100+ Gbps IPv6 Packet Forwarding on Multi-Core Platforms

Thilan Ganegedara, Viktor Prasanna
Ming Hsieh Dept. of Electrical Engineering
University of Southern California
Los Angeles, CA 90089
{ganegeda, prasanna}@usc.edu

***Abstract*—The migration from IPv4 to IPv6 addressing is gradually taking place with the exhaustion of IPv4 address space. This requires the network infrastructure to have the capability to process and route IPv6 packets. However, with the increased complexity of the lookup operation and storage requirements, performing IPv6 lookup at *wire-speed* is challenging. In this work, we propose a high-performance IPv6 lookup engine solution for multi-core platforms that deliver state-of-the-art line card throughput rates. In order to exploit the parallelism offered on modern multi-core platforms, we propose a routing table partitioning scheme that forms disjoint and balanced partitions, given a IPv6 routing table. These partitions are represented as *range trees* to perform the lookup operation. Due to the disjoint nature of the proposed partitioning scheme, the individual range trees are able to operate independently, improving the parallelism of the lookup engine. Our experimental results on state-of-the-art multi-core processors show that throughputs of $100+$ Gbps can be achieved for 2 million entry IPv6 routing tables using the proposed scheme. Compared with existing literature, the proposed solution achieves $10\times$ higher throughput and is on par with performance delivered by hardware IP lookup engines.**

Keywords*—*Packet forwarding, IPv6, Multi-core, Networking, Routers

I. Introduction

The proliferation of Internet enabled devices has caused the exhaustion of the most prevalent logical addressing space i.e., IPv4 [2]. Due to this reason, the next generation of IP addressing, IPv6, has begun its debut. IPv6 addresses are 128 bits long whereas IPv4 addresses are 32 bits long [6]. The increased bit length of IPv6 facilitates the addressing demands of the continuously growing Internet. However this necessitates the existing network infrastructure to be augmented with capabilities to process and forward IPv6 packets, which will require redesigning of packet forwarding engines.

Considering the requirements of current backbone networks, IPv6 packet forwarding engines are ought to have the ability to cope with the following two aspects.

- Increased size of the routing table — this is essential for the solution to be scalable to larger routing tables
- Increased lookup complexity — to deliver high throughput to support *wire-speed* packet forwarding

Both software and hardware platforms are used in the literature to implement high performance packet forwarding engines [1, 4, 5, 11]. While hardware platforms deliver high performance, the processing flexibility offered on such platforms is limited and the resources such as memory and logic is also limited [1]. On the other hand, software platforms offer various computational resources and offer superior processing capabilities and abundant memory resources [5]. The challenge, however, is achieving high performance. Since these platforms are General Purpose Processors (GPPs), the processing kernels need to be carefully designed such that the latency of operation is minimized and the memory hierarchy of the GPP is exploited.

Existing software solutions for packet forwarding engines suffer from low throughput mainly due to the limited parallelism despite the abundant parallel processing capabilities available on modern GPPs [5]. Further, due to the sequential nature of a single lookup operation, the lookup latency becomes a dominant factor when determining the performance of software forwarding engines. Low latency operation is desirable in order to increase the packet lookup rate [10]. Further, routing table storage becomes a critical concern with the hierarchical memory organization and exploiting cache memory (low access latency) while minimizing communication with main memory (high access latency) becomes imperative.

In this work, we propose a solution to perform *wire-speed* IPv6 packet forwarding on modern multi-core platforms. We devise an algorithm that partitions a routing table into a set of disjoint and balanced partitions, which enables us to search only one partition to find the matching prefix for a given packet header. This is highly desired for multi-core platforms since 1) a lookup operation is limited to a single partition which is of smaller size than the original routing table, and 2) this provides opportunity for more parallelism. Based on this partitioning, we propose a range tree based solution to achieve high packet forwarding rates on multi-core platforms. We evaluate our solution on two AMD Opteron 6200 series processors and show that throughputs of $100+$ Gbps can be achieved, which make them suited for state-of-the-art line cards on backbone routers.

We summarize our contributions in this paper as follows:

- An easily parallelizable solution able to exploit the parallel processing capabilities available on modern multi-core platforms
- Reduced worst case packet latency via routing table partitioning

This work is supported by the United States National Science Foundation under grant No. CCF-1116781. Equipment grant from Xilinx Inc. is gratefully acknowledged.

- A hierarchical multi-threaded lookup architecture that can be mapped onto multi-core platforms efficiently
- 230+ Million Lookups Per Second (MLPS)[1] rates for 2 million entry IPv6 routing tables on AMD Opteron 6200 series processors

II. Related Work

Recently there have been many efforts to devise memory and/or resource efficient IPv6 lookup engines that deliver high-performance. In [4], the IPv6 engine was realized using a combination of Content Addressable Memory (CAM) and Programmable Logic Device (PLD), in which the prefix search was performed by the CAMs while the PLD aggregated the results from the CAMs (priority encoder) to yield Longest Prefix Match (LPM) [9]. Although the routing table was distributed across multiple CAMs based on the subnet mask length, all the CAMs were required to perform the search increasing the amount of computations performed per lookup. The architecture yields lower throughput due to the lack of pipelining and the delays introduced by the priority encoder. A multi-bit trie based solution was explored in [5] that exploited the prefix length distribution of IPv6 routing tables. The lookup engine employed a combination of trie search and hashing to perform the search. The performance was evaluated on a Intel IXP2800 network processing unit (NPU) that runs at 700 MHz. For a 400K entry synthetic IPv6 routing table, their solution required 35 MBytes (280 Mbits) of memory and operated at 21 MLPS rates (multi-threaded).

In [10] the authors propose a partitioning and binary search tree based solution for high performance packet forwarding on multi-core platforms. They form partitions in a similar fashion as done in this paper. However, no balancing of partitions is proposed, which causes packets belonging to different partitions to experience different delays. Further, as noted by the authors, the solution is not scalable to IPv6 despite the 700+ MLPS lookup rates. [11] proposes a range tree based solution for IPv6 which partitions the routing table by exploiting the span of IPv6 prefixes. This requires them to search each partition, one after the other, in order to arrive at LPM. Such sequential operations on multi-core platforms can yield poor performance due to increased packet lookup latency.

III. Routing Table Statistics

In order to explore the viability of our solution, we first explored the features of real-life IPv6 backbone routing tables. These routing tables were obtained from RIPE Routing Information Service (RIS) project [7]. The collected routing tables are dated 07/30/2012 and the average statistics of the routing tablesare given in Table I.

As it can be seen from Table I, currently, the IPv6 addresses constitute just above 2% of the total number of prefixes on the average. However, this number is expected to grow significantly with networking companies adopting IPv6 and eventually IPv4 becoming obsolete. With such a scenario, the resource requirements and latency issues will become major concerns in packet forwarding engines. Even though the number of IPv4 prefixes are in the range of 430K on the average, after removing the duplicate prefixes, the prefix count becomes 350K on the average.

In order to understand the memory consumption, we computed the memory requirement of these routing tables if they were implemented as uni-bit trie and range tree. The memory computations were carried out as per Eq. 1a, Eq. 1b and Eq 1c.

The notations are as follows with the values used for the computations inside parenthesis: N_T - number of nodes in a trie, W_p - number of bits per pointer field (15 bits), W_{nhi} - number of bits per Next-Hop forwarding Information (NHI) field (6 bits), N_R - number of ranges produced by the prefixes, W_{ip} - number of bits per prefix (64 bits). For the experiments, we considered leaf-pushed trie, which essentially reduces the memory footprint of a trie. When converting a regular uni-bit trie to a leaf-pushed trie [8], the number of nodes increase since new nodes are created when routing information is pushed down to the leaf level [8]. For the considered IPv6 routing tables, we observed a $1.866\times$ node count inflation on the average when leaf-pushed. In a leaf-pushed trie, half of the nodes, non-leaf nodes, contain pointers to children nodes and the other half, leaf-nodes, contain NHI.

For range tree [11], we consider two approaches, namely, explicit and implicit. In both versions, the ranges are arranged similar to a BST (both versions have same structure), but the search operation is different. In the explicit version, the entire range is saved as lower and upper bounds, and the incoming key is compared against both bounds and the traversal decision is made based on the result of the two comparisons. Since the ranges are disjoint, if the input key falls within a particular range, then the search can be terminated at that node. This is helpful especially on software platforms where early termination of search yields lower packet latency, hence higher throughput. In the implicit version, only the upper bound is stored and the incoming packet is compared against the stored value and the search needs to go up to the leaf-node level for termination. This is useful on hardware platforms where linear pipelines are employed, since the entire pipeline needs to be traversed despite an early termination in search. Considering the chosen platform, in this work we use the explicit storage of ranges.

$$M_{trie} = 1.866 N_T \times (2W_p + W_{nhi}) \tag{1a}$$
$$M_{rtree-exp} = N_R \times (2W_{ip} + W_{nhi}) \tag{1b}$$
$$M_{rtree-imp} = N_R \times (W_{ip} + W_{nhi}) \tag{1c}$$

Table I reveals that the real-life IPv6 routing tables are small, hence, the full effect of a backbone routing table cannot be observed using them. For this reason and to evaluate the scalability of the proposed solution, we generated large IPv6 routing tables using an extended version of FRuG [3]. Using FRuG, the prefix distribution and the structure of the seed routing tables are preserved when generating the synthetic routing tables. We show the normalized prefix length distribution of the real and synthetic routing tables generated using the RRC00 IPv6 routing table in Figure 1 and it can be seen that the synthetic routing table has almost the same prefix length distribution as that of the real routing table. However, it should be noted that for range tree, the worst case scenario is if a particular routing table has N prefixes, the number of ranges that can exist is bound by $2N - 1$. We evaluated this empirically and found that the ranges per prefix is 1.14 and

[1]Gbps = $0.1 \times$MLPS$\times 64 \times 8$ for minimum size (64 Byte) packets

TABLE I. Average routing table statistics obtained from [7] dated 07/30/2012 for RRC0 - RRC16

# IPv4 Prefixes	# IPv6 Prefixes	IPv6 Statistics					
		#Trie nodes	Trie (Mb)	# Ranges	Ranges/prefix	Range tree - Exp. (Mb)	Range tree - Imp. (Mb)
415072	9079	110259	1.14	10209	1.14	1.39	0.74

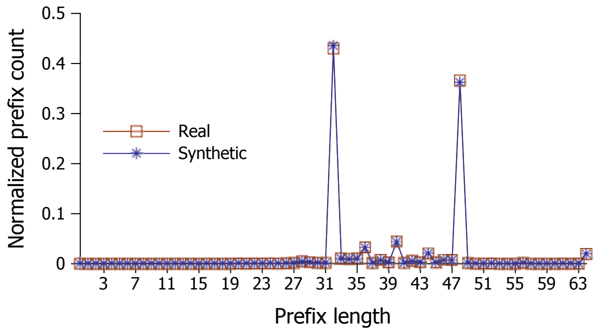


Fig. 1. Normalized prefix length distribution for real and synthetic IPv6 routing tables generated using RRC00 backbone routing table

0.84 for real and synthetic routing tables (350K routing table for each real routing table), respectively, on the average. The reason for the value to be lower for the synthetic routing table is the increased amount of prefix overlap.

The main use of the synthetic routing tables is to highlight the benefits of using range tree as opposed to trie. The memory requirement of trie can increase significantly with increasing number of prefixes as the trie size grows dramatically depending on the prefix distribution of the routing table. This is on one part due to the increased prefix length. Since a IPv6 prefix is 64 bits in length, the effect of a single prefix is much more than that of a IPv4 prefix. Further, doubling of packet lookup latency also becomes a concern. In order to evaluate this quantitatively, we generated synthetic routing tables of 350k prefixes for each real routing table and observed memory consumption of each approach. The average values indicate that using uni-bit trie requires 29.19 Mbit, explicit range tree requires 39.77 Mbit and implicit range tree requires 21.05 Mibt. Note that all memory computations follow Eq. 1a, Eq. 1b and Eq. 1c using the aforementioned bit lengths.

IV. Range Tree-based IPv6 Lookup

Tree-based solutions, are elegant in terms of the number of memory accesses required to perform a lookup, which is $O(\log N)$, where N is the number of keys/ranges [11]. In this section, we explain our approach for the IPv6 lookup engine. First we present the details of our solution and discuss how multi-core platforms can deliver high performance using such techniques.

A. *Enabling Parallelism for IP lookup*

Despite the platform and even application, considering current trends, enabling parallelism is essential for high performance computing. IP lookup is not a compute-bound, but a memory-bound operation. In such cases, parallelism is critical to exploit the multiple processing cores available on today's GPPs. However, it is not straightforward to parallelize the IP lookup operation in an efficient manner on these platforms.

The most basic method of parallelization can be thought of as search tree duplication, in which case, all partitions possess the entire routing table information and lookup can be carried on independent of other packet lookups. However, this causes the memory required to store the search tree to increase proportional to the number of partitions. While this is not a desirable solution, in most cases on software platforms, this translates to increased lookup time since the cache memory will not be sufficient to store the duplicated search trees, especially for large routing tables.

A more attractive method to perform partitioning is to place the prefixes in a set of bins in such a way that the prefixes in one bin are disjoint from those in other bins. In the context of our problem, IPv6 lookup, the state of being disjoint can be described as being able to search in only one bin and finding the corresponding routing information, without consulting the other bins. Since packets correspond to different prefixes in a given trace, when more disjoint partitions are present, the more likely for them to be processed independently and in a parallel fashion. This provides opportunities for parallelism. However, it is imperative that the formed partitions are of similar sizes. Otherwise, it gives rise to various other issues such as fairness (some packets experiencing longer latency than others) and uneven memory distribution.

B. *Disjoint Partitioning*

Disjoint partitioning can be achieved in numerous ways. For example, the leftmost bits of the IP addresses can be used and by exploiting the distribution of the prefixes in the routing table, the partitions can be formed. Instead of using the leftmost bits, a selected set of bits from the IP addresses can be chosen as well. In this work, we consider partitioning using the initial (leftmost) bits of the prefixes. The initial bits based partitioning divides the address space into multiple disjoint sections and considering subsets of prefixes belonging to each section as a smaller routing table. If p bits are used, there can be as many as $O(2^p)$ partitions, depending on the prefix distribution of a given routing table. Partitions formed using this method can be of disparate sizes. Also, the prefixes with length less than p needs to be expanded under this partitioning scheme. Our analysis of real routing tables indicate that prefixes of length 25 and shorter constitute only 1% of routing table. Hence, the effect of shorter prefixes is negligible.

The proposed disjoint partitioning has two main benefits: 1) identification of the corresponding partition for an incoming packet is simply a table lookup which can be completed in $O(1)$ time and 2) even though the initial partitioning may not be balanced (i.e. near-uniform prefix distribution across partitions), it is relatively easy to achieve balanced partitioning by aggregating initial partitions.

C. *Algorithm and Partitioning*

Once the initial partitioning using the p leftmost bits is done, we process them further to achieve balanced partitioning. The rationale behind this further processing is that the initial

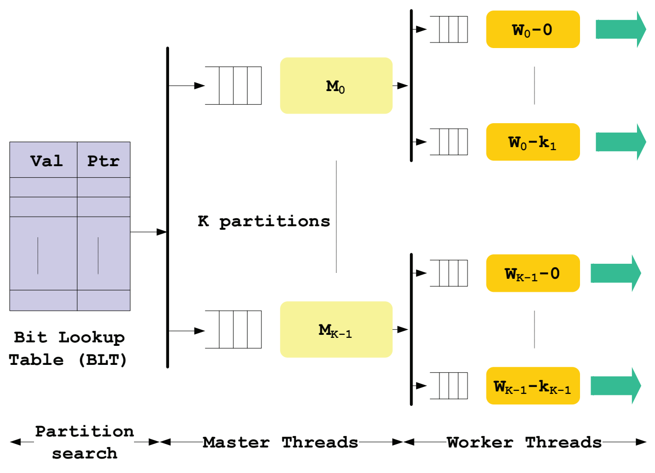


Fig. 2. Hierarchical multi-threaded architecture of the proposed IPv6 lookup engine

partitions may contain different number of prefixes which causes the packet latency for different partitions to change significantly across partitions. Hence, we perform an aggregation operation in which a subset of initial partitions are combined to form a single aggregated partition. This step may not ensure perfectly balanced partitioning depending on the distribution of the prefixes. However, perfect balancing is not critical in our application as we describe later.

The aggregation algorithm can be described as follows:

1: Define the maximum size for the aggregated partitions, $maxsize$
2: If all partitions are marked *used* go to step 8, if not, go to step 3
3: Select the largest initial partition that is not marked *used*
4: If the selected partition size is equal to or greater than $maxsize$, mark the selected partition as *used* and go to step 2
5: If all partitions are marked *used* go to step 8, if not go to step 6
6: Starting from the smallest partition that has not been marked *used*, combine the prefixes from the smaller partition into the larger partition
7: If the aggregated partition size is equal to or greater than $maxsize$ go to step 2, otherwise go to step 5
8: Algorithm complete

For a routing table with N prefixes, time complexity for initial partitioning is $O(N)$. To form the aggregated partitions, the time complexity is $O(n_i)$ where n_i is the number of initial partitions formed, hence the time taken to form the partitions is fairly small. Table II shows the effect of using the aforementioned algorithm. The number of bits used for partitioning is denoted by p, and n_i and n_a stand for initial and aggregated number of partitions, respectively. For the results shown in Table II, we varied p and set $maxsize$ to the largest initial partition size of each scenario. One can use the tuning parameters p and $maxsize$ to adjust the number of aggregated partitions (n_a) created.

Our experimental results revealed that the aggregated partitions have nearly the same size except for the last few partitions. This is due to the way the partitioning algorithm operates. As it can be seen from the aggregation algorithm steps, the remaining smallest size partitions are aggregated

TABLE II. EFFECT OF INITIAL AND AGGREGATED PARTITIONING ON REAL AND SYNTHETIC ROUTING TABLES

Routing Table	p	n_i	n_a
Real	10	7	4
	15	25	4
	20	294	6
Synthetic	10	24	5
	15	336	12
	20	4854	94

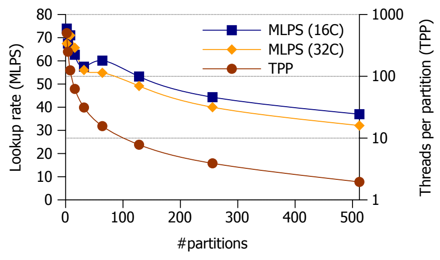
to form an aggregated partition such that the size of the aggregated partition satisfies the size requirement. The size requirement can be changed in order to vary the number of aggregated partitions generated. For the partitioning in Table II, we have used the size of the largest partition as $maxsize$. Hence, whenever the aggregated partition size reaches $maxsize$, the algorithm completes generation of one aggregated partition. This way, the last few partitions do not have adequate smaller partitions to form partitions of size in the range of the maximum size, therefore the algorithm creates few smaller partitions. The effect of this variation ultimately translates to a difference in range tree height and in our experiments we observed a maximum of 2 level difference in the range tree. For example, in the synthetic routing table case, when $p = 15$, the largest aggregated partition size is 32124 prefixes and the smallest is 10373 — a difference of two tree levels. Hence, the overall effect of partition size difference can be tolerated to some extent as long as the packet latency for different partitions does not deteriorate significantly.

V. Lookup Algorithm

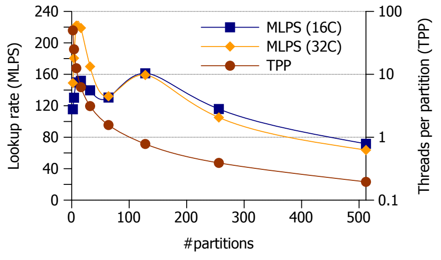
In this section, we describe how to map the partitions formed by the aggregation algorithm onto multi-core platforms and how the packet forwarding is performed.

The initial portion of the lookup is the partition identification. This can be simply realized using an initial lookup table. For example, if the first p bits were used for partitioning, then the lookup table of size 2^p will hold the sub-tree pointer information for each partition. Multiple entries may point to the same sub-tree pointer due to the aggregation step. With this approach, the partition search complexity becomes $O(1)$ which is desirable for high-speed operation.

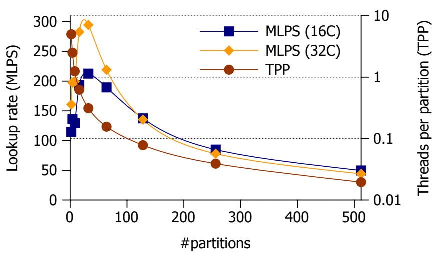
For the main lookup engine that performs range tree search,



a IPPT = 10000



b IPPT = 100000



c IPPT = 1000000

Fig. 3. Performance of the software lookup engine for varying IPPT values.

we adopt a master-worker architecture. After the partition search (described previously), the packet is forwarded to the corresponding master thread, denoted M_i, where i is the corresponding aggregated partition index. Each M_i creates a set of worker threads, denoted W_{i-j}, where j is the worker thread number. The architecture is flexible in that, the number of worker threads created, can be controlled by the user. The overall architecture is depicted in Figure 2.

On multi-core platforms, it is desirable to have more partitions (i.e. more master threads) which provides more opportunity for parallelism. Further, when the number of partitions is higher, the number of prefixes (hence the height of the range tree) per partition decreases. This effectively reduces packet lookup latency. Also, the explicit range tree approach is more suited for the software engine since the search can be terminated when the corresponding node is found. This comes at the cost of higher memory consumption. However, on GPPs, this is a minor concern.

VI. Performance Evaluation

We use two state-of-the-art platforms to evaluate the architecture presented in Section V: 1) 2× AMD Opteron 6278, 2.4 GHz, 16C, with 16 MB L2 cache, 16 MB L3 cache and 128 GB DDR3 main memory, and 2) 2× AMD Opteron 6220, 3.0 GHz, 8C, with 8 MB L2 cache, 16 MB L3 cache and 64 GB DDR3 main memory, which we name 32C and 16C, respectively. We use two platforms to highlight the performance variation with higher number of cores and clock frequency on the lookup engine performance.

For performance evaluation, we used synthetic routing tables generated using [3]. As for the packet traces, in order to evaluate the random access performance, we generated traces using the disjoint ranges generated from the routing tables in a uniformly random fashion. No temporal locality was assumed. If the number of generated ranges is N_R, then the probability of an IPv6 address being generated from a specific range is $1/N_R$. The average tree depth traversed by a packet for a given trace can be calculated as $\sum_{k=0}^{k_1} k \times \frac{2^k}{2^{k_1+1}-1} \approx k_1$ for a complete tree with k_1 levels. This ensures that most of the generated IPv6 addresses correspond to a leaf or near-leaf level of the tree, which simulates a near worst case scenario. However, in a realistic environment, the packets may terminate the search at intermediate levels, in which case, the performance will be better than what is reported in this paper. In this paper, we evaluate the lookup portion of the software engine, i.e. the range tree lookup. Since the initial partition search is $O(1)$ time and the size of the Bit-Lookup Table (BLT) is considerably small compared with the size of the search trees, this operation can be performed within few clock cycles.

A. *Performance of Master-Worker Architecture*

First, we examined the performance of the lookup engine for a fixed trace size. And note that, for this experiment we assume that the packet trace is evenly distributed across the partitions. Even though this simulates a best case scenario, later we discuss the worst case performance for the case in which all the packets are directed to a single partition. It must be noted that the worst case scenario is highly unlikely in a core router due to the high traffic volumes that traverse through the core networks.

We considered a 10 million IPv6 address trace and observed the performance variation for an increasing number of partitions. In this experiment, we controlled the number of packets each worker thread handles in order to observe its effect. We name this parameter IPs Per Thread (IPPT). Figure 3 illustrates the results of these experiments.

We calculate Threads Per Partition (TPP) as `partition trace size/IPPT`. The key observation is that the lookup rate is best when TPP $\approx$ 1. This variation is mainly due to thread creation overhead. Since a new worker thread is created for each trace subset with IPPT number of IPs, the total number of threads created is high for lower IPPT. For the case where TPP $<$ 1 and the number of partitions is high, more master threads are created with low load. Both these scenarios cause the performance to degrade, suppressing the gains achieved by enhanced parallelism and reduced search tree height. We keep the discussion on Figure 3 short due to space limitations. Rather, we describe how such thread creation overheads can be eliminated by modifying the proposed software architecture.

B. *Performance of Master-only Architecture*

The master-worker architecture requires dynamic thread creation and buffer management, creating adverse effects on lookup rate. We conducted experiments for the case where the master thread itself takes care of the IP lookup process instead of creating worker threads. The performance values obtained are shown in Figure 4. We report performance for two scenarios: 1) when all masters are fully utilized (best case - parallel) and 2) when only one master thread is utilized (worst case - serial). For both cases, the trace size was kept the same. As expected, performance increases with increasing number of partitions initially and declines due to increased context switching overhead. The 32C outperforms 16C in the best case

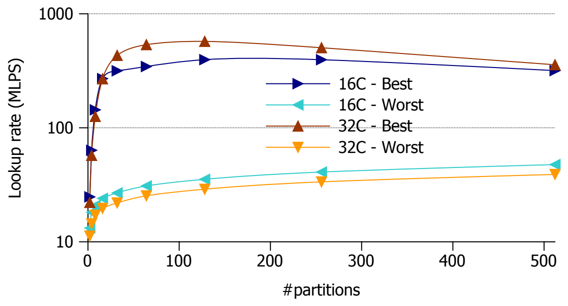


Fig. 4. Best and worst case performance of master thread only approach

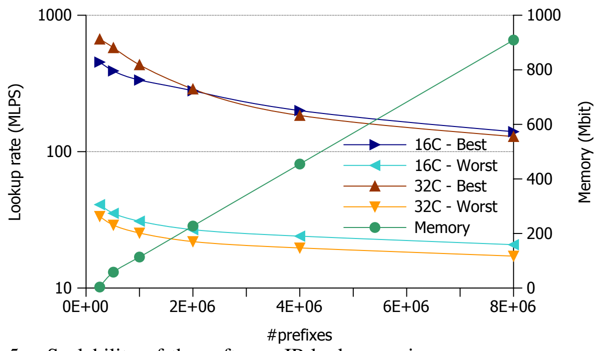


Fig. 5. Scalability of the software IP lookup engine

due to more cores, however in the worst case, the higher clock frequency of 16C gains advantage over 32C.

The memory consumption of the proposed solution is 3× lower than that of [5] and lookup rate is nearly 5× higher than both [5, 11] solutions even after scaling for technology gap. Even though the comparison is not detailed due to space limitations, it is evident that the proposed solution outperforms existing literature by fair margins. Compared with the IPv4 lookup engine proposed in [10], our IPv6 lookup engine delivers similar performance despite the increased lookup complexity and storage requirements.

In order to evaluate the performance for larger routing tables, we generated routing tables of various sizes that represent current backbone routing table sizes and beyond, and observed the performance. The results are shown in Figure 5 and for this experiment, we fixed the number of partitions to 128 since in Figure 4, 128 partitions yielded the best performance. The performance variation shown in Figure 5 is expected due to larger routing table size, which translates to increased tree depth. Also, when considering the best performance curves, the higher core count of 32C platform yields higher lookup rate due to higher parallelism available. However when the routing table size increases, due to the size increase of each sub-tree, the 16C platform with higher clock frequency, outperforms the 32C platform (faster memory operations and context switching). In the worst case performance scenario, due to higher clock frequency, the 16C platform delivers high performance. Note that, for this experiments, we ensured that the total L3 cache size of the processor is exceeded to observe performance variation even when a portion of the routing table resides in main memory.

VII. Conclusion

In this work, we proposed a range tree based solution for IPv6 lookup that is suited for modern multi-core platforms. We devised a tunable partitioning algorithm that forms a set of disjoint, yet balanced, subsets of prefixes, given a IPv6 routing table. This enabled us to enhance parallelism on software platforms. We proposed two architectures, namely master-worker and worker-only, to show different possible configurations of the lookup engine. The two architectures were evaluated on two AMD Opteron 6200 series processors and the random access performance was measured. The experimental results showed that the master-only architecture yield high performance over master-worker architecture, mainly due to thread creation overhead of the master-worker architecture. The master-only architecture was then tested extensively and the results ensures 100+ Gbps throughputs for a 2 million entry IPv6 backbone routing table. The proposed architecture can be extended to include incremental routing table updates. Routing table updates can be localized to a partition, therefore updates can be calculated faster.

References

[1] M. Bando, Y.-L. Lin, and H. J. Chao. Flashtrie: Beyond 100-gb/s ip route lookup using hash-based prefix-compressed trie. *Networking, IEEE/ACM Transactions on*, 20(4):1262 –1275, aug. 2012.

[2] American Registry for Internet Numbers (ARIN). Ipv4 address exhaustion. https://www.arin.net/announcements/2011/20110203.html.

[3] T. Ganegedara, Weirong Jiang, and V. Prasanna. Frug: A benchmark for packet forwarding in future networks. In *Performance Computing and Communications Conference (IPCCC), 2010 IEEE 29th International*, pages 231 –238, dec. 2010.

[4] T. Hayashi and T. Miyazaki. High-speed table lookup engine for ipv6 longest prefix match. In *Global Telecommunications Conference, 1999. GLOBECOM '99*, volume 2, pages 1576 –1581 vol.2, 1999.

[5] Xianghui Hu, Bei Hua, and Xinan Tang. Triec: a high-speed ipv6 lookup with fast updates using network processor. In *Proceedings of the Second international conference on Embedded Software and Systems*, ICESS'05, pages 117–128, Berlin, Heidelberg, 2005. Springer-Verlag.

[6] Internet Engineering Task Force (IETF). Ipv6 addressing architecture. https://tools.ietf.org/html/rfc4291.

[7] RIPE. Ripe routing information service (ris). http://www.ris.ripe.net/.

[8] M.A. Ruiz-Sanchez, E.W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 15(2):8 –23, 2001.

[9] Wikipedia. Longest prefix match. http://en.wikipedia.org/wiki/Longest_prefix_match.

[10] Marko Zec, Luigi Rizzo, and Miljenko Mikuc. Dxr: towards a billion routing lookups per second in software. *SIGCOMM Comput. Commun. Rev.*, 42(5):29–36, September 2012.

[11] Pingfeng Zhong. An ipv6 address lookup algorithm based on recursive balanced multi-way range trees with efficient search and update. In *Computer Science and Service System (CSSS), 2011 International Conference on*, june 2011.