if the entries of the forwarding table do not change at all or change very little, but this is rarely the case for backbone routers. Inserting and deleting prefixes degrades the improvement due to optimization, and rebuilding the structure may be necessary.

Multibit tries and compression

Multibit tries are usually implemented by using arrays. While the use of arrays allows for fast indexed accesses in each subtrie, memory requirements are increased. To save some memory, a number of approaches uses compression techniques when implementing the multibit tries. Two general techniques are used to save memory. The first technique comes from the observation that the entries in the array stores two pointers, one for the current BMP and one for the next possible subtrie (see section 3.5.1). The idea is to use a prefix transformation in such a way that we can have a multibit trie where entries in the arrays have only one pointer: a pointer to a next subtrie or a pointer to the current BMP, but not both. This prefix transformation exists. All we need to do is to transform the set of prefixes into an equivalent set of disjoint prefixes; that is, a set of non nested prefixes. But how many new prefixes can be generated? To transform a multibit trie with nested prefixes into a multibit trie with disjoint prefixes, we proceed, conceptually, as follows: We begin at the subtrie in the first level. For every array entry having a non null pointer to a next subtrie, we suppress the current BMP pointer, and we remember this BMP. Then, we follow the next subtrie pointer to arrive at a subtrie in the second level. In this subtrie, for all the array entries having a null BMP pointer, we change this null pointer with a pointer to the BMP just remembered in the last subtrie. In other words, we expand in this subtrie the remembered BMP. Once this expansion has been done, we recursively repeat the process; that is, in this subtrie, we repeat the process for every array entry having a non null pointer to a next subtrie. At the end of this process, all the subtries will have only one pointer: a pointer to a next subtrie or a pointer to a BMP, but not both. In fact, only leaves will have a pointer to BMP, while only internal nodes will store a pointer to a next subtrie. Figure 4.6 shows

the multibit trie with disjoint prefixes resulting from the multibit trie in figure 4.5.

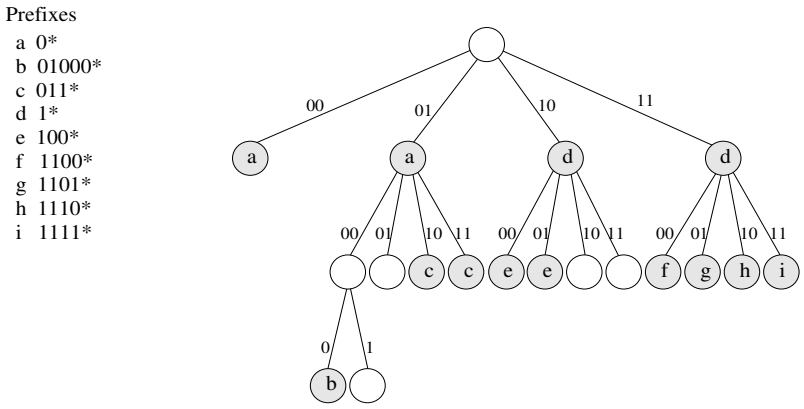

Figure 4.5: A multibit trie.

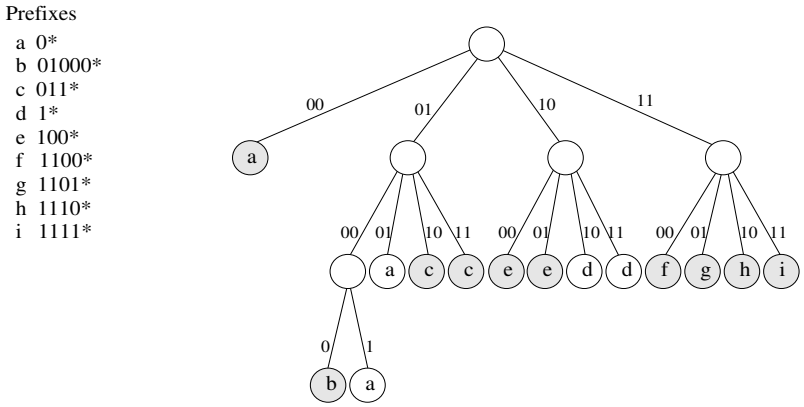

Figure 4.6: A multibit trie with disjoint prefixes.

The second technique to save memory consists in using a bit vector to represent with only one bit each entry of the array. The bits in the bit vector are set by using the Run-length encoding technique [Sto88] as follows: We scan the entries in the original array, each time that a different pointer is found, the bit corresponding to this entry is set. On the contrary, if the pointer is the same than that of the previous entry, then the corresponding bit is cleared. Once the bit vector is built, we can store only the different pointers in the array in a contiguous block, that is, in a smaller compressed array. Now, indexing the subtrie consists in two steps: In the first step, we index the bit vector to locate the target bit position, then we count the number of bits set from the beginning of the bit vector until the target bit position. In the second step, this count number is used to index the compressed array. Figure 4.8 shows the compressed multibit trie of figure 4.7.

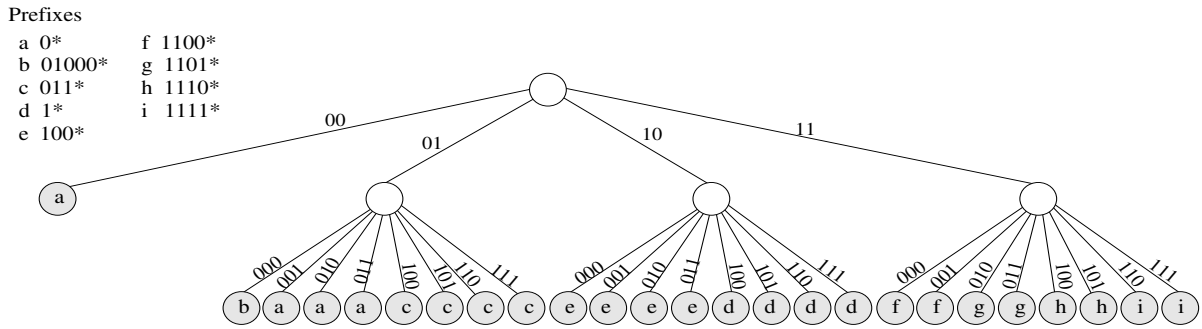

Figure 4.7: A multibit trie with disjoint prefixes.

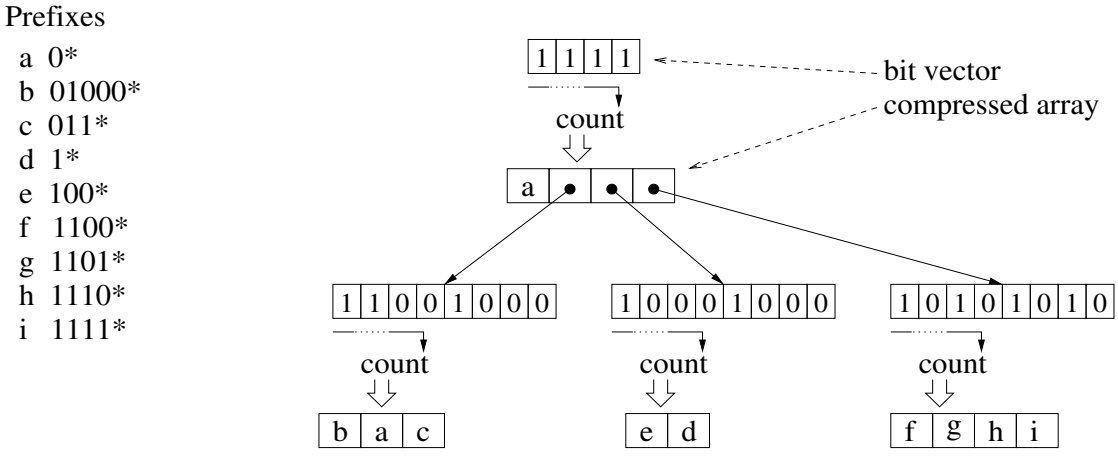

Figure 4.8: The compressed multibit trie corresponding to the multibit trie of figure 4.7.

The Lulea scheme [DBCP97] uses these two techniques to reduce memory consumption. Furthermore, to count efficiently the number of set bits, the Lulea scheme partitions the bit vector in blocks, then precomputes and stores the count of set bits in each block. That is, the count is done in several stages, with precomputed values.

Another example of compressed multibit tries is the Full expansion/Compression scheme proposed by Crescenzi et al. [CDG99]. In this approach both techniques described above are also used, but with some modifications. The scheme uses a multibit trie of two levels; and conceptually, it expands prefixes to the maximum length. We will illustrate their method with a small example where we do a maximal expansion supposing 5-bit addresses and use a two-level multibit trie. The first level uses a stride of 2 bits and the second level a stride of 3 bits, as shown in figure 4.9. The idea is to compress each of the subtries at the second level. In figure 4.10 we can see how the leaves of each subtrie at second level have been placed vertically. Each column corresponds to one of the second-level subtries. The goal is to compress the repeated occurrences of the BMPs. Nevertheless, the compression is done in such a way that at each step the number of compressed symbols is the same for each column. With this strategy the compression is not optimal for all columns, but since the compression is made in a synchronized way for all the columns, accessing any of the compressed subtries can be made with one common additional table of pointers, as shown in figure 4.10. To find the BMP of a given address, we traverse the first level of the multibit trie as usual; that is, the first two bits of the address are used to choose the correct subtrie at the second level. Then the last three bits of the address are used to find the pointer in the additional table. With this pointer we can readily find the BMP in the compressed subtrie. For example, searching for the address 10110 will guide us to the third subtrie (column) in the compressed structure; and using the pointer contained in the entry 110 of the additional table, we will find *d* as the best matching prefix.

In the actual scheme proposed by Crescenzi, prefixes are expanded to 32 bits. A multibit trie of two levels is also used, but the stride of the first and second levels is 16 bits. It is worth noting that even though compression is done, the resulting structure is not small enough to fit in the cache memory. Nevertheless, because of the way to access the information, search always takes only three memory accesses. The reported memory size for a typical backbone forwarding table is 1.2 Mbytes.

While compression saves memory requirements, update cost is very high and in general these approaches do not allow for incremental updates. Obtaining disjoint prefixes can be very costly. Insertion of a new prefix in a compressed array involves shifting already inserted prefixes to make space for the new one. When obtaining disjoint prefixes, the disaggregation

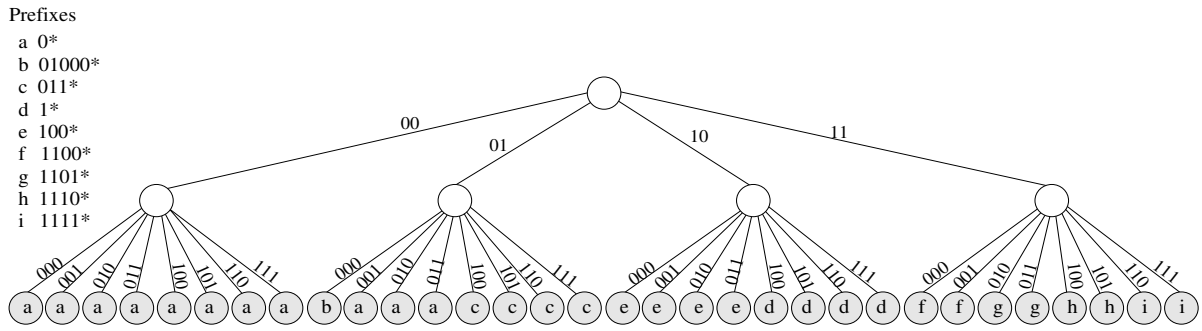

Figure 4.9: A two level full expanded multibit trie

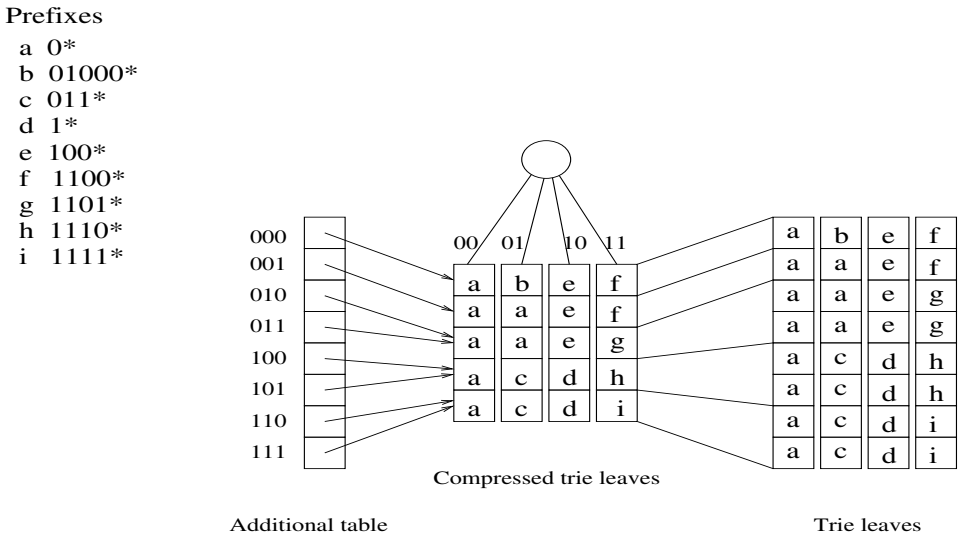

Figure 4.10: Full expansion parallel compression scheme

cannot be controlled.

4.2.3 Binary search based on lengths

The problem with arbitrary prefix lengths is that we do not know how many bits of the destination address should be taken into account when compared with the prefix values. Tries allow a sequential search on the length dimension: first we look in the set of prefixes of length 1, then in the set of length 2 prefixes, and so on. Moreover, at each step the search space is reduced because of the prefix organization in the trie. We have seen that another approach to do sequential search on lengths without using a trie is organizing the prefixes in different tables according to their lengths. In this case, a hashing technique can be used to search in each of these tables. Since we look for the longest match, we begin the search in the table holding the longest prefixes; the search ends as soon as a match is found in one of these tables. Nevertheless, the number of tables equals the number of different prefix lengths. If W is the address length (32 for IPv4), the time complexity of the search operation is $O(W)$ assuming a perfect hash function, which is the same as for a trie.

In order to reduce the search time, a binary search on lengths was proposed by Waldvogel et al. [WVTP97]. In a binary search, we reduce the search space in each step by half. On which half to continue the search depends on the result of a comparison. However, an ordering relation needs to be established before being able to make comparisons and proceed to search in a direction according to the result. Comparisons are usually done using key values, but our problem is different since we do binary search on lengths. We are restricted to checking whether a match exists at a given length. Using a match to decide what to do next is possible: if a match is found, we can reduce the search space to only longer lengths. Unfortunately, if no match is found, we cannot be sure that the search should proceed in the direction of shorter lengths, because the BMP could be of longer length as well. Waldvogel et al. insert extra prefixes of adequate length, called markers, to be sure that, when no match is found, the search must proceed necessarily in the direction of shorter prefixes.

To illustrate this approach consider the prefixes shown in Fig. 4.11. In the trie we can observe the levels at which the prefixes are located. At the right, a binary search tree shows the levels or lengths that are searched at each step of the binary search on lengths algorithm. Note that the trie is only shown to understand the relationship between markers and prefixes, but the algorithm does not use a trie data structure. Instead, for each level in the trie, a hash table is used to store the prefixes. For example, if we search for the BMP of the address 11000010, we begin by searching the table corresponding to length 4; a match will be found

because of prefix *f*, and the search proceeds in the half of longer prefixes. Then we search at length 6, where the marker 110000* has been placed. Since a match is found, the search proceeds to length 7 and finds prefix *k* as the BMP. Note that without the marker at level 6, the search procedure would fail to find prefix *k* as the BMP. In general, for each prefix entry a series of markers are needed to guide the search. Since a binary search only checks a maximum of $\log_2 W$ levels, each entry will generate a maximum of $\log_2 W$ markers. In fact, the number of markers required will be much smaller for two reasons: no marker will be inserted if the corresponding prefix entry already exists (prefix *f* in Fig. 4.11), and a single marker can be used to guide the search for several prefixes (e.g., prefixes *e* and *p*, which use the same marker at level 2). However, for the very same reasons, the search may be directed toward longer prefixes, although no longer prefix will match. For example, suppose we search for the BMP for address 11000001. We begin at level 4 and find a match with prefix *f*, so we proceed to length 6, where we find again a match with the marker, so we proceed to level 7. However, at level 7 no match will be found because the marker has guided us in a bad direction. While markers provide valid hints in some cases, they can mislead in others. To avoid backtracking when being misled, Waldvogel uses precomputation of the BMP for each marker. In our example, the marker at level 6 will have *f* as the precomputed BMP. Thus, as we search, we keep track of the precomputed BMP so far, and then in case of failure we always have the last BMP. The markers and precomputed BMP values increase the memory required. Additionally, the update operations become difficult because of the several different values that must be updated.

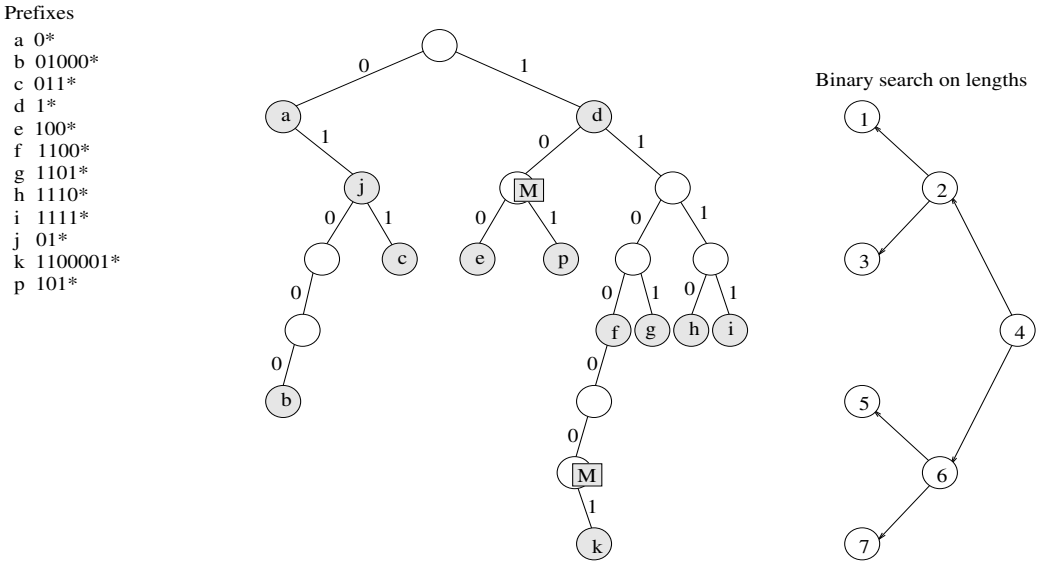

Figure 4.11: Binary search on prefix lengths

4.3 BMP search based on values

The previous section presented BMP search methods that use the length of prefixes as the main search dimension. In this section we present BMP search methods that get rid of the prefix length dimension. The idea is to transform the prefixes of the forwarding database in such a way that traditional search methods based on value comparisons can be applied. To base the search on values only, we need to get rid of the length dimension in some way. One way of doing this is by changing the prefix representation (value/length) to an explicit interval representation. In this way, instead of using value/length we use two values the first (lowest) value of the interval and the last (highest) value of the interval. Note that with this representation we get rid of the length dimension because all the elements have the same length; that is, the maximum prefix length, which is also the length of the IP addresses. Now we can use search methods based on comparison of values exclusively. Now the problem is transformed into finding any of the two endpoints of the appropriate prefix interval, for these endpoints have the associated forwarding information of the corresponding prefix. One natural way to search for one of the two appropriate endpoints is to search for the predecessor of a given address. The predecessor of a given address is defined as the greatest endpoint smaller than or equal to the given address. When we search for the predecessor of a given address, what we are trying to find is, in fact, the left endpoint of the prefix interval that contains the given address. Unfortunately, this approach does not work because the prefix intervals can be included in other prefix intervals, i.e., prefix intervals can be nested. Since prefix intervals can be nested, the predecessor of a given address is not always the endpoint of the prefix interval that contains this address. For example, figure 4.12 shows the full expansion of prefixes assuming 5-bit-length addresses. The same figure shows the endpoints of the different prefix intervals, in binary as well as decimal form. There we can see that the predecessor of the address 21 is the endpoint 19; nevertheless, the correct BMP for address 21 is not the one associated with endpoint 19 (i.e., *e*), but *d* instead. Since the idea of the search of the predecessor works only for non nested intervals, one solution is to transform the original prefix intervals into non nested intervals, i.e., disjoint intervals. Let's see with an example how this transformation can be achieved. In figure 4.12, we can see that the interval corresponding to the prefix *a* can be partitioned into 4 subintervals. Two of them are in fact the intervals of prefixes *b* and *c*; and only the other two subintervals: [0,7] and [9,11] have *a* as their closest covering prefix. While the search for the predecessor of addresses in the interval [0,7] gives the correct result, this is not the case for addresses in the interval [9,11]. The problem is that there is no explicit left endpoint for the subinterval [9,11]. Obviously,

the solution is to store a new left endpoint for this subinterval, and associate this new left point (i.e., 9) with the forwarding information of prefix *a*. What we have just done is, in fact, transforming the set of nested intervals of prefixes *a*, *b*, and *c* into an equivalent set of disjoint intervals; that is, a set of disjoint intervals with the same forwarding information. We will refer to the disjoint intervals resulting of this transformation as the basic intervals. In our example, to obtain disjoint intervals, the interval of prefix *a* was transformed into two basic intervals. In general, to obtain disjoint intervals, the interval of a prefix covering m prefixes will be transformed into a maximum of m+1 basic intervals. Figure 4.12 shows the basic intervals for an example of a set of prefixes. Note that for the algorithm to work properly, each basic interval must be associated with its BMP, which is simply the closest prefix covering this basic interval. In summary, the transformation consists in two stages. In the first stage, the basic intervals are determined by using the endpoints of the prefix intervals; and adding extra left endpoints for the basic intervals without it. In the second stage, each left endpoint of the basic intervals is associated with its BMP, which is its closest covering prefix. Now, we can use classical search methods based on value comparisons to find the correct predecessor of a given address and so obtain its correct BMP. Note that once the basic intervals are determined and the extra left endpoints are added, we do no longer need to maintain the right endpoints because they are redundant. Note also that an equivalent approach can be based on the use of right endpoints, instead of left endpoints. In this case the search for the BMP is achieved by searching for the successor of a given address; where the successor of a given address is defined as the smallest endpoint greater than or equal to the given address.

4.3.1 Linear search based on values

The simplest method to search for the predecessor of a given address is to sequentially scan an array containing the left endpoints of the basic intervals. This linear search method has time cost $O(N)$, where N is the number of prefixes of the forwarding database. Obviously, a binary search method is possible. We describe in the next section some variants for doing binary search based on values

4.3.2 Binary search based on values

Different implementations of this idea can vary in how the endpoints are managed. In the general scheme described above, once the basic intervals are determined, we need to maintain, in fact, only their left endpoints (the original and the new ones) because the right

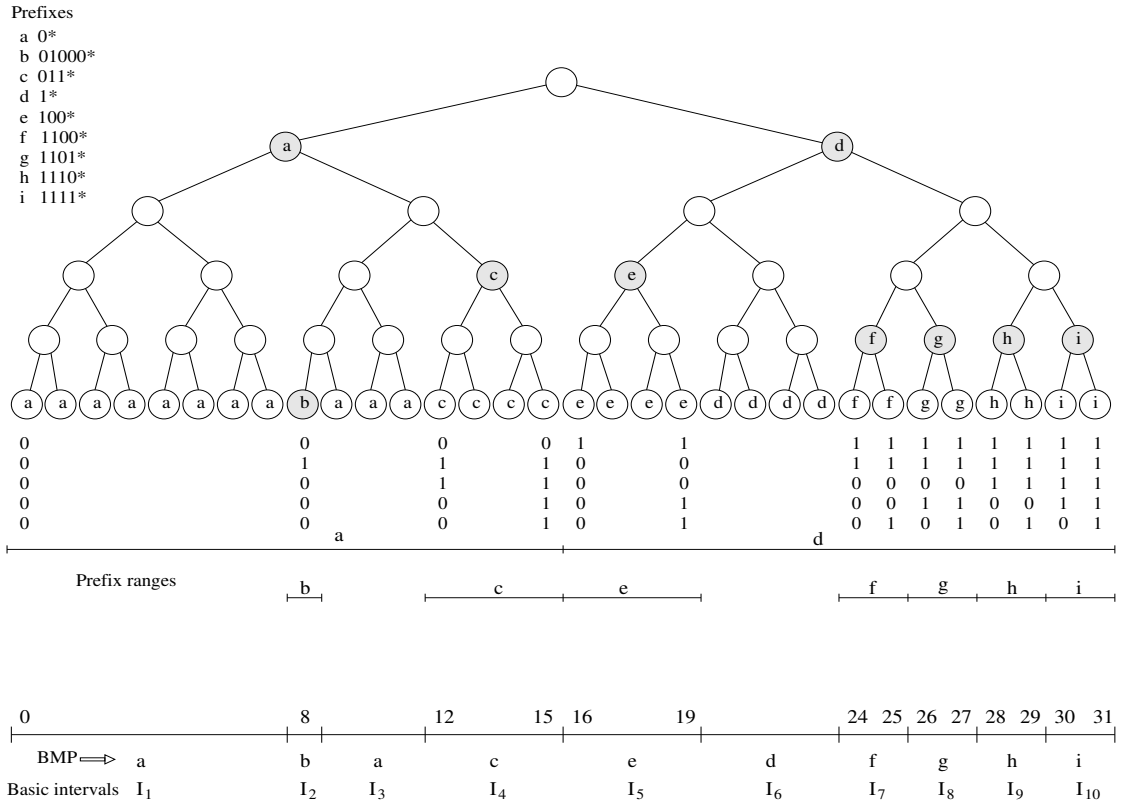

Figure 4.12: An example of Prefix Intervals

endpoints are redundant. Another variant is the scheme of Lampson et al. [LSV98]. In this scheme, the authors do not store explicitly the left endpoints of the new intervals resulting of the transformation process; instead, both left and right original endpoints maintain two BMPs, one for the interval they belong to and one for the potential next basic interval. For example, in figure 4.12, endpoint 19 will maintain two BMPs, one for basic interval I5 and one for I6. Figure 4.13 shows the search tree indicating the steps of the binary search algorithm. The leaves correspond to the endpoints, which store the two BMPs (= and >). For example, if we search the BMP for address 10110 (22), we begin comparing the address with key 26; since 22 is smaller than 26, we take the left branch in the search tree. Then we compare 22 with key 16 and go to the right; then at node 24 we go to the left arriving at node 19; and finally, we go to the right and arrive at the leaf with key 19. Because the address (22) is greater than 19, the BMP is the value associated with > (i.e., *d*). As for traditional binary search, the implementation of this scheme can be made by explicitly building the binary search tree. Moreover, instead of a binary search tree, a multiway search tree can be used to reduce the height of the tree and thus make the search faster. The idea is similar to the use of multibit tries instead of binary tries. In a multiway search tree, internal nodes have k branches and k-1 keys; this is especially attractive if an entire node fits into a single cache line because search in the node will be negligible compared to normal memory accesses.

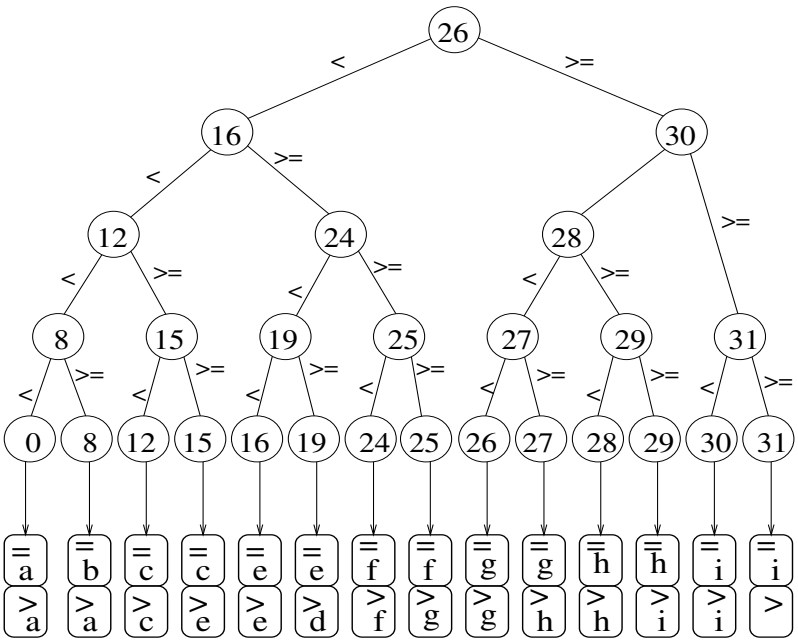

Figure 4.13: A basic interval search tree.

As previously mentioned, the transformation process consists of two stages. First, the

basic intervals are determined; and second, each basic interval is associated with its BMP, which is its closest covering prefix. While inserting a new prefix can generate in the worst case only 2 new endpoints, it may need to change the BMP of N basic intervals, in the worst case. This is because a new prefix when inserted can become the new closest covering prefix of many basic intervals. Similarly, when a prefix P is deleted, all the basic intervals whose closest covering prefix is P will need to update their BMP with a new closest covering prefix. In the worst case we would need to update the BMP for N basic intervals, N as usual being the number of prefixes. This is the case when all the $2N$ endpoints are all different and one prefix contains all the other prefixes.

Remember that a prefix interval containing m prefix intervals can be transformed into up to m+1 new disjoint subintervals. Similarly, when an existing prefix is deleted, the BMP of N basic intervals may need to change in the worst case. Hence, the problem with this approach is that inserting or deleting a single prefix may require recomputing the BMP for many basic intervals. In general, every prefix interval spans several basic intervals. The more basic intervals a prefix interval covers, the higher the number of BMPs to potentially recompute. In fact, in the worst case we would need to update the BMP for N basic intervals, N as usual being the number of prefixes. This is the case when all 2N endpoints are different and one prefix contains all the other prefixes.

Thus, to support incremental updates we need to find a way to aggregate the basic intervals in such a way that when they need to be updated we only need to update aggregates of basic intervals and not each individual basic interval. To provide incremental updates, we need to control the transformation process. Remember that to obtain disjoint intervals, a prefix interval can be transformed into N basic intervals, in the worst case. We need a way to organize hierarchically the new derived subintervals. That is, we need to maintain an additional structure to be be able to update by modifying aggregates of subintervals and not each of the basic subintervals individually.

In Fig. 4.13 we can observe that the leaves of the tree correspond to basic intervals. We can observe also that internal nodes correspond to intervals that are the union of basic intervals (Fig. 4.14). Also, all the nodes at a given level form a set of disjoint intervals. For example, at the second level the nodes marked 12, 24, and 28 correspond to the intervals [0,15], [16,25], and [26,29], respectively. So why store BMPs only at leaves? For instance, if we store *a* at the node marked 12 in the second level, we will not need to store *a* at leaves, and update performance will be better. In other words, instead of decomposing prefix intervals into basic intervals, we decompose prefix intervals into disjoint subintervals

as large as possible. Figure 4.14 shows how prefixes can be stored using this idea[1]. Search operation is almost the same, except that now it needs to keep track of the BMP encountered when traversing the path to the leaves. We can compare the basic scheme to using leaf pushing and the new method to not doing so. Again, we can see that pushing information to leaves makes update difficult, because the number of entries to modify grows. The multiway range tree approach [SVW01] presents and develops this idea to allow incremental updates.

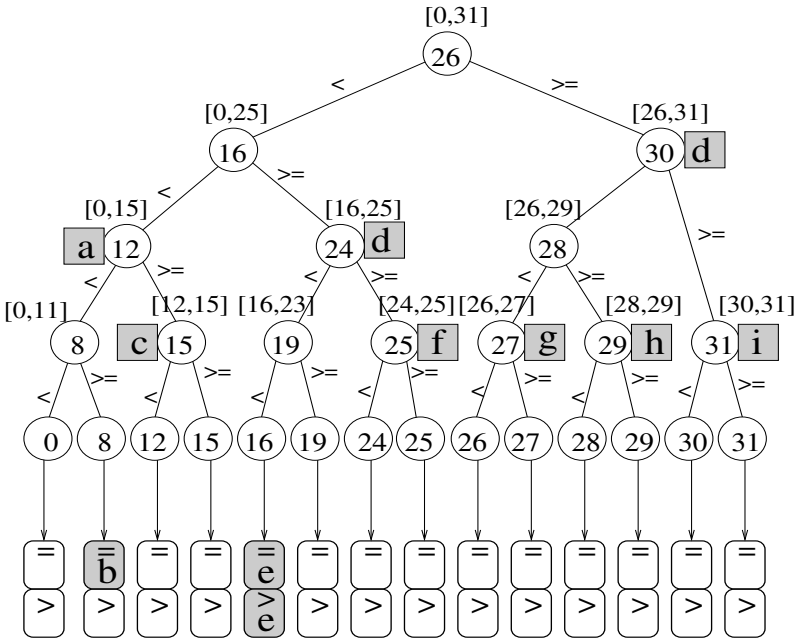

Figure 4.14: A range search tree.

Another way to aggregate the basic intervals is by using the nesting structure of prefixes itself. What changes essentially when a prefix is inserted or deleted is the nesting relationship between the prefixes and the basic intervals. For example, when a prefix is inserted a number of basic intervals can change their closest covering prefix; that is, the new prefix can become their new closest covering prefix. Similarly, when a prefix is deleted some of the basic intervals can change their closest covering prefix. As a result, insertion or deletion of a prefix can change the BMP of a number of basic intervals.

To better understand this aggregation strategy, let's define the set of basic intervals covered by a prefix P as the *cover* of P. Note that among the basic intervals in the cover of a prefix P, some have P as their closest covering prefix but some others do not. This is because prefixes can be nested and hence, some basic intervals that are in the cover of P can also be

[1]The data structure based on the idea of storing intervals at nodes as high as possible is called the segment tree, in the area of computational geometry [BKOS97].

in the cover of longer prefixes than P. Let's define the set of basic intervals covered directly by a prefix P (that is, not covered by longer prefixes) as the *direct cover* of P. In other words, the direct cover of P is the set of basic intervals whose closest covering prefix is P. We will refer also to the basic intervals in the direct cover of P as the basic intervals seized by P, for these are the basic intervals that have P as their BMP. Intuitively, when a prefix is inserted, the direct cover of some prefixes will change, but which prefixes? Suppose we have a prefix Q and we insert a prefix P such that Q and P are non nested. Clearly the insertion of P does not modify the direct cover of Q because the cover of P and the cover of Q are disjoint. Now suppose that P and Q are nested and P is shorter than Q. Again, the direct cover of Q is not modified because P cannot become a closer covering prefix for the basic intervals in the direct cover of Q (Q is longer than P). In fact, the only case in which the direct cover of Q is modified is when P and Q are nested and P is directly covered by Q (and hence, P is longer than Q). Again, we say that P is directly covered by Q when Q is the closest covering prefix of P. Let's call for convenience the closest covering prefix of a prefix P the *coverer* of P. Thus, when a prefix P is inserted, the only prefix whose direct cover is modified is the prefix which is the coverer of P. More specifically, if Q is the coverer of a new prefix P to be inserted, then the basic intervals that are in both the direct cover of P and the direct cover of Q will be seized by P. This is because these basic intervals are no longer covered directly by Q but by P. As a result, the direct cover of Q will be reduced.

Similarly, when a prefix P is deleted the only prefix whose direct cover is modified is the prefix which is the coverer of P. In the case of deletion of P, the direct cover of Q is increased because the basic intervals that are in both the direct cover of P and the direct cover of Q will be seized by Q.

In the CBST (Collection of Binary Search Trees) approach [SK02], the cover of a prefix is represented by a binary search tree. The prefix itself is represented by the header-node of the tree; and each node in the tree represents either a directly covered basic interval or a directly covered prefix interval. Since each prefix is represented by a different tree, modifying the direct cover of a prefix translates into moving nodes from one tree to another. For example, if Q is the coverer of a prefix P to be inserted, then some of the nodes in the tree of Q will be moved to the new tree of the prefix P. Similarly, when prefix P is deleted, the nodes in the tree of P will be merged into the tree of Q. The advantage of using a tree is that we can move aggregates of nodes and not only individual nodes. Moving aggregates of nodes can be performed by splitting and joining subtrees.

To provide incremental updates, the CBST scheme uses basically two data structures: The first data structure is essentially the classical basic interval tree described for the prece-

dent schemes. The second data structure is a collection of binary search trees; in which each of these binary trees represents the cover of each prefix. Every basic interval is stored at two different places: at a leaf in the basic interval tree, and at a node in one of the trees in the collection of trees. Both are related by a pointer stored in each leaf of the basic interval tree. As a result, the search of the BMP for a given address consists in two steps: First, we search the appropriate basic interval for the given address in the basic interval tree. Once the appropriate basic interval (leaf) is found by using binary search, the pointer in the leaf is followed into the corresponding tree; where the BMP is found by following parent pointers until the header-node of the tree. As described above, in the case of updates, the nodes in the collection of trees are moved from one tree to another and so the BMP of the corresponding basic intervals is automatically updated.

4.4 Transforming Original Prefixes and Incremental Updates

To cope with scalability, a forwarding database aggregates forwarding information by the use of prefixes. While the use of prefixes allows routers to reduce the size of their forwarding tables, a direct implementation of a forwarding table based on the original prefixes does not always allow for fast address lookups. We have seen in previous sections how a number of transformations can be applied to the set of original prefixes to optimize the address lookup operation:

1. Transformation of an original prefix into several longer but equal-length prefixes (prefix expansion).

2. Transformation of the set of original prefixes into a set of disjoint prefixes.

3. Transformation of a set of prefixes into a set of disjoint intervals.

4. For a given prefix, addition of redundant shorter prefixes (markers).

When a prefix is transformed, what we do in fact, is to disaggregate to some extent the original prefixes. As a result, a single prefix can be transformed into several different prefixes or intervals. While this transformation is to make the search operation faster, inserting or deleting a single prefix needs more work because we need to take into account the transformation process. To allow for incremental updates we need to limit in a controlled way the

extent of the disaggregation process and maintain additional data structures to keep track of the disaggregation. In other words we need to do hierarchical disaggregation.

4.5 Comparison and Measurements of Schemes

Each of the schemes presented has its strengths and weaknesses. In this section, we compare the different schemes and discuss the important metrics to evaluate these schemes. The ideal scheme would be one with fast searching, fast dynamic updates, and a small memory requirement. The schemes presented make different tradeoffs between these aspects. The most important metric is obviously the lookup time, but update time must also be taken into account, as well as the memory requirements. Scalability is also another important issue, with respect to both the number and length of prefixes.

4.5.1 Complexity Analysis

The complexity of the different schemes is compared in table 4.1. The next sections carry out detailed comparison.

Scheme	Worst case lookup	Update	Memory
Binary trie	$O(W)$	$O(W)$	$O(NW)$
Path-compressed tries (BSD trie)	$O(W)$	$O(W)$	$O(N)$
k stride Multibit trie	$O(\frac{W}{k})$	$O(\frac{W}{k}+2^k)$	$O(2^k N\frac{W}{k})$
LC trie	$O(\frac{W}{k})$	-	$O(2^k N\frac{W}{k})$
Lulea trie	$O(\frac{W}{k})$	-	$O(2^k N\frac{W}{k})$
Full expansion/compression	3	-	$O(2^k+N^2)$
Binary search on prefix lengths	$O(\log_2 W)$	$O(N\log_2 W)$	$O(N\log_2 W)$
Binary range search	$O(\log_2 N)$	$O(N)$	$O(N)$
Multiway range search	$O(\log_k N)$	$O(N)$	$O(N)$
Multiway range trees	$O(\log_k N)$	$O(k\log_k N)$	$O(Nk\log_k N)$

Table 4.1: Complexity comparison

4.5.1.1 Tries

In binary tries we potentially traverse a number of nodes equal to the length of addresses. Therefore, the search complexity is $O(W)$. Update operations are readily made and basically need a search, so update complexity is also $O(W)$. Since inserting a prefix potentially

creates W successive nodes (along the path that represents the prefix), the memory consumption for a set of N prefixes has complexity $O(NW)$. Note that this upper bound is not tight, since some nodes are, in fact, shared along the prefix paths. Path compression reduces the height of a sparse binary trie, but when the prefix distribution in a trie gets denser, height reduction is less effective. Hence, complexity of search and update operations in path compressed tries, is the same as in classical binary tries. Path-compressed tries are full binary tries. Full binary tries with N leaves have N-1 internal nodes. Hence, space complexity for path-compressed tries is $O(N)$.

Multibit tries still do linear search on lengths, but since the trie is traversed in larger strides the search is faster. If search is done in strides of *k* bits, the complexity of the lookup operation is $O(\frac{W}{k})$. As we have seen, updates require a search and will modify a maximum of 2^{k-1} entries (if leaf pushing is not used). Update complexity is thus $O(\frac{W}{k}+2^k)$ where *k* is the maximum stride size in bits in the multibit trie. Memory consumption increases exponentially with *k*: each prefix entry may need potentially an entire path of length $\frac{W}{k}$, and paths consist of one-level subtries of size 2^k. Hence, memory used has complexity $O(2^k N\frac{W}{k})$.

Since the Lulea and Full expansion/Compression schemes use compressed multibit tries together with leaf pushing, incremental updates are difficult if not impossible, and we have not indicated update complexity for these schemes. The LC trie scheme uses an array layout and must maintain lists of less specific prefixes. Hence, incremental updates are also very difficult.

4.5.1.2 Binary Search on Lengths

For a binary search on lengths, the complexity of the lookup operation is logarithmic in the prefix length. Notice that the lookup operation is independent of the number of entries. Nevertheless, updates are complicated due to the use of markers. As we have seen, in the worst case $\log_2 W$ markers are necessary per prefix; hence, memory consumption has complexity $O(N\log_2 W)$. For the scheme to work, we need to precompute the BMP of every marker. This precomputed BMP is a function of the entries being prefixes of the marker; specifically, the BMP is the longest among them. When one of these prefix entries is deleted or a new one is added, the precomputed BMP may change for many of the markers that are longer than the new (or deleted) prefix entry. Thus, the marker update complexity is $O(N\log_2 W)$ since theoretically an entry may potentially be prefix of N-1 longer entries, each having potentially $\log_2 W$ markers to update.

4.5.1.3 Range Search

The range search approach gets rid of the length dimension of prefixes and performs a search based on the endpoints delimiting disjoint basic intervals of addresses. The number of basic intervals depends on the covering relationship between the prefix ranges, but in the worst case it is equal to $2N$. Since a binary or a multiway search is performed, the complexity of the lookup operation is $O(\log_2 N)$ or $O(\log_k N)$, respectively, where *k* is the number of branches at each node of the search tree. Remember that the BMP must be precomputed for each basic interval, and in the worst case an update needs to recompute the BMP of N basic intervals. The update complexity is thus $O(N)$. Since the range search scheme needs to store the endpoints, the memory requirement has complexity $O(N)$.

We previously mentioned that by using intervals made of unions of the basic intervals, the approach of [SVW01] allows better update performance. In fact, the update complexity is $O(k\log_k N)$, where *k* is the number of branches at each node of the multiway search tree.

4.5.1.4 Scalability and IPv6

An important issue in the Internet is scalability. Two aspects are important: the number of entries and the prefix length. The last aspect is specially important because of the next generation of IP (IPv6), which uses 128-bit addresses. Multibit tries improve lookup speed with respect to binary tries, but only by a constant factor on the length dimension. Hence, multibit tries scale badly to longer addresses. Binary search on lengths has a logarithmic complexity with respect to the prefix length, and its scalability property is very good. The range search approaches have logarithmic lookup complexity with respect to the number of entries but independent, in principle, of the prefix length. Thus, if the number of entries does not grow excessively, the range search approach is scalable for IPv6.

4.5.2 Measured Lookup Time

While the complexity metrics of the different schemes described in the previous section are an important aspect for comparison, it is equally important to measure the performance of these schemes under "real conditions". We now show the results of a performance comparison made using a common platform and a prefix database of a typical backbone router [Tel03].

All programs are coded in C and were executed in the user space under the Linux operating system in a Pentium-III-based computer with a clock speed of 935 MHz. The code

for the path-compressed trie (BSD trie) was extracted from the FreeBSD implementation [WS95][MBKQ96], the code for the Multibit trie was implemented by us (as it was proposed in chapter 3), and the code for the other schemes was obtained from the corresponding authors.

While prefix databases in backbone routers are publicly available, this is not the case for traffic traces. Indeed, traffic statistics depend on the location of the router. Thus, what we have done to measure the performance of the lookup operation is to consider that every prefix has the same probability of being accessed. In other words, the traffic per prefix is supposed to be the same for all prefixes. Although a knowledge of the access probabilities of the forwarding table entries would allow a better evaluation of the average lookup time, assuming constant traffic per prefix still allows us to measure important characteristics, like the worst-case lookup time. In order to reduce the effects of cache locality we used a random permutation of all entries in the forwarding table (extended to 32 bits by adding zeroes). Figure 4.15 and figure 4.16 show the distributions and the cumulative distributions, respectively, of the lookup operation for 5 different schemes. The lookup time variability for the 5 different schemes is summarized in table 4.2, which shows the corresponding percentiles [Jai91].

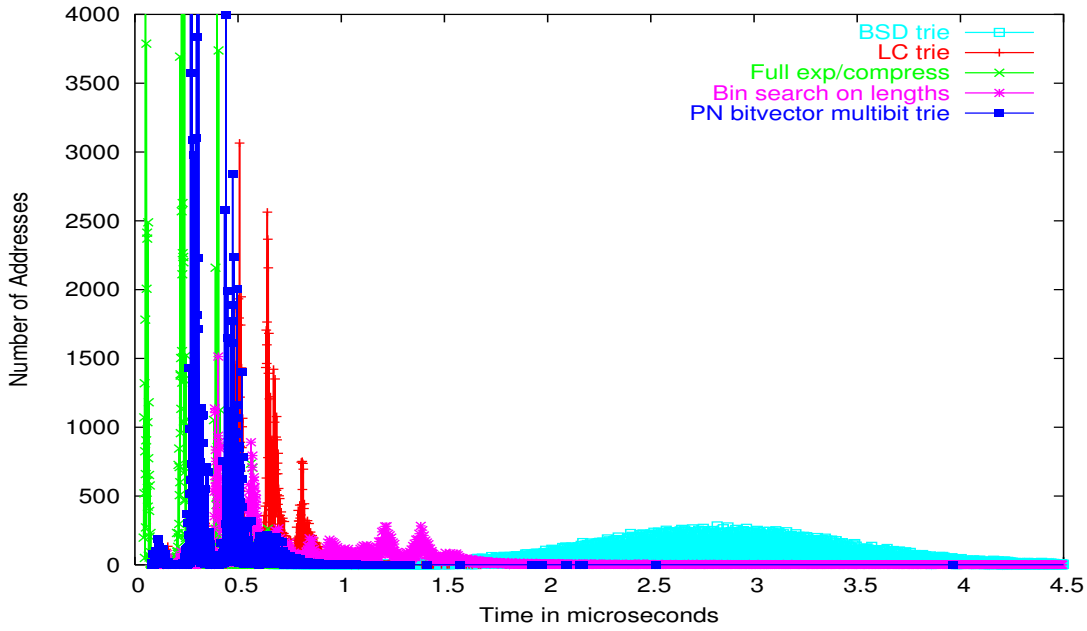

Figure 4.15: Lookup time distributions of several lookup mechanisms

Lookup time measured for the BSD trie scheme reflects the dependence on the prefix length distribution. We can observe a large variance between time for short prefixes and

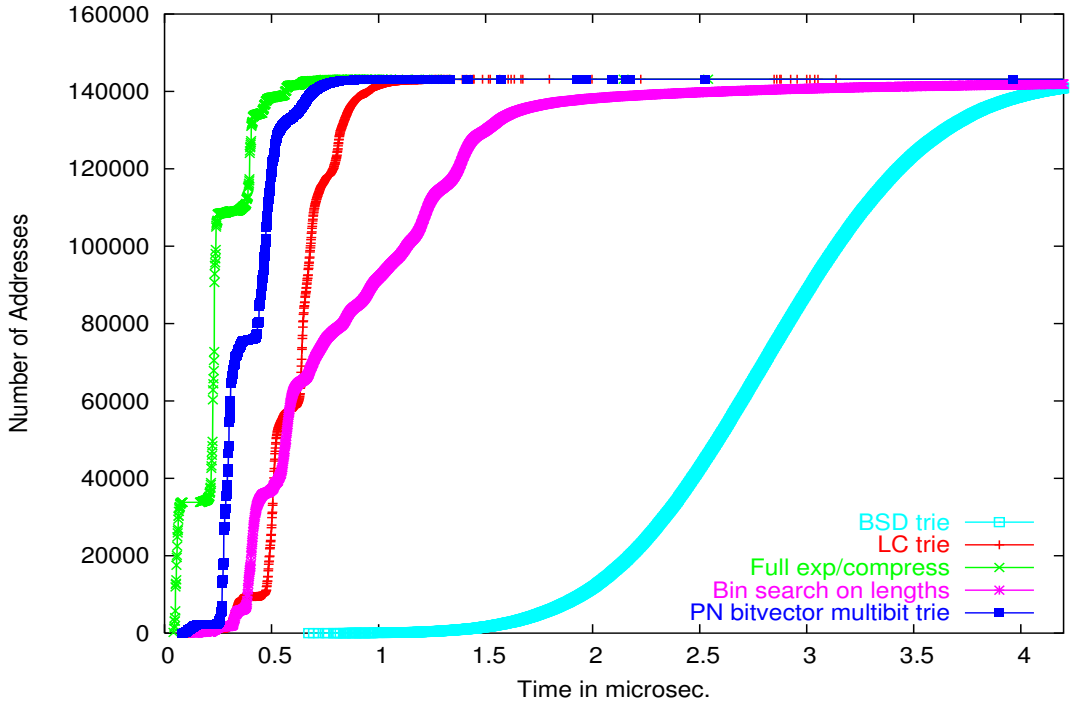

Figure 4.16: Lookup time cumulative distributions of several lookup mechanisms

time for long prefixes because of the high height of the BSD trie. On the contrary, the full expansion/compression scheme always needs exactly 3 memory accesses. This scheme has the best performance for the lookup operation in our experiment. Small variations should be due to cache misses as well as background operating system tasks.

Scheme	10-percentile	50-percentile (Median)	99-percentile
BSD trie	2.06	2.82	4.33
Multibit trie	0.27	0.33	0.75
LC trie	0.48	0.64	1.01
Full expansion/compression	0.06	0.23	0.63
Binary search on prefix lengths	0.39	0.71	4.00

Table 4.2: Percentiles of the lookup times (μseconds)

As we know, lookup times for multibit tries can be tuned by choosing different strides. We have measured the lookup time for the LC trie scheme, which uses an array layout and the path compression technique. The LC trie is a variable-stride multibit trie that uses the distribution of prefixes to guide the choice of strides. Additionally, the fill factor was chosen

such that a stride of k bits is used if at least 50 percent of the total possible nodes at level k exist (see section 4.2.2.2). Even with this simple strategy to build multibit tries, lookup times are much better than for the BSD trie.

We have also measured the lookup time for a Multibit trie implemented with a linked tree structure and without path compression (our PN bit vector multibit trie mechanism proposed in chapter 3). In this emulation we have used the PN bit vector multibit trie with the next fixed strides: 16, 8, 8 at the first,second ant third level respectively (recall that while we have used a fixed stride multibit trie in our emulations, our mechanisms can be used with any multibit trie).

Table 4.3 shows the statistics of the BSD trie and multibit tries, which explains the performance observed. Notice that the average height value of the BSD trie is very high. Hence, a path compression technique used alone, as in the BSD trie, has almost no benefit for a typical backbone router. This table explains also the best performance of the PN bit vector multibit trie with respect to the LC trie. Note that the maximum height of the LC trie is higher that that of the PN bit vector multibit trie. Recall that the higher the trie, the more the number of memory accesses, which slows the BMP lookup operation. Moreover, since the LC trie needs to do extra comparisons in some cases once at the leaves of the trie, these extra operations further degrade the BMP lookup performance.

Scheme	Average height	Maximum height
BSD trie	21.57	32
LC trie	2.28	6
Multibit trie (PN bit vector)	2.24	3

Table 4.3: Trie statistics for the Telstra router (21 March, 2003)

The binary search on lengths scheme also shows a better performance than the BSD trie scheme. However, the lookup time has a large variance. As we can see in figure 4.17, different prefix lengths need a different number of hashing operations. We can distinguish five different groups, which need from one to five hashing operations. Since hashing operations are not basic operations, the difference, between a search that needs five hashes and one that needs only one hash can be significant.

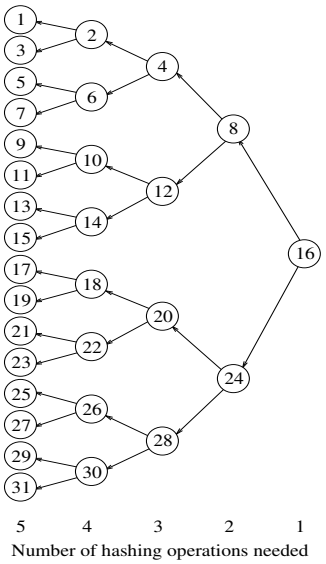

Figure 4.17: Standard binary search on lengths for IPv4

4.6 Summary

This chapter introduced a taxonomy and a framework of reference for fast BMP lookup schemes. Our taxonomy is based on the two dimensional nature of the BMP search operation: value and length. We have highlighted the principles behind the BMP lookup algorithms. We have reviewed approaches for fast BMP lookups, both static and dynamic; that is, whether they provide or not incremental update capabilities. We have seen that to provide fast BMP lookups, the different approaches perform a transformation of the prefix set, i.e., they do controlled disaggregation of the forwarding database. We have seen that to support incremental updates we must find a way to manage in an aggregated way the derived prefixes or intervals resulting of the transformation. While these ideas has been flying around, we have made an effort to treat them in depth and we have used them to establish a framework of reference in this area. The purpose of our taxonomy was two fold. In addition to analysing and surveying the different schemes, our taxonomy identifies and contrasts the different tradeoffs in the address lookup schemes. Another contribution of this chapter is the comparison of the different schemes in terms of their complexity and measured execution time on a common platform. We believe that our taxonomy and framework of reference has greatly increased understanding of the relationships among the existing fast address lookup algorithms and hence we believe that our contribution can help to point researchers in fruitful directions in this area.

While performing a BMP lookup allows routers to decide where to send a packet next,

this task is only a part of the process to achieve the actual relaying of a packet. Once the router has decided through which output port the packet will be forwarded, the router must multiplex all the packets that need to be forwarded through the same output port. Furthermore, routers must resolve possible contention for a given output port because it is possible that several packets from different inputs need to be forwarded through the same output port at the same time. In the next chapter, we concentrate in this aspect of the forwarding process of packet.

Chapter 5

Optimizing the use of buffers for flow isolation

In the previous chapter we have studied how routers search the forwarding information for every incoming packet. This forwarding information allows routers to decide where the packet should be sent next and through which output port the packet will be forwarded. Since a router has in general several input ports, it is possible that several packets need to be sent through the same output port at the same time. As a result, routers need to be able to resolve the possible contention for a given output port. Usually, routers use buffers to address the problem of output-port contention. Buffering allows routers to retain packets while one of the contending packets is transmitted.

However, buffering alleviates the output-port contention only to some extent because, in case of sustained overload, the buffer will eventually overflow. When the buffer overflows, packets are dropped. Hence, the use of buffers is effective only in the case of transient overload. Furthermore, while the use of buffers avoid packet losses to some extent, its use introduces packet delay. While packet delay is in general better than packet loss, some traffic conditions can lead to situations in which the buffer adds unnecessary packet delay. The use of buffers to alleviate the output-port contention is based on the assumption that overload is only transient and not sustained. However, this assumption is not always valid in the Internet. As a result, to make effective use of buffers we need mechanisms to control the traffic. The control of traffic can be made in two places: at end systems or/and at routers. Traditionally, the control of traffic in the Internet has been done by end systems. The sources use algorithms to try to discover the available resources in the network and so adapt their traffic pattern in a dynamic manner. In particular, if congestion occurs the sources should respond by reducing their traffic. The classical algorithm to control traffic of end systems is the

TCP protocol [Jac88][APS99]. Nevertheless, in today's Internet responding to congestion is rather a user's choice and in general there are responsive as well as unresponsive users. As a result, buffers in routers are not always used effectively. When the buffers are not used effectively, the network service is degraded in the form of packet losses and/or packet delay. Furthermore, adaptive sources are penalized because unresponsive sources, intentionally or unintentionally, abuse the cooperative nature of responsive traffic. To address this problem, routers need to provide flow isolation. Providing flow isolation is important because with flow isolation the performance perceived by users does not depend on the good behavior of other users. In this chapter we propose a mechanism to optimise the use of buffers in routers towards providing flow isolation.

5.1 The Functions of Router Buffers

To understand the rationales of our approach we first revise the forwarding process in a router. When a packet arrives at a router, the router examines the packet's destination address to determine the appropriate outgoing link for the packet and then directs the packet to the link.

Since flows share the link capacity on a need basis, packets from different flows may need to be transmitted over the same link at the same time. In order to be able to multiplex different flows, packets are queued at the buffers associated with outgoing links and scheduled for transmission.

Also, since in packet networks there is no blocking mechanism, sources do not have restrictions nor in the traffic amount nor in its traffic pattern. As a result, a source can temporarily send packets at a rate that exceeds output link bandwidth (burst) and so a buffer is also used to absorb sporadic bursts of packets from individual flows.

In summary, load at a router varies dynamically as new flows appear and end, and also as burstiness of each flow varies in time. Hence, the aggregate arrival rate of packets may exceed the output capacity of the link and not all packets can be sent immediately. Routers adapt to transient overloads by using buffers to store packets for later transmission, which otherwise would be lost. In other words, by using buffers we trade delay for delivery capacity.

It is very important to note that buffers in routers are used for two purposes. First, a buffer allows to multiplex simultaneous flows into the output link. Second, it allows to absorb bursts from individual flows. Both functions lead to packet delay. With a buffer of limited size one function necessarily degrades the other one. We will see shortly how a

router can protect the multiplexing function.

Buffers need to have free space to be useful, i.e. to absorb transient overloads. Thus, if the offered load persistently exceeds link capacity, buffers have no use because they will tend to be full or almost full most of the time.

Since users are not limited in the load they can send into the network, routers can control sustained overload only by dropping packets. A major decision to the router is to distinguish sustained overload from transient overload. This decision is crucial since it will determine when to drop packets and with which rate. Another major decision is the selection of the packets to drop.

By controlling when to drop packets, a router can improve the link utilization and its capacity to absorb transient overload. By selecting packets to drop, the router can control the buffer occupancy distribution among flows and thus indirectly control the bandwidth allocation. Hence, to protect the multiplexing function of router buffers and thus provide flow isolation, a buffer system must meet the next two requirements:

1. An ideal buffer system must always have free buffer space to absorb transient overloads, in the form of new flows or in the form of transient bursts from individual flows.

2. An ideal buffer should select packets to drop in such a way that loss distribution follows the buffer occupancy among flows.

We will see in the next section to what degree these requirements are met with the different existing queue management mechanisms.

5.2 Discussion of Existing Capacity Allocation Schemes in Routers

Drop-Tail FIFO

Currently, the Internet's best-effort service is provided by FIFO scheduling and Drop-Tail queue management in routers.

In this scheme, routers have one queue per output link and routers simply transmit packets in the order they arrive (FIFO). Also, routers have no control of how buffer is filled and packets that arrive when the buffer space is exhausted are simply dropped (Drop-Tail). In other words, routers cannot control neither when to drop packets nor which packets to drop;

instead, both decisions are controlled by the users' behavior. As a result, the service order of packets and the occupancy distribution of the buffer depend also on the users' behavior.

Clearly, with this scheme, routers cannot protect the multiplexing function of the buffer by themselves; instead, the multiplexing function can only be protected by the users' own behavior. More specifically, users should find the appropriate traffic rate to send packets, according to the network load conditions. Usually, users control its transmission rate with the TCP protocol [Jac88][APS99], which ideally ensures that buffer occupancy will be evenly distributed among the competing flows. Nevertheless, since the protection of the multiplexing function of buffers depends on the cooperation of users, an aggressive user can always monopolize the buffer space and starve cooperating flows.

Even if all users cooperate, still some problems exist with this scheme. Since Drop-Tail cannot control buffer space allocation, users can maintain buffers full or almost full most of the time, and routers will detect sustained overload only when the buffer overflows. Full buffer causes problems because it does not allow buffer space availability to absorb transient overloads. Since data traffic is bursty, losses will be also bursty. This can result in problems like "global synchronization" [SZC90], where several TCP connexions increase and decrease their load simultaneously because router drops packets from several connexions at the same time. When this happens the link utilization is reduced. By using TCP, users can more or less preserve the multiplexing function of router buffers. But even if all users utilize TCP, differences among flows like the round trip time (RTT) results in bias against some flows [FJ92].

In summary, with Drop-Tail, the multiplexing function cannot be protected in the presence of unresponsive flows because these flows can occupy all the buffer space and starve responsive flows by simply sending fast enough. With Drop-Tail, the only way to protect the multiplexing function of buffers is by having all end-systems to control their traffic. In particular, by using the TCP protocol.

RED

The major improvement of RED (Random Early Detection) [FJ93], with respect to Drop-Tail, is that space to absorb transient overload is made available (first requirement to protect the multiplexing function). RED achieves this by controlling the load in case of incipient congestion. That is, RED begins to drop packets well before the queue length reaches the total buffer size. The average queue length is used as an indicator of the level of sustained overload.

Another improvement of RED is that it uses randomization to select packets to drop. Since RED tends to drop each packet with equal probability, losses are distributed among flows in proportion to their bandwidth usage. By using randomization, RED alleviates the problems of global synchronization and phase effects, but it is assumed that users respond to packet losses, or in general to congestion indication, by reducing their sending rate.

While RED randomization allows the routers to better distribute losses among competing flows, RED randomization cannot avoid that flows using less than their fair share lose packets. The problem is that responsive flows respond to losses by reducing their load while unresponsive aggressive flows can continue to get their packets stored in the buffer. In other words, responsive flows not using their fair share are prevented from reaching it. As a result unresponsive users can still starve responsive flows with RED [LM97].

In summary, RED allows the multiplexing function of buffer to be protected only if all users are responsive.

A modified version of RED called FRED (Flow Random Early Drop) was proposed in [LM97]. By using per-active-flow accounting FRED provides better isolation from aggressive flows than RED.

Rate information in packets.

With Drop-Tail routers cannot select at all the packets to be dropped because this depends on the users' behavior. While RED improves this by using randomization in such a way that drops are better distributed among all the flows, responsive flows are still affected in the presence of unresponsive flows because RED randomization has limitations on the selection of packets to drop.

A different way to allow routers to select packets to drop is by including some flow information in the packets. In particular, flow rate information can be included in the packets in such a way that routers can decide if a packet is accepted or dropped depending on the level of overload. An example of this approach is CSFQ[SSZ98]. CSFQ still uses randomization but in such a way that: First, flows sending at less than their fair share suffer no drops. Second, flows exceeding its fair share suffer probabilistic dropping. Dropping is function of the flow rate information of the packet and of an estimated fair share rate.

The Rainbow Fair Queueing approach, RFQ [CWZ00], also label packets with flow information. Contrary to the CSFQ scheme, RFQ does not label packets with the estimated flow rate; instead, each packet has a label corresponding to a layer value. These values results from dividing conceptually a flow into a number of layers with fixed rates. Routers

determine a dynamic threshold according to the load level and packets with labels greater than this threshold are dropped.

Another scheme that includes flow information in packets is Tuf [CD01]. In the Tuf approach, packets are labeled with different values. These values determine the dropping eligibility of the packet. When the buffer overflows, the router drops the packet with the highest label from the buffer. Labelling of packets takes into account the responsiveness of the flow they belong to, in such a way that the average rates of flows remain fair.

The main improvement of these approaches is that the multiplexing function of buffers is protected by avoiding that flows using less than their fair share suffer packet losses. Nevertheless, these approaches require to insert additional information in packets and more importantly it requires that all edge routers (or end-hosts) in the system agree on a single scheme to consistently label packets.

Per-flow queuing mechanisms

A completely different way to protect the multiplexing function of router buffer is by maintaining a separate FIFO queue for each flow [Nag87][SV95]. Queues are serviced in a Round Robin order.

In this case the multiplexing function is "decoupled" from the burst absorbing function. Since each flow has a different queue, the bursts or traffic pattern of each flow will not disturb the others flows.

The problem with this approach is the difficulty of determining the number of active flows at a given moment. Additionally, state per flow must be maintained and complex scheduling is needed. Also, no FIFO scheduling introduces unnecessary delay for low-bandwidth flows with short bursts arrival [CSZ92].

5.3 Our scheme

5.3.1 Design Goals

FIFO scheduling has the strength of being simple and easy to implement because no calculation is needed to decide which is the next packet to send. With FIFO, the bandwidth allocation is made by allocating the buffer space. Thus, a queue management mechanism is necessary to provide isolation with FIFO scheduling. This is the approach we use in our mechanism.

The simple Drop-Tail queue management has the main problem that the burst absorbing function troubles the multiplexing function. Our main goal is to protect efficiently the multiplexing function from the burst absorbing function in router buffers. Protecting efficiently the multiplexing function provides isolation of flows while allowing high link utilization level.

Since bursts are an inherent characteristic of traffic in Internet, we seek to not penalize bursty traffic. Also, since there are many short lived flows in the Internet, it is very important to not penalize this kind of flows. Our mechanism seeks to not refuse packets from new flows. In fact, this is just another aspect of the protection of the multiplexing function.

We now introduce our queue management mechanism that provides flow isolation.

5.3.2 Overview

We propose a new scheme that provides isolation of flows for best-effort traffic, without requiring per-flow queuing. Our scheme is based on the main idea that the multiplexing function of a router buffer must be protected from its burst absorbing function. We called our scheme MuxQ for Multiplexing Queuing. As was stated in section 5.1, to protect the multiplexing function, routers must provide buffer space to absorb transient overloads. It also must drop or accept packets according to the buffer occupancy distribution of flows. To provide buffer space to absorb transient overloads, MuxQ controls the queue length while allowing high throughput and high link utilization. Also, MuxQ progressively controls the allocation of buffer space in a FIFO queue. The allocation decision is based only on state information of a limited number of flows: the flows that do currently have packets in the queue.

MuxQ has a FIFO queue called MUXqueue, see Figure 5.1. Although this queue works essentially in the same way that the FIFO queue in traditional IP routers, we control the buffer space of the MUXqueue in such a way that in case of overload, only a limited number of packets from each active flow is accepted. For each flow the maximum number of packets that the router accepts is function of the number of active flows and the flow's buffer occupancy.

5.3.3 Detailed operation

We explain our mechanism by starting from the traditional FIFO Drop-Tail router then we will refine the scheme to achieve our goals.

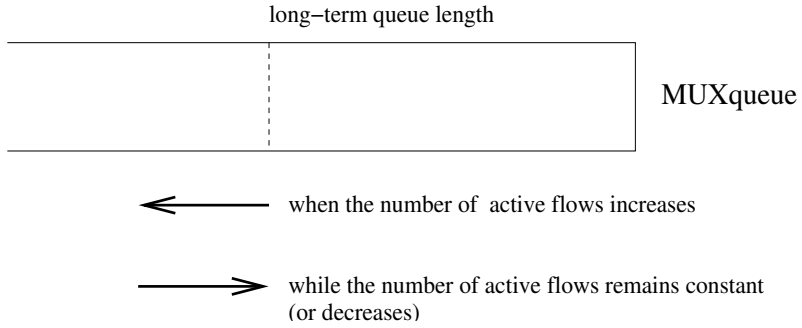

Figure 5.1: The MuxQ queue

In traditional IP routers with drop tail, the buffer occupancy distribution among flows is not controlled by the router but by the users' behavior. Among other problems this can keep the buffer full, because packets are dropped only when the buffer space gets exhausted.

As a result, a packet from a new flow may be refused while other flows have a rather large number of packets accepted. In other words, the multiplexing capacity is completely reduced because some flows have already used buffer space to absorb their bursts. To address this problem MuxQ controls the allocation of buffer space in such a way that: First, buffer space is always made available to accommodate packets from new flows. The idea is to control the queue length in such a way that in the long term the queue has a smaller length than the buffer size. We will refer to this length as *ltqlen* (long-term queue length), see Figure 5.1. Second, buffer space is progressively shared among the active flows. We explain this with an example.

Suppose the MUXqueue is empty and a packet arrives. At this moment there is only one active flow, thus the buffer can accept a maximum of *ltqlen* packets from this flow. In other words, the flow is restricted to have a maximum number of packets (denoted *maxpkts*) equal to *ltqlen*, in the MUXqueue at the same time. Since *ltqlen* is set to a value smaller than the buffer size, a packet from a second flow will always be accepted. Suppose that by the time when the packet from the second flow arrives the first flow has already *ltqlen* packets enqueued. At this moment the maximum number of packets allowed per flow will be reduced to *maxpkts*=*ltqlen*/2 because the number of active flows has changed to 2. Note that actually the first flow has more packets enqueued than the maximum allowed. Nevertheless, this is a transient overload because the first flow will not be allowed to enqueue further packets until its packet population gets smaller than the new *maxpkts* value. On the other hand, the second flow can continue to enqueue packets. Clearly if no other flow appears the queue length will tend to the *ltqlen* value, with half of the space for each flow (assuming

flows continue to send packets). But suppose that another packet from a third flow appears. In this case the number of active flows will be 3 and the maximum number of packets allowed per flow will decrease to *maxpkts*=*ltqlen*/3. Again flows with a packet population exceeding the maximum allowed will not have packets accepted for some time while flows with a number of packets below this maximum will continue to enqueue packets. Clearly, the more the number of new flows, the more the greedy flows will be penalized.

In summary, in the long term, the queue length is maintained shorter than the buffer size and tends to the value *ltqlen*. As we can see, a dynamic reserved part of the buffer is always available to accommodate new flows. This part of the buffer is dynamically delimited by the *ltqlen* value and the maximum buffer size. We protect in this way the multiplexing function of the router buffer. Controlling the long-term queue length has the additional benefit of controlling the long-term delay of packets. The MuxQ algorithm is shown in Figure 5.2.

As we have seen, *maxpkts* the maximum number of packets in the buffer from the same flow, will be reduced according to the number of active flows. This allows to share the buffer space among the competing flows. The more the number of competing flows, the less the buffer space allowed per flow.

With MuxQ the bursts are naturally accepted as long as their size do not trouble the multiplexing capacity (*maxpkts*). In case the aggregated bursts size (number of packets in the MUXqueue) for an active flow is larger than *maxpkts,* its new incoming packet is dropped. Note that packets are dropped also if a very large number of new flows arrives simultaneously at the MUXqueue.

5.3.4 Active Flows

Since routers treat packets independently, the concept of flow is not natural at the router level. Nevertheless, some notion of flow is necessary if we want to protect the multiplexing function. This notion must be more related to a dynamic soft-state than to the real end-to-end flow notion. In fact, in our scheme the notion of flow is associated to the group of packets with some defined specific subset of identifiers and which are in the MUXqueue at a given moment. In other words, a flow in our scheme lasts as long as the flow has packets in the MUXqueue.

When a packet arrives at the router, a lookup of the output port is performed and then the packet is directed to the output link. Then it is checked if packets from the same flow are already in the MUXqueue. To make this operation efficient we use a hashing table containing the number of packets of each flow in the MUXqueue. It is worth to note that

```
procedure Enqueue
begin
    Check number of packets (npkts) from this flow already in the
    MUXqueue (* by hashing *);
    if packet is from a new active flow, i.e. npkts==0 then
        increment number of active flows;
        set npkts=1 in the hash table entry for this flow;
        insert packet in the MUXqueue;
        update maxpkts=ltqlen/number of active flows;
    else if npkts < maxpkts then
        increment npkts in the hash table entry for this flow;
        insert packet in the MUXqueue;
    else
        drop packet;
    end
end

procedure Dequeue
begin
    serve packet in the front of the MUXqueue;
    decrement npkts in the hash table entry for the flow the packet belongs to;
    Check number of packets (npkts) from this flow still in the MUXqueue (* by hashing
    *);
    if npkts== 0  then
        decrement number of active flows;
        update maxpkts=ltqlen/number of active flows;
    end
end
```

Figure 5.2: The MuxQ algorithm

state information is maintained only for the flows that do have packets in the main buffer at any moment. The table is updated when a packet arrives and when a packet leaves the router using a hashing operation, see Fig.5.2.

5.4 Performance

In this section, we evaluate our queue management mechanism by simulation. More specifically, we study the ability of MuxQ to isolate different kind of flows. The performance of MuxQ is compared with those of Drop-Tail, CSFQ, FRED, and DRR by using the ns-2 simulator [nns], which we extended with a MuxQ module. CSFQ and FRED simulation code was obtained from [Sto98]. DRR and Drop-Tail are included in the standard ns-2 package. Since DRR uses per flow queuing, nearly perfect isolation can be obtained with this mechanism; it is used as a reference in our simulations. We present the results of several simulation experiments, each of which focuses on a particular traffic condition. Both TCP sources and constant bit rate (CBR) UDP sources are used. Most of our simulations use the topology shown on figure 5.3. The bottleneck link bandwidth is 10 Mbps and the link's propagation delay is 5 ms. The bottleneck router has a maximum buffer size of 100 KBytes and all packets are 1000 bytes long. For MuxQ the *ltqlen* value was fixed at 0.75 of the buffer size.

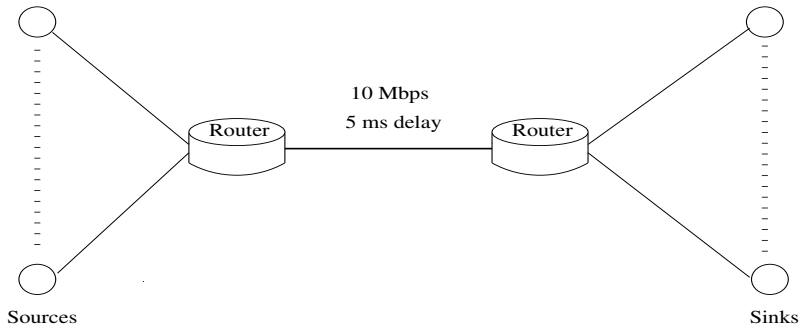

Figure 5.3: The single bottleneck link simulation topology

5.4.1 Protecting long-lived responsive flows (TCP) from each other.

In this first experiment we seek to show that our mechanism does not interfere with the end-to-end TCP congestion algorithm. For network traffic, we use FTP transfers over TCP. Each source initiates an FTP transfer at the beginning of the simulation and continues until the

end of the simulation. Figure 5.4 shows the average throughput achieved by each flow over a 50 sec interval. As we can see, MuxQ provides almost perfect isolation. DRR provides also almost perfect isolation, but DRR uses per flow queuing which is not the case for MuxQ. For this traffic, Drop-Tail provides reasonable isolation. This is because all sources are responsive and all have the same round-trip time. For CSFQ and Fred the throughput of flows shows more variability.

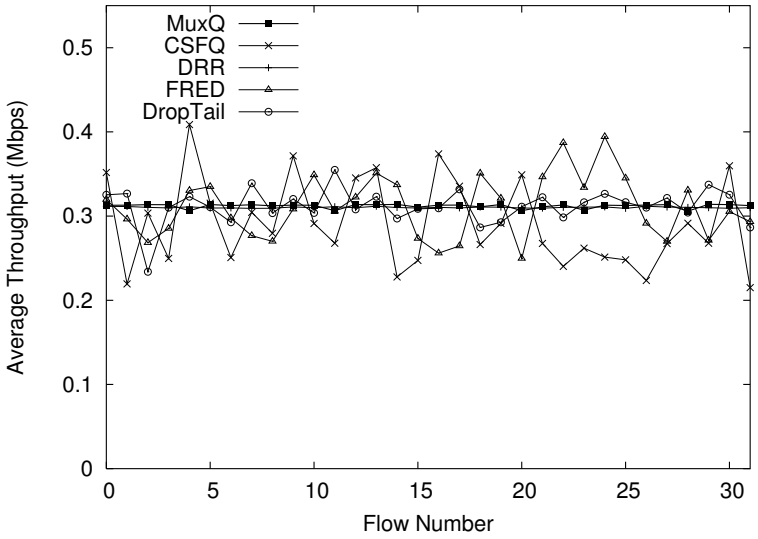

Figure 5.4: Average throughput of 32 TCP flows sharing a 10 Mbps link.

5.4.2 Protecting long-lived responsive flows (TCP) from long-lived non responsive flows

In this experiment, we examine MuxQ's ability to isolate responsive flows from non responsive flows bad effects. Five of the TCP flows in the last experiment are substituted with 5 aggressive CBR flows (flows 27 to 31) sending at 10 Mbps each one. Figure 5.5 shows the average throughput of each flow over a 50 sec interval. Again DRR performance is almost perfect while Drop-Tail does not protect at all the TCP flows from the aggressive CBR flows. Performance of MuxQ is slightly better than that of CSFQ.

In another experiment, a TCP flow competes with an increasing number of CBR flows. The TCP flow is generated by a source which always has data to send, and the CBR flows are generated by unresponsive sources which transmit packets at a constant rate of twice its fair share. In Figure 5.6 we show the normalized average throughput achieved by the

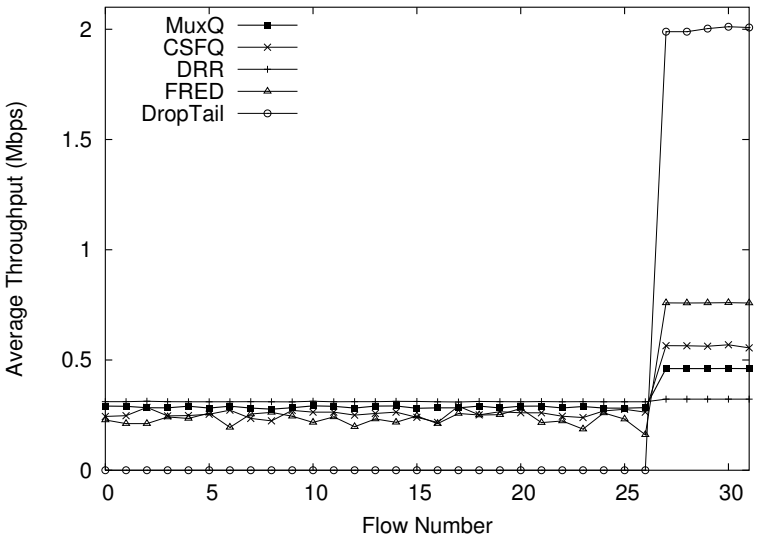

Figure 5.5: Performance degradation of several TCP flows (flows 0 to 26) when competing with 5 aggressive CBR flows (flows 27 to 31).

TCP flow. The results show again that with Drop-Tail the performance of the TCP flow is severely degraded even with only one CBR flow. DRR provides almost perfect isolation. With MuxQ and CSFQ the performance of the TCP flow is maintained at a reasonable level.

5.4.3 Protecting short-lived flows from long-lived flows (responsive and non responsive)

In this experiment, short-lived web-like TCP flows compete with long-lived flows: 5 long-lived TCP flows and 1 aggressive CBR flow sending at 10 Mbps. For the long-lived flows, each source initiates its transfer at the beginning of the simulation and continues until the end of the simulation, i.e 50 sec. 20 web clients make random requests to a web server. The server responds to each request by sending a web page with one object of 12 Kilobytes. In other words, all web-like flows are of the same size, i.e. 12 Kilobytes. The time between retrieval of two successive pages follows an exponential distribution with mean equals to 3 sec. Figure 5.7 shows the cumulative distribution of the web-like flows. Horizontal axis is the time to complete the flow transmission. The figure shows the number of finished web-like flows after 50 sec of simulation. We can observe that with MuxQ most of the web-like flows have smaller response times than with the other schemes. Note also that for the same simulation time, the number of successful finished flows is different for each scheme. MuxQ

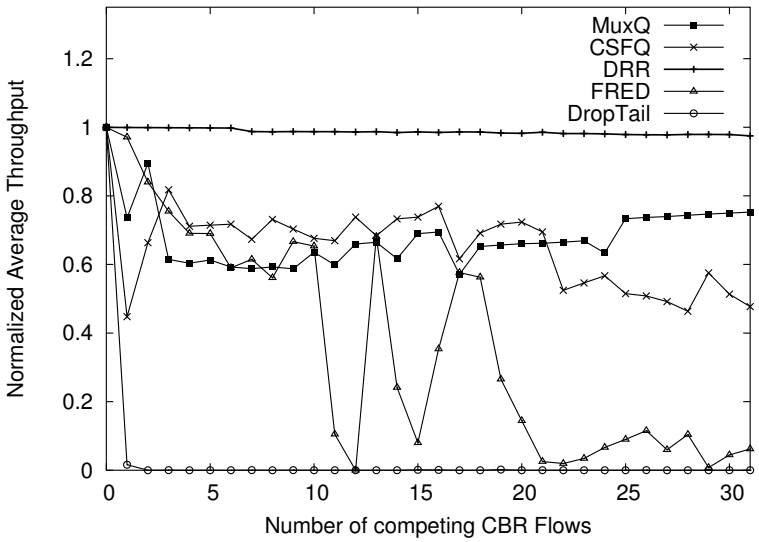

Figure 5.6: Average throughput of 1 TCP flow competing with an increasing number of CBR flows

outperforms in this aspect to DRR and Drop-Tail. Unfortunately, we were not able to run this simulation nor with CSFQ nor with Fred with the source code from [Sto98].

5.4.4 How non-responsive flows affect each other

In this experiment, we seek to show how unresponsive flows with different rates affect each other. CBR flow number i sends packets at a rate (i+1) times its fair share. In other words, flows transmit at different rates which go from the fair share rate to the maximum link rate. Note that all flows but the first one send packets at a rate greater than its fair share rate. Figure 5.8 shows the average performance of the different flows. As we can see, Drop Tail does not isolate flows. Flows which transmit at a larger rate get more bandwidth than lower rate flows. MuxQ, DRR, and CSFQ provides almost perfect isolation. Fred performance is only slightly better than that of Drop-Tail.

5.4.5 Multiple congested links

In this experiment a TCP flow traverses several congested links. The topology used in this experiment is shown in figure 5.9. Bottleneck links have a bandwidth of 10Mbps and 1ms of delay. The main source sends a TCP flow that traverses k consecutive congested links. Cross traffic is generated by several groups of sources. Each source in a group sends a flow

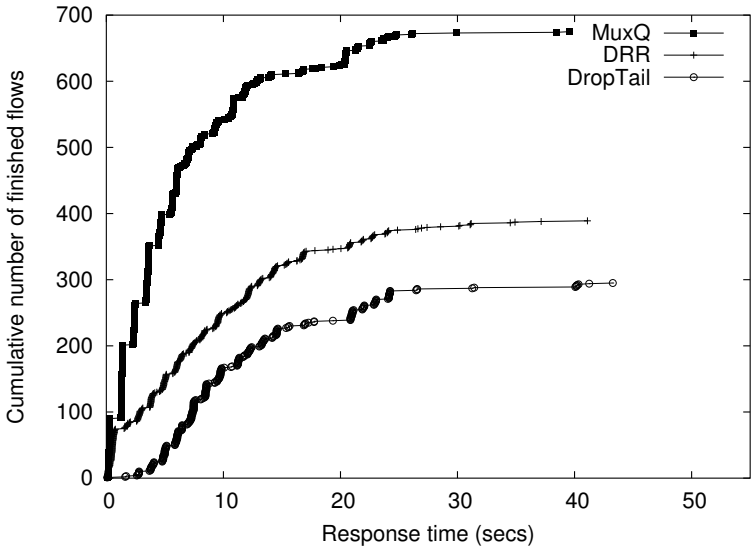

Figure 5.7: Web-like traffic competing with 5 long-lived TCP flows and 1 aggressive CBR flow sending at 10 Mbps.

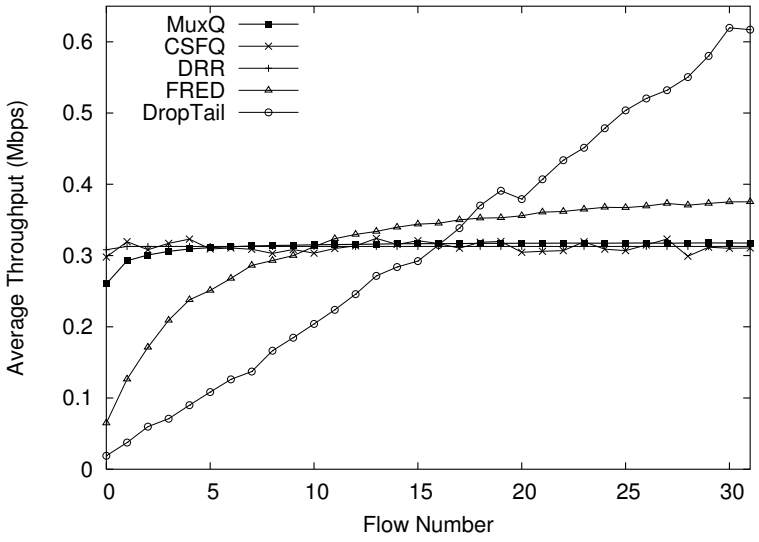

Figure 5.8: Average throughput of 32 CBR flows sending at different rates. Flow i sends packets at a rate (i+1) times its fair share.

that traverses only one congested link as shown in the same figure. Each group consists of 10 sources, which send a long-lived flow each one. In each group, five sources generates TCP flows, and the other five sources generate CBR aggressive flows at 10 Mbps. Fig 5.10 shows the normalized average throughput of the TCP flow traversing k congested links. With Drop-Tail the normalized average throughput is zero (even with only one congested link) while with DRR performance is almost perfect. Performance of MuxQ, CSFQ and Fred is better than that of Drop-Tail but it degrades rapidly with increasing congested links.

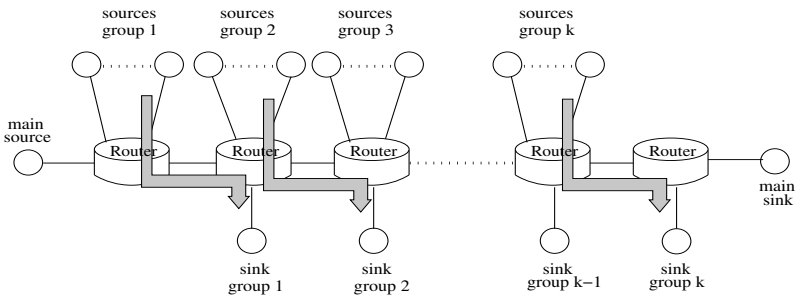

Figure 5.9: Topology with several congested links

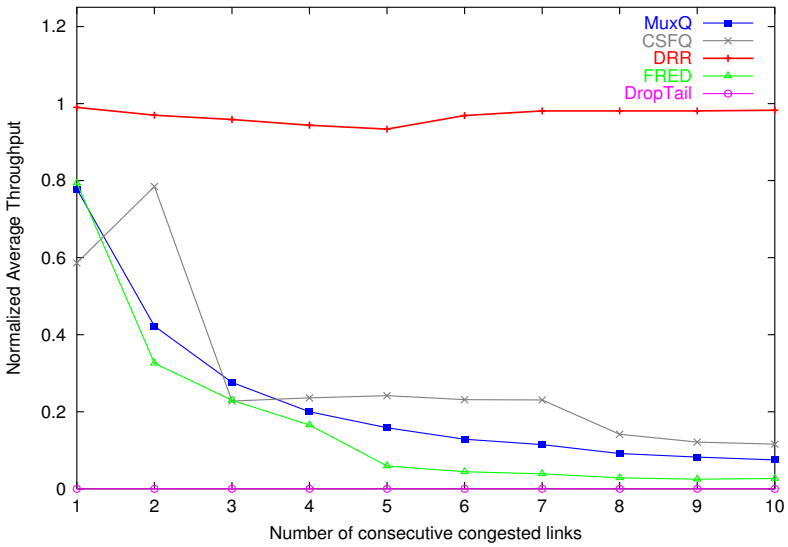

Figure 5.10: A TCP flow traversing several congested links

5.4.6 MuxQ Deployability

We finish this section by indicating why we believe MuxQ can be easily deployed. Even though some users can choose not to use TCP as their transport protocol, it is a fact that

TCP remains highly utilized. We have shown that MuxQ behaves very well when TCP flows cross a MuxQ router.

Also, MuxQ does not need modifications to the IP header. More importantly, MuxQ does not require that hosts or end-routers agree on a single scheme to consistently include flow information. In other words, MuxQ does not expect a special behavior from routers nor from hosts. Thus, MuxQ routers can be deployed incrementally in the Internet.

5.5 Summary

We have designed a new queue management mechanism for flow isolation. Our mechanism is based on the idea to protect the multiplexing function from the burst absorbing function of router buffers. MuxQ which is based on a FIFO queue, uses a very simple algorithm to allocate buffer space and control the queue length.

We compared the performance of the proposed scheme to that of classical Drop-Tail and to that of other proposed schemes, including CSFQ and DRR which provides nearly perfect isolation by using per-flow queuing. By keeping only limited flow-state our mechanism performs very much better than Drop-Tail. MuxQ achieves performance similar to that of CSFQ but MuxQ does not need modifications to the IP packet header as it is the case for CSFQ.

One of the important characteristic of a new router mechanism is its incremental deployability. MuxQ does not need modifications of the IP packet header. Moreover, since MuxQ does not expect a special behavior from other routers, MuxQ routers can interact without problem with classical Drop-Tail routers and thus MuxQ can be deployed incrementally. We believe that MuxQ is an interesting approach to achieve a high degree of flow isolation with respect to Drop-Tail by using a very simple algorithm.

Chapter 6

Conclusions

One of the key characteristics of the Internet is its capacity of adaptation. The Internet has been able to adapt to growth in terms of number of users and volume of traffic. The ability to adapt to these changes is usually referred to as scalability. The Internet also adapts in the short term to changes in the topology and load. The ability of the Internet to adapt to these changes by degrading gracefully the network service is referred to as its robustness. This thesis has been motivated by the idea of preserving the scalability and robustness of the Internet.

The Internet is a packet network, where each packet is treated independently of all others; that is, it uses the datagram paradigm. Routers play an essential role in the datagram paradigm of the Internet. Since each packet must be treated independently, the performance of the Internet depends directly on the per packet processing capacity of routers. In this thesis we have proposed several algorithms to improve the packet forwarding process in Internet routers. We conclude this dissertation with a summary of the main ideas and contributions of this thesis. Then we discuss directions for future work.

6.1 Main ideas and contributions of this thesis

Incremental updates for forwarding databases based on multibit tries

Due to installation, reconfiguration or failure of network elements, the Internet is always changing. Routers adapt to these changes by updating their forwarding databases. To cope with scalability, routers use nested prefixes to aggregate forwarding information. While the use of nested prefixes allows routers to cope with scalability issues, an implementation that maps directly these prefixes to a data structure can result in non efficient searches.

To optimize the address lookup operation the forwarding database can be disaggregated to some extent in a controlled way. Controlled disaggregation of a forwarding database can be achieved by the use of multibit tries. While disaggregation of the forwarding database optimizes the search operation, this disaggregation complicates the updating operations because inserting or deleting a single prefix may need to update many entries. Moreover, an additional structure is needed to maintain the original prefixes and their nesting structure.

One of the main contribution of this thesis is that we have a proposed a complete and efficient scheme to support incremental updates in forwarding databases based on multibit tries. In our approach we have introduced two key concepts to support incremental updates: The notions of span and coverer. In particular, we have proposed two mechanisms to support incremental updates in multibit-trie-based BMP lookup schemes: the bit-vector-array mechanism and the PN bit vector mechanism. While the bit-vector-array mechanism does not need computing overhead, it uses memory inefficiently. To use memory more efficiently, we proposed the PN bit vector mechanism. The PN bit vector mechanism obtains efficient memory usage by using our PN bit vector data structure and by computing the corresponding mapping functions. In other words, the PN bit vector mechanism trades computation for efficient memory usage. Moreover, Since the PN bit vector mechanism separates the additional data structure needed to support incremental updates from the main data structure used for doing BMP lookup operations, the performance of the BMP lookup operation is not degraded by the incremental update capabilities. Another contribution in our approach is the introduction of two key concepts in the framework of incremental updates for multibit tries: the notions of span and coverer.

Taxonomy and framework for BMP lookup schemes

To better compare the different BMP lookup schemes, we have proposed in this thesis a taxonomy based on the double dimension of the BMP search: value and length. That is, determining the best matching prefix involves not only comparing the bit pattern itself (i.e., finding a match), but also finding the appropriate length (i.e., the longest one). Our taxonomy classifies the address lookup schemes according to these two dimensions and also if a linear or a binary search is performed. We analyzed the next four cases: 1) Linear search based on values; 2) Binary search based on values; 3) Linear search based on lengths; 4) Binary search based on lengths. We analysed also how the different approaches can be explained in terms of classical well-known search algorithms. We highlighted the main ideas behind the different schemes and we emphasized whether the schemes supports incremen-

tal updates. With our taxonomy, we identified the different tradeoffs in the address lookup schemes. We compared the different schemes in terms of their complexity and measured execution time on a common platform.

It has been the purpose of our taxonomy to use consistent terminology. By using consistent terminology, the different ideas and concepts in each scheme can be easily related each other. We believe that by providing an integral vision of the BMP lookup schemes, as our taxonomy does, we can help to point researchers in fruitful directions in this area.

Optimizing the use of buffers for flow isolation

To forward a packet, routers need not only to find where the packet should be sent next, but also must resolve contentions at the output-port. Routers use buffers to solve the contention for the output-port. In general, buffers are used not only to multiplex packets from different flows, but also to absorb bursts of packets of individual flows.

The use of buffers is effective only in the case of transient overload. Since in the Internet sustained overload can exist, to make effective use of buffers one needs mechanisms to control the traffic. Traditionally, the traffic control in the Internet has been done by end systems using the TCP algorithms. Nevertheless, in today's Internet, the use of TCP is rather a user's choice and hence, in general, buffers in routers are not always used effectively. When the buffer is not used effectively, the multiplexing function of buffers is affected by its burst absorbing function. Degradation of the multiplexing function of buffer in routers results in a lack of flow isolation. Based on the idea of protecting the multiplexing function of buffers from the burst absorbing function, we have designed a mechanism to optimize the use of the buffer in routers and to provide flow isolation with only a minimum of flow information in routers.

Our mechanism, which we call MuxQ, protects the multiplexing function from the burst absorbing function by progressively and dynamically controlling the allocation of buffer space in a FIFO queue. MuxQ is a new queue management mechanism that provides flow isolation by using a very simple algorithm and without using per-flow queuing.

We compared the performance of the MuxQ scheme to that of classical Drop-Tail and to that of other proposed schemes, including CSFQ and DRR which provides nearly perfect isolation by using per-flow queuing. By keeping only limited flow-state, our mechanism performs very much better than Drop-Tail. MuxQ achieves performance similar to that of CSFQ but MuxQ does not need modifications to the IP packet header as it is the case for CSFQ.

One of the important characteristic of a new router mechanism is its incremental deployability. MuxQ does not need modifications of the IP packet header. Moreover, since MuxQ does not expect a special behavior from other routers, MuxQ routers can interact without problem with classical Drop-Tail routers and thus MuxQ can be deployed incrementally. We believe that MuxQ is an interesting approach to achieve a high degree of flow isolation with respect to Drop-Tail by using a very simple algorithm.

6.2 Future work

The performance of the packet forwarding process in routers is essential for the performance of the Internet. While we have provided algorithms to improve the packet forwarding performance of routers, ever increasing traffic demands for further improvements. We discuss in this section some directions for future work.

The address lookup operation requires to consult a table with the forwarding information. As a result, an ideal address lookup mechanism makes a single memory access when consulting the forwarding table. What are the options to obtain an address lookup mechanism with a single memory access? There are two possible ways to obtain address lookups with a single memory access. One is by implementing multibit tries with pipelined architectures and the other one is by using TCAMs (Ternary Content-Addressable Memories). However, for these approaches to work, some problems need to be solved.

To effectively implement multibit tries in a pipeline architecture, the problem of distribution of memory among the different pipeline stages must be solved. A main point in this thesis has been the capability of incremental updates. Update speed is more critical when pipeline is used. One direction for future work is to investigate how to support efficient incremental updates in pipeline architectures, without causing significant disruption to the address lookup operations.

While TCAMs allow address lookups with a single memory access, they require prefixes to be ordered by length. Hence a problem to solve is how to efficiently keep prefixes ordered by length in TCAMS based schemes, when inserting or deleting prefixes.

We have proposed MuxQ, a mechanism to protect the multiplexing function of buffers in routers, and hence provide flow isolation, without the need of end-systems cooperation. We considered that buffering happens only at the output ports. While many routers have buffers only at the output ports, the need for further high performance routers leads to routers with buffers both before and after the internal switch of routers. A future direction is designing flow isolation mechanisms which exploits the fact of having buffers both before and after

the internal switch of routers.

Bibliography

[APS99] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, April 1999.

[Awe00] J. Aweya. On the design of ip routers part 1: Router architectures. *Journal of Systems Architecture*, 46(6):483–511, April 2000.

[Bak95] F. Baker. RFC 1812: Requirements for IP version 4 routers, June 1995.

[BKOS97] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry : Algorithms and Applications*. Springer Verlag, Berlin Heidelberg, 1997.

[CD01] A. Clerget and W. Dabbous. Tuf : Tag-based unified fairness. In *Proceedings of INFOCOM 2001*, pages 498–507, April 2001.

[CDG99] Pierluigi Crescenzi, Leandro Dardini, and Roberto Grossi. IP address lookup made fast and simple. In *7th European Symposium on Algorithms*, pages 65–76, 1999.

[CM99] G. Cheung and S. McCanne. Optimal routing table design for ip address lookups under memory constraints. In *Proceedings of IEEE INFOCOM*, pages 1437–44, March 1999.

[CSZ92] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: architecture and mechanism. In *Proceedings of the ACM SIGCOMM '92 conference*, pages 14–26. ACM Press, 1992.

[CWZ00] Z. Cao, Z. Wang, and E. W. Zegura. Rainbow fair queueing: Fair bandwidth sharing without per-flow state. In *Proceedings of INFOCOM 2000*, pages 922–931, 2000.

[DBCP97] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM*, pages 3–14, 1997.

[DKN96] W. Doeringer, G. Karjoth, and M. Nassehi. Routing on longest-matching prefixes. *IEEE/ACM Transactions on Networking*, 4(1):86–97, February 1996.

[Eat99] W. N. Eatherton. Hardware-based internet protocol prefix lookups. Master's thesis, Washington University, May 1999.

[FJ92] S. Floyd and V. Jacobson. Traffic phase effects in packet-switched gateways. *Journal of Internetworking:Practice and Experience*, 3(3):115–156, September, 1992.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (TON)*, 1(4):397–413, 1993.

[FLYV93] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, September 1993.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[GLM98] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM (3)*, pages 1240–1247, 1998.

[Har] Yoichi Hariguchi. Smart multi-array routing table. In *Proceedings of INET2001*.

[Hui00] Christian Huitema. *Routing in the Internet*. Prentice Hall PTR, 2 edition, 2000.

[Hus] Geoff Huston. Bgp table report. *at http://bgp.potaroo.net/*.

[Hus01] Geoff Huston. Analyzing the internet bgp routing table. *The Internet Protocol Journal*, 4(1):24–15, March 2001.

[HZ99] Nen-Fu Huang and Shi-Ming Zhao. A novel ip-routing lookup scheme and hardware architecture for multigigabit switching routers. *IEEEJSAC: IEEE Journal on Selected Areas in Communications*, 17(6), June 1999.

[Jac88] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[Jai91] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling,*. Wiley-Interscience, New York, NY, USA, May 1991.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[Lab99] C. Labovitz. *Scalability of the Internet Backbone Routing Infrastructure*. PhD thesis, University of Michigan, 1999.

[LM97] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 127–137. ACM Press, 1997.

[LSV98] B. Lampson, V. Srinivasan, and G. Varghese. Ip lookups using multiway and multicolumn search. In *Proceedings of IEEE INFOCOM*, pages 1248–56, April 1998.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.4BSD operating system*. Addison Wesley Longman Publishing Co., Inc., 1996.

[Min01] Cyriel Minkenberg. On packet switch design. Research Report RZ3387, IBM Zurich Research Laboratory, December 2001.

[Mor68] Donald R. Morrison. Patricia–practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[MS98] Andreas Moestedt and Peter Sjodin. IP address lookup in hardware for high-speed routing. In *Hot Interconnects VI*, August 1998.

[Nag87] J. Nagle. On packet switches with infinite storage. *IEEE Trans. Communications*, 35(4):435–438, April 1987.

[NK98] S. Nilsson and G. Karlsson. Fast address look-up for internet routers. In *Proceedings of IFIP 4th International Conference on Broadband Communications*, pages 11–22, April 1998.

[NK99] S. Nilsson and G. Karlsson. Ip-address lookup using lc-tries. *IEEE Journal on Selected Areas in Communications*, 17(6):1083–1092, June 1999.

[nns] The ns-2 network simulator. release 2.1b8. *http://bgp.potaroo.net/*.

[PCB+98] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, and G. D. Troxel. A 50-gb/s ip router. *IEEE/ACM Transactions on Networking (TON)*, 6(3):237–248, 1998.

[Pos81] J. Postel. Internet protocol. *RFC 791*, September 1981.

[PZ92] T. Pei and C. Zukowski. Putting routing tables in silicon. *IEEE Network Magazine*, 6(1):42–50, January 1992.

[RL93] Y. Rekhter and T. Li. RFC 1518: An architecture for IP address allocation with CIDR, September 1993.

[RSBD01] Miguel A. Ruiz-Sánchez, Ernst W. Biersack, and Walid Dabbous. Survey and taxonomy of ip address lookup algorithms. *IEEE Network*, 15(2):8–23, March/April 2001.

[RSD00] Miguel A. Ruiz-Sánchez and Walid Dabbous. Un mécanisme optimisé de recherche de route. In *Proceedings of CFIP 2000*, pages 217–32, October 2000.

[RSD03] Miguel A. Ruiz-Sánchez and Walid Dabbous. Controlling bursts in best-effort routers for flow isolation. In *Proceedings of ISCC 2003*, pages 399–404, July 2003.

[Sed97] R. Sedgewick. *Algorithms in C, 3rd edn, parts 1-4.* Addison_Wesley, December 1997.

[Sem01] C. Semeria. Understanding ip addressing: Everything you ever wanted to know. *http://www.3com.com/other/pdfs/infra/corpinfo/en_US/501302.pdf*, 2001.

[SK02] S. Sahni and K. Kim. O(log n) dynamic packet routing. In *Proceedings of IEEE Symposium on Computers and Communications*, pages 443–448, 2002.

[Skl91] Keith Sklower. A tree-based packet routing table for Berkeley UNIX. In USENIX, editor, *Proceedings of the Winter 1991 USENIX Conference: January 21–January 25, 1991, Dallas, TX, USA*, pages 93–104, Berkeley, CA, USA, January 1991. USENIX.

[SSZ98] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM 1998*, pages 118–130, 1998.

[Sto88] James A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, Rockville, MD, 1988.

[Sto98] I. Stoica. Csfq and fred ns-2 simulation source code. In *at: http://www.cs.berkeley.edu/ istoica/csfq/*, 1998.

[SV95] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of SIGCOMM'95*, pages 231–242, 1995.

[SV98] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. In *Proceedings of ACM Sigmetrics*, pages 1–11, June 1998.

[SV99a] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems (TOCS)*, 17(1):1–40, 1999.

[SV99b] V. Srinivasan and G. Varghese. A survey of recent ip lookup schemes. In *Proceedings of Conference on Protocols for High Speed Networks*, pages 9–23, August 1999.

[SVW01] S. Suri, G. Varghese, and P. Warkhede. Multiway range trees: Scalable ip lookup with fast updates. In *Proceedings of Globcom*, 2001.

[SZC90] S. Schenker, L. Zhang, and D. D. Clark. Some observations on the dynamics of a congestion control algorithm. *ACM SIGCOMM Computer Communication Review*, 20(5):30–39, 1990.

[Tel03] Telstra. Bgp routing table. *http://bgp.potaroo.net/as1221/bgp-active.html*, 21 mar 2003.

[TP99] Tzeng and Przygienda. On fast address-lookup algorithms. *IEEEJSAC: IEEE Journal on Selected Areas in Communications*, 17, 1999.

[WS95] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated: Volume 2. The Implementation*. Addison-Wesley, Reading, MA, USA, 1995.

[WVTP97] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed IP routing table lookups. In *ACM SIGCOMM '97*, pages 25–36, September 1997.