University of Alberta

CACHE ARCHITECTURES TO IMPROVE IP LOOKUPS

by

Sunil Ravinder

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2009
Edmonton, Alberta

Examining Committee

Mike MacGregor, Computing Science

Mario A. Nascimento, Computing Science

Ehab Elmallah, Computing Science

Chinthananda Tellambura, Electrical and Computer Engineering

To Dad and Mom

Abstract

IP address lookup is an important processing function of Internet routers. The challenge lies in finding the longest prefix that matches the packet's destination address. One of the issues concerning IP address lookups is the average lookup time. In previous works, caching was shown to be an effective method to minimize the average lookup time. Caching involves storing information on recent IP lookup results in order to decrease average lookup times.

In this thesis, we present two architectures that contain a prefix cache and a dynamic substride cache. The dynamic substride cache stores longest possible substrides from previous lookups, and is used in conjunction with a prefix cache. Successful hits in both the caches help reduce the number of worst-case lookups in the low level memory containing the IP routing table in a trie data structure. From simulations, we show that the two architectures show up to 99.9% global hit rate. Furthermore we present analytical models to find optimal designs for the two architectures. We also show that the architectures can support incremental updates once appropriate modifications are made to the trie data structure.

Acknowledgements

First and foremost I would like to thank my supervisors, Professor Mike MacGregor and Professor Mario Nascimento for their support. I appreciate the help and guidance they provided me especially when I had made little progress in my research. I am grateful to them for providing an exceptionally friendly and cooperative enviornment.

I gratefully acknowledge my fellow colleagues Abhishek, Abdullah Reza, Yufeng, Qing and Zhiyu for their help and advice. My appreciation to my friends back home for their constant motivation. I also thank the department for their generous support and for all the assistance they provided me during my stay.

Last but not the least, I am indebted my family for their love and encouragement. This thesis would not have been possible without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

One of the primary functions of IP routers is packet forwarding. This requires a longest prefix lookup in a routing table based on the packet destination address. Until approximately 1993, classful routing logic was used where the lookup process is based on matching network ID's of a IP destination address with the Class A (8 bits), B (16 bits) or C (24 bits) in the routing table. The poor IPv4 address utilization of classful routing brought about the development of Classless Inter-Domain Routing (CIDR) [33] which has a better address utilization. The many advantages of CIDR come at the expense of making the lookup process much more complex: matching is done on variable length network ID's (or prefixes). Moreover, increasing Internet traffic demands a fast mechanism to perform lookups in routing tables that may contain thousands of prefixes. Hence, IP forwarding remains an interesting research problem owing to reduced time budgets.

Real-world IP traffic demonstrates two fundamental properties: temporal and spatial locality. Temporal locality means that there is a high possibility that packets destined for the same destination addresses may arrive again within a certain time interval. These occur because of traffic burstness and network congestion and was substantiated in the studies done by Jain [21]. Spatial locality means that the packets often reference the same subnet. For example, the destination addresses 172.15.16.0, 172.15.16.1, 172.15.16.11 belong to the same subnet 172.15.16.0/24. There are a few techniques that can be used to exploit locality, e.g., dynamic data structures and caching. Examples of dynamic data structures representing the routing tables are Splay trees [30], Dynamic HiCuts [9], and Binary search trees [19]. These data structures restructure themselves depending on the locality patterns of the traffic. However this restructuring consumes both time and hardware resources. Another popular approach is using a route cache. Similar to CPU caches, a route cache is a dedicated memory that is fast [3] and relatively small and is exclusively dedicated to exploit locality in the incoming IP destination addresses.

1.2 Motivation

Standard IP forwarding data structures representing an IP routing table require multiple memory accesses for a single lookup where each memory access can be very expensive and can contribute to network latency. A number of solutions have been proposed to reduce the number of memory accesses and one such solution is caching.

In conventional routing table lookup architectures, a cache is placed between the processor and the low-level memory containing the IP routing table. During lookups, the destination addresses are first looked up in the cache. If the cache lookup is unsuccessful, the address is then looked up in the low-level memory which is much slower than the cache. As a result, it is imperative that we increase the number of successful lookups (or hits) in the cache so that fewer lookups have to refer to the low-level memory.

Traditional route caches store IP addresses [45], prefixes [13], or a combination of both [23]. Over the years, caching IP addresses has drawn criticism from router architects primarily because IP address caches capture only temporal locality [26]. Another factor that does not act in favor of IP address caches is that the IP address cache should be large in order to be effective. In recent years, prefix caching has been well received by the research community. Prefix caching is effective in capturing both temporal as well as spatial locality. Some of the reasons for it to be effective are:

- Gateway routers demonstrate high locality with references to very few prefixes. Caching these popular prefixes will also decrease network latency and improve lookup rates
- ISPs assign prefixes to subnets based on geographical location. Thus the traffic flowing to a particular area is localized and will cause similar prefixes to be frequently referenced. Caching these prefixes will be useful.

The approach towards prefix caching is to store prefixes that were looked up more recently. These cached prefixes then benefit subsequent lookups that tend to match (or hit) the same prefix.

However, because of the inherent limit in the hit rate a prefix cache can yield, some lookups will not match (or hit) and so will require a reference to low-level memory containing the complete routing table. Ideally, we would like to reduce the number of lookups that have to refer to the low-level memory or reduce (significantly) the total memory accesses needed by the unmatched packets to complete the lookup process in the low-level memory. For example, a single address lookup in the lower memory may require up to 32 memory accesses where each memory access may cost up to 25 nsec if a routing table is represented as a trie and implemented in a 25 nsec-DRAM [16]. These numbers are significant considering that for OC-48 and OC-192 links, we need to ensure that an address lookup requires at most 3-8 memory accesses on average. So the idea is to reduce the number of lookups that have to do 32 memory accesses in the worst case. This goal forms the backbone of our research. We therefore present two architectures each containing a prefix cache and a dynamic substride cache (DSC) that can be beneficial in reducing the trie lookups.

1.3 Contributions

1.3.1 Architectures

An IP routing table can be represented in a data structure such as HiCuts [20], HyperCuts [42], Trie [25], Multibit trie [44] etc. For our work, we considered tries to be suitable to represent the IP routing table. This is because tries are much easier to manage and incremental updates are straightforward to implement. A prefix is represented by a path from the root of the trie down to the leaf. A simple example of a trie representing IP prefixes is shown in Figure 1.1.

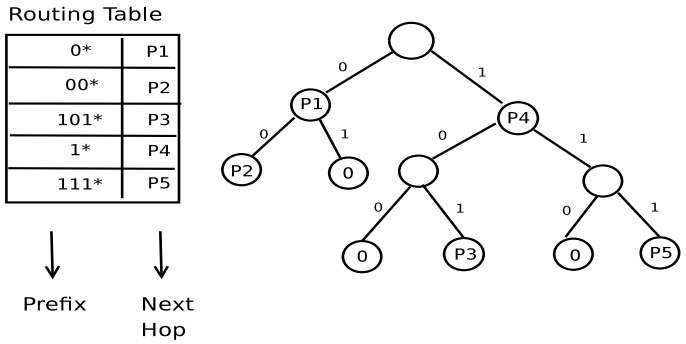

Figure 1.1: Trie representing several prefixes

IPv4 prefixes have a maximum length of 32 bits and IPv6 prefixes have a maximum length of 128 bits. We work exclusively with the former in this thesis, but recognize the growing importance of solving the same problem for IPv6 traffic. Accordingly, our work considers tries to be built using the IPv4 prefixes.

An address lookup involves traversing the complete trie starting from the root to determine the next hop. We call this traversal process a full-trie lookup. It can be anticipated that this process will be expensive if a lookup has to access multiple trie nodes during its traversal before it finds the next hop. Further, most lookups match prefixes of length greater than 18 [13]. That would mean that most lookups have to perform at least 18 memory accesses before finding the next hop. Thus, the nature of the trie as well as the traffic pattern emphasize the need to reduce lookups that do a full-trie lookup.

In this thesis we discuss two different architectures that aim to reduce the number of full-trie lookups. They are:

- Prefix cache-first architecture: This particular architecture consists of a prefix cache and the dynamic substride cache (DSC) where the DSC is placed between the prefix cache and the low-level memory containing the trie. A sketch of the architecture is given in Figure 1.2. The two caches combined work much better than an architecture that contains only a prefix cache. We introduce the DSC in Section 1.3.2.

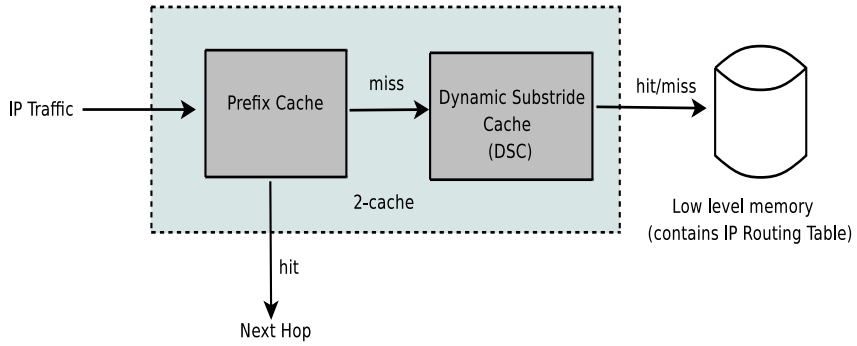

Figure 1.2: Prefix cache-first architecture

- DSC-first architecture: We alternatively look into an architecture where the prefix cache is placed between the DSC and the low-level memory. The behavior of the two caches in this architecture is quite different from the prefix cache-first architecture. A sketch of the architecture is given in Figure 1.3.

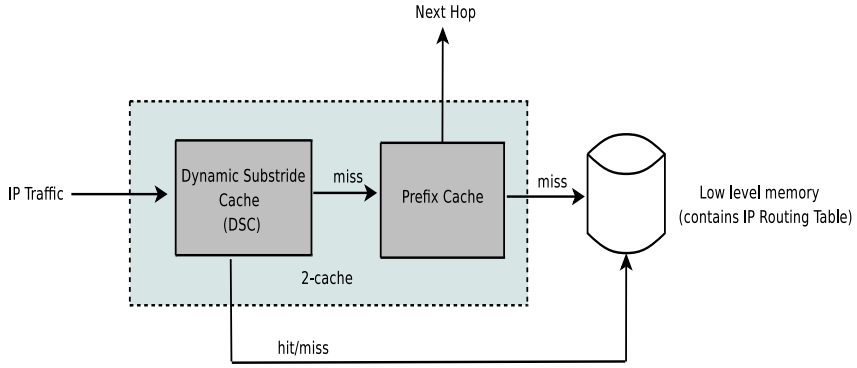

Figure 1.3: DSC-first architecture

1.3.2 Dynamic substride caching

The two architectures incorporate an unique cache organization called the dynamic substride cache (DSC). The DSC is different from the prefix cache or the IP address cache in that it contains substrides. Briefly, substrides are shortened prefixes that are derived from prefixes obtained during the previous lookups over the trie. For example, if we found that the prefix 10001101* was looked up in the trie, we store the substride 10001* in the format <substride, node address> in the DSC where the node address is the address of a node within the trie. This substride is obtained by reducing the height of the branch represented by prefix 10001101* by 3 (*shortened prefix*) as shown in Figure 1.4. In this format, the node address is the address of the node at height - 3 (represented *1xA*) within the trie from where the lookup can proceed. Later if a new lookup matches the substride <10001*, node address> present in the DSC, it will allow the new incoming packet to skip the substride 10001* and

start the lookup from the node at height - 3 within the trie. This prevents full-trie lookups starting from the root of the trie. The issue of determining the height of the substride is studied and results are presented in this thesis.

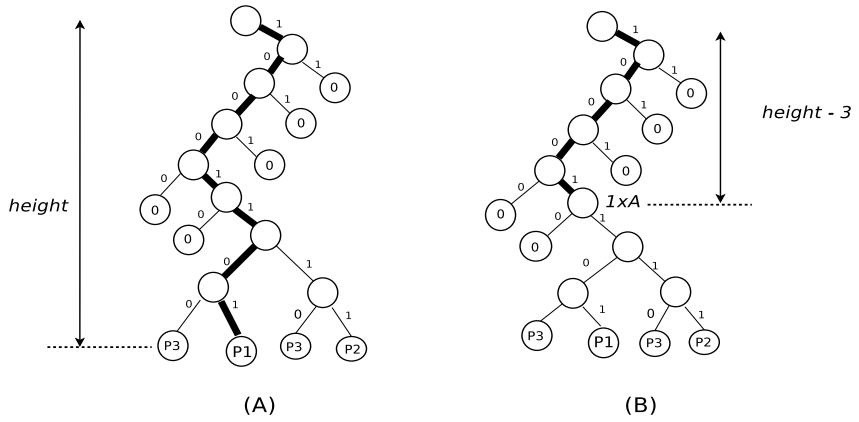

Figure 1.4: (A) The height of a branch (B) Branch at height - 3

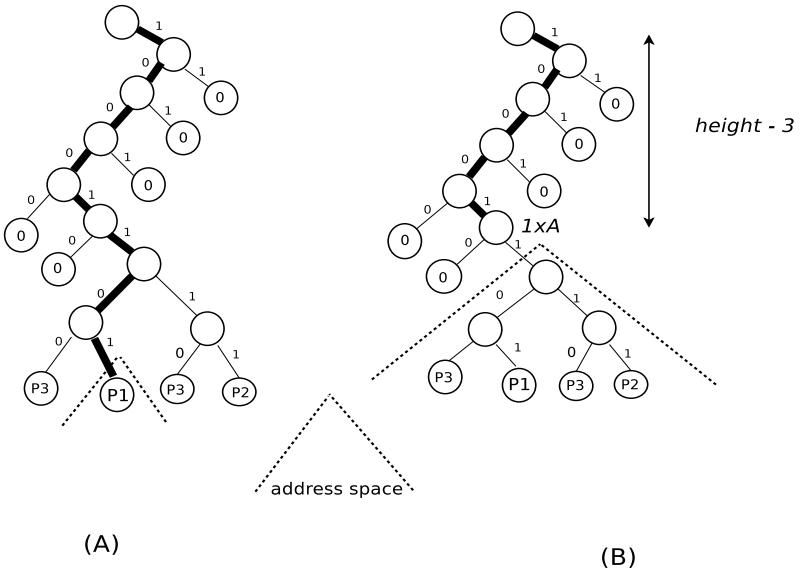

Figure 1.5: (A) IP address space coverage of a prefix (B) IP address space coverage of a substride

The DSC is effective because it covers a wider “IP address space”. The substride 10001* in Figure 1.5(B) has a wider IP address space coverage than the prefix 10001101* in Figure 1.5(A). As a result, there is a good chance that the lookups that miss the prefix cache for the prefix 10001101* may find the substride 10001* useful and can skip 10001* to proceed from node 1xA.

1.3.3 Preliminary observations

Apparently the DSC has different functions in the two architectures. In the prefix cache-first architecture, the prefix cache is more likely to benefit from locality in the destination address stream. In

this situation, the DSC cannot benefit from locality. As a result, the DSC plays a different role in assisting lookups that fall in particular IP address spaces.

In the DSC-first architecture, the DSC resolves lookups resulting from locality as well as lookups that fall in particular IP address spaces. The dual responsibility has a marked effect on the performance of the DSC. The prefix cache in this case acts as a container and holds on to prefixes relating to victim substrides that were aged-out from the DSC. This aids lookups that may re-appear in the IP traffic after a length of time.

A possible question that may arise on the use of a DSC in a architecture is the cost (in dollars) associated with placing a new hardware (DSC) in an already existing prefix cache architecture. Will the cost of such hardware be acceptable ? Are there any alternative architectures that are more worthwhile considering the implementation costs ? The answers to these questions are very important, however, these comparisons are not within the scope of our thesis work.

1.3.4 Optimal design for the architectures

Through experiments we found that using the two caches in conjunction works well in reducing the number of full-trie lookups. Still, it is not always clear how much memory should be allocated to the prefix cache and the DSC when the total available memory is fixed (Figure 1.6). One simplistic approach investigated in this thesis is to allocate equal cache memory to both the prefix cache and the DSC. However, extensive experiments on different datasets convinced us that the design decision of having "equal allocations" for the caches is not necessarily the best.

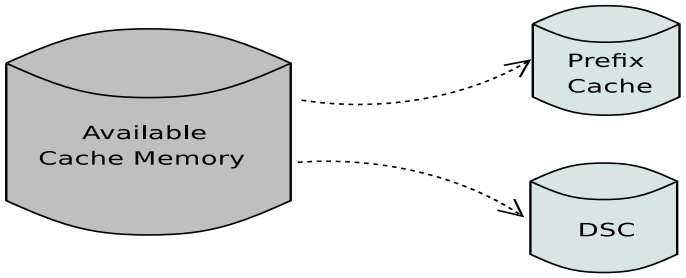

Figure 1.6: Cache memory distribution

A straightforward approach to determine the appropriate sizes for the two caches in the two architectures is exhaustive search. This process is tedious and we need to re-run the search for every single dataset. As an alternative, we developed an analytical method for optimal design based on a non-linear integer optimization technique. The primary task of the design method is to predict the hit rates for the two caches under varying cache size. Once we have predicted the hit rates, we compute the global hit rate to determine which combination of cache sizes gives us the best global hit rate. Specifically, global hit rate is the total number of hits in the two caches as a fraction of the total number of address lookups. Broadly, the higher the global hit rate the lower will be the number of full-trie lookups.

To determine the optimal distribution of memory between the prefix cache and the DSC, we analyze the IP address traces and the selected routing tables. We first try to capture the temporal and spatial locality of the traces. We follow by choosing a suitable model that will best replicate the locality properties of the traces. Thereafter, a constrained optimization technique is used to decide the optimal memory distribution between the two caches. Given the different behaviors of the two architectures, we use a different optimization technique for each of them.

1.3.5 Incremental updates - caches and trie

Incremental updates consisting of insertions or deletions are frequent. Updates are complex and require that the process be as fast as possible. In addition, during incremental updates there should be no need to completely recreate the data structure.

Incremental updates become more complex in architectures that maintain "extra" information about the IP routing table. In our case, the extra information constitutes the prefix cache and the DSC. For our architectures, updates should be reflected in the prefix cache [8] and the DSC as well as the trie that stores the IP routing table. The DSC needs to be searched for substrides that are inconsistent with the restructured IP routing table. In addition, we need to update the trie without taking a considerable amount of time. Given that we do not have have much control information in the DSC, we follow a conservative approach to reflect changes due to updates. Further, we also show how storing control information in the leaves and internal nodes of the trie can aid insertion and deletion of prefixes. It should be noted that our approach to incremental updates is applicable to both the prefix cache-first and the DSC-first architectures.

1.4 Other contributions

We also present a subsidiary work on improving the hit rates of a prefix cache. Various methods have been proposed to increase the hit rates of a prefix cache [26, 40]. This is primarily achieved by increasing the IP address space captured by a prefix. One method to increase the IP address space for a prefix is to reduce the trie [26]. In this thesis we present an alternative method that uses "skip-nodes" to reduce a trie. Skip-nodes are specialized nodes in the trie that store skip-strings to reduce the trie size.

A simple example of using skip-nodes is shown in Figure 1.7 which allows us to reduce the size of the trie. For instance in Figure 1.7, the skip-node contains the skip-string 101* which is sufficient for us to decide the packet's next hop (P2 or P5). However, a different architecture will be required to store prefixes involving skip-nodes. This is detailed in subsequent chapters.

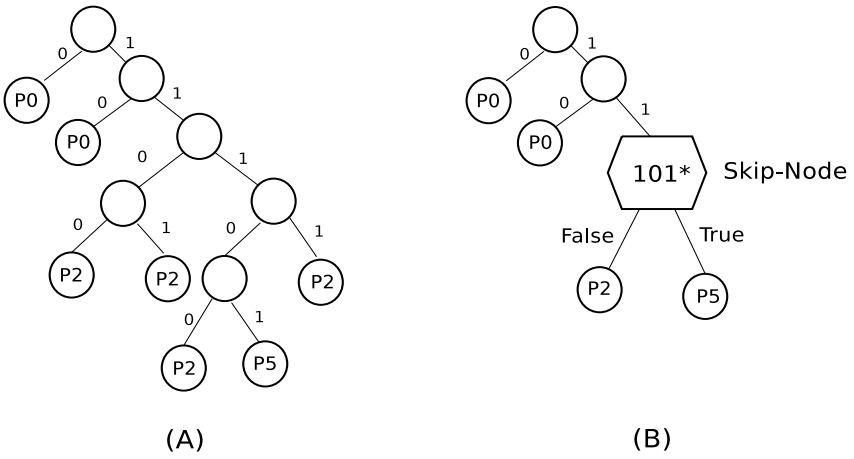

Figure 1.7: (A) Trie representation for prefixes (B) Reduced trie using skip-node which is encoded by skip-string 101*

1.5 Average-case performance

Throughout this document we stress the importance of considering the average-case lookup speed as a performance criterion. This is not to say that worst-case lookup speeds are not important. Worst-case lookup speeds ensure that all lookups complete in a acceptable amount of time. However, we emphasize that improving average-case lookup speeds is also beneficial and should be considered a vital performance metric for any lookup architecture.

It is evident from looking at our architectures that not all lookups will find hits in the two caches. Consequently, there will be lookups that will still have to do a full-trie lookup. As a result, the worst-case lookup speeds are not necessarily decreased.

A good average-case performance can be extremely beneficial for the performance of small and medium-sized routers used in campus and enterprise networks [14]. These networks support applications such as firewalls, virus-scanning, file transfers that involve packets of different types and with different processing requirements. Supplying worst-case processing times to networks supporting such different packet types is expensive. Further, reducing the average lookup time for each address will decrease the amount of time a resource (e.g hardware threads, memory) is held during lookups. This will benefit other applications in the router competing for the same resource.

An interesting example is that of matching query patterns with strings in a database. Assume we use linear search and trie search for the above purpose. Linear search will simply match each query pattern with the strings in the database. On the other hand, trie search will lookup the query pattern in a trie built from strings in the database. Clearly the trie search performs better in the average case, however, asymptotically both the methods perform the same in the worst-case. Thus, trie search should be preferred over linear search.

Consider another example where the number of memory accesses required by six different pack-

ets are 8, 8, 7, 8, 7, 6 where the worst-case bound is 8 memory accesses. On average the six lookups will need 7.33 memory accesses whereas in the worst-case each will require 8 memory accesses. Now if we can some how reduce the number of memory accesses (say, using our architectures) for the six lookups to: 2, 2, 4, 5, 6, 8 then the average number of memory accesses is reduced to 4.5 while the worst-case is still 8. The method has improved the average case significantly, but the worst-case has stayed the same.

1.6 Thesis overview

In this document we discuss different techniques to reduce the number of full-trie lookups. Recognizing that reducing full-trie lookups can help decrease the network latency associated with IP routing table lookups, we introduce methods and improvements that are relevant and effective. Our work primarily focuses on taking advantage of the fact that lookups tend to refer few IP address spaces. To this end, we show that the two architectures are interesting approaches to reduce full-trie lookups.

Before we present our work in detail, it is also extremely important to understand how the problem was tackled in previous work. There are a number of interesting ideas and concepts that were exploited in previous studies on IP routing table lookups and we illustrate them in Chapter 2. These previous works provide extensive details on the nature and pattern of IP traffic as well as the IP routing table. They also bring to notice the importance of incremental updates and lookup speeds. We also briefly present some popular approaches to representing IP routing tables.

We then proceed to explain our work in detail in Chapter 3. We first introduce the traditional prefix caching method, its utility and move towards explaining the need for encoding additional information in the trie that would aid incremental updates. Then, we introduce our idea of using skip-nodes to reduce the trie size and detail its hardware requirements.

Next, we present the overall design of our architectures in Chapter 4 and general strategies that should be followed to get the best results. We also discuss the importance of tuning certain parameters for the DSC in order to achieve desired results. Additionally, incremental updates are critical and any/every architecture should be able to support them. Subsequently, in Section 4.4 we illustrate our idea for incremental updates for the entire architecture using suitable examples.

We present our experiment results in Chapter 5 by choosing the cache sizes in some naive ways. However, it is important to decide the best memory configurations for the two caches. To achieve that, we first find an analytical model that help us in making those decisions. Chapter 6 is dedicated to our model with some explanations as well as experiments. Further in Chapter 6 we outline a constrained optimization method that incorporates the chosen models to determine the optimal memory distribution for the two caches.

Finally, in Chapter 7 we summarize our work.

Chapter 2

Related Work

2.1 Problem statement

In the previous chapter we briefly discussed IP address lookups. We define the problem of IP address lookup as determining the best (longest) matching prefix for a packet based on its destination address. A typical IP routing table with prefixes would appear as in Figure 2.1. For example, a packet containing the destination address 172.192.14.70 will have 172.192.14.0/24 as the longest matching prefix. The packet will then be routed to the next hop 2. Though the process seems straightforward, the longest prefix lookup is much harder given that the arriving destination address does not carry the length of the longest matching prefix. Hence, the lookup requires search to be among the space of all prefix lengths. The problem is further aggravated with the increasing routing table sizes. We therefore require longest prefix searches to be as fast as possible. IP address lookup speedups can be achieved by using a prefix cache where lookups for destination addresses showing locality can be looked up within few clock cycles [3]. However, not all destination addresses will find a successful match in the prefix cache. As a result, the unsuccessful lookups in the prefix cache may have to perform multiple memory accesses to complete the lookups. We explore the idea of using the DSC that may ensure that these unsuccessful lookups require fewer memory accesses to complete compared to a full-trie lookup.

Prefix	Next Hop
172.192.14.0/24	2 (P1)
172.192.0.0/16	2 (P2)
172.188.10.0/22	16 (P3)
172.188.100.0/18	8 (P4)

Figure 2.1: Example routing table

2.2 Related research

In the following sections we introduce some proposals that are aimed to tackle the IP address lookup problem. These techniques vary based on the type of algorithms and the type of hardware they use. We try to illustrate some research proposals (caching and others) that are relevant for the understanding of our work. We also mention some other proposals that may be necessary to get a general perspective of this research area.

2.2.1 Linked list

The most naive solution for IP address lookup is to store all the prefixes in a linked list in the decreasing order of their prefix lengths. A destination address is compared to all the prefixes and the prefix with the highest priority (longest length) is selected. This method is storage efficient requiring only *O(N)* storage locations (*N* is the number of prefixes). However, the lookup time scales linearly with the number of prefixes.

2.2.2 Trees and tries

A simple algorithmic solution is the trie [25] as shown in Figure 2.2 (A). Tries are used to represent IP routing tables as a tree where the results (the next hops) are either stored in the internal nodes or the leaves. Internal nodes that do not store next hops are regular nodes and are used for traversal decisions. During the traversal, regular nodes are examined to decide whether to proceed to the left branch or the right branch. A "0" as the next bit in the address means that the traversal should proceed to the left branch and a "1" means the traversal should proceed to the right branch. The path starting from the root to a leaf node gives the bit string that represents a prefix and a path from the root to an internal node containing a next hop also represents a prefix.

Some drawbacks of tries are that they take a lot of memory. Further, the worst-case depth of the trie is 32 and a lookup may require 32 memory accesses in the worst-case to determine the next hop. However, tries are well suited for research in the area of caching techniques because no complicated piece of hardware is needed to store them.

One extension to the trie is the multibit trie [35, 44]. Multibit tries are compressed tries which ensure that the worst-case depths are much lower than 32. The multibit compresses the trie where the internal nodes are densely populated. The idea is to allow a packet to skip multiple bits during traversal. These multiple bits are called strides. Instead of having just two child nodes, a multibit trie can have 2^n children where n is the fixed length stride. For example the fixed length stride of the multibit trie in Figure 2.2(B) is 2. An altogether different approach can be to have variable length strides where the stride length is not fixed as shown in Figure 2.2(C). Both the fixed length and the variable length strides reduce the number of memory accesses required by a lookup to determine the next hop. Even though the internal nodes become more "flexible", managing such tries is difficult.

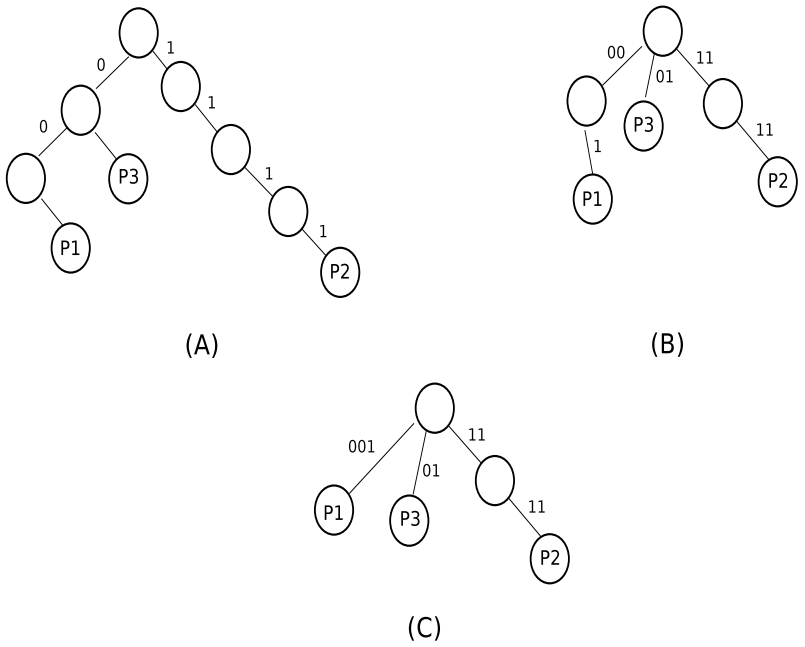

Figure 2.2: (A) Unibit trie (B) Multibit trie with stride length 2 (C) Multibit trie with variable stride length

Incremental updates are time consuming especially in the case of tries with variable length strides and updates often result in rebuilding the whole data structure.

Recently, Wuu *et al.* proposed a longest prefix first search tree (LPFST) [46]. Unlike the tries, the internal nodes in the LPFST tree encode prefixes. Moreover in this method, longer prefixes are stored in the nodes present at the upper level of the tree. This allows the IP lookup procedure to terminate as soon as the packet's destination address matches one of the prefixes. Prefix trees resemble heaps where the prefix present in an internal node is greater or equal in length than the prefixes present in its children.

An example LPFST is shown in Figure 2.3. Initially, the tree is considered empty and prefix 00* becomes the root as in Figure 2.3(a). Next, we insert 01* but since the the two prefixes 00* and 01* have the same prefix lengths, 00* remains the root of the tree. Thereafter, 010* becomes the root since it has the longest prefix length (see Figure 2.3(c)). After 010* has been inserted, the insertion procedure continues and 00* is swapped with 01* since 00* < 01* (in value). Similarly, in Figure 2.3(d) with the incoming prefix 0111*, the prefix 010* is pushed to second level of the tree. Since 01* is less specific than 010* and 00* < 01*, it is pushed to the right side of the tree.

It is quite evident looking at the structure of LPFST that the height of the tree depends upon the number and type of prefixes. The depth may increase or decrease depending upon the the prefix distribution. Moreover, Wuu *et al.* report that for medium-sized routing tables containing about 16000 prefixes, lookups require 18 memory accesses on average.

Mehrotra *et al.* [28] developed a method that organizes the prefixes as a binary search tree. To

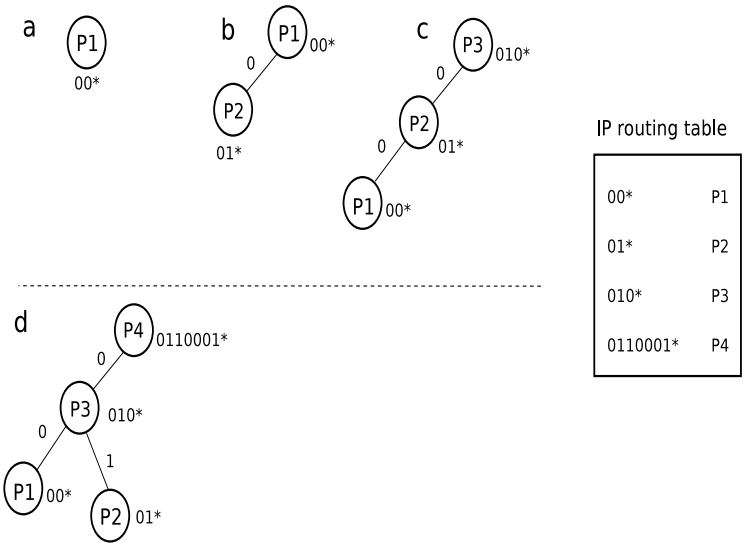

Figure 2.3: An example of LPFST (reproduced from [46])

build the binary search tree the following two conditions were used:

1. Given two prefixes $A = a_1a_2.....a_m$ and $b = b_1b_2.....b_m$, if $A \subset B$ then A is a parent of B.

2. If $A \not\subset B$ and $A < B$ (i.e less than in numerical value) , then A lies on the subtree left of B. If A and B are equal in length, then the numerical values of the prefixes are compared. If the lengths of A and B are not same, then the longer prefix is chopped to the length of the shorter prefix and then the numerical values are compared.

An example of a binary search tree is shown in Figure 2.4. Searching an address in the binary search tree, however, is not straightforward. For example, the destination address 101100*/32 lies between 011* and 101101* in the tree and it becomes difficult to determine the next hop. This is because the tree does not contain sufficient information about the parent prefixes. Mehrotra *et al.* provide an extra field for each prefix known as the Path Information Field (PIF). This PIF indicates the parent prefixes for each of the prefixes. So for example, during the binary search the address matches the prefix 101101* up to 5 bits (longest) and therefore we look into the PIF of 101101* which indicates that 1011* is the parent prefix. The parent prefix 1011* is considered the longest matching prefix and the packet is routed to the next hop associated with that prefix. PIF's are bit strings where a set bit in any bit position means that there is a parent prefix up to that bit position. In addition, the prefixes and the PIF are stored in an array.

Unfortunately, the above technique has a large build time. First, all the prefixes need to be sorted before a binary search tree can be built. Further, the authors specify a list of preprocessing steps that need to be performed to build the binary search tree. Deletion of prefixes is complicated since it will require a binary tree traversal followed by regeneration of the PIF field. Also, this technique will show a high average number of memory accesses.

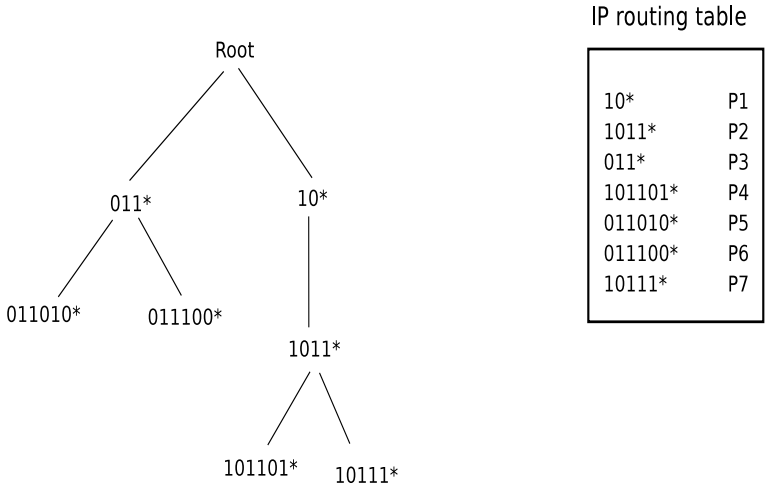

Figure 2.4: An example of binary search tree (reproduced from [28])

2.2.3 Hardware based solutions

Ternary CAM (Ternary Content Addressable Memory) is a specialized piece of hardware sometimes used in routers for performing high speed searches. Ternary CAMs, unlike binary CAMs, provide ternary logic with three matching states “1” , “0” and “Don’t Care (x)”. This allows TCAMs to store binary strings of the form “1x01” which will match both of the two binary strings 1001 and 1101.

TCAMs have been popular amongst researchers. First, because ternary CAM lookups are quick and they offer throughputs of up to 125 million lookups per second [37]. Secondly, the use of a TCAM is simple since it only needs to be prepopulated with the prefixes (see Figure 2.5). An incoming address is looked up in parallel across all the prefixes in the TCAM resulting in a number of successful matches (or outputs). Thereafter, a priority encoder returns the highest priority output.

The major disadvantage of a TCAM-based lookup is its high power consumption. Present day TCAM devices with a capacity of 18 Mb consume up to 18W of power. This is significantly more than SRAM or DRAM [31]. The other disadvantage of a TCAM is its cost. A 36 Mb TCAM will cost around 800 US dollars, while the same amount of SRAM will cost 80 US dollars. While TCAMs are expensive for storing a full IP routing table, a smaller sized TCAM (say, with 9 Mb storage) is well-equipped for the purposes of caching (Chapter 3) and is not as expensive.

Zane *et al.* [47] provide an index-based TCAM architecture to reduce overall power consumption. The idea is to divide the TCAM into buckets obtained by using a partitioning algorithm. An initial lookup uses hashing which specifies the buckets that should be further investigated to complete the lookup. This reduces the number of “active” buckets for a single lookup thereby reducing the power consumption.

The authors use a trie-based algorithm that partitions an IP routing table into a number of buckets. Based on the partitioning algorithm, a relatively large number of buckets (larger than 8) are

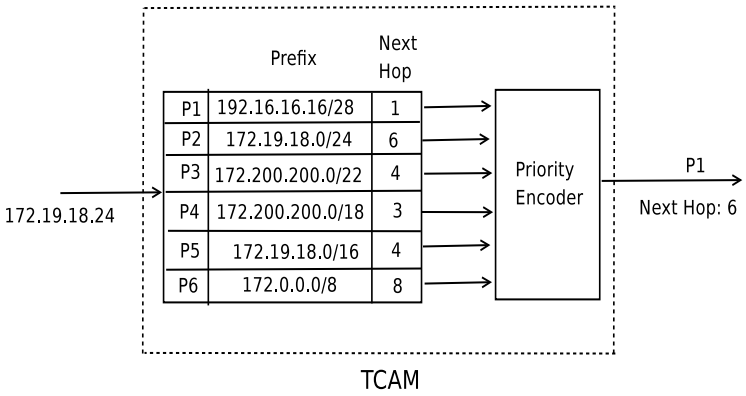

Figure 2.5: Logical design of a TCAM

produced. However, current TCAM devices allow only 8 partitions so that supplying more buckets is infeasible. Further, the architecture may not be robust to frequent updates since recomputation of hashing bits are required and the buckets need to be made consistent with the updates. In the recent few years many proposals have been made to improve the TCAM architecture itself. Some of the latest works in this area can be found in [17], [31], [48].

An IP packet forwarding based on partitioned lookup table architecture (IFPLUT) is proposed by Akhbarizadeh *et al.* [7]. The authors partition the routing table based on the next hops. For a list of ports $P = p_1, p_2, p_3, ..., p_m$ present in the routing table, their algorithm partitions the routing table into m lookup tables. These m lookup tables (LUTs) are then looked up in parallel using specialized circuitry. In addition, they remove all the parent prefixes from each of the LUT to ensure that the longest prefixes are matched.

However, the partitioning scheme may require large number of LUTs in large network access points. Though updates are simple, creating a new LUT when a "new" prefix with a "new" next hop is inserted is not feasible. The same holds true during prefix deletion where a LUT may remain empty when all the prefixes from a LUT are deleted. In addition, the authors discuss the possibility of implementing IFPLUT in a TCAM. Still, using the TCAM for the architecture will make the IFPLUT trivial. Further, IFPLUT's build time is high as all the prefixes need to be sorted based on the next hops.

2.2.4 Caching

Caching to accelerate IP lookups has been a relatively popular field of research and a number of methods have been developed to improve caching. As discussed in Chapter 1, caching prefixes will not reduce the worst-case lookup time but is extremely valuable in decreasing the average-case lookup time. Caching is particularly useful for distribution and access routers which are faced with traffic of a high degree of locality (temporal and spatial). The idea behind caching is to store recent lookup results which can be reused later for future incoming packets. In this section we introduce

historically the different developments in the area of caching for IP lookups and their contributions. Later, we also explain why it is important to have a DSC in an architecture that already contains a prefix cache. It is noteworthy to mention that the DSC can be used with any prefix cache mechanism described in this section.

Early research on caching was spurred by the work of Talbot *et al.* in [45]. The authors proposed the use of IP address caching for terabit speed routers. They attempted to demonstrate that real IP traffic exhibits temporal locality of destination addresses and developed a cache simulator to cache frequently referenced addresses. The authors showed that the cache hit rates were significant (greater than 80%) for a cache of reasonable size. However, IP address caching will demonstrate significantly lower hit rates when compared to IP prefix caching since it does not capture spatial locality.

Chiueh *et al.* in [11] developed an independent architecture for IP address caches known as a host address cache (HAC). The architecture was aimed to exploit CPU caching. The authors treated IP addresses as virtual memory addresses and developed a lookup algorithm based on two data structures: HAC and NART (network address routing table). The HAC used the Level-1 (L1) cache in the CPU. For lookups, the 32-bit IP addresses were considered as 32-bit virtual memory addresses in the L1 cache. If the L1 lookup did not succeed then the lookup proceeded to the NART. The authors treated the NART as a Level-2 cache. The NART consisted of three "hardware" tables each containing the prefixes from the IP routing table. The prefixes were assigned to these three tables using a complex indexing and tagging method. The authors improved their work on the HAC architecture by developing HARC (host address range cache) [12]. This architecture based on CPU caching is suited particularly well to a cluster of IP routers implemented using commodity shelf PCs.

As with IP address caching the hit rates for IP address caches will always be lower than those for prefix caching. Also, inconsistent virtual-to-physical address mapping will result in conflict misses. Further, supporting incremental updates is difficult due to the complex indexing as well as the full prefix expansion scheme.

Cache hit rates can be maximized by providing plenty of cache entries to hold as much information from recent results. However, providing too many cache entries for the cache may not be practical. Nonetheless, there are alternative techniques of "splitting the cache" that are reported by Shyu *et al.* [40] and Chvets *et al.* [13].

Shyu *et al.'s* aligned-prefix caching (APC) splits the cache based on the prefix lengths. APC splitting creates two caches - aligned-24 cache that caches all prefixes of length 24 or less and aligned-32 cache that caches all prefixes of length between 24 and 32. Since (probably) most of the traffic matches prefixes of length greater than 24, the aligned-24 cache is not (overly) polluted by prefixes of length less than 24 and therefore can be much better utilized. As a result, APC shows high cache hit rates. Still, the scheme is less flexible (or dynamic) since it is not certain that the traffic will always prefer prefixes of length greater than 24. MacGregor [27] however proposed a dynamic approach to split the cache. The "splits" based on prefix lengths are no longer fixed but

vary depending upon the traffic.

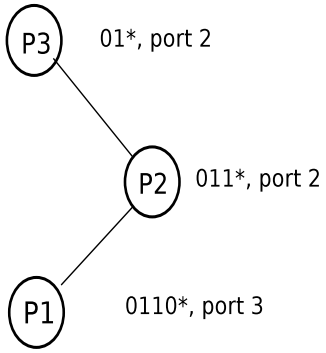

Figure 2.6: Pruning example (reproduced from [26])

The idea of increasing the cache hit rates by routing table compaction (reduction) was first explored by Liu [26]. They demonstrated that routing tables can be reduced by 48% by using - pruning and mask extension. A simple example of pruning is shown in Figure 2.6. The prefix P3 is a subset of prefix P2 and they have the same next hop. Thus P2 is redundant and can be deleted from the IP routing table. The second technique is mask extension that uses the EXPRESSO tool [34] for logic minimization. The minimizer combines two compatible prefixes into a single routing prefix entry. For example, the two prefixes 1100 and 1101 can be combined as 1x00 where x is the don't care bit. The two techniques result in cache hit rates increasing by nearly 15%. However, the logic minimization process using EXPRESSO is slow. As a result, the routing table build time is very high.

Kasnavi *et al.* [23] developed a multizone pipelined cache (MPC), a two-cache architecture comprised of a TCAM (Ternary CAM) and a CAM. The TCAM is used to cache the prefixes and the CAM is used to cache the IP addresses. In terms of memory, they provide an equal number of "entries" to both the TCAM as well as the CAM. Basically, their method can be termed a half prefix/half full-address cache. In their architecture, the CAM is used to exploit temporal locality and the TCAM is used to exploit spatial locality. They further partition the CAM into two equal sized CAMs, each storing 16 bits of the 32 bits of the IPv4 address.

The authors state that matching the first 16 bits of the cache will help decide whether it is worth looking (matching) the next 16 bits of an IP address which may reduce latency during CAM lookups. Further, they use Chvets *et al.'s* [13] multizone method to split the two caches (though not a lot of information is provided regarding the the configuration/scheme they use for multizone caches). For the pipelining in the MPC architecture, they place a buffer between the two caches and the IP routing table. This buffer stores recent lookup results obtained from the two caches. The primary use of the buffer is to prevent stalling of lookups in the two caches when the lookup proceeds to the slow IP routing table. While the IP routing table lookup is performed, the two caches continue lookups and store all the unsuccessful matches in the buffer. Once the IP routing table lookup is finished, the

lookup result is used to clear the buffer to ensure similar lookups need not proceed to the IP routing table. This essentially reduces the latency that exists between the two caches and the IP routing table.

Apart from the hardware-based improvements suggested by the authors for the two caches, they employ a short-prefix expansion (SPE) technique on a trie (storing the IP routing table) where they expand all the prefixes until they become disjoint prefixes (or leaves) or 17 bits in length.

Understandably, the above architecture does not scale well in terms of performance and has nearly twice the miss rates of a prefix cache. In any event, a prefix cache will always show a better hit rate than an MPC. Further, the SPE technique, unlike full-trie expansion [44], has no clear benefits and may only marginally improve the MPC cache hit rate since it provides a slightly better chance to cache "parent" prefixes. In addition, it is not clear whether splitting of the two CAMs is useful since jumping from one CAM to another will incur latency which may outweigh the advantages of having two CAMs.

Uga *et al.* in [39] proposed storing intermediate nodes of a Patricia tree [36] in CAMs. The authors propose using three CAMs that store prefixes of different lengths. More specifically, the authors prepopulate prefixes of length 8, 16 and 24 in three different CAMs. When a packet arrives, the intermediate nodes are looked up in the three caches using the first 8, 16 or 24 bits of the packet. The highest priority match is selected and the search then continues by jumping at the intermediate node (found from the cache) within the Patricia Tree.

However, it is not possible to hold all the intermediate nodes especially in the case of a CAM storing prefixes of length 24. This is true especially for a medium-sized routing table (~30,000 prefixes) that may easily show about 15,000 prefixes (about 50%) with lengths 24 or more [29]. In order to cope with the above problem, prefix aggregation technique is used where prefixes of length 8 to 15 are aggregated into prefixes of length 8. Likewise, the prefixes of length 16 to 23 and 24 to 32 are aggregated into prefixes of length 16 and 24 respectively. For example, if there are three prefixes: 10.64.0.0/12, 12.96.0.0/14 and 12.32.0.0/15, then the second and third prefix are aggregated resulting in the prefix 12.0.0.0/8. The first prefix 10.64.0.0/12 is stored as 10.0.0.0/8.

Unfortunately, the build time of the method is extremely high. To perform the required aggregation, three different exhaustive (entire Patricia tree) depth-first traversals will be required for the prefix lengths 8,16 and 24. Further, incremental updates can become complicated considering that aggregated prefixes in three CAMs can become inconsistent with the addition or deletion of prefixes in the Patricia tree.

Peng *et al.* [32] proposed a supernode caching scheme to efficiently reduce IP lookup latency in processors. Supernodes are nodes of a tree bitmap [15]. A tree bitmap is a compressed subtree that reduces the number of levels in a tree. During IP lookups, recently visited supernodes in the bitmap tree are stored in a SRAM based cache. These supernodes are later used by subsequent lookups. Still, the highly compressed tree bitmap by itself (without the cache) ensures that lookups require

only 4 or less memory accesses. So, there is no substantial benefit in having a cache for supernodes. Our approach is different in that we cache the longest possible branches at minimal height from the leaf of the trie to obtain the longest possible jumps within the trie.

RRC (Reverse Routing Cache) was proposed by Akhbarizadeh *et al.* [8]. RRC employs a minimal expansion (ME) technique to deem parent prefixes as cacheable. "Parent prefixes" are prefixes that are a prefix of other prefixes. For example, 172.19.24.0/8 is a prefix of 172.19.24.0/24. In prior work it was illustrated that caching of parent prefixes should be avoided as it could result in incorrect lookups. For example, if we cache prefix 172.19.24.0/8, then subsequent lookups for the subnet 172.19.24.0/8 will find a match. However, a lookup for the subnet 172.19.24.0/8 will match the prefix 172.19.24.0/24, yielding an incorrect result.

The Minimal Expansion (ME) technique proposed by the authors provides a more intelligent way to handle parent prefixes. A simple example of ME is shown in Figure 2.7 where the parent prefix is 1*. When an address (e.g, 1110) is looked up in the trie, we first find that the parent prefix 1* is a match. However, we cannot place 1* in the cache. As a result, we continue the traversal and generate the prefix n = 11*. Again, 11* is not cacheable since there exists a more specific prefix 110* in P1. We then add the bit "1" (third bit of the address 1110) to the prefix n = 11* . The expanded prefix 111* is therefore a match for 1110. Since, the minimally expanded prefix 111* is not a parent prefix, we can safely cache the prefix 111*. Thereafter, packets destined to the subnet 111* find a match in the cache.

Though the authors provide a suitable method to cache parent prefixes, their method cannot avoid full-trie lookups for a significant percentage of the packets. For example, for a cache size of 128 entries, they show that 92% of the packets were looked up in the cache. That is, 8% of the lookups still had to do a full-trie lookup. However, in later chapters we present our architectures (prefix cache-first and DSC-first) which are primarily aimed at reducing the number of full-trie lookups.

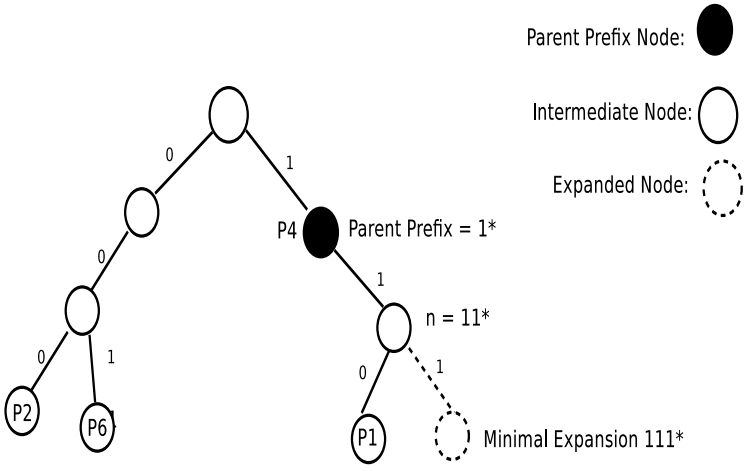

Figure 2.7: Minimal Prefix Expansion (reproduced from [8])

Chapter 3

Prefix Caching

A prefix cache is an integral part of our architectures (prefix cache-first and DSC-first) and we dedicate this chapter to describe the workings of a prefix cache and a prefix caching scheme that uses skip-nodes. Skip-nodes help reduce the size of the leaf-pushed trie and helps increase the prefix cache hit rates. The technique of using skip-node especially for prefix caching is not tried out previously.

We first outline the overall process to build trie using skip-nodes as well as the architecture required for it to be feasible. In addition we provide various metrics to evaluate the performance of a prefix cache. We also detail the experimental results and its implications.

3.1 Leaf-pushed tries

As discussed earlier, tries can be used to represent prefixes in a routing table. A prefix is simply a path in a trie starting from the root to a node that contains a next-hop. We call nodes containing a next-hop as decision nodes. In addition, we refer to all nodes other than leaves as internal nodes. Further, a decision node can be an internal node of a trie as well as its leaf as shown in Figure 3.1(A). Before we get into the details, we introduce some notation that we will be using in subsequent sections.

Suppose $R = r_1, r_2, ..., r_N$ is a set of N prefixes and $H = h_1, h_2, ..., h_M$ is a set of M unique next hops. Then we represent a single entry in a routing table R by the prefix-next hop pair $< r_i, h_i >$. The following rules are followed during an address lookup as well as during the construction of a trie:

- For an entry $< r_p, h_p >$ in R, r_p is a parent prefix if there exists an entry $< r_i, h_i >$ in R such that r_p is a prefix of r_i and the length of r_p is less than the length of r_i. The above rule holds irrespective of the values of h_p and h_i.
- A parent prefix r_p of a prefix r_i in R has a lower priority than prefix r_i during address lookups.

- For two entries $< r_i, h_i >$ and $< r_j, h_j >$ in R, the prefixes r_i and r_j are independent if neither of them is a parent prefix of each other.

- If there exist two entries $< r_i, h_i >$ and $< r_j, h_j >$ in R such that the prefix $r_i = r_j$ (similar) and the length of prefix r_i is equal to the length of prefix r_j then both the prefix r_i and r_j have equal priority.

In a trie, a parent prefix r_p is represented by a path from the root to a decision node that is an internal node. In Figure 3.1(A), the prefix $r_p = 10*$ is the parent prefix of 1001* and 101*. Take a hypothetical case where a packet with destination address 10001 arrives at the router ingress. Looking at Figure 3.1(A) and following the rules stated above, we traverse up to the leaf node storing 0. Since the traversal had already seen the parent prefix $r_p = 10*$, we route the packet to $h_p = P3$.

However, we use a different scheme to represent our routing table. We use a leaf-pushed trie where the next hops of the parent prefixes are pushed to the leaves of the trie instead of storing them in the internal nodes [44]. The next hop h_p of a parent prefix r_p is pushed within the subtrie rooted by the internal node that was containing the next hop h_p. For example, in Figure 3.1(A) the internal node containing the next hop $h_p = P3$ is a root of a subtrie. During the leaf-pushing process, if a leaf turns out to be a decision node, we do not push the next hop $h_p = P3$ to that leaf. On the other hand, if a leaf is storing a 0, then we replace 0 with the next hop $h_p = P3$. An example of a leaf-pushed trie is illustrated in Figure 3.1(B) where the parent prefixes 10* and 0* are pushed to the leaves.

During prefix caching, we keep note of the prefixes that were generated from the trie during address lookups. For example in Figure 3.1(B), for a packet with destination address 10001, we generate the prefix 1000* during the traversal. Thereafter, we store the pair $< 1000*, P3 >$ in the prefix cache which can be used for future address lookups. It should be noted that the leaf-pushed trie for prefix caching is an extension of the work done in [8]. For example in Figure 3.1(B), a traversal along the path 01110 generates the minimally expanded (ME) prefix 01*. Then the pair $< 01*, P6 >$ can be safely stored in the prefix cache since 01* is an independent prefix. Even though leaf-pushing increases processing during trie build, we show in Chapter 4 that the leaf-pushed trie is indeed necessary for architectures containing the DSC.

Apart from storing the next-hop h_i in a leaf for a prefix r_i, we also store the prefix lengths as shown in Figure 3.1(B). This additional control information is useful during incremental updates and is discussed in detail in Section 4.4.

Traditionally, a trie with no leaf pushing is preferred over a leaf-pushed trie since it is convenient to delete parent prefixes during incremental updates. For example, if the routing table entry <10*, P3> needs to be deleted from the trie, we traverse up to the internal node storing P3. Thereafter, we delete P3 from the internal node and make it a "non-decision" node. The worst-case complex-

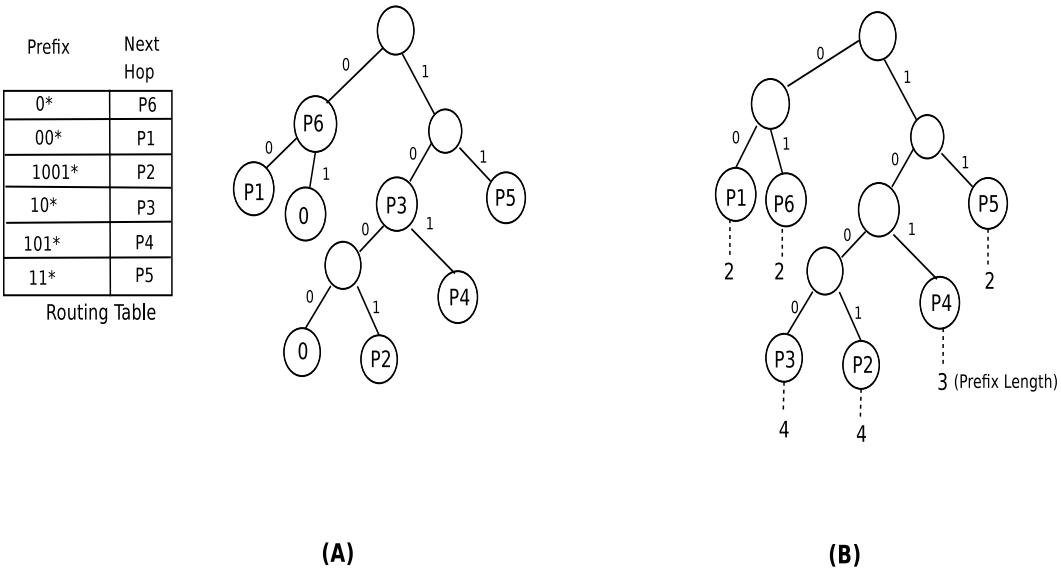

Figure 3.1: (A) Traditional Trie (B) Leaf-pushed trie with prefix length encoding

ity of deleting a prefix from a trie is *O(W)* where W is the worst-case length of a prefix (i.e 32). Nonetheless, we show in Chapter 4 that parent prefixes can be deleted from a leaf-pushed trie with a marginal increase in complexity.

A major component that affects the performance of a cache is the replacement strategy. In our work we used least recently used (LRU) stack. The primary advantage of a LRU stack is that it is well suited to exploit locality due to packets arriving in short bursts. As per the strategy, the most recently referenced (matched) prefix is moved to the top of the cache while other prefixes are moved down by one position in the cache. This ensures that popular prefixes remain near the top of the cache. At the same time, not so popular prefixes tend to remain at the bottom of the cache and gradually age out from the cache. We also tried FIFO (First-In First-Out) replacement strategy where a prefix at the top of the cache gradually moves down the cache with time. However, prefixes that have a chance to become popular once again (because of similar "previous" bursts) may get aged out from the cache. As a result, the cache will show relatively lower hit rates. This was evident from empirical results where the LRU stack out-performed FIFO in terms of hit rates.

3.2 Prefix caching using skip-nodes

Trie compression is one way to improve the performance of a prefix cache. A compressed trie will have fewer prefixes than the original trie. This will improve the hit rates since more prefixes can occupy a single cache entry. During compression we also ensure that address lookups return correct results.

First, we compress subtries that have leaves containing the same next hops. We call this method initial compression (IC). This technique was also investigated by Kasnavi *et al.* [23]. We describe

the technique using a simple example from Figure 3.2 (A). As shown the leaves in the subtrie rooted by the nodes 1xC and 1xB contain the next hops $h_1 = P1$ and $h_2 = P2$ respectively. We compress these subtries into singular nodes containing their respective next hops (P1 and P2 respectively) as shown in Figure 3.2 (B). After the compression, the number of prefixes in the trie is reduced from 6 to 3. This scheme is simple and provides an "initial" compressed trie before we generate skip-nodes on the trie. It may at first seem that this initial compression is self-sufficient. However, we show from empirical results that by using skip-nodes we improve the performance of the prefix cache.

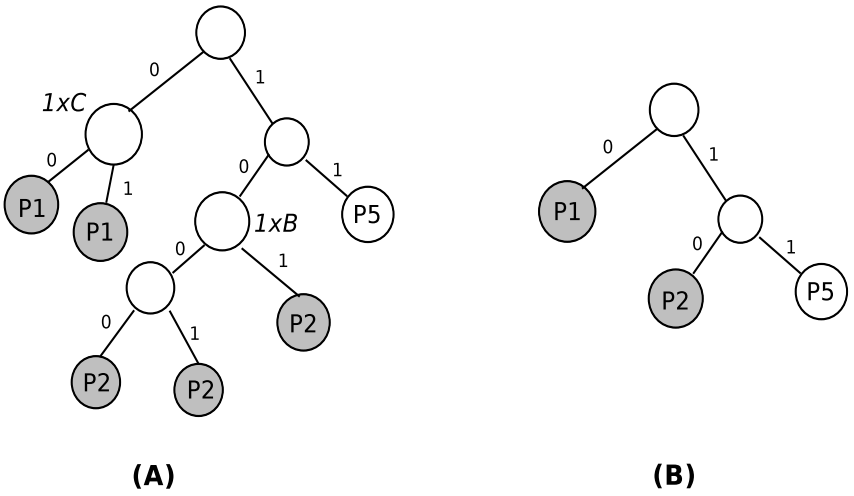

Figure 3.2: (A) Leaf-pushed trie with redundancy (B) Leaf-pushed trie without redundancy.

Skip-nodes in the leaf-pushed trie are generated after we apply the IC technique. We use the following steps to generate the skip-nodes:

- For two distinct next hops h_1 and h_2, we search the trie depth-first for subtries containing leaves that store either h_1 or h_2.
- If a subtrie has just two leaves each storing either h_1 or h_2, then the two leaves are replaced with a single skip-node.
- If a subtrie contains multiple leaves storing h_1 and a single leaf storing h_2 or multiple leaves storing h_2 and a single leaf storing h_1, then the subtrie is replaced with a single skip-node.
- Other cases that do not satisfy the above three rules are are not considered.

A simple example of a trie containing skip-nodes is shown in Figure 3.3. The two subtries that satisfy the above rules are reduced to skip-nodes containing skip-strings 01* and 101*. A lookup for a destination address 1010100 in the trie will find the skip-node containing the skip-string 101*. Thereafter, the most significant bits of the address is compared with the skip-string 101*. A match will result in the packet being routed to the next hop $h_1 = P3$. Otherwise, the packet is routed to the next hop $h_2 = P2$. The above mechanism compresses the trie by representing a group of prefixes

by a skip-node. In Figure 3.3(B) we can see that we require just two skip-nodes in place of five different prefixes.

However, we would need a different architecture for prefix caching. This can be explained from Figure 3.3(B). For a prefix cache architecture supporting entries of the form $< r_i, h_i >$ pairs, we can store the entry $< 11*, P5 >$ in the prefix cache. However, if we find P5 as the lookup result for a destination address, we cannot store the prefix 001* in the prefix cache since the architecture does not support assigning two ports (P1 and P5) for a single prefix entry. Moreover, no extra logic is available with the architecture to decide the correct next hop for a destination address (P1 or P5) that matches the prefix 001* .

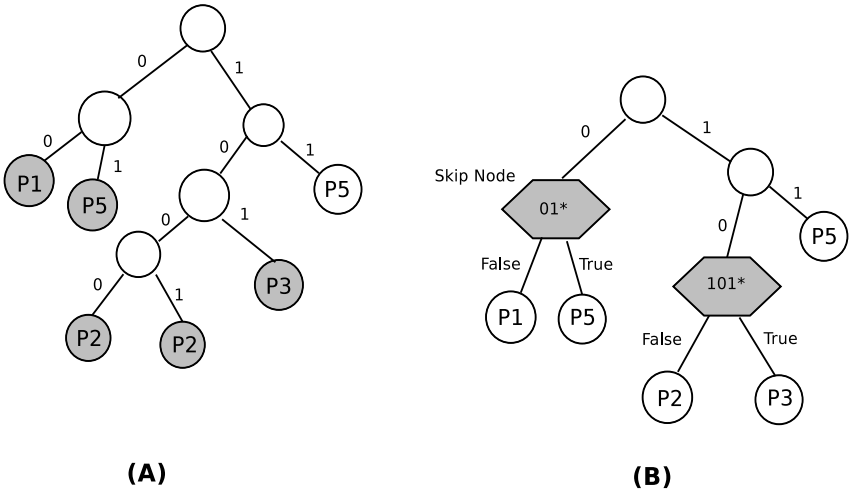

Figure 3.3: (A) Compressed trie obtained after the "initial" compression (B) Trie with skip-nodes.

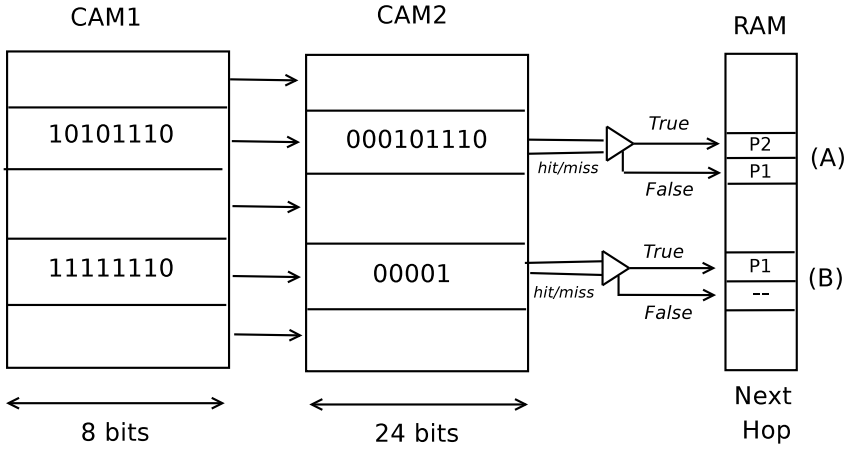

Figure 3.4: Architecture required for prefix caching using skip nodes

Alternatively, a different architecture can be used and is illustrated in Figure 3.4. Such architectures are prominent in the area of caching and have been investigated previously [23, 24]. For

our work, we use two CAMs (content addressable memories) for the cache. For caching, the first 8 bits of a prefix are stored in CAM1 and the remaining bits of a prefix are stored in CAM2. For a successful address lookup, a packet's destination address should match a single "continuous" entry in both the CAMs.

In the architecture, CAM2 can be used to store skip-strings. For instance in Figure 3.3(B) if a lookup found the skip-node containing the skip-string 101* during lookup, then we cache the skip-string 101* in CAM2 and 10* in CAM1. The logic required to choose the correct next hop is placed between CAM2 and the RAM. A match in CAM2 will cause the packet to be routed to the next hop $h_2 = P3$. Otherwise, the packet is routed to the next hop $h_2 = P2$. Further, the next hops h_1 and h_2 associated with a skip-node is placed adjacently in the RAM. For the above example, $h_1 = P2$ and $h_2 = P3$ will be placed adjacently in the RAM. The "–" in the RAM indicates that the lookup needs to proceed to the low-level memory containing the trie and is specifically meant for entries like <11*, P5> that contain a single next hop.

The problem with using skip-nodes in a trie is that we lose considerable control information regarding the prefixes. One obvious loss is that we no longer know about the prefixes that were present in the trie (i.e after the compression). Further, we can no longer reliably associate priorities to the prefixes that were compressed.

The loss of vital control information is a hindrance for incremental updates. This is true because for adding or deleting a prefix from the trie, we might have to recreate the compressed subtrie to make the trie reflect the latest routing table. For instance in Figure 3.3, it is not easy to recreate the subtrie that was replaced by the skip-node. This is true even if the subtrie had just two prefixes since we cannot be sure about the priority (prefix lengths) of the two prefixes. At the same time it is essential for any architecture to support incremental updates. Consequently, in the prefix cache-first or the DSC-first architecture we do not perform any compression of the trie .

3.3 Metrics and Implementation

There are two different metrics that are used to evaluate the performance of the technique described in this chapter.

Firstly, in order to test the effectiveness of skip-nodes, we calculate the size of the of trie after generating skip-nodes within the trie. The reason for selecting such a metric is that it indicates whether the number of prefixes in the compressed trie has indeed reduced when compared to the original trie.

The metric mentioned above is obtained by simply searching depth-first the number of prefixes in the trie. The prefix as a result of a skip-node is considered to be a single prefix since it effectively consumes just a single cache line in the two CAMs irrespective of its representation in the RAM. Similarly, we consider a prefix that does not result from a skip-node to be just one prefix. The measurements described above may slightly vary depending upon the properties of packet trace and

routing table that we employ.

Secondly, we calculate the number of times a match is found in the prefix cache. To express the numbers in fractions of the total, we count the number of matches in the prefix cache of the total number of searches in the prefix cache. For better readability, we term the fraction as the hit rate. A higher hit rate would mean that many lookups are able to find a match in the prefix cache. It should also be noted that the performance of a cache also depends upon the cache size (i.e number of cache entries). Thereby we also show how and why performances vary with increasing cache size.

To test our idea we performed experiments on different traces and routing tables (datasets). For evaluation, we first built the leaf-pushed trie without generating the skip-nodes. We then determined the number of prefixes in the trie. At first it may seem that the number of prefixes should be equal to the number of prefixes in the original (raw) routing table and so there is no need to perform this search. However, the leaf-pushing process described previously causes "prefix-expansion" [23]. This results in the number of prefixes in the trie to be larger than that present in the original routing table. Once the trie is built, we perform tests by simulating the lookups in the prefix cache. Lookups that find a hit in the cache are considered as completed lookups. Lookups that miss the prefix cache are then simulated to be traversing down the the trie. While a lookup is being performed, we simultaneously generate the pair $< r_i, h_i >$. A lookup that finds a leaf is considered complete. The pair $< r_i, h_i >$ that is generated as a result is then moved to the prefix cache.

We simulate the prefix cache using C programming language and consider the prefix cache to be residing in a contiguous memory where the beginning of this contiguous memory is the top (first-entry) of the prefix cache. Henceforth, to simulate the LRU replacement strategy, the most recently looked up pair $< r_i, h_i >$ is moved to the top of the contiguous memory and other $< r_i, h_i >$ pairs were moved down by one position. Further, we simulate the size of the prefix cache by the size of the contiguous memory.

A similar process is followed while evaluating the prefix cache using skip-nodes. This time we build a trie first applying the IC technique followed by our proposed method. Further, the prefix cache is simulated as having two contiguous regions of memory instead of just one. For the next hop information we considered a contiguous memory as well, where two adjacent memory locations stored the two next hops associated with a prefix.

3.4 Dataset

3.4.1 Routing tables

For the experiments, we use three different routing tables downloaded from different sources. These routing tables are real-world and each are drawn from different routers. This allows us to verify whether the proposed technique for prefix caching has relevance under real-world circumstances. These routing tables vary in their size and prefix length distribution.

We first used the Funet routing table that was made available in [2]. Funet is a backbone network providing Internet connections for Finnish universities and other research institutes. For our work, we downloaded a routing table that contained 41362 prefixes (medium-sized). This routing table has a significant number of parent prefixes. This gives us a good opportunity to compress the trie. Moreover, the number of distinct next hops $H = h_1, h_2, ..., h_M$ is relatively small $|H| = 11$. In addition, like modern day routers, the routing table has no prefix of length less than 8 [22].

The other two routing tables ISP1 and ISP3 are drawn from distribution routers made available by local service providers. These routing tables have fewer prefixes (10166) than the Funet table. Unlike Funet, these routing tables have very few parent prefixes. That is to say most of the prefixes are independent. The number of distinct next hops in ISP1 and ISP3 is $|H| = 32$. However, the smaller number of parent prefixes causes the initial compression to be less effective.

3.4.2 Packet traces

The real-world packet traces Funet, ISP1, and ISP3 used for the experiment are related to their respective routing tables. The three packet traces are approximately 100,000 packets in length and contain destination addresses only. The traces are different in terms of locality properties. The Funet trace has less locality compared to the ISP1 and ISP3 traces. In particular, ISP1 and ISP3 demonstrate high degree of temporal locality. It is not surprising that the Funet trace shows less temporal locality because the IP traffic at the Funet backbone router sees significantly more flows as compared to ISP1 and ISP3. A distinctly greater number of flows increases the inter-arrival times of similar packets resulting in reduced temporal locality. At the same time, the Funet trace has fewer unique destination addresses. Consequently, the Funet trace requires a smaller cache to achieve peak performance.

3.5 Results and discussion

Table 3.1 gives a fair idea of the amount of compression in the trie after we apply the two compression techniques. The amount of compression is relatively good for all three routing tables. As hinted earlier about the presence of redundancy in the Funet routing table, nearly 46% of the prefixes have been removed after applying the IC technique. This clearly suggests that there was significant redundancy in the trie where the overlap to a great extent is caused by parent prefixes. As ISP1 and ISP3 had a large percentage of independent prefixes, they performed relatively worse than the Funet routing table. However, the level of compression is still good and this demonstrates the effectiveness of the proposed method. The amount of compression can have a marked improvement on the performance of the prefix cache. As for applying our proposed method on top of the IC technique, we can see the results are still relatively good. We get nearly 20-30% reduction in the size of the trie for all three routing tables. ISP1 and ISP3 show slightly lower compression than Funet because there are more distinct next hops present in the routing tables. This reduces the probability of finding subtries

Table 3.1: Percentage decrease in trie size

Routing Table	Initial Compression (IC)	Proposed (After IC)
Funet	46%	31%
ISP1	27.2%	26%
ISP3	27.2%	26%

that satisfy the rules stated in Section 3.2. Further, the number of independent prefixes in the routing tables also affects the amount of compression.

Tables 3.2 to 3.4 shows the hit rates for all three traces for the IC technique as well as the proposed technique. For the Funet trace, the IC technique gives a reasonable improvement in hit rates. This is correlated with the amount of compression that was achieved (46%). At the same time, ISP1 and ISP3 show significantly less improvement in hit rates. This again is because the IC technique was not overly successful in performing compression. It is interesting to see that ISP3 hardly shows any improvement even though there was some amount of compression after using the IC technique. This is because the destination addresses tended to fall in the part of the trie that was not compressed. This also suggests that redundancy was not uniform throughout the routing table.

As with the proposed technique, we can see that the hit rates have increased by up to 5% when compared to the IC technique. The improvement in hit rates using the proposed technique gradually decreases with increasing cache size. This is because as the cache size grows, we get nearer to the peak performance and improving upon it gets harder. The same holds true for the performance of the IC technique. The Funet trace has a better performance since the compression achieved using the proposed method is much higher. In addition, since the Funet routing table had more parent prefixes, the chances of subtries satisfying the compression rules are greater. This is substantiated by the extent of compression resulting from the proposed technique. Further, ISP3 shows up to 4% increase in the prefix cache hit rate. This also indicates that the destination addresses tend to fall in the part of the trie that was compressed by the skip-nodes. In contrast, we see the pattern of cache hit rates for ISP1 to be similar to that for the IC technique. The IP traffic, again, does not seem to prefer the address space that was compressed using the skip node.

Table 3.2: Performance: Hit rates (%) - Funet

Cache Size	No Compression	Initial Compression (IC)	Proposed (skip-node + IC)
32	58	60	63.8
64	68.4	70.2	75.2
128	77.4	80.5	84.4
256	89	91.6	93.8
512	96.7	97.8	98.4
1024	98.9	99.1	99.3

Table 3.3: Performance: Hit rates (%) - ISP1

Cache Size	No Compression	Initial Compression (IC)	Proposed (skip-node + IC)
32	65	65	65.8
64	72	73	76
128	84	85	87.3
256	92.6	93.3	94.8
512	97	97.4	98.4
1024	98.6	98.8	99.1

Table 3.4: Performance: Hit rates (%) - ISP3

Cache Size	No Compression	Initial Compression (IC)	Proposed (skip-node+ IC)
32	51.8	52	54
64	61.5	61.5	64
128	71.9	72	76
256	83.5	83.7	87.3
512	92.6	92.6	95
1024	97.57	97.57	98.9

Chapter 4

Architecture

In this chapter we introduce the idea of using the prefix cache-first and DSC-first architectures to reduce the number of full-trie lookups. The important component in the two architectures is the DSC. The DSC exploits locality as well as assists lookups that fall in particular address spaces. We are further helped by the fact that the DSC can be used in processors that already have a prefix cache.

In Sections 4.1 and 4.2 we give the datapath for our architectures. In Section 4.3 we discuss substrides and the way we generate them. In Section 4.4, we describe incremental updates that are applicable to both the architectures.

4.1 Prefix cache-first architecture

Figure 4.1 presents the datapath for our prefix cache-first architecture. The architecture consists of two major components. The first component consists of the prefix cache and the DSC that will reside in a processor. The second component consists of the leaf-pushed trie that is made available in the low-level memory. The <prefix, next hop> pairs are stored in the prefix cache while the <substride, node address> pairs are stored in the DSC. Other components in the architecture such as $g1^{'}$, $g2$, $g3$ and $g4^{'}$ are required for providing decision logic. Further, they control the flow of information within the architecture. It should be noted that we have not applied any pipelining in the above architecture and the direction of flow of information is from left to right. In any case, the DSC always waits for the prefix cache lookup results. By information we mean the data such as IP destination addresses and lookup results. Further, information can flow through a line (say, T) if and only if it is enabled.

Initially when a destination address needs to be looked up, the incoming lines to the prefix cache and $g1^{'}$ from T are enabled. A decision in $g1^{'}$ is made only after results are obtained from the prefix cache. If a destination address finds a match in the prefix cache, the output lines are enabled. Hence, the line U is enabled and the lookup result (next-hop) is forwarded to the output interface. Since the lookup process no longer needs to be continued, the input line to $g1^{'}$ via the prefix cache is enabled. Consequently, the lookup does not proceed to the DSC. If the lookup in the prefix cache fails, the

input line to $g1^{'}$ via the prefix cache is disabled. As a result, the input signals to $g1^{'}$ remain enabled and the lookup then proceeds to the DSC.

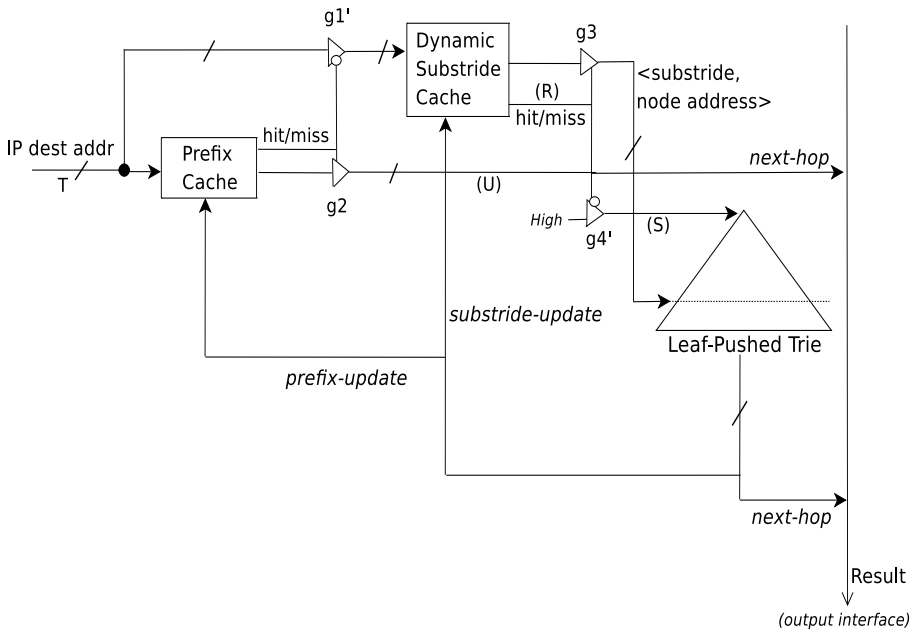

Figure 4.1: Prefix cache-first architecture

A similar logic is followed when we perform a lookup in the DSC. If a lookup in the DSC is successful, the output lines are enabled. Subsequently, lookups proceed by jumping within the trie. If line R is disabled then $g4^{'}$ will enable output line S. This will result in the lookups continuing from the root of the trie. For caching, we update the prefix cache and the DSC with prefixes and substrides (respectively) generated during trie lookups. This update happens if only if a lookup was successful in the leaf-pushed trie.

The following are the characteristics of the prefix cache-first architecture:

- In Figure 4.1 we can see that the prefix cache sees the destination address stream first. Thereby in this architecture, the prefix cache remains focused in exploiting locality.
- The DSC in this architecture is focused to assist lookups left-over from the prefix cache. Based on the empirical results, we found that the DSC benefits from the fact that most of the left over lookups fall in particular address spaces. In fact, some of these lookups that are assisted by the DSC result from compulsory misses in the prefix cache.
- The performance of the prefix cache does not depend upon the performance of the DSC. However, the DSC performance may vary depending upon the prefix cache performance.

4.2 DSC-first architecture

Figure 4.2 presents the datapath for our DSC-first architecture. The architecture is different from the prefix cache-first architecture in that the DSC is placed before the prefix cache. Moreover, the information flow is slightly different for this architecture. The main difference is that we refer to the leaf-pushed trie when the output line U is enabled. Furthermore a successful lookup in the prefix cache will give the next hop result which is immediately forwarded to the output line.

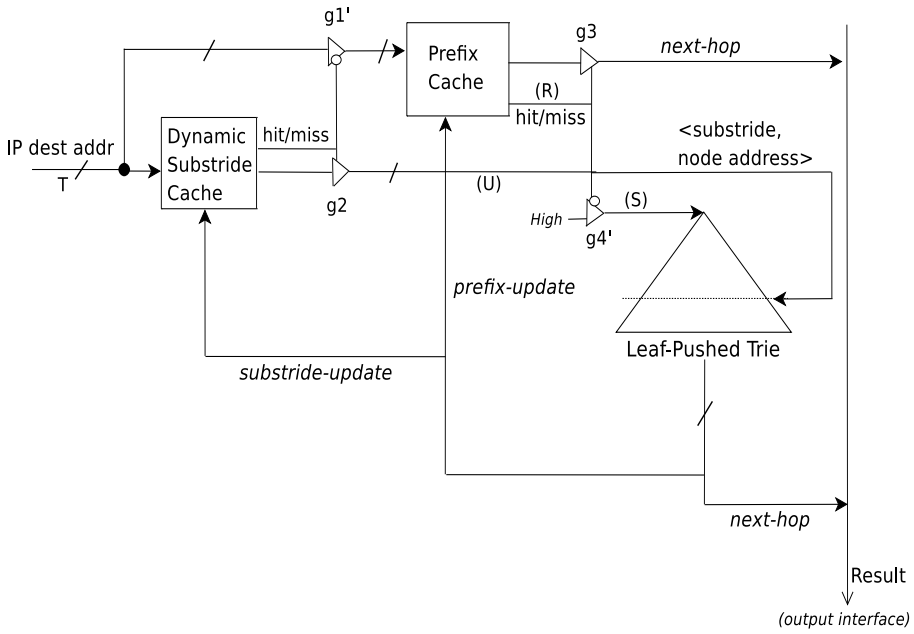

Figure 4.2: DSC-first architecture

The following are the characteristics of the DSC-first architecture:

- The DSC in this architecture sees the destination address stream first. Without doubt the DSC will exploit the locality in the destination address stream. The DSC will also be active in assisting lookups that fall in particular address spaces.
- The prefix cache in this architecture may act as a support to the DSC and may assist lookups that do not find corresponding substrides in the DSC. This behavior can be attributed to a more-active DSC and a less-active prefix cache. We discuss this in greater detail in Chapter 5 and Chapter 6.
- The two caches in this architecture are inclusive i.e the DSC performance depends upon the prefix cache and vice-versa (Chapter 5 and Chapter 6).
- It should not be surprising if we see a DSC alone performing adequately. In fact, we see this behavior for few datasets (Chapter 6).

4.3 Understanding the DSC

An important functional component in the two architectures is the DSC which is used to store substrides. Before we get into details of how we generate substrides, it is important to understand some notation that we will be using throughout this chapter.

Firstly, we represent a substride by the pair <substride, node address>. The substride in the <substride, node address> format is a shortened prefix generated during lookups whereas the node address is the address of an internal node within the leaf-pushed trie. Similar to a prefix, we represent a substride by a bit string. Since substrides are shortened prefixes, their bit length is not more than 32. These pairs, in concept, point to different subtries within the trie. Moreover, the substride and the node address in the pair are obtained dynamically based on the IP traffic seen by the trie rather than by preloading the DSC.

Secondly, we represent the height of a substride by $height - k$ which is measured from the root of the trie. The term k in the notation indicates skipping k levels from the leaf of a trie. The values of k have an affect on the performance of the DSC and we later show the importance of choosing a good value for k. An illustration is provided in Figure 4.3.

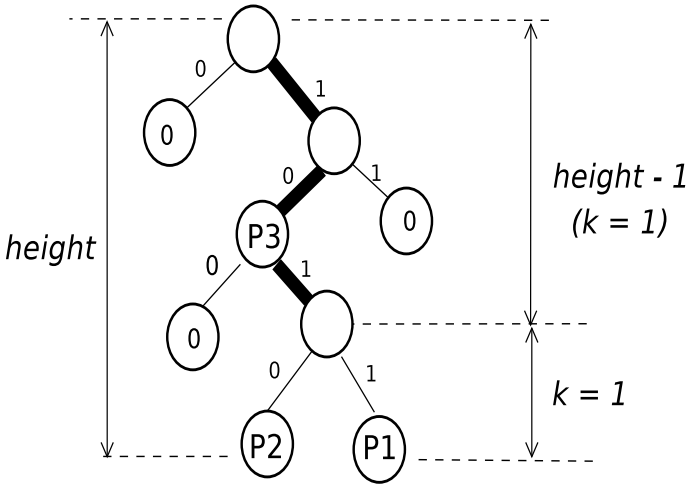

Figure 4.3: Height of a substride 101*: *height - k* where *k = 1*

Dynamic substride caching is a technique which helps alleviate the problems of full-trie lookups. The DSC stores substrides obtained from recent lookups in the trie. Leveraging substrides, lookups can proceed directly from an internal node within a trie, skipping many internal nodes along the path. This, in effect, reduces the number of memory accesses needed by a single address lookup. Consider a simple example in Figure 4.4 where the lookup is being performed for a packet with destination address 10100. Also assume that we have a DSC which contains an entry <101*, 0xB>.The address lookup will find a hit in the DSC because the most significant bits of the destination address match the substride 101*. This allows the lookup to skip 101 and proceed from node 0xB in the trie. Then the lookup has to do two more memory accesses to find the leaf containing the next hop $h_i = P3$.

Clearly, in this case we skip three memory accesses.

Once a lookup has completed we store the prefix and the substride in the prefix cache and the DSC respectively. Consider the previous example where a lookup for the packet with destination address 10100 finds the next hop $h_i = P3$. Firstly, the pair <10100*, P3> is stored in the prefix cache. Secondly, we store the pair <1010*, 1xC> in the DSC where the substride is 1010* and the node address is 1xC. The substride 1010* is obtained by reducing the height of the path represented by the prefix 10100* by 1. For the previous example we considered the value of k to be 1.

The lookup requires one top-down traversal to determine the prefix as well as the substride. To achieve the above, we initialize two pointers to the root of the trie - one for the prefix (p_c) and the other for the substride (p_s). The p_c always begins the traversal down the trie. The p_s starts its traversal only when p_c has completed k edge traversals. Thereafter, p_s follows p_c one edge at a time until p_c finds a leaf node. The internal node where p_s points when p_c reaches the leaf node is the required node address for the pair <substride, node address>.

While we cache the prefixes resulting from a full-trie lookup, we also want to ensure that the prefixes that were successfully looked up because of the substrides are also moved to the prefix cache. This allows us to fully utilize the prefix cache and not have it affected because of the DSC. For the same, we perform a full-trie lookup and use p_c and p_s to traverse and generate the prefix and the substride. However, if the lookup was assisted by the DSC then we do not have the opportunity to generate the prefix or the substride using the two pointers.

Alternatively we can use a different architecture for the DSC (see Figure 4.5). As shown, the tag array in the DSC stores the substride and the data array stores the <substride, node address> pair. A lookup that finds a hit in the tag array retrieves the corresponding <substride, node address> from the data array. Once a lookup completes, we can extract the substride from the <substride, node address> pair to generate the prefix as well as the substride. For example in Figure 4.4, 10100* for the prefix cache is generated using the substride 101* from the DSC and the remaining trie path (00) traversed starting from 0xB due to the DSC lookup 101*.

Previously we mentioned that it is important to have a leaf-pushed trie to prevent incorrect lookups. Incorrect lookups can be explained by a simple example using Figure 4.6(A) where prefix 10* is the parent prefix of 1001* and 1011*. If we receive a packet for which P4 is the best match and we cache the pair <101*, C> in the DSC ($height - 1$) then subsequent lookups that should have found $h_i = P3$ as the next hop result may end up finding 101* in the DSC. As a result, lookups in the future may fail to find the next hop $h_i = P3$. Such an outcome is erroneous and may result in a large number of misses. We therefore solve this by using a leaf-pushed trie as shown in Figure 4.6(B). Lookups assisted by the DSC can find any of the three prefixes (P2, P3 and P4) without being incorrect.

An important parameter that effects the performance of the DSC is the value of k in $height - k$. The value of k indicates the number of levels we skip starting from the leaf of a trie. The larger

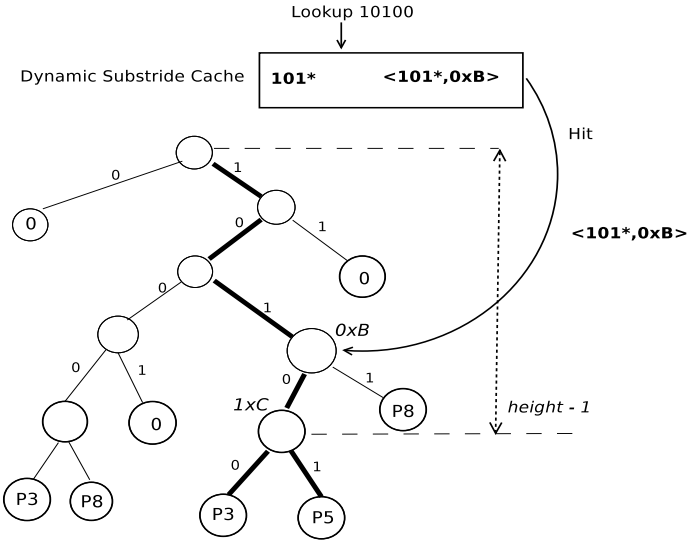

Figure 4.4: Dynamic Substride Caching for $k = 1$

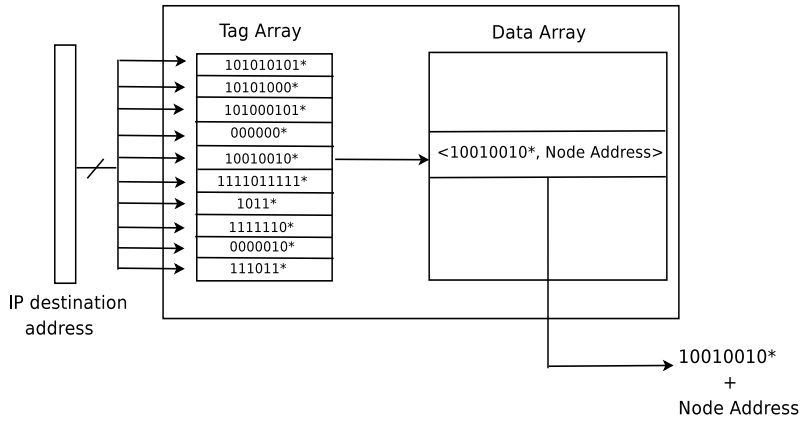

Figure 4.5: Dynamic Substride Caching

the value of k , the more levels we skip while generating the substride. This increases the number of additional memory accesses that lookups need to perform after they jump within the trie. At the same time, if we choose a smaller value of k , the DSC will show relatively lower hit rates. This scenario can be explained from Figure 4.4 where the DSC will demonstrate a better hit rate when we cache substride up to 0xB rather than 1xC. This is because the substride up to 0xB covers a larger address space when compared to the substride 1xC. However, lookups that benefit from the substride up to 0xB have to perform at most two more memory accesses in order to find the leaves. On the other hand, lookups that benefit from 1xC have to perform at most one memory access in order to find the leaves.

Our choice for the value of k is based on empirical results. We found that the value of k can be selected anywhere between 2 and 4. By choosing k between these values lookups that use the substrides are able to jump as deep as possible within the trie. By doing this, we found that only an

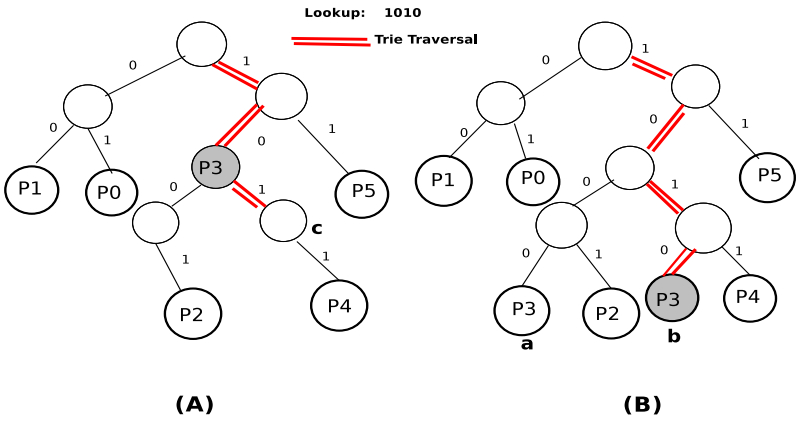

Figure 4.6: Dynamic Substride Caching

additional 3 to 5 memory accesses on average are required to finish the lookup. While we could have chosen $k = 1$, we realized that the DSC then shows relatively lower hit rates. Also, values of k more than 4 were found not suitable because they induced an average of more than 5 memory accesses. We consider average memory accesses of less than 5 to be suitable because it is permissible for the OC-192 line rates [16]. Further, we identified the value of $k = 3$ to be the best because then lookups need to perform at most an additional 5 average memory accesses. Further discussion on the parameter k is provided in Section 5.3.1.2.

4.4 Incremental updates

A routing table is subject to change during updates. These updates can lead to inconsistency in the prefix cache-first or the DSC-first architectures. Two situations where the data can become inconsistent are addition and deletion of prefixes. The task of ensuring consistency gets even harder since we store information in three different components in our architecture - prefix cache, DSC, and the leaf-pushed trie. Given the importance, we present an idea for incremental updates that is applicable to both the prefix cache-first as well as the DSC-first architectures. Some views on incremental updates on a leaf-pushed trie are presented in [44]. However, they do not present views in cases where parent prefixes are deleted from the leaf-pushed trie. Alternatively, we show that encoding extra information in the internal nodes and leaves will be sufficient to ensure proper deletion of parent prefixes from the leaf-pushed trie. Further, since DSC is a unique cache organization, we provide a new mechanism to ensure consistency in DSC during incremental updates. In addition, Akbarizadeh *et al.* [8] provide schemes for incremental updates in a prefix cache. However, they use an approach where they do not consider next hop information when making decisions. This results in removal of prefixes from the prefix cache that can actually be done without if we consider the next hops. We show that using next hop information of the newly updated prefix and the prefix in the prefix cache, we can reduce the number of prefix removals from the prefix cache. This will only reduce latency

during updates.

Moreover, we have do not do any implementations for incremental updates due to lack of relevant datasets. Nonetheless, we present all the possible cases where incremental updates can cause inconsistency and provide solutions for them.

For readability, we call prefixes in the leaf-pushed trie r_t and prefixes in the prefix cache r_j. Further, we call prefixes to be added or deleted as r_i.

4.4.1 Addition

When a new prefix r_i in $< r_i, h_i >$ is to be added, we take the following steps to make the leaf-pushed trie consistent. First, we traverse down the trie based on the prefix r_i in $< r_i, h_i >$. If the traversal finishes at a leaf encoded with 0, then we replace 0 with h_i. In this case, the prefix r_i is an independent prefix. We also store the prefix length of r_i in the leaf. If the traversal finishes at a leaf node which is a decision node, we compare the prefix length of r_i with prefix length of r_t present in the leaf. The prefix that turns out to be the longest gets the priority and the next hop associated with it is encoded in the leaf. In the case where the prefix lengths are the same, we encode the leaf with both r_i and r_t since both the prefixes have the same priority. In addition, if the traversal exceeds beyond a leaf with 0, then we continue by creating the new nodes for the prefix r_i. The leaf in the new branch will encode h_i The above process has the complexity *O(W)* where *W* is the worst case depth of the trie.

The update becomes much more complex in the event the traversal finishes at an internal node. This would mean that prefix r_i is a parent prefix. The internal node where the traversal finishes, in effect, is a root of a subtrie. Then we traverse the entire subtrie depth-first starting from the root of the subtrie. Subsequently, we follow a similar technique as described above. Any leaf in the subtrie encoded with 0, is replaced with h_i. For other cases, we compare the prefix lengths present in the leaves with that of r_i. Changes are made to leaves depending upon the outcomes of the prefix length comparisons. This multiple traversal has complexity *O(NW)* where *N* is the number of paths in the subtrie. Further if the new prefix r_i makes an independent prefix r_t in the leaf-pushed trie a parent prefix, then we follow the method described above. We traverse the leaf-pushed trie using the prefix r_t instead of r_i to find the root of a subtrie. Then we traverse the entire subtrie in depth-first order starting from the root of the subtrie and update the leaves based on the rules stated above.

The DSC is looked up for substrides in <substrides, node address> such that either one of them - substride or r_i is a parent of the other. For readability, we make the distinction that “parent” is meant for comparisons between a substride and a prefix. On the other hand, “parent prefix” is meant for comparisons between two prefixes.

Firstly, in the case where the new prefix r_i is a parent of the substride, we do not remove the corresponding <substride, node address> from the DSC. This is because the new prefix r_i in this case is a parent prefix and does not effect the leaf-pushed trie in terms of the number of prefixes

(i.e the address space). Since the leaf-pushed trie has not changed, the effects of the jump from the substride will also not change. This is shown in Figure 4.7.

Secondly, if the substride is a parent of r_i, we still need to verify whether the new prefix r_i is a parent prefix or not. This information can be obtained from the recent addition process for r_i in the leaf-pushed trie. If the leaf-pushed trie "saw" that r_i was a parent prefix, we still do not remove <substride, node address> from the DSC. This is because even in this case the leaf-pushed trie is not altered due to the prefix r_i. This shown in Figure 4.8. Otherwise, if r_i is not a parent prefix we follow a conservative approach. We check how much longer is r_i in length than the substride. If it is more than *k* levels, we delete the corresponding <substride, node address>. This is to ensure that we (approximately) maintain the property of *height - k*.

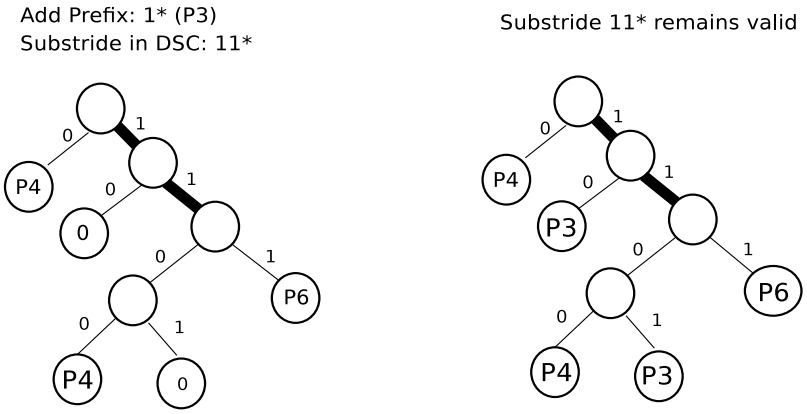

Figure 4.7: Effects on substride during parent prefixes addition in trie

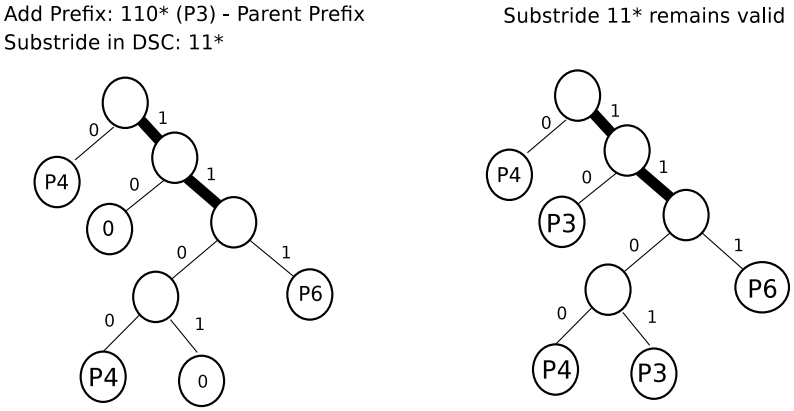

Figure 4.8: Effects on substride during parent prefixes addition in trie

Preliminary work on making the prefix cache consistent after incremental updates is provided in [8]. However, they do not cover all the possible scenarios that can make the prefix cache inconsistent.

We look at each of the scenarios one at a time.

We look out for a prefix r_j in the prefix cache such that either r_i or r_j is the parent prefix of the other. It should also be noted that the leaf-pushed trie has only independent prefixes even after the new prefix r_i is added to the leaf-pushed trie. The following cases may arise:

1. If we do not find a prefix r_j where either r_i or r_j is a parent of each other, then we stop the search in the prefix cache. In this situation the prefix cache is consistent.

2. If r_i is a parent prefix of r_j and $h_i \neq h_j$, then $< r_j, h_j >$ is removed from the prefix cache. Since we do not have enough information about r_j i.e its prefix length, it is not possible to decide the highest priority prefix - r_i or r_j. However, if $h_i = h_j$ then the entry $< r_j, h_j >$ is not removed from the prefix cache. This is because r_i has not effected the leaf containing the next hop h_j for prefix r_j.

3. If r_j is a parent prefix of r_i, then again we need to remove the entry $< r_j, h_j >$ from the prefix cache. This is because r_i has a higher priority and the the entry $< r_j, h_j >$ can make the prefix cache inconsistent.

4. If both the prefix are same and $h_i = h_j$, then the prefix is not removed from the prefix cache. However, if $h_i \neq h_j$, then the entry $< r_j, h_j >$ is removed from the prefix cache since the packet should be routed to h_i and h_j, not h_j alone.

4.4.2 Deletion

Deletion of prefixes from a leaf-pushed trie is much more complicated. To ensure correct deletions in a leaf-pushed trie, we need to store additional information about parent prefixes in the leaf-pushed trie. As a result, we encode an internal node with a flag indicating the presence of a parent prefix. In addition, we store the next hop h_k of the parent prefix in the internal node. However, internal nodes encoded with the above information are not decision nodes and do not participate during address lookups in the two architectures. This additional information is meant for incremental updates only. A simple example is shown in Figure 4.9. This information can be encoded during leaf-pushing (trie build) and during prefix addition (incremental updates).

During deletion, we traverse down the leaf-pushed trie based on the prefix r_i in $< r_i, h_i >$. During the traversal we check whether any of the internal nodes in the path have $flag = 1$. If yes, we keep note of the parent prefix information. In addition, we remember the number of edges we saw before we met the internal node. The remembering is done for the most recently visited internal node having the $flag = 1$. Further, information pertaining to previous internal nodes is forgotten.

The following cases occur when a prefix r_i is to be deleted from a leaf-pushed trie:

1. Suppose we traverse down the leaf-pushed trie for a given prefix r_i where r_i is not a parent prefix. When the traversal finishes at a leaf, we first check whether we met an internal node

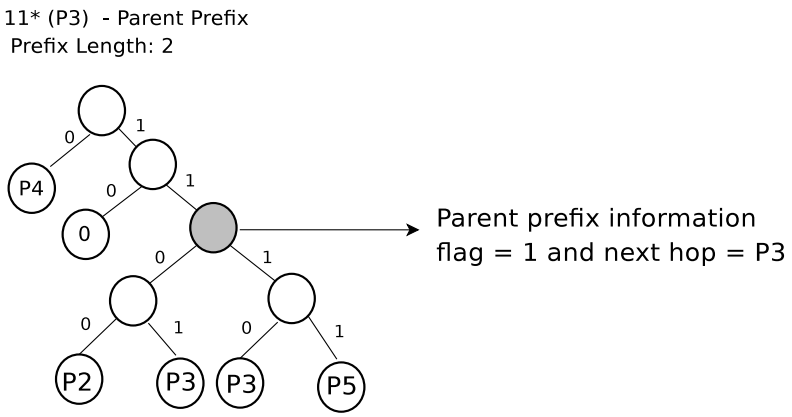

Figure 4.9: Leaf-pushed trie with parent prefix information

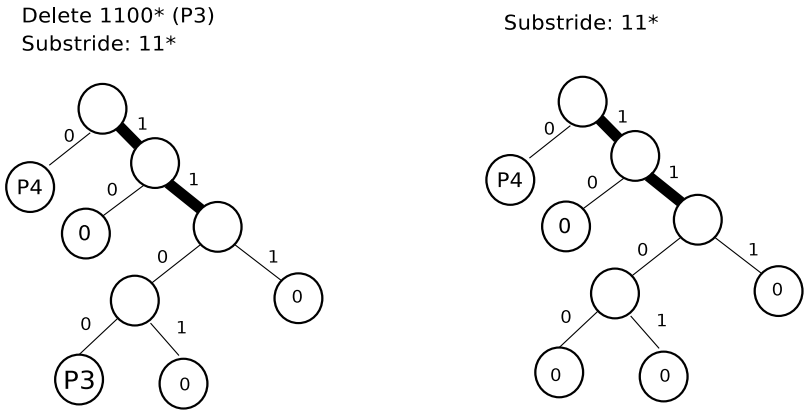

Figure 4.10: Subtrie after prefix deletion

with $flag = 1$ during the traversal. If not, we clear the next hop h_j (i.e we make it a non decision node 0) and the prefix length encoded in the leaf. If we did see an (most-recent) internal node with $flag = 1$ during the traversal, we replace the next hop h_j with the next hop h_k we saw in the internal node. We also replace the previous prefix length in the leaf with the one we remembered for the internal node.

2. Similarly, if we traverse down the leaf-pushed trie for a given prefix r_i where r_i is a parent prefix, then again, we check whether we met an internal node with $flag = 1$ during the traversal. The internal node where the traversal finishes is the root of the subtrie which is then traversed for the deletion of the prefixes. If we did not meet an internal node with $flag = 1$ during the traversal, then we clear the next hops from all those leaves that contain h_i. We also clear the prefix lengths. This clearing is done only if the prefix lengths of r_i is equal to r_j that is encoded in the leaves. In the event, we saw an internal node with $flag = 1$ during the

traversal, we replace the next hops from leaves containing h_i with h_k. Further, we replace the prefix lengths in the leaves with the one we remembered. Again, we do the replacement only if the prefix lengths match.

For the DSC, we follow a conservative approach during deletion. After the deletion process, we cannot accurately determine whether a substride is useful or not. This is shown in Figure 4.10. The subtrie rooted by the substride has leaves storing 0. Seemingly, the substride is no longer useful for address lookups. As a result, we follow a simple approach where we remove all <substride, node address> entries from the DSC where either the substride or r_i is a parent of the other.

The two type of prefixes that can be deleted from a leaf-pushed trie are parent or independent prefixes. A prefix cache at any particular time can store either independent prefixes or prefixes grafted from parent prefixes (minimally expanded prefix). The following cases may occur when a prefix r_i is deleted from a leaf-pushed trie which is also investigated in [8]:

1. For a prefix r_j in the prefix cache, if r_i and r_j are same and $h_i = h_j$, then we remove the entry $< r_i, h_i >$ from the prefix cache.

2. For a prefix r_j in the prefix cache, if r_i is a parent prefix of r_j and $h_i = h_j$, then $< r_i, h_i >$ is removed from the prefix cache. Otherwise, if $h_i \neq h_j$, then we do not remove the entry $< r_i, h_i >$ from the prefix cache.

3. In any event, no prefix r_j can be a parent prefix of r_i because the prefix cache contains only independent prefixes.

Chapter 5

Experiments

In this chapter, we first describe the different metrics we use to evaluate our idea of using the two different architectures (prefix cache-first and DSC-first). The metrics are used to determine the effectiveness of the two caches in the two architectures as well as their contribution in reducing the number of full-trie lookups. Further, in Section 5.2 we describe the packet traces and routing tables we use for the experiments. Section 5.3 onwards we present the empirical results as well as detailed analysis.

5.1 Metrics and Implementation

In order to test our technique, several different measurements were taken into account in order to evaluate the performance of the architectures (prefix cache-first and DSC-first).

Firstly, we determine the hit rates of the prefix cache as well the DSC. These hit rates are local to the cache. For readability we call them local hit rates $l(c_i)$ where c_i is either the prefix cache or the DSC. A higher local hit rate in both the prefix cache and the DSC would mean that a significant percentage of address lookups were assisted by both the prefix cache as well as the DSC.

Secondly, the local hit rates are not sufficient on their own to indicate the performance of the architectures as a whole. As a result, we also use the global hit rate $g(c)$ to evaluate the performance of the two caches in combination. The global hit rate can be computed using the following equation:

$$\textit{global hit rate} = \frac{\textit{Total number of hits in prefix cache} + \textit{Total number of hits in DSC}}{\textit{Total number of address lookups}} \times 100 \quad (5.1)$$

Quantitatively, the global hit rates indicate the percentage reduction in the number of full-trie lookups.

Finally, we measure the average number of memory accesses required by lookups that are assisted by the DSC. As mentioned earlier, address lookups that use the DSC jump to an internal node within the leaf-pushed trie. Thereafter, these lookups require few more memory accesses to find the next hop. Preferably, we would like all lookups to cost not more than 5 memory accesses on average considering the requirements of the OC-192 links [16].

One aspect that needs to be considered is the value of k. This value does not necessarily indicate the performance of the DSC. However, it does have a profound impact on the performance of the DSC. Thus, it is imperative to find an appropriate value of k. A high value of k would result in higher hit rates in the DSC. However, the improvements in the hit rates come at the expense of an increased number of average memory accesses. At the same time, choosing a lower value for k can lead to lower hit rates in the DSC. Since the performance of the DSC varies with the value of k, we therefore perform exhaustive simulations on different datasets to arrive at an appropriate value of k.

We consider the implementation of the DSC to be similar to the prefix cache where the DSC is considered to be residing in a contiguous memory. The beginning of the contiguous memory is the top (first-entry) of the DSC. Henceforth, to simulate the LRU replacement strategy, the most recently looked up pair <substride , node address> is moved to the top of the contiguous memory and other <substride , node address> pairs were moved down by one position. It should be noted that we do not measure the number of clock cycles required for a lookup in a cache. In addition we do not take into consideration the cache power consumption nor the latency associated with the prefix cache and the DSC.

5.2 Dataset

5.2.1 Routing tables

For the experiments, we consider a few more routing tables along with the ones mentioned in Chapter 3. The new routing tables, again, are real-world and are drawn from different routers. These routing tables vary depending on size and prefix length distribution. These routing tables are larger in size and demonstrate significantly less redundancy.

We downloaded two routing tables, rrc03 and rrc11, from [5] each containing 132210 and 126687 (large-sized) prefixes respectively. These routing tables, rrc03 and rrc11, were resident in routers in the Internet exchanges located at Amsterdam and New York respectively. Further, we downloaded an as1221 routing table from [1] containing 292496 prefixes. This routing table was resident in a router located in Sydney, Australia. Significantly, rrc03, rrc11 and as1221 have nearly 80% of the prefixes of length greater than 18. We also use ISP2 routing table which contains 6342 prefixes. Like ISP1 and ISP3, ISP2 is also drawn from distribution routers.

5.2.2 Packet traces

We downloaded three traces upcb.1, upcb.2 and bell from [4]. The bell trace was collected at Bell Labs whereas the upcb traces were collected at a Catalan research network. The upcb.1 and bell traces had around 0.9 million packets whereas the upcb.2 trace had around 0.6 million packets. It should be noted that the three traces have relatively lower amount of temporal locality and thereby require the caches to be comparatively larger in size than the one used for ISP1, ISP2, ISP3 and the

Dataset	Average memory accesses	Standard Deviation
Funet	16.0	2.7
ISP1	14.8	4.4
ISP2	19.7	4.2
ISP3	15.8	4.9
rrc03 - upcb.1	16.4	3.1
rrc11 - bell	15.9	2.2
as1221- upcb.2	17.4	3.6

Table 5.1: Average memory accesses for all packet traces

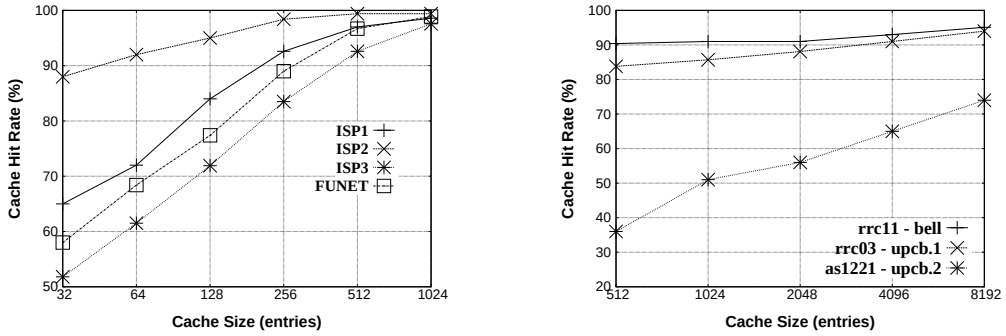

Figure 5.1: Prefix cache hit rates

Funet trace. Further, the destination addresses in the bell traces are destined to fewer address spaces when compared to the upcb traces. Consequently, the upcb traces should find the DSC more useful as opposed to the bell trace. For the experiments, we ran the upcb.1 trace over rrc03 routing table, the upcb.2 trace over as1221 routing table and the bell trace over rrc11 routing table. Further, we use the ISP2 trace which we use it over the ISP2 routing table. ISP2 contains around 100000 packets and has more locality in comparison to ISP1 and ISP3.

5.3 Experiments

Table 5.1 gives the average number of memory accesses required by all the packet traces. We also present the standard deviation with respect to the average value. These values are obtained by performing address lookups on a leaf-pushed trie. It is interesting to note that ISP1 requires only 14.8 average memory accesses. This suggests that most lookups in ISP1 match prefixes of length less than 16. However, for the as1221 routing table, the upcb.2 packet trace requires 19.7 memory accesses on average. Clearly, address lookups for all the packet traces are expensive.

5.3.1 Prefix cache-first architecture

Figure 5.1 shows the prefix cache hit rates for all the packet traces. As discussed earlier, the upcb.1, upcb.2, and bell packet traces have substantially less locality. As a result, the hit rates are comparatively lower when compared to the other packet traces. Consequently, we require a prefix cache that is much larger in size. As shown, the hit rates for the packet trace upcb.2 is comparatively lower

than upcb.1 and bell packet traces. This suggests that upcb.2 has lower temporal and spatial locality. Further, the bell trace and the upcb.1 trace show less improvement in hit rate with increasing cache size. This in fact shows that we are closer to the inherent limit and improving upon it will only get harder.

Clearly, by using the prefix cache a significant percentage of the lookups will not perform full-trie lookups. For instance for the packet trace upcb.2, nearly 74% of the lookups will not proceed to the leaf-pushed trie when the prefix cache has 8192 entries. Similarly, for the Funet trace 98.9% of the lookups will not perform full-trie lookups.

For the packet traces ISP1, ISP2, ISP3 and Funet, the prefix cache attains peak hit rates at cache size 1024. We found that we do not improve on the hit rate even after increasing the size of the prefix cache. Similarly, for the remaining packet traces, the prefix cache should have at most 8192 entries to achieve peak hit rates. At this point, however, we require something new.

We first explore the idea of using a prefix cache with a smaller size and a DSC with varying size. There are couple of advantages of doing this. Firstly, a small prefix cache can be used to exploit temporal locality. This can be seen in Figure 5.1 where a prefix cache size with 32 entries is beneficial for the ISP1, ISP2, ISP3 and Funet packet traces. This is because a small prefix cache will be sufficient to consume most of the temporal locality. Similar results hold true for the packet traces upcb.1, upcb.2 and bell. This ensures that straightforward lookups need not proceed to the DSC. Secondly, a small prefix cache will reduce the likelihood of the DSC getting polluted with substrides resulting from temporal locality. As a result, the DSC can then be used exclusively to exploit spatial locality.

Figure 5.2 shows the DSC hit rates when the prefix cache size is small. For the experiment we consider $k = 3$. Further, for the packet traces ISP1, ISP2, ISP3 and Funet we allocate 32 entries to the prefix cache. In addition, we allocate 512 entries to the prefix cache for the remaining traces. It should be noted that the DSC exploits two different patterns in the traces. First is the spatial locality where destinations addresses belong to the same subnet. Second is the case where destination addresses fall in fewer address spaces.

Figure 5.2(A) shows that the DSC hit rates increase with increasing DSC size. For example, the DSC with 256 entries has nearly 95% local hit rate for ISP1. This is primarily due to better locality demonstrated by the ISP1 trace. In contrast, the DSC hit rates for the traces upcb.1, upcb.2, and bell are comparatively lower (Figure 5.2(B)). This is because the prefix cache is not very effective in this case. The reason is that the upcb.1, upcb.2 and bell traces have lower spatial and temporal locality. As a result, most of the lookups miss the prefix cache and proceed to the DSC. Consequently, the DSC serves most of the address lookups that could have been exploited by a prefix cache had it had more entries. Similarly, in Figure 5.2(B), the DSC with 8192 entries shows only 66% local hit rates for the upcb.1 trace. This suggests that these 8192 entries were mostly useful in exploiting temporal locality and to some extent spatial locality. However, the DSC was not effective in exploiting

addresses that belong to a specific address space. Unfortunately here, we find that having a small prefix cache size can also be counter-productive.

It is not surprising that the DSC demonstrates a higher hit rate for the upcb.2 trace. This is because the prefix cache with 512 entries shows lower hit rates for upcb.2 trace when compared to the upcb.1 and bell traces. As a result, the DSC has more opportunities to exploit locality. Had we allocated more entries to the prefix cache, the DSC would have been comparatively less effective.

In addition, if we look into the performance of the overall architecture (prefix cache-first), we can see that the global hit rates are up to 99.66% (Table 5.2).

Table 5.3 gives the average number of memory accesses that lookups may require if we do not consider the DSC in the architecture. Clearly, lookups may require up to 22.5 memory accesses on average. This indicates that most of the lookups that miss the prefix cache match prefixes of length greater than 18. As it turns out, even a small percentage of misses in the prefix cache can prove expensive. For example for upcb.1 trace, the prefix cache with 512 entries yields a hit rate of 84%. This means 16% of the address lookups missed the prefix cache and may require up to 17 memory accesses on average. However, if we have a DSC in the prefix cache-first architecture with 8192 entries then 65% of the lookups for the upcb.1 trace will find a hit in the DSC. As a result, only 5.6% of the lookups would require 17 memory accesses.

It is interesting to see the average number of memory accesses that a lookup requires when it finds a hit in the DSC. Table 5.4 gives the average number of memory accesses for all the packet traces. It should be noted that this value may vary depending on the DSC size. These values change because different DSC cache sizes will see different types of lookups. Hence, a particular DSC size will demonstrate a slightly different average number of memory accesses. In the table we have recorded the worst-case average memory accesses (WC) required by lookups amongst the different DSC sizes we considered in Figure 5.2.

The upcb.2 trace (Table 5.3) shows a high average number of memory accesses. Moreover, the deviation about the average is also very high. The reason behind the high value is that as1221 is a comparatively large-sized routing table with many distinct prefixes. As a result, some of the substrides do not provide a “deep” enough jump within the leaf-pushed trie. This leads to lookups performing comparatively more memory accesses.

We use the empirical results showed in Figure 5.2 and Table 5.4 to demonstrate the full benefits of the DSC. Consider a simple example where lookups for upcb.1 trace may require 4.29 memory accesses on average for a DSC with 8192 entries. As noted earlier, the DSC shows 65% hit rates for the upcb.1 packet trace. Consequently, 65% of the lookups will require only 4.29 memory accesses on average as opposed to 17.

It may appear that it is wiser to allocate all the available memory to the prefix cache rather than distributing part of it to the DSC. This is not necessarily true. This can be explained from Figure 5.2(B). Consider a prefix cache with 2048 entries that shows 87% hit rate for the upcb.1

trace. However, if we allocate 512 entries to the prefix cache and the remaining to the DSC, then the global hit rate will be around 92%. The same holds true for the upcb.2 trace where the prefix cache of 2048 entries has a 56% hit rate. However, if we allocate 512 entries to the prefix cache and the remaining to the DSC, then we achieve 71% global hit rate.

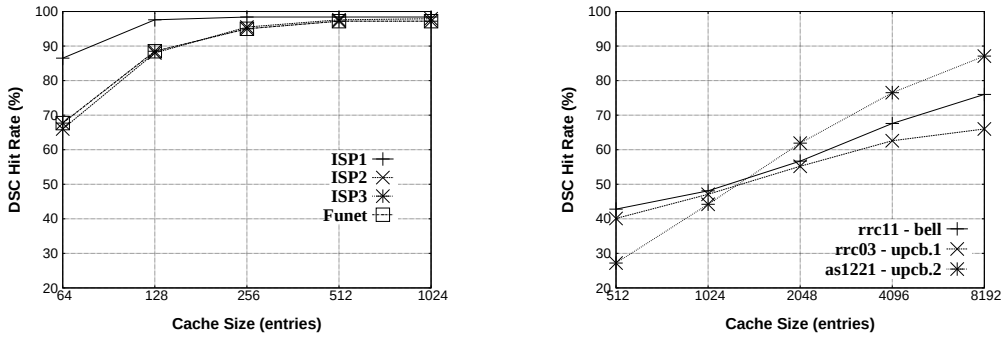

Figure 5.2: (A) DSC hit rates when prefix cache size is 32 ($k = 3$) (B) DSC hit rates when prefix cache size is 512 ($k = 3$)

Dataset	DSC size (Maximum) - Prefix cache size	Global Hit Rate (g_c)
Funet	2048 - 32	98.27
ISP1	2048 - 32	99.26
ISP2	2048 - 32	99.66
ISP3	2048 - 32	99.22
rrc03 -upcb.1	8192 - 512	96.11
rrc11-bell	8192 - 512	96.73
as1221-upcb.2	8192 - 512	90.46

Table 5.2: Global Hit Rates (%)

Dataset	Avg. memory accesses	Std. Dev.
Funet	18.7	4.3
ISP1	22.5	3.2
ISP2	18.1	4.7
ISP3	19.4	3.9
rrc03 - upcb.1	17.0	2.9
rrc11 - bell	15.9	4.4
as1221 - upcb.2	17.7	3.2

Table 5.3: Actual average (additional) memory accesses required by lookups that miss the prefix cache

Dataset	Avg. memory accesses (WC)	Std. Dev.
Funet	4.1	1.8
ISP1	4.2	1.6
ISP2	3.9	2.2
ISP3	4.4	1.8
rrc03 - upcb.1	4.29	2.4
rrc11 - bell	4.22	2.4
as1221 - upcb.2	4.9	3.1

Table 5.4: Average (additional) memory accesses required by lookups that find a hit in the DSC

Dataset	Prefix Cache Size	Avg. memory accesses	Std. Dev.
Funet	2048	19	3.8
ISP1	2048	18.3	4.6
ISP2	2048	16.8	3.1
ISP3	2048	17.1	4.4
rrc03 - upcb.1	8192	18.2	2.9
rrc11 - bell	8192	17.9	3.6
as1221 - upcb.2	8192	19.1	4.2

Table 5.5: Actual average (additional) memory accesses required by lookups that miss the prefix cache when the prefix cache size is maximum

5.3.1.1 Equal cache sizes

It is important to explore the individual improvements that a DSC can bring when the prefix cache in itself is very effective. To test the idea, we keep the size of both the prefix cache and the DSC the same. By doing this we give equal priorities to both the caches and we aim to achieve high global hit rates. This essentially means we need not necessarily prefer an under-performing cache within the prefix cache-first architecture. Further, this will keep the prefix cache completely focused in exploiting the spatial as well as temporal locality. Now, it is to be seen whether the DSC still manages to be effective.

Figures 5.3 and 5.4 shows the hit rates in the prefix cache as well as the DSC. We initially assume the value of k as 1. As discussed previously, the prefix cache hit rates increase with size. However, the downward trends of DSC hit rates in Figure 5.3 are different from those in Figure 5.2. This exemplifies the idea of using equal cache sizes. Initially, when the prefix cache is small, say 256 entries (Figure 5.3), the prefix cache is not at its best. Most locality in the destination addresses is not exploited by the prefix cache. As a result, the DSC which is similar in size, more or less aids the prefix cache and exploits the left-over locality in the destination addresses. Thus, the hits in the DSC are mostly due to left-over locality in the destination addresses and to some extent the address lookups that fall in fewer address spaces. However, as the prefix cache size increases, most locality is captured by the prefix cache. This causes the DSC hit rates to decrease. The other important reason for the trend is that the substrides in the DSC were not useful for the left-over lookups. This is partly because $k = 1$ (low) and partly because the left-over destination addresses were destined to different IP address spaces. Thus, the increased efficiency of the prefix cache and the reduced effectiveness of the substrides together cause the DSC hit rates to decrease.

However, in Figure 5.4 we see that the DSC hit rates for the upcb.1, upcb.2 and bell traces improve with increasing cache size. The primary reason for the trend is that the prefix cache hit rates improve with increasing size. Consequently, most of the locality in the destination addresses is exploited by then. What is left-over are mostly lookups that fall in fewer address spaces. Fortunately, the substrides in the DSC are able to exploit it even with the value of k being 1.

In the Figures 5.3 and 5.4, the prefix cache hit rates are maximum at sizes 1024 and 8192

respectively. The prefix cache hit rates do not improve beyond this point. It is interesting to see how the DSC performs in this scenario. For reference, we have recorded the average number of memory accesses that lookups will require when they miss the prefix cache. This entry in the table is made when the prefix cache hit rates are at their peak i.e at maximum prefix cache size (Table 5.5). However, these values may change depending upon the prefix cache size. Again, we see that the average number of memory accesses for the lookups are high. This clearly indicates the need and scope of reducing the number of full-trie lookups.

It can be seen that the DSC hit rates are not very high for all the traces when the prefix cache hit rates are maximum. This is because we skip only a single level ($k = 1$) when generating the substrides from the prefixes. Moreover for the ISP1, ISP2, ISP3 and Funet traces, the left-over destination addresses are destined to different IP address spaces. As a result, most of the lookups are not able find the substrides useful. Hence, for all the traces the DSC hit rates do not go beyond 50%. Further, we observe that the DSC hit rates are not greater than 10% for the Funet trace. Clearly, it shows that addresses in the Funet trace are destined to different IP address spaces within the leaf-pushed trie.

The lower hit rates in the DSC also reflect in the global hit rates (Table 5.6). Though there is improvement, it is only marginal. For the upcb.1 and bell traces, the global hit rates do not exceed 97%. As a result, we have nearly 3% of the lookups performing full-trie lookups. Though the numbers are small, this, it is costly since each lookup may require more than 15 memory accesses (Table 5.5). Similarly, the upcb.2 trace shows only 81.2% global hit rate. That means nearly 19% of the lookups proceed to the leaf-pushed trie for a full-trie lookup. This again is very expensive. The above discussion also holds true for the ISP1, ISP2, ISP3 and Funet traces.

The average number of memory accesses recorded in Table 5.7 is significantly lower. The table also shows the standard deviations which again is not extremely high. As previously, we have recorded only the worst-case average number of memory accesses seen by lookups under any prefix cache - DSC size combination. It is shown that the address lookups that find a hit in the DSC do not require more than 3 memory accesses on average. The reasons are similar to the situation above. We choose to skip only a single level $k = 1$ when generating the substrides. As a result, most of the jumps are nearer to the leaves. Consequently, most of the lookups do not require more than 3 memory accesses to find the next hop.

However, we would very much want a situation where the DSC has high hit rates. At the same time, we would want the average number of memory accesses not to exceed a certain quantitative point. Clearly, there is a trade off. If we skip more levels to generate the substrides, we will get better hit rates. Again, this comes at the expense of increased average number of memory accesses. Given its importance we will be discussing this trade-off in greater detail in the following sections.

In Figures 5.5 and 5.6, we show the DSC hit rates for $k = 3$. Clearly, the DSC hit rates have improved substantially up to a maximum of 88%. The DSC hit rates have improved especially for

the upcb.1, upcb.2 and bell traces. It shows that the value of k has a marked effect on the DSC hit rates. However, the DSC does not show considerable improvements for the Funet trace. Further, increasing the DSC size does not help either. Unfortunately, the lookups for the Funet trace do not find the substrides extremely useful. The primary reason is that most of the addresses are not destined to the address spaces captured by the substrides. This has to do with the nature of the source of Funet traffic stream.

Not surprisingly, the DSC hit rates for the upcb.1, upcb.2 and bell traces have improved. This is mostly due to the fact that substrides are more effective when $k = 3$. In addition, the prefix cache plays its part by taking care of lookups originating due to spatial and temporal locality.

Table 5.8 gives the global hit rates for all the packet traces. As shown, the global hit rates reach 99.88%. In addition, Table 5.9 gives the average number of memory accesses that lookups may require when they find a hit in the DSC. Again, the values are not extremely high and do not exceed 5 memory accesses. This also suggests that any increase in the value of k will only increase the average memory accesses. Thus, we can consider $k = 3$ to be suitable given that the lookups may not require more than 5 average memory accesses.

Taking a cue from the previous discussions, we would also like to further highlight the importance of the DSC. We see that the DSC hit rates are significant even when the prefix cache hit rates are maximum. This clearly shows that the DSC is beneficial even after having high prefix cache performance. For instance, the DSC shows 88% hit rates for the ISP1 trace even when the prefix cache is at its peak. This also indicates that the the prefix cache does not really hamper the performance of the DSC. Practically, for an infinite trace it is not possible to identify the quantitative point where a prefix cache may show peak hit rates. However, we show using a finite packet trace that the DSC is beneficial even when used in conjunction with a prefix cache.

However, a higher value of k can increase the average memory accesses appreciably. We show that by considering the value of k of 4. The average number of memory accesses for $k = 4$ are recorded in Table 5.11. The increase is primarily due to the number of levels we skip from the leaves when we generate the substrides. As a result, most of the substrides do not provide a jump within the trie that is reasonable in terms of average number of memory accesses. However, skipping more levels from the leaves increases the DSC hit rates considerably. The DSC hit rates for most of the traces exceed 90% (Figure 5.7 & Figure 5.8). Further, the global hit rates are nearly up to 99.9% (Table 5.10). However, the DSC hit rates for the Funet trace do not show sufficient improvement. This certainly confirms that the addresses in the Funet trace do not find the substrides extremely useful.

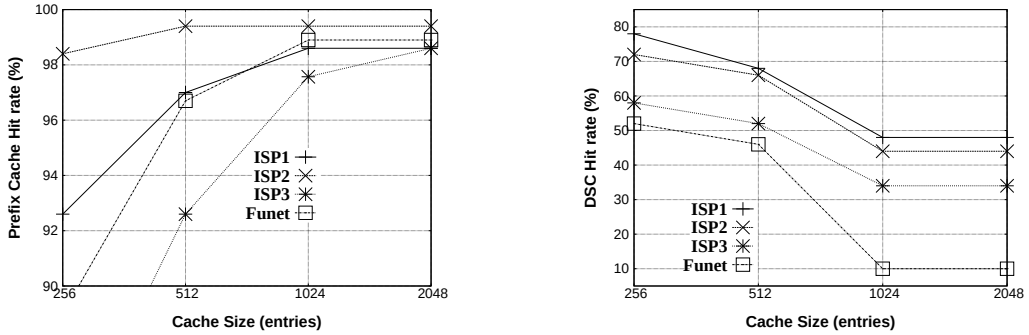

Figure 5.3: Prefix cache hit rates and DSC hit rates *(k = 1)*

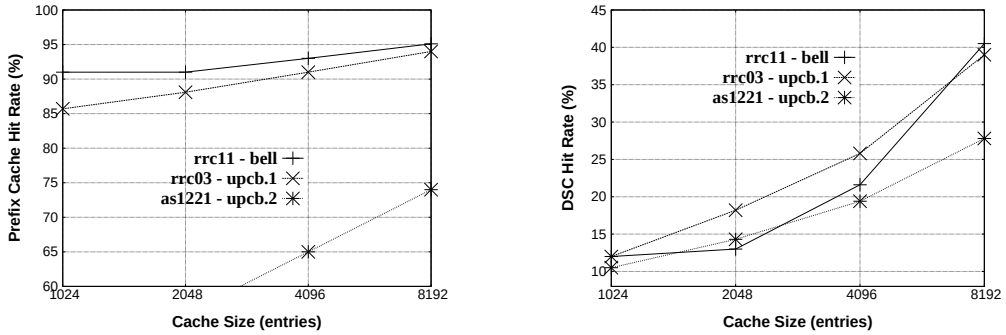

Figure 5.4: Prefix cache hit rates and DSC hit rates *(k = 1)*

Dataset	Cache Size (Prefix cache & DSC)	Global Hit Rate (g_c)
Funet	2048	99
ISP1	2048	99.2
ISP2	2048	99.6
ISP3	2048	99.0
rrc03 - upcb.1	8192	96.34
rrc11 - bell	8192	97
as1221 - upcb.2	8192	81.2

Table 5.6: Global Hit Rates (%) when cache sizes are maximum

Dataset	Avg. memory accesses (WC)	Std. Dev.
Funet	2.3	1.7
ISP1	1.9	2.1
ISP2	1.5	2.3
ISP3	2.1	1.8
rrc03 -upcb.1	2.4	2.2
rrc11 - bell	2.2	2.7
as1221 - upcb.2	3.1	2.4

Table 5.7: Average (additional) memory accesses required by lookups that find a hit in the DSC when cache sizes are maximum

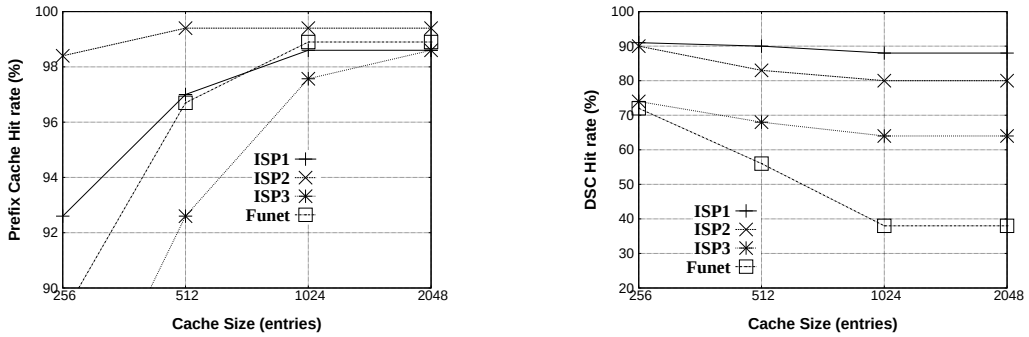

Figure 5.5: Prefix cache hit rates and DSC hit rates *(k = 3)*

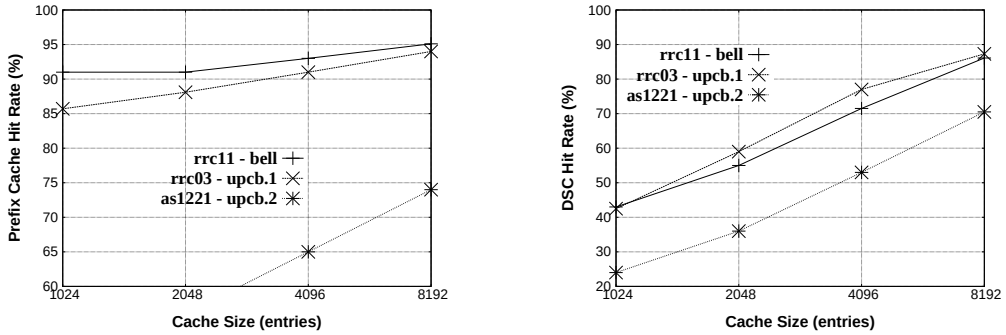

Figure 5.6: Prefix cache hit rates and DSC hit rates *(k = 3)*

Dataset	Cache Size (Prefix cache & DSC)	Global Hit Rate (g_c)
Funet	2048	99.3
ISP1	2048	99.83
ISP2	2048	99.88
ISP3	2048	99.49
rrc03 -upcb.1	8192	99.23
rrc11-bell	8192	99.34
as1221-upcb.2	8192	92.2

Table 5.8: Global Hit Rates (%) when cache sizes are maximum

Dataset	Avg. memory accesses (WC)	Std. Dev.
Funet	4.1	2.6
ISP1	3.5	1.9
ISP2	3.6	1.7
ISP3	3.9	2.1
rrc03 -upcb.1	4.66	2.9
rrc11-bell	4.49	3.1
as1221-upcb.2	4.52	3.4

Table 5.9: Average (additional) memory accesses required by lookups that find a hit in the DSC when cache sizes are maximum

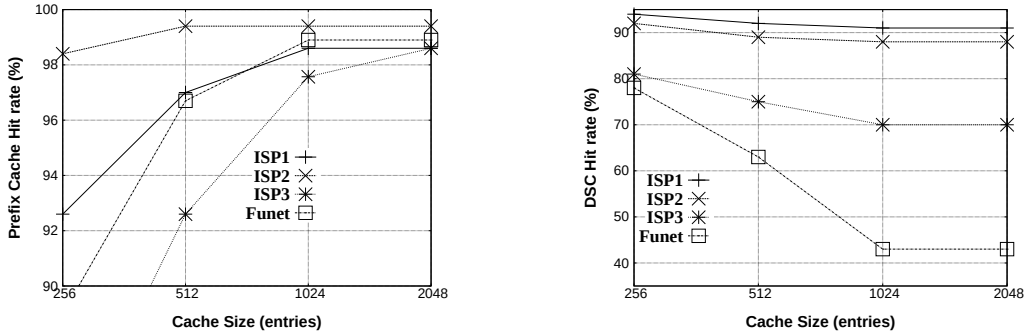

Figure 5.7: Prefix cache hit rates and DSC hit rates *(k = 4)*

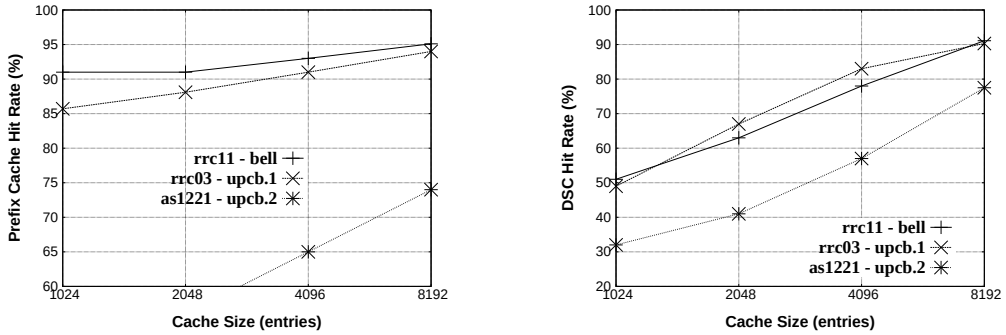

Figure 5.8: Prefix cache hit rates and DSC hit rates *(k = 4)*

Dataset	Cache Size (Prefix cache & DSC)	Global Hit Rate (g_c)
Funet	2048	99.38
ISP1	2048	99.88
ISP2	2048	99.94
ISP3	2048	99.58
rrc03 - upcb.1	8192	99.4
rrc11- bell	8192	99.5
as1221- upcb.2	8192	94.1

Table 5.10: Global Hit Rates (%) when cache sizes are maximum

Dataset	Avg. memory accesses (WC)	Std. Dev.
Funet	6.0	2.6
ISP1	5.1	3.2
ISP2	5.6	2.4
ISP3	5.8	2.8
rrc03 - upcb.1	6.3	3.8
rrc11- bell	6.1	2.2
as1221-upcb.2	6.5	4.2

Table 5.11: Average (additional) memory accesses required by lookups that find a hit in the DSC when cache sizes are maximum

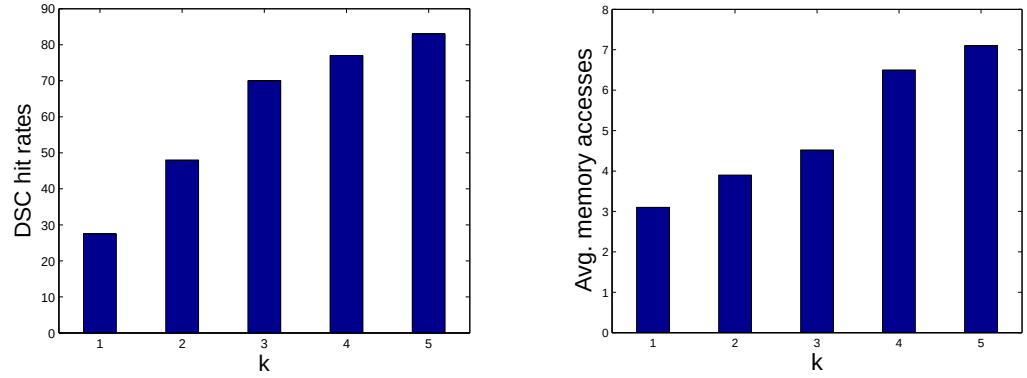

Figure 5.9: (A) DSC hit rates for increasing value of *k* when prefix cache size is maximum (upcb.2) (B) Avg. memory accesses for increasing value of *k* when prefix cache size is maximum (upcb.2)

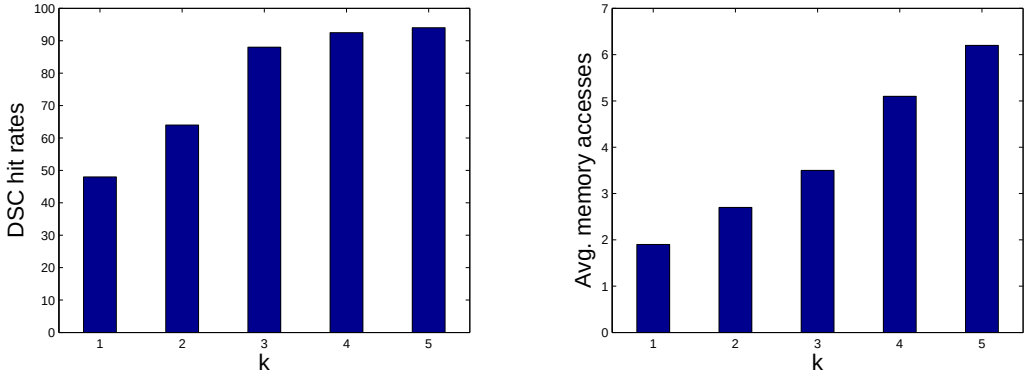

Figure 5.10: (A) DSC hit rates for increasing value of *k* when prefix cache size is maximum (ISP1) (B) Avg. memory accesses for increasing value of *k* when prefix cache size is maximum (ISP1)

5.3.1.2 Effects of value of *k*

In Section 5.3.1.1, we summarized the average number of memory accesses required by destination addresses for all the packet traces for $k = 3$. We also demonstrated that the DSC hit rates improved when we selected $k = 3$ over $k = 1$. But in order to achieve further improvements in performance, we might be tempted to choose a larger value for k. However, choosing a larger value of k can be counter-productive. In Figures 5.9 and 5.10, we show the effects of k for the upcb.2 and ISP1 traces respectively. For the experiment we considered equal-sized caches. Further, we allocated 8192 entries to both the caches when considering the upcb.2 trace. In addition, for the ISP1 trace, we allocated 1024 entries to both caches. It can be seen that for $k = 5$, the average number of memory accesses is as high as 7.0 for the upcb.1 trace even though the global hit rates have increased (Table 5.12). Similarly, for $k = 4$ the average number of memory accesses goes beyond 5 for both the traces. Hence, we can safely choose a value of k that is between 1 and 3 considering that we decided not to exceed 5 memory accesses.

DSC hit rate is one important parameter that can help us decide a suitable value of k. In Figure

Dataset	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
Funet	99	99.21	99.3	99.38	99.46
ISP1	99.2	99.49	99.83	99.88	99.91
ISP2	99.6	99.79	99.88	99.94	99.96
ISP3	99.0	99.30	99.49	99.58	99.67
rrc03 - upcb.1	96.34	97.72	99.23	99.4	99.67
rrc11 - bell	97	98.21	99.34	99.5	99.75
as1221 - upcb.2	81.2	86.74	92.2	94.1	95.84

Table 5.12: Global Hit Rates (%) for different values of k when cache sizes are maximum

5.9 (A) and Figure 5.10 (A), we see that the DSC hit rates for values of $k = 1$ or $k = 2$ are lower when compared to those when $k = 3$. However, $k = 3$ increases the average number of memory accesses when compared to $k = 2$. Nonetheless, the change in the average number of memory accesses is not drastic and $k = 3$ can be considered to be suitable.

It may seem from Figures 5.9 and 5.10 that there is no real benefit in choosing $k = 3$ over $k = 2$ since the average number of memory accesses do not change drastically between the two values of k. For instance, for the upcb.2 trace there is only 0.9 increase in average memory accesses when we choose $k = 3$ over $k = 2$. The same discussion holds true for the ISP1 trace. However, the DSC hit rates corresponding to the values of k provides a strong reason for our choice. The DSC hit rates for upcb.2 trace for $k = 3$ are nearly 20% more than for what we have for $k = 2$. That means 20% more lookups that missed the prefix cache did not perform a full-trie lookup. Consider Table 5.5 where the upcb.2 trace requires 19.1 memory accesses for full-trie lookups. Suppose we have 1000 lookups that missed the prefix cache. So, when considering $k = 2$, we require $(3.9 * 0.49 * 1000) + (1000 - 0.49 * 1000) * 19.1 = 11652$ memory accesses. On the other hand, when $k = 3$, we require $(4.52 * 0.70 * 1000) + (1000 - 0.70 * 1000) * 19.1 = 8894$ memory accesses. Thus, by choosing $k = 3$ we save up to 23.6% memory accesses. If we follow the same procedure, we will find that for the ISP1 trace, we save up to 37% memory accesses when we consider $k = 3$.

5.3.2 DSC-first architecture

In the experiments described previously, we considered the prefix cache to be placed before the DSC. These experiments showed that the DSC is not necessarily dependent on the prefix cache. This demonstrates that the DSC can be a valuable part of an architecture that employs a prefix cache. Then how about about a design scenario where the DSC comes before the prefix cache ? This design scenario seems interesting and it is to be seen whether the DSC has a good influence on the prefix cache.

In Figure 5.11 and 5.12 we show the hit rates demonstrated by the prefix cache and the DSC for all the packet traces when $k = 3$ in an architecture where the DSC precedes the prefix cache. It should be noted that we have considered the sizes of both the caches as equal. We first evaluate the results for the traces ISP1, ISP2, ISP3 and Funet. Apparently, the hit rates in the prefix cache

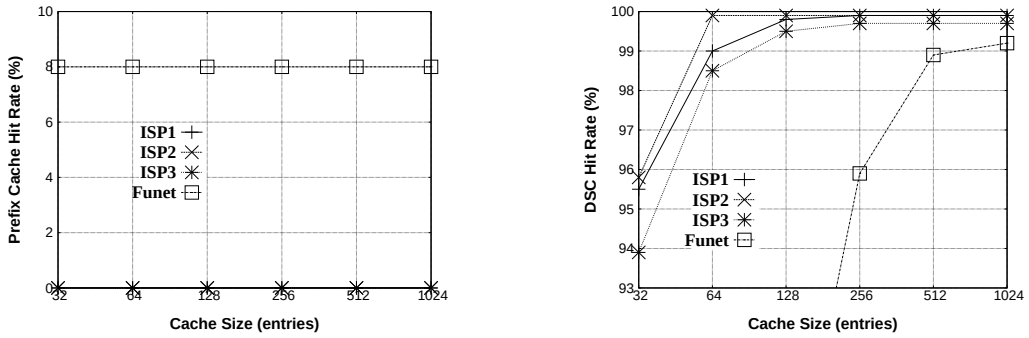

Figure 5.11: Prefix cache hit rates and DSC hit rates with both the cache with equal sizes *(k = 3)*

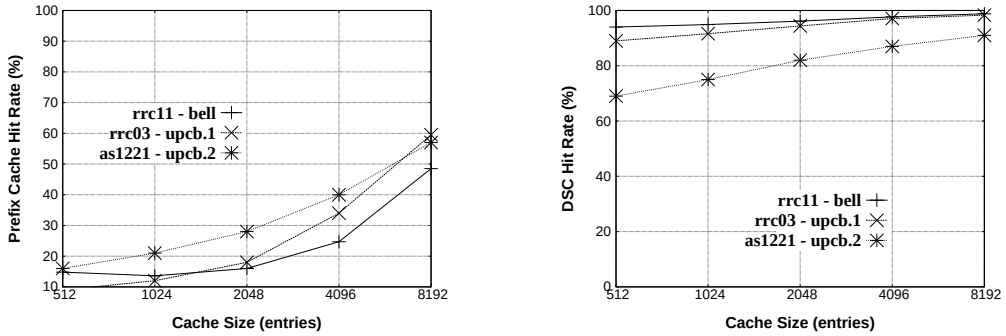

Figure 5.12: Prefix cache hit rates and DSC hit rates with both the cache with equal sizes *(k = 3)*

for the traces ISP1, ISP2, and ISP3 are marginal. Actually, for 256 entries, the prefix cache has a hit rate of 0.017% hit rates for the ISP1 trace. If we further increase the size of the DSC, the prefix cache hit rates approach 0%. This shows that the DSC is capable of achieving high global hit rates entirely on its own. That is, the DSC exploits all of the temporal and spatial locality in the addresses. Furthermore, it does more by assisting lookups for addresses that fall in fewer address spaces.

The improved performance in the DSC has a negative impact on the prefix cache. Higher DSC hit rates mean that the prefix cache have very few lookups to assist. To add to that, the prefixes in the prefix cache are more or less not useful for the lookups. This is because prefixes can be used to exploit locality only. Since the DSC does well on that front, the prefixes in the prefix cache are generally never referenced. In this case, we can consider the prefix cache redundant for the ISP1, ISP2 and ISP3 traces.

For the Funet trace, the prefix cache shows only 8% hit rate and this value remains stable even if we increase the size of the prefix cache. Seemingly, the prefix cache is not useful for any of these four traces. However, the DSC shows significant results. We have recorded the global hit rate values in Table 5.13 for the four traces. These values are mostly due to the DSC. Further, these values are not lower than the one recorded in Table 5.8. In fact, the global hit rates in Table 5.13 are slightly better. Moreover, we can see from Figure 5.11 that to achieve 99.9% global hit rate we need not require a large DSC cache. For instance, for the ISP1 trace we require only 256 entries to get near to peak hit rates. Similarly, for other traces the DSC cache size need not be more than 512 entries

Dataset	Cache Size (Prefix cache & DSC)	Global Hit Rate (g_c)
Funet	2048	99.26
ISP1	2048	99.9
ISP2	2048	99.9
ISP3	2048	99.7
rrc03 - upcb.1	8192	99.31
rrc11 - bell	8192	99.38
as1221- upcb.2	8192	96.13

Table 5.13: Global Hit Rates (%) when cache sizes are maximum

Dataset	Avg. memory accesses (WC)	Std. Dev.
Funet	3.25	1.77
ISP1	3.33	1.56
ISP2	3.12	1.21
ISP3	3.23	1.41
rrc03 - upcb.1	3.4	1.73
rrc11 - bell	3.34	1.66
as1221 - upcb.2	3.78	1.91

Table 5.14: Average (additional) memory accesses required by lookups that find a hit in the DSC when cache sizes are maximum

in order to obtain peak performance. But again, this comes at the expense of a few more memory accesses. Table 5.14 gives the average number of memory accesses that lookups will require if they hit the DSC. The values for the traces ISP1, ISP2, ISP3 and Funet are lower than 5. The question then arises: Is it reasonable to have a DSC alone in the architecture without being overly concerned about the average number of memory accesses (lower than 5) that it may yield for each lookup ? This again is a design choice which may depend upon the needs of the problem at hand. Alternatively, there is the prefix cache-first architecture to fall back upon if router architects believe that memory accesses may induce unnecessary latency.

The situation is slightly different for the upcb.1, upcb.2 and bell traces. For explanation, we use Figure 5.12 which shows the prefix cache hit rates as well as the DSC hit rates. Clearly, the prefix cache hit rates are substantially lower than those recorded in Figure 5.6. For the upcb.2 trace, the prefix cache hit rates gradually increase with the increasing prefix cache size. Provided that we allocate enough entries to the prefix cache, we may witness some decent contributions from the prefix cache towards the global hit rates. The same result holds true for the upcb.2 and bell traces. These numbers are mostly because of a more-active DSC and a less-active prefix cache. An active DSC results in some substrides getting aged out. Yet these substrides may be referenced again in future lookups. However, the lookups that missed matching these substrides find a matching prefix in the prefix cache. These matching prefixes remained resident in the prefix cache and did not get aged out because of less activity in the prefix cache. But for this to happen we may need a substantially larger prefix cache to hold on to those important prefixes.

These results suggest that we might place a sufficiently large prefix cache between the DSC and

the leaf-pushed trie. Even though the prefix cache alone may not achieve high hit rates, it may have some reasonable contribution towards the global hit rates.

5.3.3 Commentary

We showed through empirical results that both the architectures perform reasonably well in reducing the number of full-trie lookups. We achieve up to 99.9% global hit rates for both the architectures. Further, the DSC-first architecture shows up to 4% increase in the global hit rates when compared to the prefix cache-first architecture.

However, for the DSC-first architecture we will find that the average number of memory accesses per lookup is higher in comparison to the prefix cache-architecture. This can be explained using Figures 5.6 and 5.12. Consider the as1221-upcb.2 dataset and the case where we allocate 8192 entries to both the prefix cache and the DSC. For the prefix cache-first architecture, we have 74% local hit rates in the prefix cache and 70% local hit rates in the DSC. Similarly, for the DSC-first architecture, we have 91% local hits rates in the DSC and 57% local hit rates in the prefix cache. Using these local hit rate values we can calculate the average number of memory accesses per lookup for both the architectures. We use the following equation:

$$\textit{Average number of memory accesses/lookup} = \frac{\textit{num(PC) * 0 + num(DSC) * t + num(trie) * wc}}{\textit{Total number of lookups}} \tag{5.2}$$

In the equation num indicates the number of hits in the individual components i.e the prefix cache (PC), the DSC and the trie. The variables t and wc are the number of memory accesses required by a lookup that hits the DSC and the trie respectively. In a realistic scenario the values of t and wc can vary depending upon the type of lookup. Nonetheless, we use the worst-case values for t and wc to lend more clarity to our argument. So for convenience, we consider $t = 5$ which simply indicates that a lookup that finds a hit in the DSC will take at most 5 memory accesses to find the next hop. Similarly, we consider $wc = 32$ which is the number of memory accesses a lookup will require in the worst-case if it misses both the prefix cache and the DSC.

For the upcb.2 with 0.9 million lookups, a single lookup in the prefix cache-first architecture will require only 3.4 memory access on average. In contrast, in the DSC-first architecture the average number of memory accesses per lookup is nearly 5.7. Using the prefix cache-first architecture, we can achieve up to 40% reduction in the average number of memory accesses per lookup. Clearly, a lookup in the DSC-first architecture will require few more memory accesses on average in comparison to the prefix cache-first architecture.

Evidently, the choice of a suitable architecture should be made only after realistic decisions regarding hardware implementations. If the router architects come up with a hardware design that focuses more on average memory accesses per lookup, then the prefix cache-first architecture is a definite choice for them. However, if global hit rates are the stronger consideration then the DSC-first architecture should be preferred.

Chapter 6

Cache Modeling and Optimization

In the previous chapter we presented empirical results for two different architectures (prefix cache-first and DSC-first) where we primarily considered a scenario where we allocate equal number of entries to both the prefix cache and the DSC. However, allocating equal cache entries to both caches may not yield the best global hit rates. Clearly, for the two architectures the design decision is to optimally allocate cache entries between the two caches for a given number of cache entries. In this chapter we present an analytical method that can be used to arrive an optimal cache design that gives the maximum global hit rates.

The idea is to first choose an analytical method that best models the hit rates in the two caches. Once we have decided upon an analytical method, we can formulate the design decision as an optimization problem.

In Section 6.1, we briefly describe some previous works that model cache memories. In Section 6.2, we describe our analytical method and we shows its effectiveness. In Sections 6.3 and 6.4 we present the optimization technique and follow it with the experimental results. In Section 6.5 we present results to validate the discussions in Section 5.3.3.

6.1 Related models

It is a known fact that cache behavior depends upon the locality in the destination address stream that the cache sees. Models that can measure these locality properties can help understand cache behavior. In fact, such models provide means to predict the cache hit rates.

Singh *et al.* [41] provide a model to characterize locality in reference stream. This model is in turn used to derive a method to predict hit rates. They provide a power function $u(t)$ that gives the number of unique references in the reference stream of length t:

$$u(t) = Wt^a \tag{6.1}$$

This power function captures the temporal locality in the reference stream. For instance, if $u(1000) = 300$ then at the 1000^{th} reference, we have already seen 300 unique references. In order

to obtain the parameter values W and a, they fit the power function using linear regression. They further illustrate how the power function can be used to arrive at a model for the cache miss rates. Specifically, they take the derivative of the power function given a cache size C:

$$M(C) = \frac{u(t)}{dt} = aW^{1/a}C^{1-1/a} \tag{6.2}$$

$M(C)$ gives the miss rate for a given cache of size C. In other words, this is the rate at which the unique references occur in a trace at the time when the cache is filled with unique references and is also the rate at which cache misses occur. This is true because a cache with C entries will contain at most C unique addresss where each cache entry can hold a single address reference only. Apparently, the model has shortcomings. In Eq. 6.2, if we consider $C = 0$ then the miss rate $M(C)$ should be 1. However, it turns out $M(C)$ will give zero. Clearly, the equation does not satisfy the boundary conditions. Meeting the boundary conditions is important else this will lead the optimizer to predict that the miss rates are least when cache size is zero.

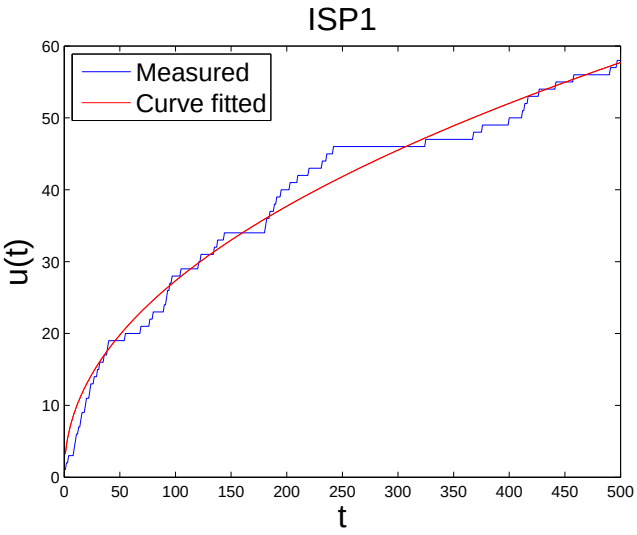

Figure 6.1: Power function $u(t)$ with respect to t for ISP1 trace

Another concern is that Eq. 6.2 does not hold for an initial transient. This can be shown in Figure 6.1 where we take a subsample of trace ISP1 containing 500 destination addresses. We then fit Eq. 6.1 with the measured $u(t)$ values. Apparently, the fit is relatively poor especially when t is less than 50. Actually, when t is less than 50, the slope of the fitted curve is above 1. This implies that the cache miss rate is in excess of 100% which is erroneous.

Another approach towards characterizing locality is reuse distance [10][38]. Reuse distance gives the measure of temporal locality in the references and is defined as $sizeof(stack)+1$. Assume we have a stack with infinite size and we push each reference onto this stack. If an incoming reference has been seen before, we remove the similar reference from its current position in the stack and we push it onto the top of the stack. The reuse distance here is the index of the stack entry

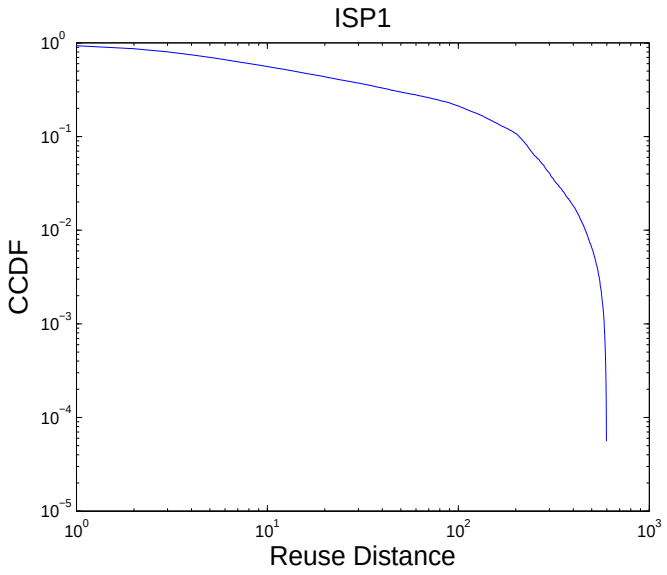

Figure 6.2: CCDF for ISP1 trace

where the reference was found. If the incoming reference has never been seen before, then we push this new entry onto the top of the stack. If the reference present at the top of the stack is seen again, then we define the reuse distance as 0. The greater the number of reuse distances closer to the top of the stack, the better is the temporal locality in the reference stream.

The distribution of the reuse distances in the packet traces can be used to predict hit rates. Specifically, we can use the CCDF (complementary cumulative distribution function) of reuse distances:

$$S(x) = Pr[X > x] = 1 - F(x) \tag{6.3}$$

where x is the reuse distance. Thus, for a given cache size, the cache miss rates can be predicted from the CCDF curve. For example, in Figure 6.2 we show the CCDF curve for the ISP1 trace. The reuse distance here can be considered as the cache size whereas the CCDF as the cache miss rates. Given the CCDF for the trace, we can fit the CCDF curve using some distribution (Pareto, Weibull etc.).

However, the CCDF is not accurate for larger reuse distances. For instance at reuse distance 1000, the cache miss rate predicted is almost 0%. Again, this will force the optimizer to believe that the best miss rate seen by the cache is 0%.

Attempts have been made to use *delays* and *strides* to measure memory reference locality in programs that use a small part of a memory. Delay is the distance (time) between two references. This is calculated by counting the total number of references between two items. Stride indicates the distance in memory between the same two references. Strides are meant to capture spatial locality. Grimsrud *et al.* [18] use these concepts to arrive at a locality surface. They count the number of times each delay and stride combination occur in the memory reference trace. Each of these combinations

are divided by the total number of references in the trace. The results are shown on a log-log scale which indicates the extent of locality. However, they do not provide a mechanism to predict cache miss rates. Further, strides are not useful to quantify spatial locality in destination address streams.

6.2 Our approach

For our work, we model cache hit rates as a power-law function:

$$H(x) = 1 - Ax^{\theta} \tag{6.4}$$

where x is the cache size and A and θ are constants. The above equation is useful for cache hit rate modeling and is also investigated by Smith [43]. However, similar to other models, Eq. 6.4 does not satisfy the boundary conditions. At cache size 0, $H(x)$ will give 100% cache hit rate. Clearly, we would need a model which gives 0% cache hit rate for a cache of size 0. In addition, it is important to predict cache hit rates of 100% when the cache size is infinity. Considering the need, we make the following changes to Eq. 6.4:

- To ensure that the equation meets the boundary condition when cache size is 0, we replace the term x^{θ} with $(x+1)^{\theta}$.
- To ensure that the equation meets the boundary condition when cache size is infinity, we place the term $(x+1)^{\theta}$ in the denominator.
- We scale the value of x directly by A. So we move A to the denominator of x. Thus, the cache hit rates increase with increasing value of θ or decrease with increasing value of A.

This gives us the following equation:

$$H(x) = 1 - \frac{1}{(\frac{x}{A}+1)^{\theta}} \tag{6.5}$$

The changes to Eq. 6.4 ensure that the modeling will not fail when the cache size is 0 or infinity. We use Eq. 6.5 to model the local hit rates for both the prefix cache as well as the DSC.

6.3 Prefix cache-first architecture

In the following section we show the effectiveness of the model for the architecture where the prefix cache precedes the DSC. We first show whether Eq. 6.5 properly models the prefix cache hit rates. For demonstration, we first try to fit the model with the observed prefix cache hit rates for the packet trace ISP3. We consider a prefix cache with 240 entries. Clearly, the model fits with the observed values very well (Figure 6.3). For the fitted curve we found $A = 11.7422$ and $\theta = 0.565$. Moreover, the curve suggests that the hit rates follow the 80/20 rule where cache hit rates increase by 20% with every increase (doubling) of the cache size.

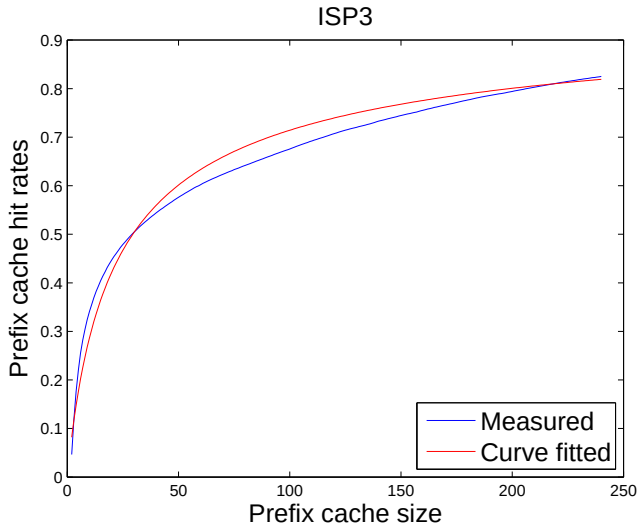

Figure 6.3: Prefix cache hit rates - Measured v. Curve fitted

Consider a scenario where we have a total of 240 cache entries. We decide to allocate 25 entries to the prefix cache and the remaining to the DSC. We also assume that the value of k to be 2. We demonstrate that Eq. 6.5 can fit very well with the observed DSC hit rates for the packet trace ISP3. Figure 6.4 shows the results of the curve fit. For the fit, we find $\theta = 1.2266$ and $A = 80.44$ for the part of the destination addresses seen by the DSC when the prefix cache has 25 entries. Apparently, the fit is good when the DSC cache size is beyond 200 entries. There is a certain degree of misfit when the DSC size is less than 50. This is due to the peculiar nature of the curve that was seen during the simulation.

We first see how the cache entries get distributed between the prefix cache and the DSC when we use an exhaustive search considering that we wish to achieve the best global hit rates. For this particular problem instance, we selected a total cache size of 240 entries. We also considered the value of k to be 2. Figure 6.5 gives the global hit rates for the simulated cache configurations for the ISP3 trace. The x-axis indicates the number of cache entries allocated to the prefix cache out of the total available cache entries. As shown, the best global hit rate is obtained when 49 entries are allocated to the prefix cache and the remaining 191 entries to the DSC. This indicates that a small prefix cache in front of the DSC is beneficial. This substantiates the point that was made in Chapter 5 regarding a small prefix cache. A small prefix cache here manages to exploit most of the temporal locality seen in the ISP3 trace. This enables the DSC to be more focused on exploiting spatial locality as well as lookups that fall in particular IP address spaces. The two behaviors combined help to increase the global hit rates.

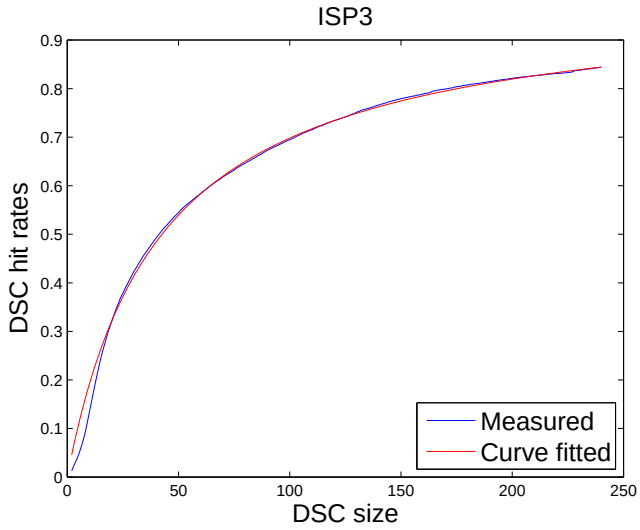

Figure 6.4: DSC hit rates - Measured v. Curve fitted when prefix cache has 25 entries

6.3.1 Formulation

The cache entry distribution between the prefix cache and the DSC can be formulated as a constrained non-linear integer optimization problem. This will essentially prevent us from using exhaustive search which is much more tedious. We use the TOMLAB [6] optimization environment for the formulation. Considering Eq. 6.5 as a method to predict hit rates, we formulate the problem of finding the optimal cache entry allocations in the following way:

$$maximize\ [y_1 \times H_1(c_1) + y_2 \times H_2(c_2)] \tag{6.6}$$

subject to:

$y_1 = L \qquad y_2 = L - H_1(c_1) \times y_1$

$H_1(c_1) = 1 - (\frac{c_1}{\beta(s)} + 1)^{-\alpha(s)}$

$H_2(c_2) = 1 - (\frac{c_2}{\beta_\phi(c_1)} + 1)^{-\alpha_\phi(c_1)}$

$\alpha_\phi(c_1) = [\alpha_s, \alpha_e]$

$\beta_\phi(c_1) = [\beta_s, \beta_e]$

$j = \theta, 2\theta, 3\theta, ..., c$

$i = j - (\theta - 1)$

$c_1 = [s, e]$ where $s = max(i)$ and $e = min(j)$

$c_1 + c_2 = c$

$c_1$and c_2 are integers

$c_1, c_2 \geq 0$

y_1 is the fraction of address references seen by the prefix cache and y_2 is the fraction of address references left-over from the prefix cache. c is the total available cache entries. $\alpha(s)$ and $\beta(s)$ are the parameters from Eq. 6.5 for the address references that find a hit in the prefix cache. Similarly, $\alpha_\phi(c_1)$ and $\beta_\phi(c_1)$ are the parameters from Eq. 6.5 for the DSC. However, $\alpha_\phi(c_1)$ and $\beta_\phi(c_1)$ depend upon the prefix cache size. More specifically, the intervals associated with $\alpha_\phi(c_1)$ and $\beta_\phi(c_1)$ are used to select the values of $\alpha_\phi(c_1)$ and $\beta_\phi(c_1)$ depending upon the value of c_1. Further, the size of the interval is defined by θ.

Figure 6.6 gives the optimization surface for the design of a prefix cache-first architecture ($c = 240$). The z-axis shows the global hit rates. The other axes show the number of cache entries allocated to the prefix cache and the DSC respectively. From the figure we can see that the optimal design decision is to allocate 47 entries to the prefix cache and the remaining 193 entries to the DSC. Also, the global hit rate is up to 90.75%. In contrast, if we allocate all the memory to the prefix cache, then we see a global hit rate of only 82.56%. Similarly, if we allocate all the memory to the DSC, the global hit rate is just around 87.24%. Clearly, allocating some entries to the prefix cache is indeed beneficial.

In fact, the cache distribution predicted by the optimization technique is very close to what we had seen through exhaustive search. The exhaustive search demonstrated that 49 entries needed to be allocated to the prefix cache. Moreover, the global hit rates seen through exhaustive search is near to 90.22%. This shows that the values determined using our optimization technique are quite close to what we see using exhaustive search.

6.4 DSC-first architecture

In this section, we describe a method to determine optimal cache entry allocation in a scenario where the DSC precedes the prefix cache. In this situation, the DSC sees all the address references and has the opportunity to exploit locality. It will be interesting to see the performance of the prefix cache given that the DSC exploits all the locality in the address references.

However, this architecture turns out to be much more complex to model due to its inclusive nature. In other words, the performance of the prefix cache has a marked effect on how the DSC performs. This can be explained from Figure 6.7. For illustration, consider that the DSC and the prefix cache have been provided with three entries each. Assume that the initial state of the caches is as in Figure 6.7(A). Suppose a new reference arrives (Figure 6.7(B)) that is already present in the DSC. As a result, there is some activity in the DSC. Further, when a reference 01111001010 arrives, the substride 1010 is evicted from the DSC (Figure 6.7(C)). However, the prefix cache still holds the prefix 101000 corresponding to the substride 1010. Consequently, any future address references that match 101000 will find the prefix cache useful (Figure 6.7(D)). This will prevent any updates for substride 1010 in the DSC. Subsequently, the DSC will always show a miss for a reference that matches the 1010 substride and this will lead to a lower DSC hit rate. We can term substrides such

as 1010 as “victim substrides”.

This cache behavior can be attributed to fortunate occurrences. Had we allocated only two entries to the DSC and the prefix cache, the DSC would had been updated with the substride 1010. Not surprisingly, the DSC in this case will show a different hit rate. Hence, the DSC hit rates may fluctuate depending upon how the prefix cache performs.

This inclusive nature of the two caches can be seen in Figure 6.8 and 6.9. For illustration, we allocate 230 entries to the DSC and we then vary the size of the prefix cache($k = 3$). Initially, when the prefix cache is less active, the DSC shows relatively stable hit rates. However, as the prefix cache size grows, the chances of it being effective increases. This leads to a sudden drop in the DSC hit rate. This is because many substrides are not updated in the DSC once the prefix cache size grows beyond a certain point. Thus, modeling such curves can be very complex.

In order to reduce the complexity, we suggest using a DSC of sufficiently large size so that the prefix cache has a meager effect on the DSC hit rates. This can be explained from Figure 6.10 which shows the global hit rates obtained from exhaustive search for the upcb.1 trace. In this case, we consider the total available memory as 400 entries. Further, we keep the value of k as 3. As shown, the global hit rates are at their peak when most of the entries are allocated to the DSC. This means that the DSC is less affected by the prefix cache which is relatively smaller in size. The result is that the DSC hit rate and the prefix cache hit rate curves are relatively less complex.

We proceed with our experiment with the criteria that the DSC will have at least 300 entries out of the available 400 entries. This assumption helps significantly in predicting global hit rates. Figure 6.11 shows the DSC hit rate curve for the upcb.1 trace. The DSC hit rates increase linearly with increasing size. This simplifies the problem and allows us to use a simple linear equation to fit the curve. The prefix cache curves are also less complex. Figure 6.12 shows the prefix cache hit rates when the DSC has 300 entries. The prefix cache hit rate curve, again, varies if the DSC has more entries. However, we found that the curves follow the power-law nature even when the DSC has 395 entries. As a result, we use Eq. 6.5 for curve fitting.

6.4.1 Formulation

The optimization problem is in ways similar to the one described in Section 6.3.1. We have:

$$maximize\ [y_1 \times H_1(c_1) + y_2 \times H_2(c_2)]$$

subject to:

$y_1 = L \qquad\qquad y_2 = L - H_1(c_1) \times y_1$

$H_1(c_1) = M(c_1) \times c_1 + A(c_1)$

$H_2(c_2) = 1 - (\frac{c_2}{\beta_\phi(c_1)} + 1)^{-\alpha_\phi(c_1)}$

$M(c_1) = [M_s, M_e]$

$A(c_1) = [A_s, A_e]$

$\alpha_\phi(c_1) = [\alpha_s, \alpha_e]$

$\beta_\phi(c_1) = [\beta_s, \beta_e]$

$j = \theta, 2\theta, 3\theta, ..., c$

$i = j - (\theta - 1)$

$c_1 = [s, e]$ where $s = max(i)$ and $e = min(j)$

$c_1 + c_2 = c$

$c_1$and c_2 are integers

$c_1, c_2 \geq 0$ and $c_1 \geq P$

$M(c_1)$and $A(c_1)$ are the parameters for the address references seen by the DSC. The intervals for $M(c_1)$and $A(c_1)$ select the values of the parameters depending upon the value of c_1 . P indicates the lower bound on the number of cache entries pre-allocated to the DSC.

Figure 6.13 gives the global hit rate curves after the optimization ($c = 400$). For the experiment, we considered $P = 300$ and as a result at most 100 entries could have been allocated to the prefix cache. The optimal global hit rate is 0.881 where 25 entries were allocated to the prefix cache and the remaining 375 entries to the DSC. This is close to what we see using the exhaustive search where 29 entries were allocated to the prefix cache (Figure 6.10).

The results show that a small prefix cache after a DSC is sufficient to hold some prefixes corresponding to the victim substrides that were evicted from the DSC. For example, for the upcb.1 trace a small prefix cache has a contribution towards to the global hit rate. However, a small prefix cache after the DSC may not be always beneficial especially in scenarios where the DSC is self-sufficient. Given this behavior, the optimization technique can prove to be useful to determine the optimal balance of cache entries for different traces.

6.5 Supplementary experiments

In Section 5.3.3 we argued that the prefix cache-first architecture performs better than the DSC-first architecture in terms of the average number of memory accesses per lookup. We further validate that argument by providing some supplementary experiment results.

We focus on finding an optimal distribution of cache entries between the prefix cache and the DSC that minimizes the number of memory accesses. To be precise, our objective is to find an optimal cache entry allocations between the prefix cache and the DSC such that the average number of memory accesses required by a single lookup is the least. We compute them using Eq. 5.2. This exercise will also show whether the DSC has a reasonable contribution in reducing the average number of memory accesses per lookup.

6.5.1 Prefix cache-first architecture

First we see how the cache entries are distributed between the prefix cache and the DSC when we use exhaustive search. For illustration we consider the total available cache entries as 240 and we select the value of k to be 2. Moreover, for Eq. 5.2 we consider the values $t = 5$ and $wc = 32$. Figure 6.14 demonstrates the distribution of cache entries for the ISP3 trace using exhaustive search. We can see that the lowest average number of memory accesses per lookup is seen when 78 entries are allocated to the prefix cache and the remaining 162 entries to the DSC.

The task of finding the optimal distribution of cache entries can be formulated as a non-linear integer optimization problem. The optimization tableau is similar to the one described in Section 6.3.1 with the only change being in the objective function. The objective function for this particular problem is:

$$minimize\ [y_1 \times H_1(c_1) \times 0 + y_2 \times H_2(c_2) \times t + (L - y_1 \times H_1(c_1) - y_2 \times H_2(c_2)) \times wc] \quad (6.7)$$

Figure 6.15 gives the optimization surface for the design of the prefix cache-first architecture. The optimizer predicts that 155 entries should be allocated to the DSC and the remaining 85 entries to the prefix cache. For the optimal distribution, the average number of memory accesses per lookup is 4.363. However, if we allocate all the entries to the prefix cache a lookup will require up to 5.6223 memory accesses. Quantitatively, the DSC in the architecture helps reduce the average number of memory accesses per lookup by 22.5% i.e we decrease the memory traffic by 22.5%. Not surprisingly, the DSC in this architecture has a better contribution in reducing the memory accesses for each lookup. This conforms to the argument we made in Section 5.3.3 regarding the prefix cache-first architecture.

6.5.2 DSC-first architecture

For the DSC-first architecture we use the optimization tableau listed in Section 6.4.1 and we use the objective function as in Eq. 6.7. Similar to Section 6.4.1, we consider that only 400 cache entries are available. We consider $k = 3$, $t = 5$ and $wc = 32$. Further, we allocate a minimum of 300 entries to the DSC as a starting point for the optimization. Figures 6.16 is the result from the optimization. Clearly, from the experiment we see that the DSC does not decrease the average number of memory accesses per lookup substantially. However, the starting point for the optimization does not shed light on the actual results. Given the current limitations in our optimization method, we use exhaustive search. Using exhaustive search we can get results on all possible cache configurations. Figure 6.17 is the result from the exhaustive search for all possible cache configurations. Evidently, from the exhaustive search we find that 395 entries need to be allocated to the prefix cache and 5 entries to the DSC. This indicates that there are less benefits of a DSC in a DSC-first architecture if we consider average number of memory accesses per lookup as a criteria. This is precisely what we expected from the discussions in Section 5.3.3 and this is primarily due to additional memory

accesses (up to 5) required by lookups that hit the DSC. In conclusion, if the router architects believe that average number of memory accesses per lookup is a concern, then they should opt for the prefix cache-first architecture.

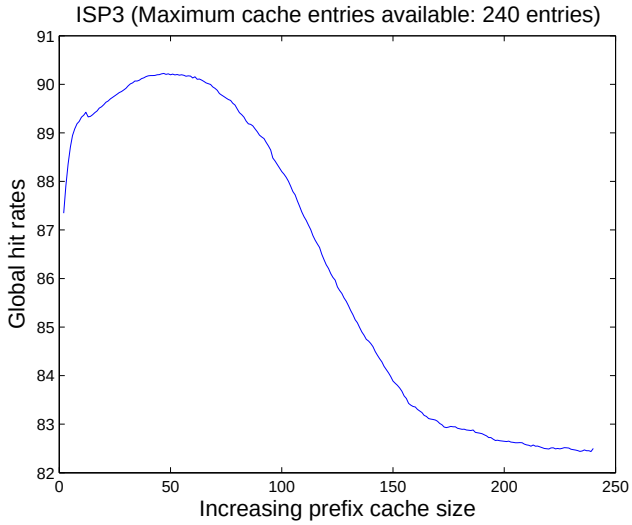

Figure 6.5: Global hit rates measured for ISP3 trace through exhaustive search

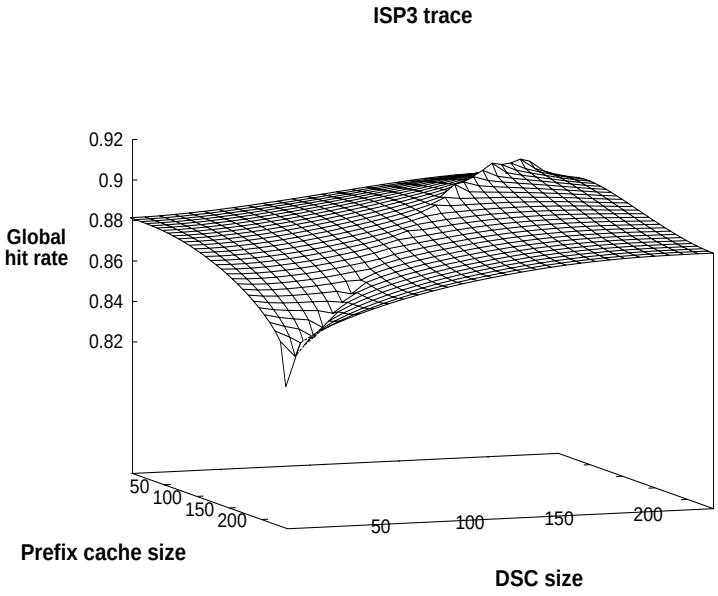

Figure 6.6: Optimization surface for the prefix cache - first architecture (global hit rates)

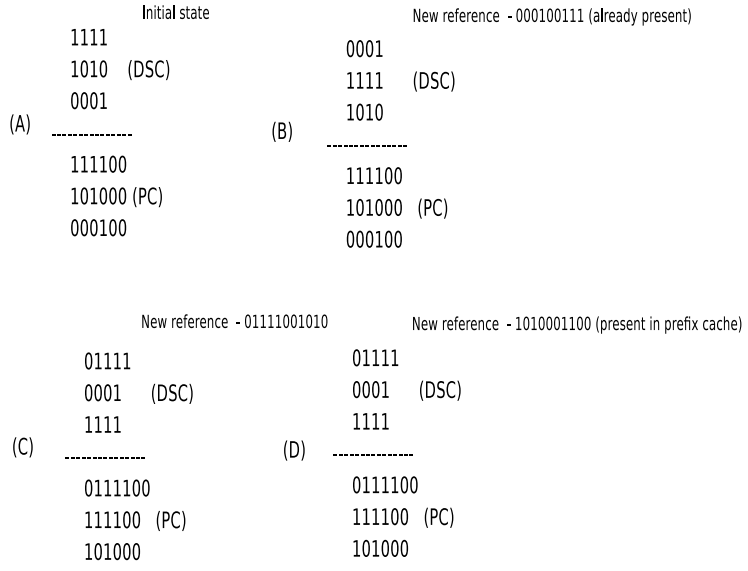

Figure 6.7: DSC and prefix cache behavior (k=3)

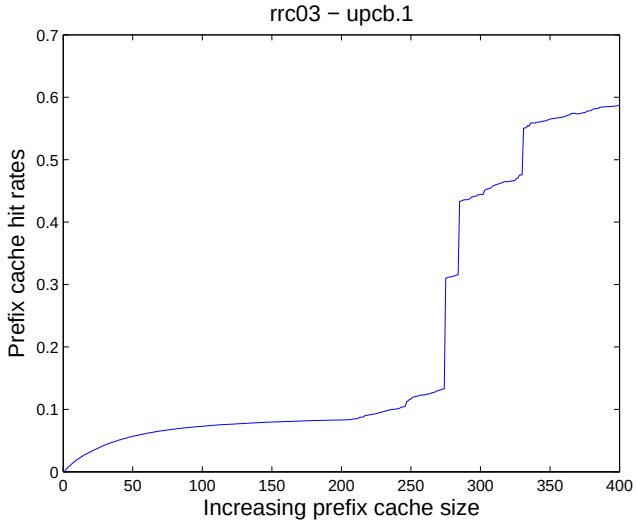

Figure 6.8: Prefix cache hit rates when DSC has 230 entries

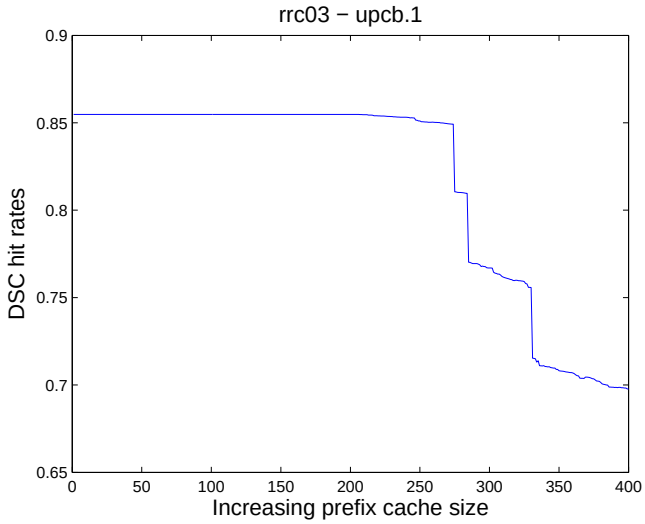

Figure 6.9: DSC hit rates on varying prefix cache sizes when DSC has 230 entries

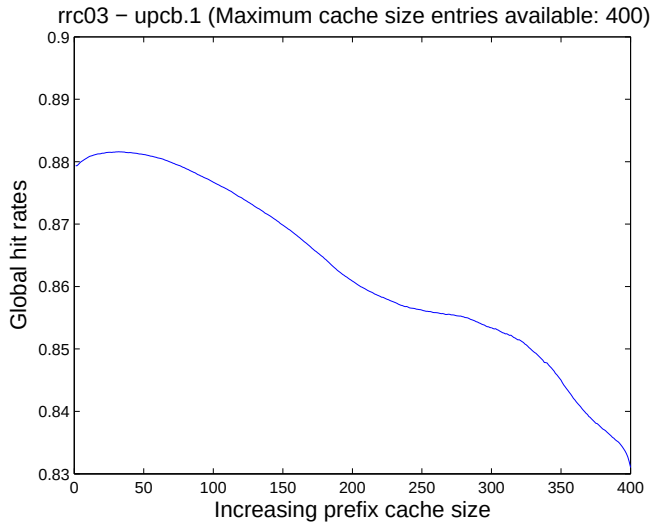

Figure 6.10: Global hit rates measured for upcb.1 trace through exhaustive search

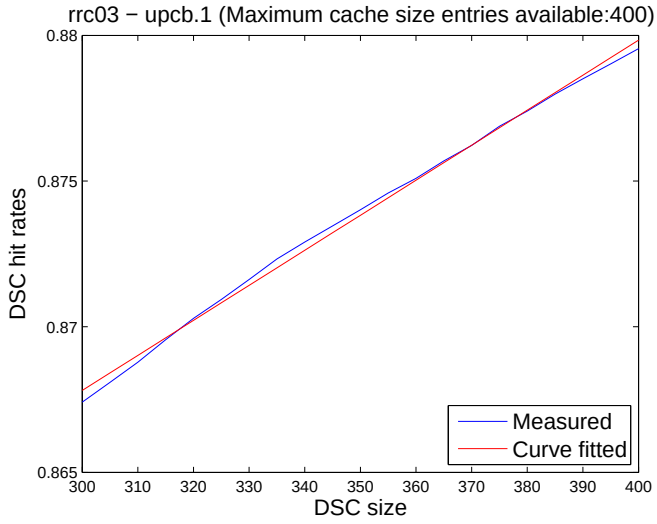

Figure 6.11: DSC hit rates - Measured v. Curve fitted when DSC size is 300 entries or more

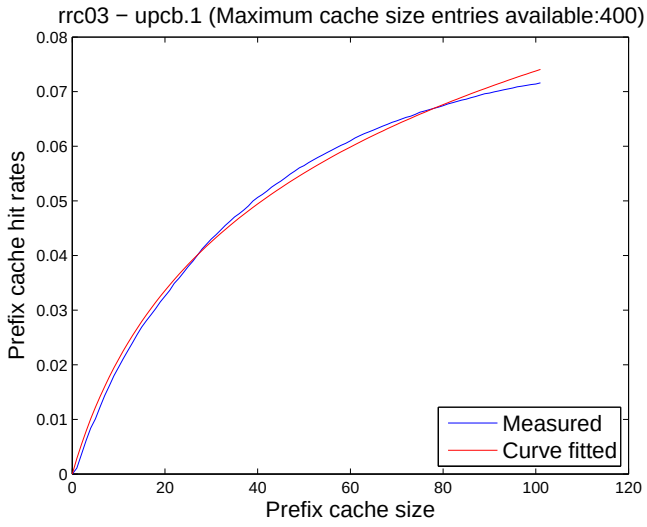

Figure 6.12: Prefix cache hit rates - Measured v. Curve fitted when DSC has 300 entries

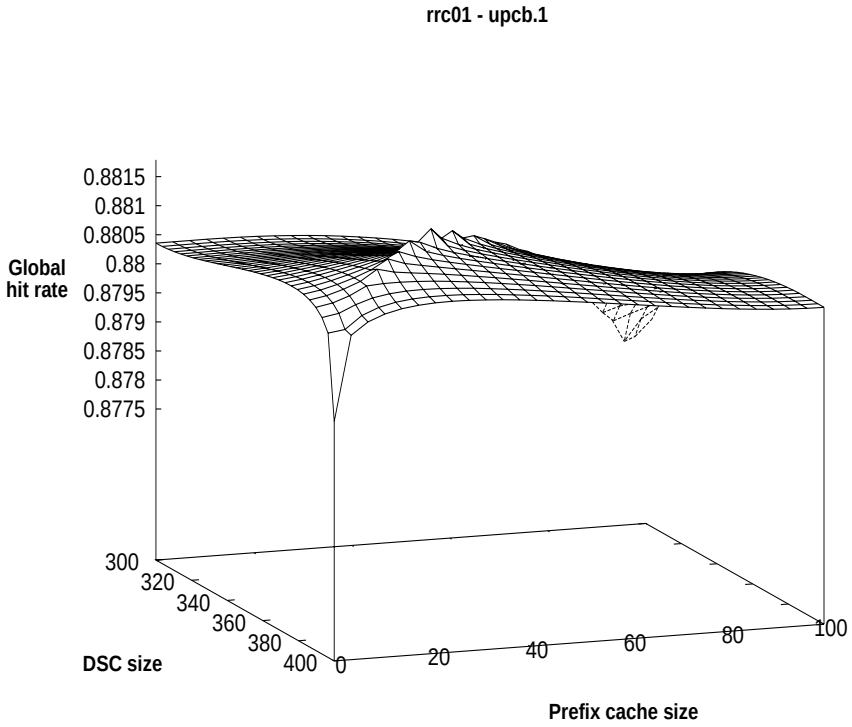

Figure 6.13: Optimization surface for the DSC - first architecture (global hit rates)

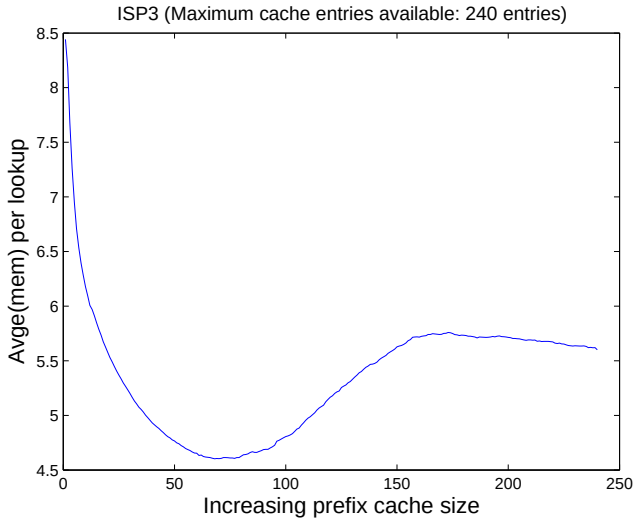

Figure 6.14: Average number of memory accesses per lookup measured for ISP3 trace through exhaustive search (prefix cache-first architecture)

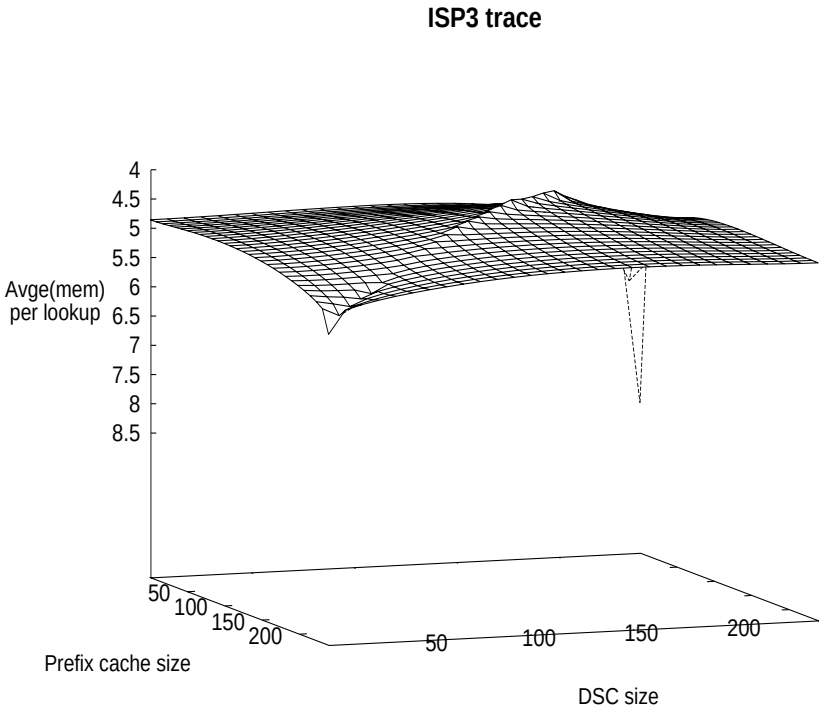

Figure 6.15: Optimization surface for the prefix cache - first architecture (Average number of memory accesses per lookup)

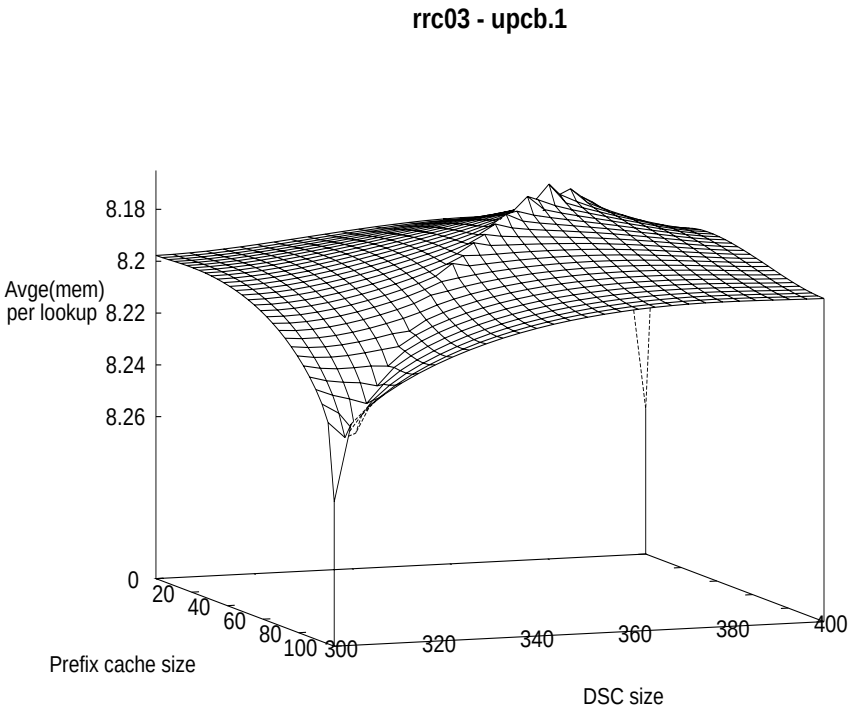

Figure 6.16: Optimization surface for the DSC - first architecture with minimum of 300 entries allocated to the DSC(Average number of memory accesses per lookup)

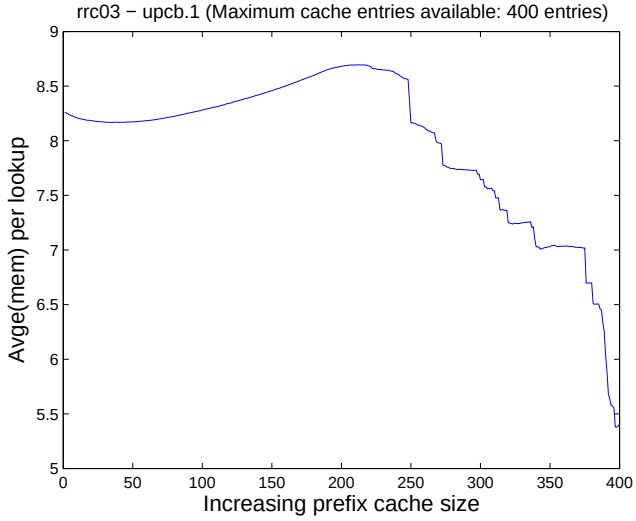

Figure 6.17: Average number of memory accesses per lookup measured for rrc03-upcb.1 trace through exhaustive search (DSC-first architecture)

Chapter 7

Conclusions & Future Work

7.1 Conclusion

In this thesis, we propose two architectures that can be used to reduce IP lookup latencies for incoming destination addresses. In particular, we use a novel cache organization called the dynamic substride cache (DSC) along with a prefix cache to reduce the number of full-trie lookups. Firstly, we tested the utility of a DSC when it is placed between the prefix cache and the leaf-pushed trie (prefix cache-first architecture). We show experimentally using different traces that the DSC is indeed effective in ensuring that fewer full-trie lookups are performed. A clear benefit of the DSC is that it is has great utility even when we have a prefix cache that is delivering good hit rates. In fact, we achieve nearly 88% hit rates in the DSC even when the prefix cache has reached its maximum hit rate. This shows that the DSC is able to assist a good percentage of lookups that are misses in the prefix cache. We also demonstrated that any lookup that is assisted by the DSC via a substride has to do at most of 5 memory accesses on average instead of a full-trie lookup that may cost up to 19.7 memory accesses on average.

Our experiments rely on finding a suitable value for the parameter k that is used to generate the substrides. We showed that the DSC performance indeed varies in terms of the local hit rates and the average number of memory accesses under different values of k. We use these results to indicate the suitable value of k for generating the substrides. Importantly, using this architecture we achieve a global hit rate of up to 99.88% when we set the value of k as 3. This result is significant since the architecture ensures that only 0.12% of the lookups are required to do the full-trie lookups. We put up a strong case especially for traces that show less locality. For these traces we achieve global hit rates up to 95% if we consider only a prefix cache in the architecture. However, if we use the prefix cache in conjunction with the DSC, we achieve global hit rates up to 99.34%. This also suggests that our architecture can perform better even when there is a limited amount of locality in the incoming destination address stream.

Additionally, we introduced a DSC-first architecture where the prefix cache is placed between the DSC and the leaf-pushed trie. This architecture retains the prefix cache which enables it to

exploit the locality in destination address streams. We showed through empirical results that for some traces the DSC alone can achieve significant global hit rates. In contrast, we showed that for some traces the prefix cache is certainly beneficial and does make some contribution towards the global hit rates. Comparatively, we find that the DSC-first architecture is better than the prefix cache-first architecture where we improve up to 4% in terms of global hit rates.

Another important consideration in an IP lookup scheme is the requirement for incremental updates. Incremental updates are frequent and it is important for our two architectures to support addition and deletion of prefixes. In order to make incremental updates possible in a leaf-pushed trie, we suggested storing extra information on prefixes in the leaves as well as the internal nodes. We demonstrated that these additional information on prefixes is essential for incremental updates in a leaf-pushed trie.

In addition, we provide schemes that need to be followed in order to ensure that the DSC is consistent after the addition or deletion of prefixes from a leaf-pushed trie. However, we required the scheme to be conservative as the DSC has little control information on the prefixes that were added or deleted.

Lastly, we investigated the optimal distribution of memory between the prefix cache and the DSC for the two architectures. We proceeded first by finding a model that can help us predict the hit rates in the two caches. Initially, we presented some existing models but showed that they may not be suitable for our work. As an alternative, we presented a model that satisfied the boundary conditions and was able to adequately mimic the hit rates in the prefix cache and the DSC. Thereafter, we used optimization techniques to arrive at an optimal distribution of cache entries. We also argued that we arrive at an optimal distribution that is quite close to what we find using exhaustive search.

7.2 Future Work

There is an important issue that could be addressed as part of our future work. A potential concern in both architectures is that we need to stall lookups while misses or DSC hits are being serviced. For instance in the prefix cache-first architecture, we need to stall lookups in the prefix cache while lookups are being performed in the DSC or the leaf-pushed trie. Further, we need to stall lookups in the DSC when a previous successful lookup in the DSC proceeds to the leaf-pushed trie. Such frequent stalls in between lookups can induce latency.

A potential solution to this problem is using buffers between the prefix cache and the DSC for pipelining purposes. As an example, we demonstrate the use of two cache miss buffers (CMB1 and CMB2) between the prefix cache and the DSC in Figure 7.1 for a prefix cache-first architecture. The CMB1 buffer can be used to store lookups from prefix cache misses in case any lookup is still in process in the DSC. The CMB2 buffer can be used to store lookups from DSC in case a lookup in the leaf-pushed trie is still in process. The CMB1 buffer allows the prefix cache to perform lookups even when lookups are being performed in the DSC or the leaf-pushed trie. However, the prefix cache

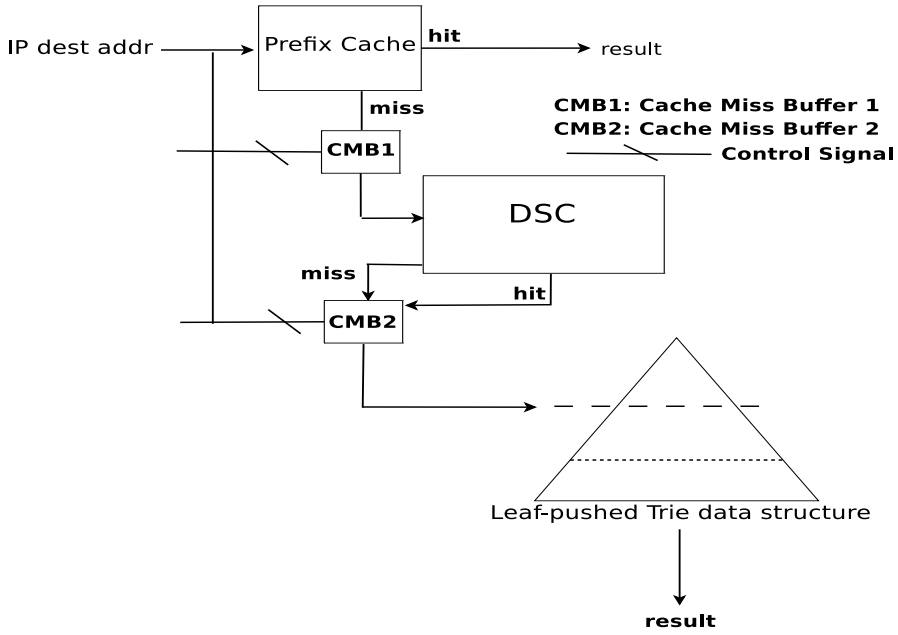

Figure 7.1: A prefix cache-first architecture with cache miss buffers

needs to stall if the CMB1 buffer gets full. Similarly, the CMB2 buffer allows the DSC to perform lookups even when lookups are in process in the leaf-pushed trie. Again, DSC lookups need to stall when the CMB2 buffer gets full. It should be noted that we need to stall the lookup process in the two caches while we are updating a prefix and a substride from a previous lookup in the leaf-pushed trie. The above discussion of using buffers is also applicable to the DSC-first architecture.

Even though the preliminary data path is available to us, we still need to experimentally evaluate the improvements the two buffers might provide in terms of the number of clock cycles. Moreover, we still need to determine at what intervals the two buffers need to be signaled to flush out the lookups for further processing.

Bibliography

[1] BGP routing table analysis report. http://bgp.potaroo.net.

[2] Finnish University and Research Network (Funet). http://www.csc.fi/english/funet/.

[3] Freescale semiconductor. PowerPC MPC7451. http://www.freescale.com/.

[4] Merit Network Inc. Internet performance measurement and Analysis (IPMA) statistics and daily reports . http://www.merit.edu.

[5] Ripe Network Centre. http://www.ripe.net/projects/ris/rawdata.html.

[6] Tomlab optimization. http://tomopt.com/tomlab/.

[7] M. Akhbarizadeh and M. Nourani. An IP packet forwarding technique based on partitioned lookup table. In *IEEE International Conference on Communications*, volume 4, pages 2263–2267, 2002.

[8] M. Akhbarizadeh and M. Nourani. Efficient prefix caching for network processors. In *IEEE Symposium on High Performance Interconnects*, pages 41–46, August 2004.

[9] K. Andrusky and M.H MacGregor. Improving packet classification: Learning from traffic. In *Internet Technologies and applications*, September 2005.

[10] M. Brehob and R. Enbody. An analytical model of locality and caching. Technical Report MSU-CSE-99-31, CSE Department. Michigan State University, 1999.

[11] T.C Chiueh and P. Pradhan. High-performance IP routing table lookup using CPU caching. In *IEEE INFOCOM*, volume 3, pages 1421–1428, May 1999.

[12] T.C Chiueh and P. Pradhan. Cache memory design for Internet processors. *IEEE Micro*, 20(1):28–33, Jan/Feb 2000.

[13] I.L Chvets and M.H MacGregor. Multi-zone caches for accelerating IP routing table lookups. In *High Performance Switching and Routing*, pages 121–126, May 2002.

[14] E. Cohen and C. Lund. Packet classification in large ISPs: design and evaluation of decision tree classifiers. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 73–84, 2005.

[15] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 97–122, 2004.

[16] W.N Eatherton. Abstract hardware-based Internet protocol prefix lookups. Master's thesis, Washington University, May 1999.

[17] M. Faeizipour and M. Nourani. Wire-speed TCAM based architectures for multimatch packet classification. *IEEE Transactions on Computers*, 58(1):5–17, January 2009.

[18] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. In *Computer performance evaluation, modeling techniques and tools*, pages 369–388, May 1994.

[19] P. Gupta. *Algorithms for Routing Lookups and Packet Classification*. PhD thesis, Stanford University, 2000.

[20] P. Gupta and N. McKeown. Algorithms for packet classification. In *IEEE Network Special Issue*, volume 15, pages 24–32, March/April 2000.

[21] R. Jain and S.A Routhier. Packet trains: Measurements and a new model for computer network traffic. *IEEE Journal on Selected Areas of Communications*, 4:286–295, 1986.

[22] W. Jiang, Q. Wang, and V.K Prasanna. Beyond TCAMs: An SRAM-based parallel multi-pipeline architecture for terabit IP lookup. In *IEEE INFOCOM*, pages 1786–1794, 2008.

[23] S. Kasnavi, P. Berube, V. Gaudet, and J.N Amaral. A cache-based Internet Protocol address lookup architecture. *Computer Networks*, 52(2):303–326, 2008.

[24] G. Keramidas, P. Petoumenos, and S. Kaxiras. Cache replacement based on reuse-distance prediction. In *International Conference on Computer Design*, pages 245–250, October 2007.

[25] D. Knuth. The art of computer programming. Addison-Wesley Professional.1997.

[26] H. Liu. Reducing cache miss ratio for routing prefix cache. In *IEEE GLOBECOM*, volume 3, pages 2323–2327, November 2002.

[27] M.H MacGregor. Design algorithms for multi-zone IP address caches. In *High Performance Switching and Routing*, pages 281–285, June 2003.

[28] P. Mehrotra and P.D Franzon. Binary search schemes for fast IP lookups. In *IEEE GLOBECOM*, volume 2, pages 2005–2009, November 2002.

[29] X. Meng, Z. Xu, B. Zhang, G. Huston, S. Lu, and L. Zhang. IPv4 address allocation and the BGP routing table evolution. In *ACM SIGCOMM Computer Communication Review*, volume 35, pages 71–80, January 2005.

[30] N.B Neji and A. Bouhoula. Dynamic scheme for packet classification using splay trees. In *International Workshop on Computational Intelligence in Security for Information Systems CISIS*, volume 53, pages 211–218, October 2008.

[31] D. Pao, Y. Li, and P. Zhou. Efficient packet classification using TCAMs. *Computer Networks*, 50(18):3523–3535, December 2006.

[32] L. Peng, W. Lu, and L. Duan. Power efficient IP lookup with supernode caching. In *IEEE GLOBECOM*, volume 48, pages 215–219, November 2007.

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

[34] R.L Rudell. Multiple-valued logic minimization for PLA synthesis. Technical Report UCB/ERL M86/65, EECS Department, University of California, Berkeley, 1986.

[35] S. Sahni and K.S Kim. Efficient construction of variable-stride multibit tries for IP lookup. In *IEEE Symposium on Applications and Internet*, pages 220–227, 2002.

[36] R. Sedgewick. Algorithms in C++. Addison-Wesley. 1990.

[37] D. Shah and P. Gupta. Fast incremental updates on Ternary CAMs for routing table lookups and packet classification. In *Proceedings of Hot Interconnects 8*, pages 145–153, August 2000.

[38] W. Shi, M.H MacGregor, and P. Gburzynski. Traffic locality characteristics in a parallel forwarding system. *International Journal of Communications systems*, 16(9):823–839, 2003.

[39] K. Shiomoto, M. Uga, M. Omatani, S. Shimizu, and T. Chamaru. Scalable Multi-Qos IP + ATM switch router architecture. In *IEEE Communications Magazine*, volume 38, pages 86–92, December 1999.

[40] W.L Shyu, C.S Wu, and T.C Hou. Multilevel aligned IP prefix caching based on singleton information. In *IEEE GLOBECOM*, volume 3, pages 2345–2349, November 2002.

[41] J.P Singh, H.S Stone, and D.F Thiebaut. A model of workload and its use in miss-rate prediction for fully associative caches. *IEEE Transactions on Computers*, 41(7):811–825, July 1992.

[42] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet classification using multidimensional cutting. Technical Report CS2003-0736, CSE Department. UCSD, 2003.

[43] A. J Smith. Two methods for the efficient analysis of memory address data. *IEEE Transactions on Software Engineering*, 3(1):94–101, 1977.

[44] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 17(1):1–40, February 1999.

[45] B. Talbot, T. Sherwood, and B. Lin. IP caching for terabit speed routers. In *IEEE GLOBECOM*, volume 2, pages 1565–1569, May 1999.

[46] L.C Wuu, T.J Liu, and K.M Chen. A longest prefix first search tree for IP lookup. *Computer Networks*, 51(12):3354–3367, August 2007.

[47] F. Zane, G. Narlikar, and A.Basu. CoolCAMs: A power efficient TCAMs for forwarding engines. In *IEEE INFOCOM*, volume 1, pages 42–52, March-April 2003.

[48] K. Zheng, H. Che, Z. Wang, and B. Liu. TCAM-based distributed parallel packet classification algorithm with range matching solution. In *IEEE INFOCOM*, volume 1, pages 5–17, March 2005.