A Heterogeneously Segmented Cache Architecture for a Packet Forwarding Engine

Kaushik Rajan† and **R. Govindarajan**†‡
†Supercomputer Education and Research Centre
‡Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560 012, India
kaushik@hpc.serc.iisc.ernet.in
govind@csa.iisc.ernet.in

Abstract

As network traffic continues to increase and with the requirement to process packets at line rates, high performance routers need to forward millions of packets every second. Even with an efficient lookup algorithm like the LC-trie, each packet needs upto 5 memory accesses. Earlier work shows that a single cache for the nodes of an LC-trie can reduce the number of external memory accesses.

We observe that the locality characteristics of the level-one nodes of an LC-trie are significantly different from those of lower-level nodes. Hence, we propose a heterogeneously segmented cache architecture (HSCA) which uses separate caches for level-one and lower-level nodes each with carefully chosen sizes. We further improve the hit rate of the level-one nodes cache by introducing a weight-based replacement policy and an intelligent index bit selection scheme. To evaluate our cache scheme with realistic traces, we propose a synthetic trace generation method which emulates real traces and can generate traces with varying locality characteristics. The base HSCA scheme gives us upto 16% reduction in misses over the unified scheme. The optimizations further enhance this improvement to upto 25% for core router traces.

1 Introduction

The enormous growth in Internet traffic and support for multimedia data has led to higher bandwidth requirements and increased line rates. As line rates increase the performance demands on routers, typically supporting multiple gigabit lines, also increase. IP forwarding is one of the key applications performed by a router. Routers take packets from the input line and make a decision on which output line should transmit them. This involves looking up a table of prefixes and finding the longest prefix match (LPM) for the destination IP address. LPM is typically done by traversing through a trie data structure. A good introduction to various algorithms used for LPM is given in [13]. We briefly describe one of the schemes, the Level Compressed trie (LC trie) in the next section.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. ICS 05, June 20-22, Boston, MA, USA

In high performance routers, where packets arrive at rates of millions per second, forwarding poses a major challenge. Assuming a line rate of 2.5Gbps (OC 48) and packet sizes of 64B, these routers have to deal with roughly 5 million packets every second. As the trie used for forwarding resides in memory and each packet involves visiting one or more trie levels, the IP lookup algorithm is memory-intensive and slow memory access times can cause a major performance bottleneck.

Earlier work shows that the use a single unified cache to store all-level nodes of an LC trie can improve the throughput of routers [1]. In an LC trie the root node typically has a large number of children (2^{16}) while other nodes have relatively smaller number of children. Every lookup involves at least an access to a Level-One (LO) node (direct children of root node) and may access lower trie nodes. We observe that the locality characteristics of LO nodes are significantly different from those of Lower-Level (LL) nodes (nodes other than LO nodes and root node).

Based on these observations, in this paper, we propose a Heterogeneous Segmented Cache Architecture (HSCA) which uses separate caches for LO and LL nodes with sizes chosen based on number of accesses to each partition. As more than 80% accesses are to LO nodes, we skew the size of LO cache to be eight times the size of LL cache. This difference in sizes makes the cache heterogeneous in nature. We further optimize the LO cache by introducing a weight-based replacement policy instead of LRU, that assigns weights to nodes based on their importance in the trie. We also try to make the distribution of the *useful LO* nodes (nodes that contain relevant information) into the cache sets more uniform. We achieve this by intelligently identifying the bits to index into the cache. It should be noted here that the cache architecture proposed in this paper is for network specific applications and is specifically designed to work with processors in edge and core routers.

The locality characteristics of IP address traces has been well studied [6][8]. IP address traces tend to have no spatial locality but do tend to exhibit temporal locality. The temporal locality characteristics vary based location of the router in consideration. It is observed that at the edge routers, the IP addresses tend to exhibit much greater temporal locality than at core routers For example, traces observed at the edge routers tend to have less than 1% of unique addresses in them, whereas the core router traces contain 10% or greater unique addresses. In order to properly evaluate the design considerations for routers, we develop a parameterized syn-

thetic trace generation methodology which mimics the characteristics of real traces. Traces generated by our generator obey the prefix length distribution of real traces [11]. Further by varying the parameters, we can generate traces which correspond to edge or core routers.

We compare the performance of HSCA with two contemporary caching schemes, namely the single unified cache [1] and the Intelligent Host Address Range Caching (IHARC) [4]. Performance evaluation reveals that the base HSCA scheme gives us upto 16% reduction in misses over the unified scheme. Relative to IHARC, HSCA results in upto 90% savings in number of misses for edge router traces and 50% savings in number of misses for core router traces. For core router traces, the weight-based replacement further enhances this improvement to upto 25% lesser misses than the unified cache and 60% lesser misses than the IHARC scheme.

The rest of the paper is organized as follows. In Section 2 we provide a background of existing caching schemes and motivate our work. In Section 3 we introduce HSCA and elaborate on the optimizations considered for the cache segments. Section 4 deals with the parameterized synthetic trace generation methodology. In Section 5 we present the experimental results for cache simulations comparing our scheme with the existing schemes. Section 6 reviews the related work. In Section 7 we provide concluding remarks.

2 Background and Motivation

In this section we briefly describe how LC-tries are used for forwarding. Next we review the existing caching schemes for IP forwarding [1][4]. Lastly we present a set of arguments that motivate our work.

2.1 Forwarding with LC Tries

A routing table can be represented by a simple binary trie. This trie is traversed using the bits of the IP address being looked up. The traversal takes one bit at a time and goes to the left/right child if the bit is 0/1 respectively. The search ends when a leaf node is reached.

A simple binary trie has a large number of nodes and a high average depth. Two compression techniques, namely path compression and level compression, are used to overcome these shortcomings [10].

Path Compression: Nodes which have only one child are removed from the trie. The number of such nodes that have been skipped, the skip value, is stored in a node along the path. While performing the lookup the skip value information is used to determine which is the next bit to use for traversal.

Level Compression: Instead of using one bit at a time, k bits are used at one go to proceed to one of the 2^k (called the branching factor or bf) children. This would save k-1 visits in a binary trie. In most cases some of these nodes may not exist in the original trie. However we can add superfluous nodes by expanding existing nodes upto the required depth (prefix expansion). A branching factor of 2^k is used if it produces atmost $bf * (1-x)$ superfluous leaves. x is called the fill factor. We use a value of $x = 0.5$ throughout this paper.

In addition to compression, prefix expansion is also used at the root. This expansion is based on the observation that very few prefixes are of length less than 16. Hence typically a branching factor of 2^{16} is used for the root node. After reaching a leaf node, two auxiliary tables, the forwarding table and prefix table, are looked into to get output port information. Alternatively, as suggested in [12], additional information can be embedded in the trie itself and the auxiliary tables can be completely eliminated.

2.2 Caching Schemes for IP Forwarding

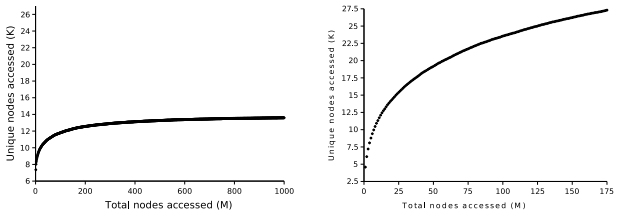
Attempts at caching for IP forwarding can be broadly classified into two categories, caching address ranges [3][4] and caching data structures used for IP lookup [1]. We briefly discuss one scheme from either category [1][4].

2.2.1 Intelligent Host Address Range Caching (IHARC)

In [4] the authors present the IHARC scheme where they cache address ranges and their routing decisions. By carefully choosing the bits to index the cache they minimize the number of address ranges in each cache set. This they achieve as ranges that are not originally adjacent may become adjacent once the index bits are removed. If adjacent ranges share the same routing decision they are merged together. In IHARC, unlike a conventional cache, we would have one entry in cache matching many addresses. This slightly complicates the cache logic as the tag match now involves applying a mask to the address tag before doing a comparison. Also a programmable hash engine is needed to select the bits to index the cache. Both these functionalities are on the critical path and add to the hit time latency. In addition the IHARC scheme uses a parameter w that can be tuned based on the routing table [3]. However, this tuning process is complex and we simply use $w = 1$ as done in [4].

2.2.2 Caching the LC-trie

In [1] the authors introduce a cache scheme (henceforth referred to as the unified cache scheme) to store the nodes of the LC trie. They observe that even with traces having no locality, the trie levels closer to the root are accessed more frequently. Hence they suggest caching nodes of the trie as opposed to address ranges. They use a single cache to store all-level nodes and use LRU replacement. Two synthetic traces, randNET and randIP are used for evaluating their scheme. Both randNET and randIP do not have prefix length distribution characteristics of realistic traces [11]. Also both these schemes do not model the locality characteristics of real traces. In Section 4 we introduce our synthetic trace generation scheme which overcomes the problems of randNET and randIP.



(a) LO nodes (b) LL nodes

Figure 1. Footprint Curves

2.3 Why HSCA?

The unified cache scheme of [1] does not take into consideration the locality characteristics of the nodes of trie. Each lookup always incurs a LO node access but may not necessarily visit lower levels. We observe that in realistic traces only 17% of the lookups visit LL nodes. In addition less than 25% of LO nodes are useful nodes (also observed in [1]). This implies that a large number of accesses go to a small set of nodes. A plot of the number of unique nodes visited in an LC-trie (footprint curve) for an address trace of 1 billion packets (refer Figure 3) reflects this observation. These observations naturally lead us to the choice of a partitioned cache memory with two segments, one cache explicitly for storing the LO nodes and the other cache for storing LL nodes. As only 17% of accesses that are incurred by LO cache go to the LL cache, it is natural to have a much smaller LL cache.

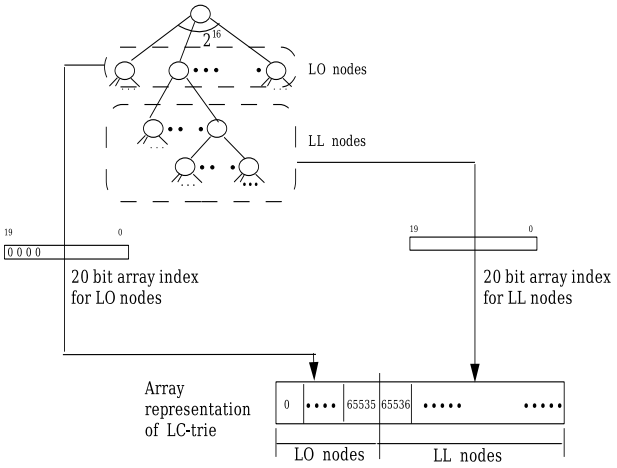


Figure 2. LC-trie and its array representation

3 HSCA and Optimizations

Here we describe the working of the base HSCA. The HSCA caches the nodes of the array representing an LC-trie. The array representation of an LC-trie is shown in Figure 2. Throughout this paper we assume that the root branching factor is 2^{16}. Hence, the LO nodes will have an array index value in the range $0-65535$. While the LL nodes will have indexes greater than 65535. The address used to access HSCA is this *array index* and not a memory address. Using the trie array index to index the cache does not limit the generality of the cache. By choosing an appropriate base trie array address (like 0x80000000) and a node size which is a power of two (4Bytes), every trie array index will translate to the middle bits in the corresponding memory address. For example, trie array index 63 (or 00111111 in binary representation) will map to memory address 0x800000fc, with bits 9 to 2 being 00111111.

We assume a 20 bit representation for the array index that can accommodate an LC-trie of 2^{20} nodes. This many nodes is more than sufficient for contemporary routing tables [1][10]. To form a k bit index for either LO cache or LL cache we use the least significant k bits (bits 0 to k-1) of the *array index*. As only bits 0 to 15 can vary for LO nodes it is sufficient to tag bits k to 16 for the LO cache. However, for the LL cache, $20-k$ most significant bits form the tag. In all our experiments we keep the number of lines in LL cache to be $1/8^{th}$ of the number of lines of the LO cache. Hence if we use 12 bits to index LO cache we need to use only 9 bits to index the LL cache.

In all our configurations we keep the associativity of the LO and LL cache the same. We experiment with associativities of 1,2,4 and 8. Note that as the node accesses do not exhibit any spatial locality we use the size of a single node as the size of a cache block. Though we evaluate the HSCA for an LC-trie data structure, it is to be noted that the same cache organization can be used for other trie based lookup algorithms as well as most trie based schemes use a large root branching factor [13].

3.1 Weight-based Replacement Policy

Most replacement policies can be thought of as replacing the cache line with the least *weight*. In LRU the weight is based on the recency of use, with most recently used line being given the maximum weight. In FIFO the weight is the reciprocal of the lifetime in cache and in LFU it is the number of accesses.

At core routers, where locality is expected to be less, these replacement schemes can be improved by finding better weight assignment schemes. We propose a weight-based replacement policy for the LO cache that statically assigns a weight to each of the LO nodes. These weights are based on two observations :

1. A majority of the traffic from real traces hits prefixes whose length ranges from $13-24$. Within this range, as prefix length increases the average number of hits for a prefix of a given prefix length decreases [11].
2. Since we use a branching factor of 2^{16} at root, prefixes of length $p < 16$ would have 2^{16-p} LO nodes pointing to the same forwarding table entry. Therefore, an LO node representing a prefix of length p is only covering $1/2^{16-p}$ prefixes.

Based on the above, we propose the following weight assignment

- LO nodes corresponding to prefixes of length $8-12$ are given weights from $0-4$ respectively.
- LO nodes corresponding to prefixes of length $13-16$ are given a weight of 7.
- The remaining LO nodes are sorted in ascending order of the number of prefixes the sub-trie rooted at that node covers. The lower half of this sorted set is given a weight 5 and the upper half is given a weight of 6.

As the weights are from 0 to 7, we need 3 bits to represent them. This weight can be accommodated as an additional field within the node.

3.2 Index Bit Selection

As the root node of the LC trie uses a large branching factor, there are a large number of LO nodes that do not contain any useful information. These nodes are basically leaf nodes that do not match any valid prefixes. The percentage of useful nodes reduces as we increase the root branching factor. For the routing table that we use, a branching factor of 2^{16} yields less than 25% of useful LO nodes.

When a k-bit index is used for the LO cache, it partitions the LO nodes into 2^k sets. It is possible that the useful nodes are not uni-

formly distributed among these sets, i.e. a large number of useful nodes could map to some sets while others could have very few nodes mapping to them. We use a greedy bit selection algorithm to intelligently choose the index set.

The pseudo-code for bit selection algorithm that selects k index bits out of 16 bits is presented in Algorithm 1. S represents the set of bits selected thus far. To select the i^{th} bit we try out each bit that is not already in S. This bit along with the previously chosen i-1 bits distributes the LO nodes into 2^i partitions. For each such bit the standard deviation of the number of nodes in the desired node set (some subset of set of all useful LO nodes) per partition is calculated. The bit which has the least $stddev$ is then chosen and added to the set S. In the bit-selection schemes that we use we

Algorithm 1 simple bit selection

```
S = ∅
min = ∞
for i = 1 to k do
  for j = 1 to 16 do
    if j ∉ S then
      T = S ∪ {j}
      partition nodes based on T
      mean = nodesetsize/2^i
      s = stddev(mean, partition)
      if s < min then
        min = s
        index = j
      end if
    end if
  end for
  S=S∪{index}
end for
```

do not consider all useful nodes but a subset of useful nodes. By changing the set of nodes considered, we come up with different bit selection schemes.

- Instead of using all the useful nodes if we consider only the set of nodes which are accessed by the first one million addresses of the trace we get a bit selection scheme that is dynamic (bsd).
- In addition to just considering the nodes accessed in first million lookups, if we weight the nodes according to number of times they are accessed, we get a weighted dynamic bit selection scheme (bsdw).
- Each prefix of length 8 covers to 256 LO nodes. As 8-bit prefixes are rarely accessed and form a large percent of LO nodes, we ignore all such nodes and try to select bits based on the remaining useful LO nodes (bs ign8).

It should be noted here that the bit selection algorithm needs to be rerun only when the routing table has changed significantly. The bit selection algorithm takes very little computation time (less than 2 seconds on a 2GHz Pentium 4 desktop) and can be run off-line, and the appropriate bit selection be enabled through a multiplexer.

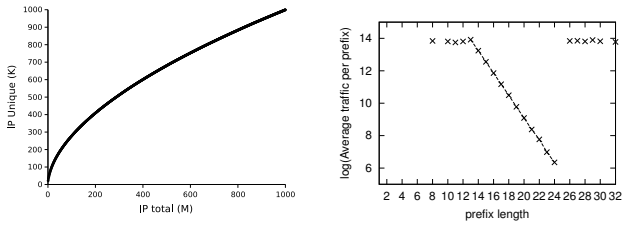
4 Synthetic Trace Generation

One major difficulty in the performance evaluation of routers and routing algorithms is the lack of publicly available traces that correspond to publicly available routing tables. This difficulty can be overcome by using synthetic traces. However, in order for these evaluations to be useful, the synthetic trace should match the characteristics of real traces.

Two trace generation methods, randIP and randNET, have been in use in literature [1][9][18]. In randIP a random 32bit IP address is generated and included in the trace if it matches an entry in the routing table. In randNET a random prefix is chosen from the routing table and expanded to 32 bits. Traces generated by these schemes do not match the characteristics of real traces [11][15]. More specifically :

1. Both randNET and randIP do not follow the *prefix length distribution* — the log of the *average number of addresses that hit a prefix of a given length* decreases linearly with increasing *prefix length*, for prefix lengths ranging from 13 to 24 — of real traces [11]. In particular, in this range, as the *prefix length* increases by one, the log of the *mean number of hits per prefix* decreases by 0.69. The authors of [11] also observe that, unlike randNET and randIP, only a small percentage of the traffic (about 6%) hits prefixes outside the prefix length range of 13 to 24 bits.
2. Both the randNET and randIP schemes do not preserve locality characteristics of real traces. With a uniform random number generator used in randNET and randIP, it is highly unlikely that an address that is referred in the recent past will be encountered again in the near future. Hence, both the schemes do not conform to the tenets of temporal locality.

Our trace generation procedure plugs in observation 1 into the LRU stack model of [15] to produce a methodology which can emulate both the *prefix length distribution* and the *locality characteristics* of real traces. The algorithm in [15] uses two primary parameters, A and θ, which are used to determine the cumulative distribution function (cdf) for the *hit index*. The cdf is such that smaller hit indexes are preferred over larger ones. A random variate *index* which follows the cdf of the *hit index* is generated. If *index* points to an entry in the stack, that entry is moved to the top of the stack and the stack is readjusted by moving down all entries upto $(index-1)$ by one position. If the *index* is beyond the current size of the stack, all entries are moved down by one position and a new entry is accommodated at the empty top position. The physical significance of A and θ, and the derivation of the cdf of the *hit index* can be found in [16]. The important point to note is that by increasing the value of θ the locality in trace can be increased. Figure 3 shows the footprint curve for one of our synthetically generated traces with $A = 10$ and $\theta = 1.8$. The trace contains roughly 1 million unique addresses out of a total of 1 billion entries, hence we refer to it as pld1M. It can be seen that this curve depicts a trend similar to the footprint curve of a real trace [15].



(a) Footprint Curve (b) Prefix Length Distribution

Figure 3. Characteristics of pld1M Trace($A = 10$,$\theta = 1.8$)

One question that is not answered by [15] is *How to choose a new address to add to the stack?*. Observation 1 provides us with a natural choice. The new entry should be chosen so that the *prefix length distribution* of the synthetic trace follows that of realistic traces. We make use of observation 1 to construct the probability mass function for prefix lengths by assigning probabilities for hitting each prefix length. While doing so we ensure that 94% of traffic hits prefixes of prefix length $13-24$. The remaining 6% traffic is uniformly distributed among the prefixes outside this range. All prefixes of a given prefix length are assumed to be equally likely to be picked, hence once a prefix length is chosen, a prefix of the given prefix length n can chosen randomly. This prefix is then expanded to 32 bits by padding it with a uniformly distributed random number between 0 and 2^{32-n}.

In Figure 3(b) we plot the log of *average traffic per prefix* against the *prefix length* for one of our synthetically generated trace (pld1M). The slope of the line for prefix lengths in the range $13-24$ matches well with the real trace used in [11]. It is to be noted that though the *average traffic per prefix* outside the range $13-24$ is high, there are a very few prefixes outside this range and these prefixes contribute to only 6% of the traffic.

5 Performance Evaluation Results

5.1 Simulation Methodology

To evaluate the performance of HSCA and the optimizations proposed in section 3, we use a trace driven simulation methodology. To measure the misses incurred we use the dineroIV cache simulator with some modifications. The modifications are made to accommodate our weight-based replacement policy and to simulate for caches of odd sizes (number of sets is not a power of two). For the odd sized caches, indexing is done using the remainder obtained by dividing the address (trie index) by the number of sets. We use the FUNET routing table [19] used for performance evaluation of the LC-trie [10]. The routing table contains 41578 entries, and its LC-trie representation leads to 128865 trie nodes.

Table 1. Traces used for performance evaluation

Name	A	θ	Total addresses
pld1M	10	1.8	1B
pld10M	10	1.5	1B
pld100M	10	1.2857	275M
pld200M	10	1.2327	140M
pld1B	-	-	1B

We use a total of 5 traces (refer to Table 1). The first 4 were generated using the methodology of the previous section. Pld1B, the fifth trace, is generated without using an LRU stack but following the prefix length distribution of real traces. The suffix for the traces represent the number of unique entries if the trace had a total of one billion IP addresses. The traces pld100M and pld200M contain 250M and 140M entries respectively, while the rest have 1B entries. This is so because it takes a longer time to generate traces with low localities. For the lower locality traces (pld100M and pld200M) we need a larger LRU stack and the stack readjustment operation takes a long time.

Throughout this section we will refer to the size of the HSCA in terms of the number of lines in the LO cache. For example, a 8k HSCA refers to an HSCA with 8k lines for the LO cache and 1k lines for the LL cache. We evaluate the performance of 2k, 4k and 8k HSCA with associativities 1, 2, 4 and 8.

Throughout this paper we report only normalized percentage miss rates to make the comparisons easier. As can be seen from Table 2 the miss rates of caches used for this application are fairly high. (also observed in [1]).

Table 2. Absolute number of misses for a 8k unified cache

Trace	Number of nodes accessed	Associativity			
		1	2	4	8
Pld1M	1.175B	39.8M	12.5M	6.3M	3.9M
pld10M	1.175B	148.6M	81.9M	52.7M	36.9M
pld100M	323.3M	87.4M	71.4M	61.6M	52.4M
pld1B	1.175B	371.7M	306.6M	265.7M	226.7M

We do not report other performance metrics, such as average lookup time as with such high miss rates, lookup time translates to the average memory access time which in turn is directly related to the number of misses.

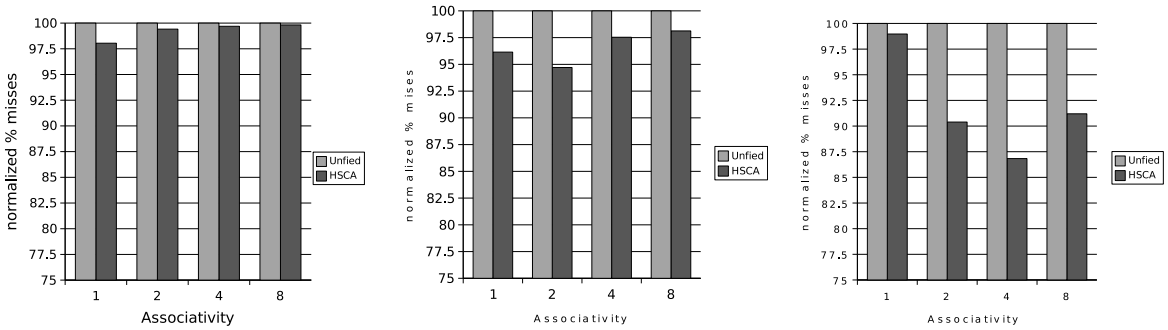
5.2 Experimental Results

5.2.1 Performance of HSCA and Unified Cache

First we compare the number of misses incurred by HSCA with that of a unified cache [1]. To make a fair comparison we compare a HSCA with a unified cache having the same number of total lines (2k,4k and 8k HSCA would correspond to a 2.25k, 4.5 and 9k entry unified caches respectively) and same associativity. The misses incurred by HSCA normalized relative to a unified cache is plotted in Figure 4. For all cache sizes and associativities HSCA performs better. A performance improvement of upto 13% is observed. It can be seen that the performance improvement of HSCA gets better and better as cache size increases. This implies that HSCA makes better use of the available cache lines than the unified cache.

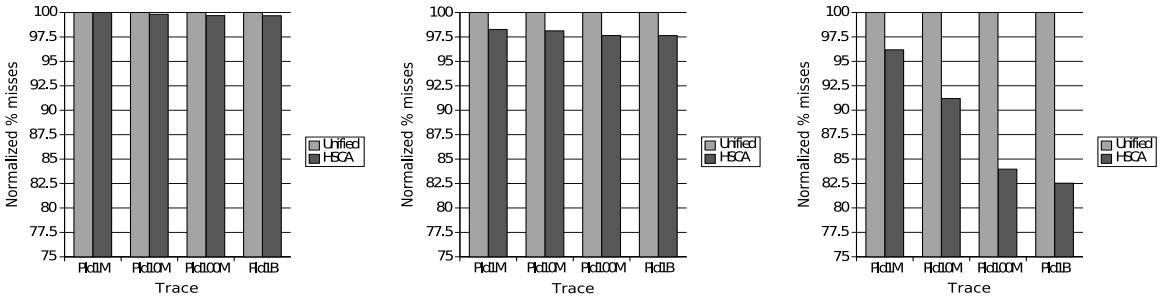
It can also be observed that as size increases the associativity at which the gains saturate, increases. A careful examination reveals that for all cache sizes this saturation occurs when the number of sets in the LO cache comes down below 2k. This could probably be because as the number of sets in LO cache reduces the number of bits used to index the cache reduces. The performance then starts to depend more on how well the index bits partition the useful LO nodes.

Figure 5 shows the impact of locality in trace on the performance



(a) 2k HSCA (b) 4k HSCA (c) 8k HSCA

Figure 4. Comparison of HSCA with Unified cache (pld10M)



(a) 2k HSCA (b) 4k HSCA (c) 8k HSCA

Figure 5. Impact of Locality in Trace

of HSCA. Although this plot is for an 8 way associative cache, no significant variations are seen when associativity is reduced. The plot shows that HSCA performs better than the unified cache for all traces, the best case being a 8k HSCA for a pld1B trace which gives 17.5% reduction in number of misses. The plot also shows that there is an increase in the relative performance of HSCA as locality reduces (especially for higher cache sizes). This happens because HSCA does not allow accesses to the LL nodes to interfere with LO cache thus preserving the LO node in the cache for longer. This pays in lower locality traces.

5.2.2 Comparison of HSCA with IHARC

The comparison between HSCA and the unified cache is simple as both cache the nodes of an LC-trie. However, the comparison of HSCA with IHARC is not as straightforward. Below we describe the complications that arise and provide a work around.

1. **Accesses to auxiliary tables** : The IHARC scheme directly caches the result of a lookup, while HSCA only caches the nodes of an LC-trie which is used to index into the auxiliary forwarding and prefix tables. Hence with HSCA additional memory accesses are needed to read the auxiliary tables. On the other hand, addresses that miss the IHARC need to go through a trie to find an LPM. Therefore it is not fair to directly compare the misses of HSCA with those of IHARC.

   We measure the IHARC misses as number of LC-trie accesses incurred to complete an LPM for the missed addresses. It is proposed in [12] that information contained in the auxiliary tables can be embedded into the trie itself using the modified LC-trie. The modified LC-trie makes a few minor modifications to the LC-trie and doubles the size of a node to store all information of the auxiliary tables with the node itself. One possible structure for the nodes is shown in Figure 6. This contains 2 extra fields, *port* and *string*, in addition to the LC-trie node. The field *Port* stores the result of the lookup, while *string* is needed to verify if the address indeed matches the prefix of the node, as some of the address bits could have been skipped.

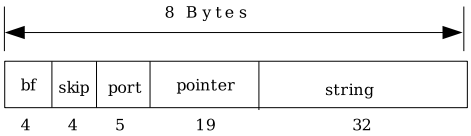
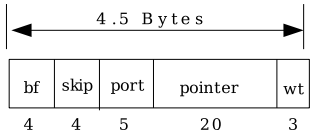
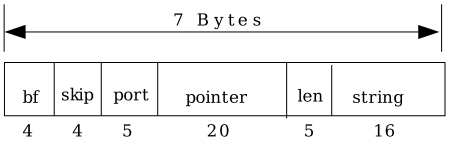


Figure 6. Node structure for modified LC-trie



(a) LO node



(b) LL node

Figure 7. Node structure for HSCA

We can further cut down on the size of the nodes by utilizing the properties of an LC-trie. The *string* field needs to be stored only if there has been a path compression along the path. As from root the first 16 bits are all used to reach a LO node, there is no need to store the *string* for LO nodes. For the same reason, even for the LL nodes it is sufficient to store only the lower 16 bits. This leads us to the node structure of Figure 7. The length field (*len*) is used to identify how many of the bits of the address should match the *string* field. This field is added to save on the need to remember how many bits of the address have been utilized. Because of the different sizes of the nodes we would now need to use two arrays to store the trie, one for LO nodes and one for the LL nodes. However the changes needed are minor, therefore it is fair to assume that these may-not significantly impact the nature of the accesses to HSCA.

2. **Making a fair comparison between IHARC and HSCA** : A fair comparison can only be made if the two caches are of identical sizes. The IHARC scheme requires a large tag space as it stores a mask in addition to the conventional tag. The tag and the mask field together require $2*(32-k)$ bits per line for a cache with 2^k sets. In comparison HSCA only needs $(16-k)$ bits for each LO cache line and $(20-k)$ bits for each LL cache line. On the other hand each data element of IHARC only needs to store a 5 bit output port field, whereas each entry in HSCA is a node of comparatively larger width (refer Figure 7).

   Based on the above we calculate the actual size (including both tag and data space) needed for each cache configuration (refer to Table 3). All sizes shown in this table are for a direct mapped cache. Increasing the associativity simply increases the sizes of both caches but does not affect the size ratio. We find that it is reasonable to compare a IHARC scheme with 2^k cache sets with an HSCA with 2^k sets in LO cache and 2^{k-3} sets in the LL cache, as even in the worst case the HSCA size is within 10% of the IHARC size.

In Figure 8 we compare the misses incurred by HSCA with those of IHARC. HSCA performs 45% to 90% better for the pld10M trace. As size increases we find the relative performance of HSCA gets better and better, suggesting that HSCA makes more efficient use of the cache lines.

Table 3. Total sizes of HSCA and IHARC (in bytes)

number of sets	IHARCsize(B)	HSCAsize(B)	Size Ratio
256	1696	1692	99.7
512	3264	3312	101.5
1024	6272	6480	103.3
2048	12032	12672	105.3
4096	23040	24768	107.5
8192	44032	48384	109.9

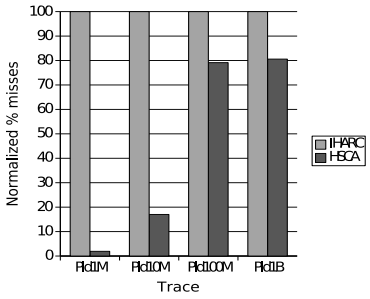


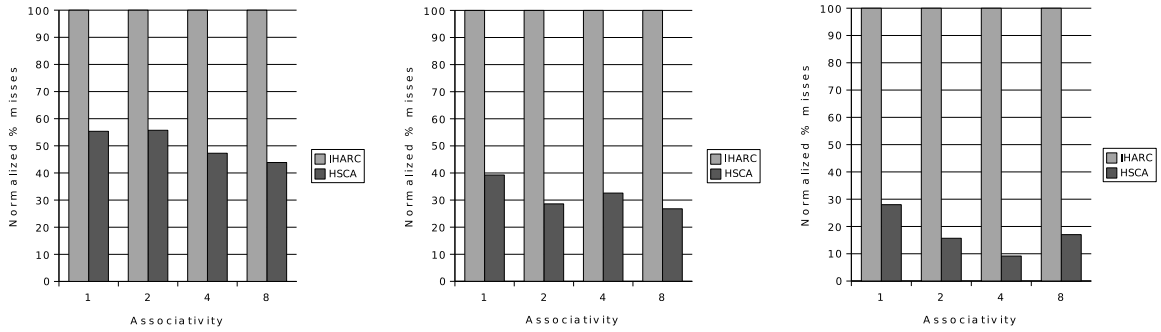
Figure 9. HSCA vs IHARC under Different Localities

In Figure 9 we plot the impact of locality in trace on performance for 8k, 8-way associative cache. The performance curves for other cache sizes and associativities exhibit similar trend. The plot shows that HSCA performs better than IHARC for all traces. It is observed that for traces with higher locality (pld1M and pld10M) HSCA reduces the number of misses by 60 to 80%. Unlike the unified cache, the relative performance of IHARC improves relative to HSCA for lower locality traces.

5.2.3 Impact of HSCA Optimizations

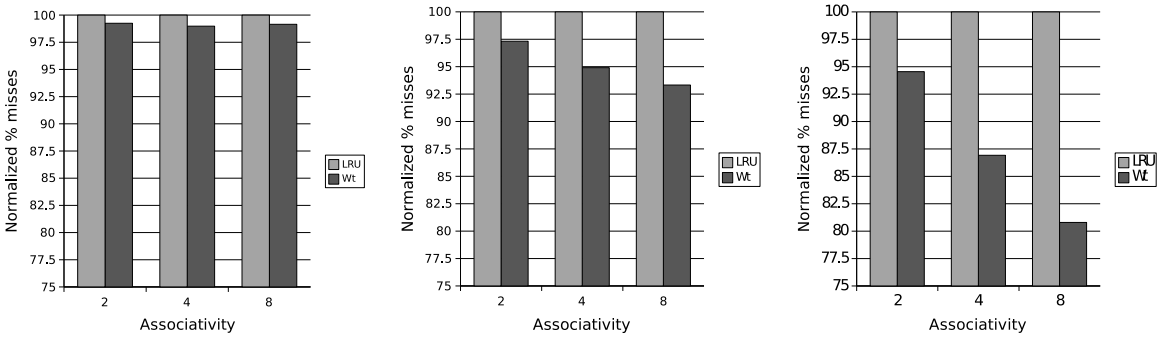
In this section we present the performance impact of bit selection schemes proposed in Section 3.2 and the weight-based replacement policy (Section 3.1). As both these optimizations impact only the LO cache, we present only the cache misses of an LO cache with and without these optimizations.

In Figure 10 we show the percentage misses of the three bit selection schemes *bsd*, *bsdw* and *bs ign8*, relative to an LO cache with no bit-selection. It can be seen that no significant gain is obtained from any of the index bit-selection schemes. Though there is a small improvement seen for some configurations, this improvement is not consistent for all traces or all sizes. A careful examination of the selected bits for indexing reveals that most of the bits selected are lower order bits that would have been used by the base scheme. For all three schemes we find that the selected index set differs from the



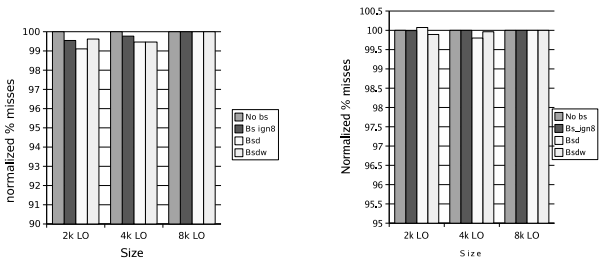
(a) 2k IHARC (b) 4k IHARC (c) 8k IHARC

Figure 8. Comparison of HSCA with IHARC (pld10M)



(a) 2k LO (b) 4k LO (c) 8k LO

Figure 11. Weight-based Replacement vs LRU (pld100M trace)



(a) pld100M (assoc. = 8) (b) pld1M (assoc. = 4)

Figure 10. Impact of bsd, bsdw and bs ign8 schemes

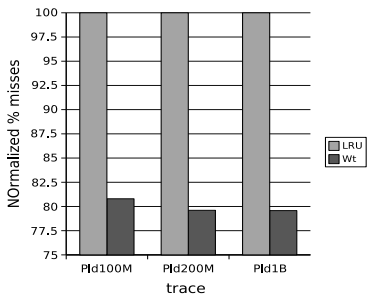
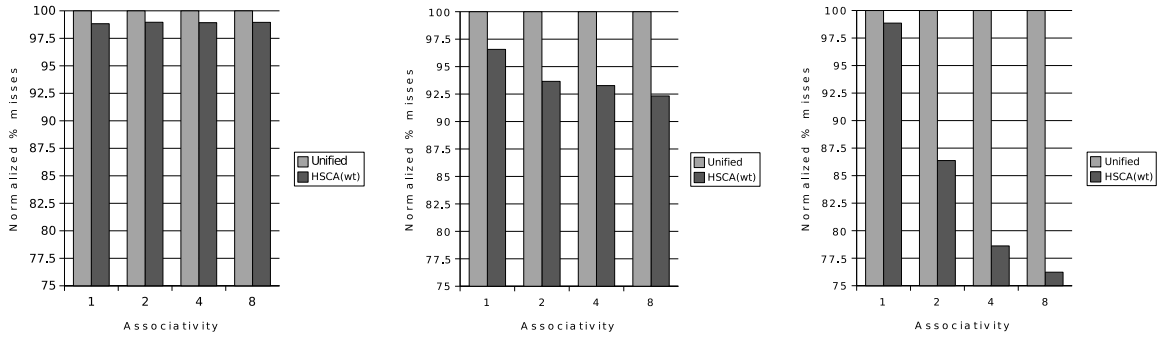


Figure 12. Impact of Locality on Weight Based Replacement

conventional index set in a maximum of two bits. This explains the lack of a significant performance improvement.

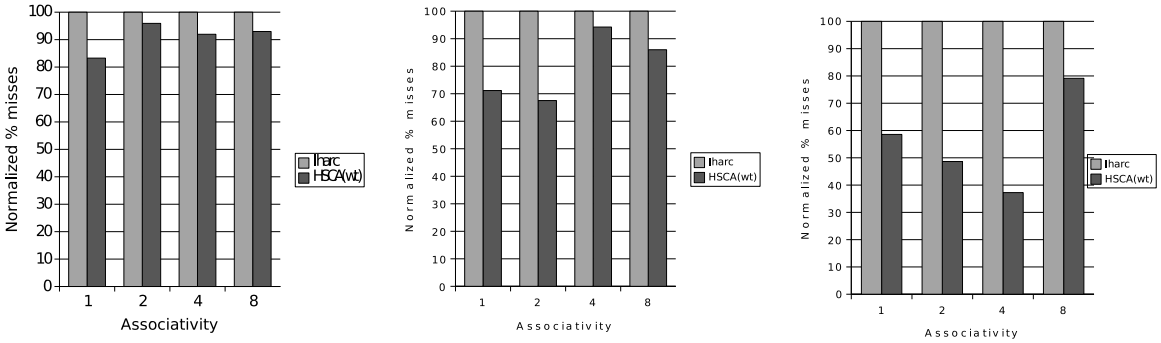
Figure 11 shows the percentage misses of the weight-based replacement policy relative to LRU for the pld100M trace. It can be seen that for large cache sizes (4k and 8k) weight-based replacement outperforms LRU by upto 20%. As size or associativity increases, the relative performance improvement gets better. A similar trend is seen for all traces having lesser locality than pld100M (refer to Figure 12) . For traces having more locality (pld1M and pld10M) LRU replacement performs better.

This implies that for core routers we should use a weight-based replacement for the LO cache of the HSCA while at the edge we



(a) 2k HSCA (b) 4k HSCA (c) 8k HSCA

Figure 13. Comparing HSCA with Weight-based replacement and Unified Cache (pld100M trace)



(a) 2k HSCA (b) 4k HSCA (c) 8k HSCA

Figure 14. Comparing HSCA with Weight-based Replacement and IHARC (pld100M trace)

should use LRU. An important point to note is that our performance measures are somewhat pessimistic in nature as the distribution function of the hit index used for trace generation has a natural tendency to favor LRU replacement.

5.2.4 Performance of HSCA at Core Routers

In this section we compare the misses incurred by HSCA with weigh-based replacement, with the misses of the unified scheme (refer Figure 13) and IHARC (refer Figure 14). Though the results here are shown for pld100M, the results for pld200M and pld1B are similar. The HSCA performs upto 60 % better than the IHARC scheme, and 24% better than the unified cache scheme. The 8k HSCA gives a large reduction in number of misses ($>$14%) from both the IHARC and the unified scheme for associativity greater than 1 where the replacement scheme comes into picture.

6 Related Work

Several approaches in literature have introduced a cache to speed up IP lookup [1][3][4][5][17]. Initial attempts at caching use conventional CPU caches and cache IP addresses along with their routing decision [5][17]. In [5] the authors suggest checking with a Host Address Cache (HAC) before searching through the routing table. In [17] the authors try to identify which bits of the address would be the best to index into the cache.

More recent attempts at caching can be broadly classified into caches for address ranges [3][4] and caches for data structures used for IP lookup [1][2]. In [3][4] the authors introduce Intelligent Host Address Range Caching (IHARC) which improves over HAC by first converting prefixes into distinct address ranges and then intelligently merging non-adjacent ranges that share a common routing decision. In an orthogonal attempt, [1] tries to exploit the additional locality in accesses to the nodes of a trie data structure. They suggest using a cache to store the recently accessed nodes of an LC-trie and evaluate its performance in a multiprocessor environment. In [2] an algorithm for LPM is introduced which uses a clever compression technique to compress the trie so as to fit it into an L2 cache. However this comes at the additional cost of accessing tables residing in a fast memory.

Our work belongs to the second classification, caches for IP lookup data structures. However, our work is different from other such

work in that we make use of the characteristics of node accesses in a trie to determine the cache organization. We also evaluate our cache architecture with realistic traces of varying locality and perform a comparative performance evaluation of various caching schemes for IP forwarding.

An interesting performance evaluation is done in [11] where different LPM algorithms are compared in terms of their suitability for caching. Other efforts at enhancing the overall throughput of the memory system of packet forwarding engines have been made [7][14]. The bandwidth of the DRAM memory is improved by exploiting row locality among incoming accesses in [7]. A pipelined memory architecture is introduced in [14] so as to enhance the memory throughput.

7 Conclusions

This paper presents a Heterogeneously Segmented Cache Architecture to cache the nodes of a trie data structure, used for IP forwarding. We choose a segmented cache architecture as the locality characteristics of the Level-One nodes (LO nodes) are significantly different from those of the Lower Level nodes (LL nodes). As only 17% of accesses to LO nodes are incurred by the LL nodes, size of the LL cache is fixed at $1/8^{th}$ of the size of the LO cache. We further enhance the performance of HSCA by introducing a weight-based replacement policy for the LO cache. To evaluate our caching scheme we propose a synthetic trace generation methodology which not only generates traces that have the characteristics of real traces, but also allows us to vary the inherent locality in the traces. This enables us perform a comprehensive performance evaluation of the HSCA. Our performance results show:

- HSCA outperforms the unified cache by incurring 17% fewer memory accesses than the unified cache.
- Relative to IHARC, HSCA results in upto 90% savings in number of misses for edge router traces and 50% savings in number of misses for core router traces.
- HSCA with weight-based replacement further reduces the misses incurred by the LO cache by upto 20% for core router traces, resulting in an overall improvement of upto 24% over the unified cache and upto 60% over the IHARC scheme.

As future work, we plan to evaluate the benefits of HSCA in a multiprocessor environment where the segmented nature of the cache would be able to service multiple memory accesses at a time. We also plan to investigate possible enhancements for the LL cache and find a better replacement policy than LRU for edge router traces.

Acknowledgments

This work was partly supported by a research grant from the Consortium for Embedded and Internetworking Technologies (CEINT) and Arizona State University, Tempe, USA. The first author acknowledges the travel grant provided by IBM India Research Laboratory to attend the ICS-05 conference. The authors are thankful to the members of the High Performance Computing Laboratory for their useful comments and discussions.

8 References

[1] J-L.Baer, D.Low, P.crowley and N.Sidhwaney. Memory hierarchy design for a multiprocessor look-up engine. In *Proc. of 12th Int. Conf. on Parallel Architectures and Compilation Techniques*, 2003.

[2] A.Brodnik, S.Carlsson, M.Degermark, and S.Pink. Small forwarding tables for fast routing lookups.In *Proc. of ACM SIGCOMM'97*, 1997.

[3] T.Chieueh and K.Gopalan. Improving Route Lookup performance using network processer cache. In *IEEE/ACM SC2002 Conf.*, 2002.

[4] T.Chieueh and P.Pradhan. Cache memory design for network processors. In *Proc. 6th Int. Symp. on High Performance Computer Architecture*, 2000.

[5] T.Chieueh and P.Pradhan. High-performance IP routing table lookup using CPU caching. In *Proc. of ACM Communication Architectures, Protocols, and Applications (SIGCOMM'97)*, 1997.

[6] K.C.Claffy. Internet Traffic Characterization. PhD thesis, University of California, San Diego, 1994

[7] J.Hasan, S.Chandra, and T.N.Vijaykumar. Efficient use of memory bandwidth to improve network processor throughput. In *Proc. of Int. Symp. on Computer Architecture*, 2003.

[8] R.Jain. Characteristics of destination address locality in computer networks: a comparison of caching schemes. *Computer Networks and ISDN Systems*, 1990.

[9] B.Lampson, V.Srinivasan, and G.Varghese. IP lookup using multiway and multicolumn search. In *Proc. of IEEE Infocom*, 1998

[10] S.Nilsson and G.Karlsson. IP-address lookup using LC-tries. *IEEE Journal on Selected Areas in Communications*, 1999.

[11] G.Narlikar and F.Zane. Performance modeling for fast IP lookups. In *Proc. of ACM SIGMETRICS*, 2001

[12] V.C.Ravikumar, R.Mahapatra, J.C.Liu. Modified LC-trie based efficient routing lookup. *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'02)*, 2002.

[13] M.Ruiz-Sanchez, E.Biersack, and W.Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Nagazine*, 2001.

[14] T.Sherwood, G.Varghese, and B.Calder. A pipelined memory architecture for high throughput network processors. In *Proc. of Int. Symp. on Computer Architecture*, 2003.

[15] W.Shi, M.H.MacGregor, and P.Gburzynski. Synthetic trace generation for internet.*IEEE Workshop on Workload Characterization (WWC-4)*, 2001.

[16] D.Thiebaut, J.L.wolf, H.S.Stone. Synthetic traces for trace-driven simulation of cache memories. *IEEE Transactions on Computers*, 1992.

[17] B.Talbot, T.Sherwood, and B.Lin. IP caching for terabit speed routers. *Proc. of IEEE Globcom*, 1999.

[18] M.Waldvogel, G.Varghese, J.Turner and B.Plattner. Scalable high speed IP routing lookup. In *Proc. of ACM SIGMETRICS*, 2001

[19] http://www.nadh.kth.se/∼snilsson.