Two-level Cache Architecture to Reduce Memory Accesses for IP Lookups

Sunil Ravinder
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Email: sravinde@cs.ualberta.ca

Mario A. Nascimento
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Email: mario.nascimento@ualberta.ca

M.H. MacGregor
Dept. of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2E8
Email: mike.macgregor@ualberta.ca

***Abstract*—Longest-prefix matching (LPM) is a key processing function of Internet routers. This is an important step in determining which outbound port to use for a given destination address. The time required to look up the outbound port must be less than the minimum inter-arrival time between packets on a given input port. Lookup times can be reduced by caching address prefixes from previous lookups. However all misses in the prefix cache (PC) will initiate a traversal of the routing table to find the longest matching prefix for the destination address. This table is stored in memory so a traversal requires multiple (perhaps many) memory references. These memory references become an increasingly serious bottleneck as line rates increase. In this paper we present a novel second level of caching that can be used to expedite lookups that miss in the PC. We call this second level a dynamic substride cache (DSC). Extensive experiments using real traffic traces and real routing tables show that the DSC is extremely effective in reducing the number of memory references required by a stream of lookups. We also present analytical models to find the optimal partition of a fixed amount of memory between the PC and DSC.**

I. Introduction

One of the primary functions of IP routers is packet forwarding. This requires finding the longest prefix in the routing table that matches the destination address of the packet. Once the longest matching prefix is found, the incoming packet is forwarded to its destination via the outbound port associated with that longest prefix. However, increasing line rates and increasing routing table sizes demand increasingly fast mechanisms to perform lookups.

It is often a good strategy in longest-prefix matching (LPM) to cache prefixes from previous lookups. The prefix cache (PC) is used to accelerate subsequent lookups [7], [13]. This exploits the temporal and spatial locality in a stream of destination addresses. In the context of Internet traffic, *temporal locality* means that there is a high probability that packets destined for the same address may arrive at the router again within a short time interval. This situation occurs because of traffic burstness and network congestion [14]. Spatial locality means that packets often reference the same subnet.

However, some destination addresses will not match any prefix currently in the PC. These are referred to as cache misses. Each miss will require a traversal of the routing table, which is stored in memory. Servicing a miss will require 32 memory accesses in the worst case. If we store the routing table in a 25 nsec-DRAM, a single worst-case traversal will require 800 nsec. However, even for an OC-192 link we have to ensure that lookups do not take more than 100 nsec. This represents approximately three to eight memory accesses, if sufficient pipelining is used in the DRAM implementation[10]. We propose an architecture that requires just 0.4 to 4.3 memory accesses per lookup in all the cases we examined.

PC hit rates are generally expected to decrease as line rates increase because traffic locality is expected to decrease as larger amounts of traffic are aggregated at higher rates. A similar effect may arise in the transition from IPv4 to IPv6 addresses due to the tremendous increase in the address space enabled by this transition. Thus, the effectiveness of prefix caching will likely decline in the future. Additional methods for expediting address lookups are urgently required.

We propose to add a second cache called a distributed substride cache (DSC) to accelerate handling lookups that miss in the PC. We show that the DSC is very effective in reducing the average number of memory accesses per IP lookup for typical traffic streams. We also show that the DSC is very effective even for traffic with low locality.

Rather than prefixes, the DSC contains substrides that are obtained dynamically by reducing the length of prefixes matched by previous lookups. This process reduces the length of a prefix by a fixed value k to produce a **substride**.

These substrides are of length $(32 - k)$ and are associated with a pointer that gives a short-cut into the routing table. Any lookup that misses the PC and then finds a hit in the DSC will be able to skip $(32 - k)$ memory accesses during the routing table traversal.

For example, suppose that we are dealing with 10-bit addresses rather than 32-bit IPv4 addreses. Also suppose that, for a lookup with destination address 1001001101, we find that the longest matching prefix is 100100110. After the lookup has completed, we create the substride 1001001 by reducing the length of the prefix by a fixed value, for example $k = 2$. This substride would then be stored in the DSC to be used by future PC misses. Of course, the performance of the DSC varies with different values of k. We use trace-based simulations and real-world data (traces and routing tables) to arrive at an appropriate value for k.

A possible concern about using a two-level cache architec-

978-0-9836283-0-9

This paper was peer reviewed by subject matter experts for publication in the Proceedings of ITC 2011

ture may be that the architecture will suffer from additional latency due to adding the DSC to the data path. However, new router architectures, such as ViAggre [12], Pipelined caches [16], and SEATTLE [18], show that two-level caching is viable. These proposals make caching increasingly feasible by reducing the cost of a cache miss [8][17].

This paper is organized as follows: Sections II and III present related work and the routing table representation. Section IV describes the DSC and how it operates. In Section V, we present an analytical model for predicting cache hit rates and a nonlinear optimization tableau that can be used to arrive at optimal designs for the PC/DSC architecture. We also present experimental results for some example designs. Finally, we present our conclusions in Section VI.

II. Related Work

IP address prefix caching has been a popular field of research, and many methods have been developed to improve its effectiveness. Early research on prefix caching was spurred by the work of Talbot et al.[30]. The authors proposed the use of IP address caching for terabit-speed routers. They demonstrated that real IP traffic exhibits temporal locality of destination addresses and developed a cache to evaluate the performance of caching frequently-referenced addresses. The authors showed that cache hit rates were greater than 80% for a cache of reasonable size.

Chiueh et al.[6] developed an architecture for IP address caches known as a *host address cache*. The architecture exploits the hardware caches in commodity CPUs and treats IP addresses as virtual memory addresses. Their lookup algorithm was based on two data structures: the *host address cache* (HAC) and *network address routing table* (NART). The HAC used the Level-1 (L1) cache in the CPU. The NART consisted of three tables each containing the prefixes from the IP routing table. A disadvantage of this architecture is that changes in virtual-to-physical address mappings will result in conflict misses.

Shyu et al.'s aligned-prefix caching (APC) splits the cache based on prefix lengths [25]. The APC uses two caches: an aligned-24 cache that caches all prefixes of length 24 or less and an aligned-32 cache that caches all prefixes of length between 24 and 32. This results in better utilization of the aligned-24 cache because most of the traffic will likely match prefixes of length greater than 24. As a result, the aligned-24 cache is not polluted by prefixes of length less than 24. MacGregor [20] proposed a dynamic approach to split the cache. Here, the lengths at which prefixes are split are no longer fixed but vary depending upon the traffic.

Liu [19] proposed the idea of IP prefix caching using a routing table of reduced size. He demonstrated that pruning and mask extension techniques could be used to reduce routing table size while increasing prefix cache miss rates by only 11%-14%. Unlike our method, Liu's scheme requires a significant amount of computation to build the routing table.

Kasnavi et al. [15] provide a different scheme for prefix caching. They expand all the prefixes until they become disjoint prefixes of 17 bits in length. The expansion scheme is different from that in [19], but there is no significant advantage in terms of routing table processing overhead when compared to a leaf-pushed trie. Further, they propose allocating a prefix cache and IP address cache together in the same block of cache memory. They give half of the memory to each cache. Understandably, using half of the memory for an IP address cache does not scale well in terms of performance, and yields nearly twice the miss rate of a prefix cache of the same total size. Kasnavi et al. use Chvets et al.'s [7] multizone method to split the two caches.

Akhbarizadeh et al. [5] developed software-based full-prefix caching for parent prefixes. Instead of restricting the parent prefix to be cached, they generate a minimally expanded parent prefix which can be cached and requires no routing table preprocessing. Though the authors give a suitable method to cache parent prefixes, their method does not significantly increase the hit rates in the prefix cache.

TCAM-based caching techniques are presented in [19],[29]. While TCAMs require just one clock cycle for a prefix lookup, they suffer from high power consumption [11][31]. In addition, TCAM-based methods require preprocessing before a new prefix can be cached.

Peng et al. [21] proposed supernode caching to efficiently reduce IP lookup latency. Supernodes are nodes of a tree bitmap [9]. 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 used by subsequent lookups.

Uga et al. [24] proposed storing intermediate nodes of a Patricia tree [23] 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. This would require considerable preprocessing. Further, the authors fail to establish the benefits of using three different CAMs instead of one large CAM.

III. Routing table representation

We use tries to represent routing tables because they have a low processing time, especially during updates. A prefix in a routing table is represented by a path from the root of the trie down to a node that contains the next-hop information. We call the nodes containing next-hop information **decision nodes**. In addition, we refer to all nodes other than leaves as *internal nodes*. Suppose $R = \{r_1, r_2, ..., r_N\}$ is a set of N prefixes in a routing table, and $H = \{h_1, h_2, ..., h_M\}$ is a set of M unique next hops. Then:

- For an entry $\langle r_p, h_p \rangle$ in R, r_p is a **parent prefix** if there exists an entry $\langle r_i, h_i \rangle$ 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 $\langle r_i, h_i \rangle$ and $\langle r_j, h_j \rangle$ in *R*, the prefixes r_i and r_j are **independent** if neither of them is a parent prefix of the other.

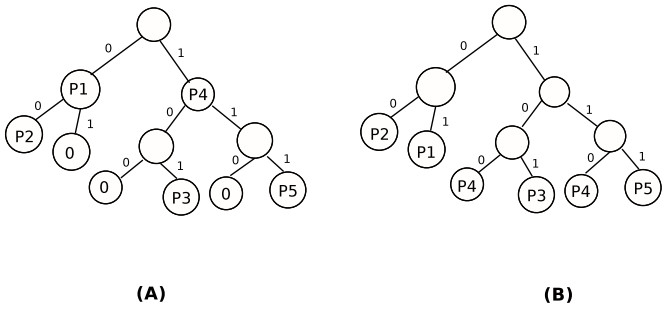

Fig. 1: (A) Trie representing prefixes (B) Leaf-pushed trie

An example of a trie data structure representing a routing table is shown in Figure 1-A. Here, some of the internal nodes contain next-hop information. We use a leaf-pushed trie (Fig. 1-B) where the next hops of the parent prefixes are pushed to the leaves of the trie instead of storing them in the internal nodes [28]. By definition, the next hop h_p of a parent prefix r_p is pushed within the subtrie rooted by the internal node that initially contained the next hop h_p.

For example, in Figure 1-A, the internal node containing the next hop $h_p = P4$ is a root of a subtrie. We push the next hop $h_p = P4$ to the empty leaves of the subtrie (those with a null, or "0" entry, in Fig. 1-A). During this leaf-pushing process, if a leaf in the subtrie turns out to be a decision node, we do not push the next hop $h_p = P4$ to that leaf because the information from the parent has lower priority. On the other hand, if a leaf is storing a 0, then we replace 0 with the next hop $h_p = P4$. In the example illustrated in Figure 1-B the parent prefixes 10* and 0* have been pushed to the leaves. An independent prefix obtained from a parent prefix after leaf-pushing is also called a minimally expanded prefix [5].

In our architecture, it is important to use a leaf-pushed trie to prevent incorrect lookups. We do leaf-pushing in the trie to prevent any prefix r_i in the trie from being a parent of a substride. If a prefix is a parent of substride, then a lookup that should match r_i may end up using an incorrect reference, with the result that it cannot find the next hop. For example if the parent prefix 100100 exists in the routing table, then the destination address 100100101 will match the substride 1001001*. Traversing a non-leaf-pushed trie starting from this substride will fail to find a match, though logically it should have matched the parent prefix 100100.

IV. DSC Operation

The DSC is very different in design from the conventional PC. A PC stores prefixes in its tag array, and the data array contains the next hop corresponding to each prefix. The DSC stores substrides in its tag array, and the data array contains the corresponding $\langle substride, \textit{trie node address} \rangle$ pair. The substrides cached in the DSC are used to assist lookups that are compulsory misses in the PC. This assistance comes in the form of a short-cut into the trie that represents the routing table. This short-cut reduces the number of memory accesses required to resolve the next hop.

Once a lookup completes via the DSC, we can extract the substride from the $\langle substride, \textit{trie node address} \rangle$ pair to generate the prefix to be stored in the PC, as well as the substride to be stored in the DSC.

The substrides in the DSC are obtained dynamically by applying a length-reduce transformation to a prefix recently looked up. This is illustrated in Figure 2, where we apply length-reduce with *k = 3* to the prefix 10001101*. We get the substride 10001*, which is a common ground for the prefixes that have the next hops P1, P2, and P3. If we cache substride 10001*, it can assist future address lookups that refer to the address space represented by the substride 10001*. This is unlike the prefix 10001101*, which can only assist lookups that refer to the address space captured by 10001101*. The substride 10001* captures a wider address space in comparison to the prefix, so more addresses will be covered by the substride.

With the choice of *k = 3*, a lookup that hits in the DSC will require one memory reference to retrieve the trie node at 1xA, plus at most three more memory accesses to find the matching leaf, and a final memory access to retrieve the next hop information. That is, choosing *k = 3* means that a hit in the DSC will result in at most five memory references to retrieve the matching next hop. Without this shortcut, a lookup that misses in the PC could be forced to traverse the full height of the trie to find a match. In the case of IPv4, this could result in up to 32 memory references.

Once a lookup has completed we know the prefix that matched, and we therefore store this prefix in the PC. We also transform the prefix to a substride, and store it in the DSC.

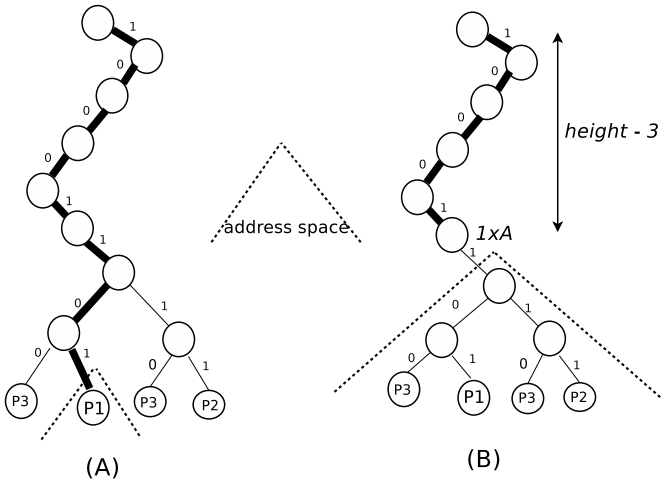

Fig. 2: (A) IP address space captured by prefix (B) IP address space captured by substride

The value chosen for k in the length-reduce operation has a significant effect on the performance of the DSC. The larger the value of k, the more levels we skip while generating the substride, and the greater the number of hits in the DSC. However, lookups that hit in a DSC that uses $k = 3$ will have to perform at least one more memory access than hits in a DSC that uses $k = 2$. Thus, there is a natural tension between increasing DSC hit rates and decreasing the number of memory references for a lookup.

We found experimentally that the value of k can be selected anywhere between two and four. We found that this results in three to five memory accesses per lookup, on average. While we could have chosen $k = 1$, at this value the DSC would show relatively low hit rates, meaning more memory accesses due to worst-case lookups. The upper limit on k is set by the permissible number of memory accesses at the interface line rate. To meet the requirement for no more than eight memory accesses at OC-192 line rates [10], we chose $k = 3$. Different memory cycle times or interface rates will result in different values for k. In [22], we present extensive results to support our use of $k = 3$, based on real traffic traces and real routing tables.

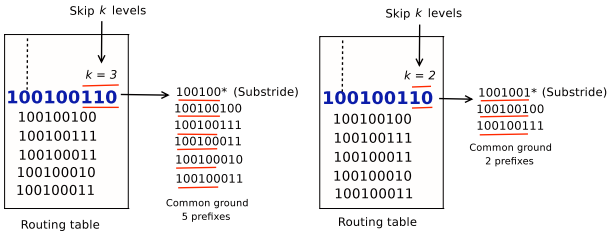

Fig. 3: Effects of value of *k*

V. Optimal Design and Experimental Results

For the PC/DSC architecture, the design challenge is to optimally allocate a fixed total budget of cache lines between the two caches. In this section, we present analytical methods that can be used to arrive at a PC/DSC design that is optimal in the sense that it gives the lowest average number of memory accesses per lookup at a given value of k. We first develop an analytical model that can predict the hit rates in the two caches. Then we formulate the design decision as an optimization tableau.

A. Modeling Cache Hit Rates

It is known 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 a means to predict cache hit rates.

Singh *et al.* [26] provide a model to characterize locality in a reference stream. They use a power function $u(t)$ that gives the number of unique references in a reference stream of length t:

$$u(t) = Wt^a \tag{1}$$

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 to the reference stream using linear regression. They further illustrate how the power function can be used to arrive at a model to predict cache miss rates by taking the derivative of the power function, and then setting $u(t)$ equal to the given cache size C:

$$M(C) = \frac{d}{dt}u(t) = aW^{1/a}C^{1-1/a} \tag{2}$$

$M(C)$ is the miss rate for a 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 it is also the rate at which cache misses occur. However, this model has some significant shortcomings. Equation (2) predicts a hit rate of 100% at a cache size of zero. Meeting this boundary condition is crucial to prevent the optimizer from predicting that the miss rate will be lowest when the cache size is zero.

A solution for this shortcoming of the footprint function was proposed in [20]:

$$u(t) = \left(\frac{t}{a}\right)^a - 1 \tag{3}$$

$$M(C) = (C+1)^{1-\frac{1}{a}} \tag{4}$$

The author then takes the first derivative of the above equation to derive the miss rate model. It was shown that (4) models hit rate curves reasonably well. However, we develop a new model in the following and demonstrate that it is superior to (4). We start with the power-law function from [27]:

$$H(x) = 1 - Ax^{\theta} \tag{5}$$

where x is the number of cache lines and A and θ are constants.

However, we need to make some modifications to satisfy the boundary conditions. First, to ensure that the equation meets the boundary condition of a hit rate of zero when the cache size is zero, we replace the term x^{θ} with $(x+1)^{\theta}$. Next, to ensure that the equation meets the boundary condition of 100% hit rate when the cache size is infinity, we place the term $(x+1)^{\theta}$ in the denominator. We also scale the value of x directly by A. Thus, cache hit rates will increase with increasing values of θ and they will decrease with increasing values of A. θ captures the degree of locality of the reference stream (higher values reflect higher locality) while A is used to enable better fit to cases of slow initial increases in hit rate. This gives us the following equation:

$$H(x) = 1 - \left(\frac{x}{A} + 1\right)^{-\theta} \tag{6}$$

We use (6) to model hit rates for both the PC and the DSC. The values of the two parameters, A and θ are arrived at by fitting this equation to the PC or DSC hit-rate behavior of any trace of interest, across a range of cache sizes, x. We tested this model by examining the residual sum of squares between the actual and predicted hit rates for PC and DSC sizes ranging from zero to 250 lines, and for several different packet traces. The fit is very good, improving on the residual sum of squares for the footprint function proposed in [20] by one or two orders of magnitude.

B. Optimal Design

Our strategy is to adjust the cache configuration to minimize the number of memory references required by a given trace of IP lookups. The metric we use is to rank designs is the average number of memory accesses per lookup (*AvgMemLookup*):

$$AvgMemLookup = \frac{num(PC) \times 0 + num(DSC) \times t + num(trie) \times wc}{\textit{Total number of lookups}} \quad (7)$$

In (7), $num(PC)$ and $num(DSC)$ are the number of hits in the PC and DSC, respectively, and $num(trie)$ is the number of worst-case lookups in the leaf-pushed trie. A hit in the PC incurs very few clock cycles, so we associate zero memory accesses with it. The variable t is the number of memory accesses required by a lookup to complete after it finds a hit in the DSC. In reality, the value of t will vary depending upon the specific lookup. For example, some lookups that hit the DSC may require three memory accesses to complete, while others may require just two. In the worst case, after a DSC hit, at most 5 memory accesses will be required. In the results presented here, we set $t = 5$ so that our results reflect the lower bound of performance of the PC/DSC architecture for any given trace. Lastly, we set $wc = 32$; this is the worst-case number of memory accesses a lookup requires, when it proceeds to the root of the leaf-pushed trie after missing both caches.

We used the TOMLAB optimization environment within Matlab to solve the following constrained nonlinear integer optimization problem of partitioning a fixed budget of cache lines, c, between the PC and the DSC to yield the minimum number of memory references:

$$\text{minimize} \sum_{i=1}^{b} F_b \times m_b \quad (8)$$

subject to:

$$0 \le F_1 = (L - L \times H_1(c_1)) \times H_2(c_2) \le L \quad (9)$$

$$0 \le F_2 = L - L \times H_1(c_1) - F_1 \le L \quad (10)$$

$$m_1 = t,\ m_2 = wc \quad (11)$$

$$0 \le H_1(c_1) = 1 - \left(\frac{c_1}{A_1(\mathcal{L})} + 1\right)^{-\theta_1(\mathcal{L})} \le 1 \quad (12)$$

$$0 \le H_2(c_2) = 1 - \left(\frac{c_2}{A_2(c_1)} + 1\right)^{-\theta_2(c_1)} \le 1 \quad (13)$$

$$\sum_{i=1}^{b} c_i = c \quad (14)$$

$$c_1, c_2 \ge 0 \quad (15)$$

The objective function, (8), is the sum of the number of memory accesses due to hits in the DSC, plus memory accesses due to traversals of the trie after misses in the DSC. For a trace of length *L*, F_1 is the number of references that are hits in the DSC, and F_2 is the number of references looked up in the trie. We set $b = 2$ because there are two caches in this architecture (the PC and the DSC).

The constraints in (9) and (10) bound the number of DSC hits and the number of full trie traversals, respectively. The number of lookups in the DSC is bounded below by zero, and above by the number of references in the trace. The lookups seen by the DSC are those that miss the PC, while the actual number of hits in the DSC is the number of lookups it sees multiplied by the DSC hit rate, $H_2(c_2)$. Only the references that hit in the DSC go on to reference memory through the shortcut pointer held in the DSC line. The number of lookups that have to go through a full trie traversal, F_2, are those left over from the PC and the DSC. The equality constraints in (11) set the values for m_1 and m_2 as explained in the notes to (7).

The subsequent constraints in (12) and (13) deal with the hit rates in the PC and DSC. The hit rates of both caches, H_1 and H_2, are bounded by 0 and 1. The hit rates themselves are functions of the number of lines in each cache, c_1 and c_2.

The parameters in the hit rate model for the PC, $A_1(\mathcal{L})$ and $\theta_1(\mathcal{L})$, are calculated by a best-fit of the references in the particular trace being studied, $\mathcal{L}$, to (6). Similarly, the parameters in the hit rate model for the DSC, $A_2(c_1)$ and $\theta_2(c_1)$, are calculated by a best fit of the trace of PC misses to (6). Thus, both $A_2(c_1)$ and $\theta_2(c_1)$ are functions of the number of lines in the PC.

We break both $A_2(c_1)$ and $\theta_2(c_1)$ into sets of piecewise-linear segments so they can be represented in the optimization tableau, and add two sets of binary selector variables to indicate the linear segment selected by the optimizer. This extra level of detail is presented in [22]. As noted, these changes are required to make the constraints linear. They make the tableau substantially more complex, both to present and to solve, but the reward is in very good solutions, as measured by how closely the results from the optimizer match those from exhaustive search.

TABLE I: Traces and Routing Tables Used

Packet Trace	Routing Table	Number of prefixes
FUNET	FUNET	41362
ISP1	ISP1	10166
ISP2	ISP2	6342
ISP3	ISP3	10166
bell	rrc11	126687
upcb.1	rrc03	132210
upcb.2	as1221	292496

Lastly, c_1 and c_2 are the number of cache lines allocated to the PC and DSC respectively out of the fixed total budget, c.

C. Experiments

For our experiments we used seven real-world packet traces downloaded from public sources. We paired each trace with the actual routing table used on the corresponding router (see Table I). The traces vary substantially in their size and degree of locality, and the routing tables vary correspondingly in their size and prefix length distribution.

Firstly, we used a trace and associated routing table from FUNET [2]. FUNET is a backbone network providing Internet connections for Finnish universities and other research institutes. FUNET makes a number of traces and routing tables available online. We downloaded a medium-sized routing table from FUNET that contained 41362 prefixes. The other three routing tables ISP1, ISP2, and ISP3 are drawn from distribution routers used by local service providers [15]. These four packet traces are each approximately 100,000 packets in length, and they all demonstrate a high degree of locality.

Present day routing tables are very large; for example, a recent report in [1] suggests that routing tables can contain upwards of 200,000 prefixes. Hence, to test the scalability of our method, we also performed experiments on larger routing tables - rrc11 [4], rrc03 [4], and as1221 [1]. The traces used for these experiments were bell, upcb.1, and upcb.2, downloaded from [3]. The bell and upcb.1 traces each had approximately 0.9 million packets, whereas upcb.2 had approximately 0.6 million packets.

The bell, upcb.1, and upcb.2 traces have much less locality, and require caches that are significantly larger than for the other traces we studied to reach comparable hit rates.

Figure 4 gives the optimization surface for the design of the PC/DSC architecture when we are given a budget of 240 cache lines to allocate between the PC and DSC. These are the results for the ISP3 trace, and a PC/DSC design with the length-reduce parameter set to $k = 3$. The optimizer predicts that 155 lines should be allocated to the DSC, and the remaining 85 lines to the PC. This allocation gives $AvgMemLookup = 3.26$. If we allocate all 240 lines to the PC, each lookup will require 5.62 memory accesses on average. In this particular case, the DSC reduces *AvgMemLookup*, and thus overall memory traffic for lookups, by 42%. The results for the FUNET, ISP1, and ISP2 traces, and their associated routing tables, show similar, substantial benefits. The full results are available in [22].

We conducted a second set of experiments with a budget of 8192 cache lines, while keeping *k = 3* (see Figure 5). The solution from the optimizer results in the lowest *AvgMemLookup* for the ISP3 trace when 6771 lines are allocated to the DSC, and the remaining 1421 lines are allocated to the PC. Here, we record an improvement of 56% over the PC alone in terms of *AvgMemLookup*.

To capture our results for all the traces examined, and to highlight the performance of the PC/DSC architecture in contrast to other current caching architectures, we present the results in Table II. The FUNET, ISP1, ISP2, and ISP3 tests used a budget of 240 cache lines. The tests for the low-locality traces (bell, upcb.1 and upcb.2) used a budget of 8192 cache lines. To make the comparisons fair, we have recorded *AvgMemLookup* values for optimal designs of all the cache architectures in the table.

This comparison demonstrates conclusively the effectiveness of the PC/DSC architecture in both high-locality traffic and low-locality traffic (see Table II). The PC/DSC architecture performs much better than any other, and shows a reduction of up to 30% in *AvgMemLookup* when compared to its nearest competitor. This considerable improvement is primarily attributed to the high hit rates in the DSC, even when there is low locality in the traffic. The PC/DSC is by far the best caching architecture in all the test cases examined, and it performs very well even with low-locality traffic.

We recommend a value of $k = 3$ for the datasets in this study instead of $k = 2$, primarily because of the benefit from the increased DSC hit rate. This is illustrated by the results for optimal design of the PC/DSC architecture for the ISP3 trace at $k = 2$ (see Figure 6). The optimum is obtained by allocating 176 lines to the DSC, and the remaining 64 lines to the PC. The use of $k = 2$ has increased *AvgMemLookup* from the value of 3.26 at $k = 3$ to 4.36. This is a 34% relative increase in the number of memory accesses, so it is clear that in this case, $k = 3$ is a much better choice. We obtained similar results with all the traces we examined.

A further question is whether the optimization tableau produces accurate answers, because there are some approximations in the model. Firstly, cache hit rates are predicted from a best-fit model. Secondly, we use worst-case values for the number of memory references after a hit in the DSC, or a traversal of the trie. To resolve this question, we placed a simulation of a PC/DSC architecture facing a stream of IP address lookups inside a simple exhaustive search. The search ran over the space of all possible joint allocations to the PC

and DSC, as constrained by the given cache line budget. In all cases examined, the designs produced by the optimization tableau are quite close to what we find using exhaustive search.

For example, at $k = 3$ and with a budget of 240 lines, exhaustive search finds the minimum $AvgMemLookup$ for the ISP3 trace when 162 lines are allocated to the DSC, and the remaining 78 lines are placed in the PC. In comparison, the tableau finds the minimum with 155 lines allocated to the DSC, and the remaining 85 are allocated to the PC. At $k = 3$ and with a budget of 8192 lines, exhaustive search finds the minimum value of $AvgMemLookup$ for the ISP3 trace when 6836 lines are allocated to the DSC, and the remaining 1356 are placed in the PC. For this case, the tableau finds the minimum with 6771 lines allocated to the DSC, and the remaining 1421 allocated to the PC. These results clearly indicate that our method is quite accurate, returning results within 5% of exhaustive search in the cases examined, despite the approximations used in the optimization tableau.

Lastly, to make sure there is a benefit to running the optimizer compared to using exhaustive search, we measured the running times for both methods in cases with budgets of 240 cache lines, and 8192 cache lines. For the ISP3 experiment with a budget of 240 lines, the optimizer required 12 seconds, while exhaustive search required 286 seconds. Similarly, for the ISP3 experiment involving 8192 lines, the optimizer required 1123 seconds, while exhaustive search required 17065 seconds. We ran these experiments on a Pentium 3, 1.3 Ghz machine.

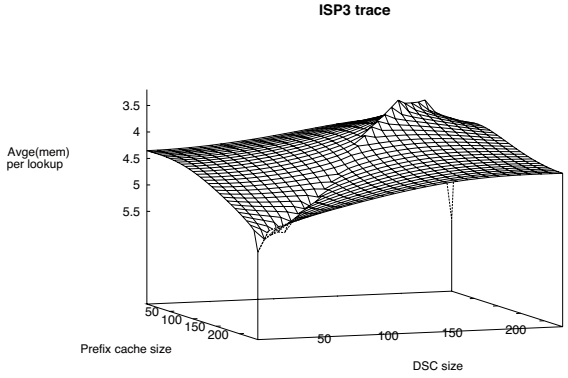

Fig. 4: Optimization surface for the PC/DSC architecture (AvgMemLookup) when *k = 3*

In summary, our experiments show that the PC/DSC architecture performs much better than all other state-of-the-art caching architectures. Specifically, our scheme results in significantly fewer memory accesses for a stream of IP address lookups. Our experiments also show our optimization tableau finds optimal PC/DSC designs, and is much faster than exhaustive search.

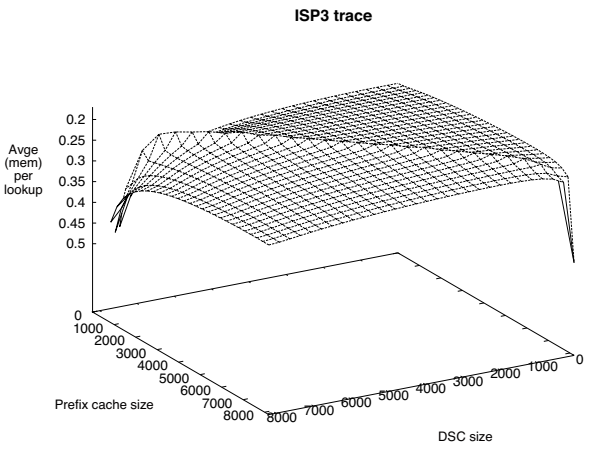

Fig. 5: Optimization surface for the PC/DSC architecture (AvgMemLookup) when *k = 3*

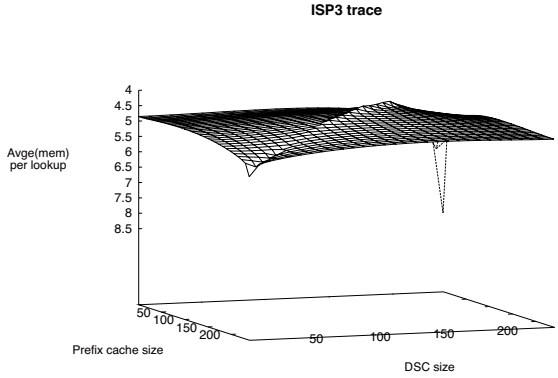

Fig. 6: Optimization surface for the PC/DSC architecture (AvgMemLookup) when *k = 2*

VI. Conclusions

In this paper, we presented a novel second layer of caching called a dynamic substride cache (DSC). The DSC significantly reduces the average number of memory accesses for IP lookups. This is extremely important because, in routers as in general-purpose computers, memory bandwidth is a very scarce resource. We expect this type of cache to become more useful over time as the locality of reference in a stream of IP addresses decreases. We demonstrated the performance benefits of the DSC with both high and low locality streams.

We provided a nonlinear optimization tableau for optimally allocating a given amount of memory between a prefix cache (PC) and a DSC. The approximations used in formulating the tableau yield good results: our method provides results within 5% of the naturally much slower exhaustive search. As part of this optimization technique, we presented a new method for accurately predicting cache hit rates. This method is applicable

Schemes	*FUNET*	*ISP1*	*ISP2*	*ISP3*	*bell-rrc11*	*upcb.1-rrc03*	*upcb.2-as1221*
Proposed architecture (PC/DSC)	2.64	0.89	1.38	3.26	0.37	0.44	4.26
Multizone Prefix Caching [20]	2.78	1.36	1.52	4.23	0.48	0.61	6.03
Shyu *et al.* [25]	4.25	1.49	1.71	4.85	0.97	1.13	6.42
Akhbarizadeh *et al.* [5]	4.32	3.68	1.79	5.62	1.56	1.92	7.68
Kasnavi *et al.* [15]	5.69	5.18	2.84	7.63	3.67	4.39	8.34
Multizone IP address Caching [7]	6.04	5.77	2.95	8.47	5.02	6.13	10.91
IP address cache [6][30]	6.76	6.65	3.26	9.73	8.94	9.16	16.67

TABLE II: *AvgMemLookup* for PC/DSC and other current caching architectures

to caches generally – not just to IP address or prefix caches.

The designs presented in this paper achieve a significantly lower number of memory accesses per lookup than optimal designs for six current competing architectures: optimal PC/DSC designs reduce the average number of memory accesses per lookup by over 40% compared to these other approaches. That is, memory traffic can be decreased by over 40% by using a PC/DSC, even when faced with low-locality traffic. In terms of worst-case performance, DSC-assisted lookups require at most five memory accesses to complete, while without the DSC, in the worst case a full trie traversal could require up to 32 memory accesses.

Lastly, we note that it is very important to ensure the consistency of any caching architecture when updates are being made to the routing table. These updates are common because of configuration changes, route flaps, etc. The task of ensuring consistency becomes challenging with the PC/DSC architecture because we store information in three different places: the PC, the DSC, and the leaf-pushed trie. Given the importance of this aspect of the problem, we have developed a scheme that can cope with incremental updates; this is discussed in detail in [22].

REFERENCES

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

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

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] H. Ballani and P. Francis and T. Cao and J. Wang. Making Routers Last Longer with ViAggre. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation*, Apr 2009.

[13] R. Jain. Characteristics of destination address locality in computer networks. In *Computer Networks*, volume 18, pages 243–254, May 1990.

[14] 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.

[15] 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.

[16] S. Kasnavi, P. Berube, V.C Gaudet, and J.N Amaral. A multizone pipeline cache for IP routing. In *IFIP Networking Conference*, volume 3462, pages 574–585, May 2005.

[17] C. Kim, M. Caesar, A. Gerber, and J. Rexford. Revisiting route caching: The world should be flat. In *Passive and Active Measurement (PAM)*, pages 3–12, April 2009.

[18] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: A scalable ethernet architecture for large enterprises. In *ACM SIGCOMM*, August 2008.

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

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

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

[22] S. Ravinder. Cache architectures to improve ip lookups. http://repository.library.ualberta.ca/dspace/bitstream/10048/667/1/. Master's thesis, University of Alberta, September 2009.

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

[24] 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.

[25] 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.

[26] 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.

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

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

[29] J. M Surprise. An energy efficient TCAM enhanced cache architecture. Master's thesis, Texas A&M University, May 2005.

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

[31] 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.