Parallel and Efficient IP Lookup using Bloom Filters on Intel® Xeon Phi™

Alexandre Lucchesi[1], **André C. Drummond**[1], **George Teodoro**[1]

[1]Department of Computer Science
University of Brasília
Brasília, Brazil

lucchesi@aluno.unb.br, {andred, teodoro}@unb.br

Abstract. *The IP lookup phase is the core operation in packet forwarding, which is implemented via a Longest Prefix Matching (LPM) to find the next hop for every input address. In this work, we evaluate the use of parallel techniques to develop a highly optimized IP lookup algorithm that employs Bloom filters and hash tables. More specifically, we investigate the implementation of our algorithm on multi-core CPUs and on the Intel® Xeon Phi™ (Intel Phi) many-core coprocessor. Our analysis includes the efficient parallelization of our Bloom filters algorithm on both devices, and the experimental results show that we were able to attain high performance with this solution (over 88 million lookups per second on a single Intel Phi for IPv6). We also compared the Bloom filters optimized solution to an efficient approach based on the Multi-Index Hybrid Trie (MIHT). This comparison shows that the most efficient sequential algorithm may not be the best option in a parallel setting. Instead, it is necessary to evaluate the processors characteristics, algorithms compute/data demands and data structures employed to analyze how the algorithms will benefit from the target computing device. These findings are also important to new efforts in algorithmic developments in the topic, which have been highly focused on sequential solutions.*

1. Introduction

The use of software routers is motivated due to its easy extensivity, programmability, and good cost-benefit. However, these routers are required to attain high packet forwarding rates, which is a challenging task that may be reached with more efficient algorithms and/or with the use of high-performance parallel computing techniques. The calculation of the next hop for input packets is a core operation in the forwarding phase of routers. Since the development of CIDR, routers are required to run a Longest Prefix Matching (LPM) algorithm in order to determine the next hop for each received packet. This task is commonly referred to as IP lookup and is often the performance bottleneck in high-performance software routers.

In this work, we investigate the Intel Phi coprocessor as a platform for the implementation of efficient LPM algorithms for IP lookup. The Intel Phi is a highly parallel platform that supports up to 244 threads, provides 512-bit SIMD instructions, and has a large memory bandwidth. These characteristics make the Intel Phi an attractive platform for the implementation of LPM for multiple incoming IP addresses. We have designed a parallel algorithm that uses Bloom filters and hash tables to efficiently find the LPM for both IPv4 and IPv6. The implementation leverages the Intel Phi capabilities to mitigate

the main drawbacks of the algorithm — namely, the high costs to compute hashes during lookup/store operations and the high memory bandwidth requirements. This is achieved with the use of vectorization to reduce the costs of hashing and thread-level parallelism to increase concurrency and throughput.

In order to evaluate our algorithm, we have compared the implementation to a parallel version of the Multi-Index Hybrid Trie (MIHT). The MIHT is a state-of-the-art sequential lookup algorithm that has been shown to attain better performance than well-known tree/trie based algorithms: the Binary Trie, the Prefix Tree, the Priority Trie, the DTBM, the 4-MPT and the 4-PCMST [Lin et al., 2014]. The experimental evaluation of the algorithms have shown that our optimized Bloom filters algorithm was able to outperform the MIHT approach in both sequential and parallel executions on the Intel Phi. In a parallel execution using IPv4 and IPv6 prefix datasets our Bloom filters optimized algorithm was, respectively, up to 4.31× and 5.39× faster than MIHT. The results show that, although the MIHT is a very memory-efficient algorithm, it presents fewer opportunities for optimizations and worse scalability on Intel Phi. For instance, the use of vector SIMD instructions available in most of the modern device architectures can be leveraged to improve the Bloom filters approach, but is not effective for MIHT because of the irregular nature of the data structures it uses. The main contributions of this paper can be summarized as:

- We implement an efficient version of the Bloom filters based LPM algorithm that fully exploits the Intel Phi capabilities.
- We evaluate the compromises of using the Intel Phi coprocessor and modern CPUs to the IP lookup problem using two algorithmic strategies.
- We propose a novel approach combining dynamic programming and Controlled Prefix Expansion [Venkatachary and Varghese, 1998] (DPCPE) to enhance the performance of IPv6 lookups in the Bloom filters based algorithm.
- We show in our experimental evaluation that the most efficient sequential algorithm may not be the better solution in a parallel setting. Instead, the possibility of adapting the algorithm to fully utilize the processor features and parallelism can lead to higher performance.

The rest of this paper is organized as follows. Section 2 describes the Intel Phi coprocessor and related works. Section 3 describes the use of Bloom filters to solve the IP lookup problem. Section 4 details the algorithm design, including the optimizations, parallelization strategies, and relevant implementation details. We experimentally evaluate our algorithm in Section 5 and present conclusions and future directions in Section 6.

2. Background and Related Work

2.1. Intel® Xeon Phi™

The Intel Phi coprocessor is based on the Intel Many Integrated Core architecture (MIC), which consists of many simplified, power efficient, and in-order computing cores equipped with a 512-bit vector processing unit (SIMD unit). The Intel Phi 7120P used in this work has 61 cores with a four-way hyperthreading that leads to the simultaneous execution of up to 244 threads, and a theoretical peek performance of 1.2 teraflops. Each core is clocked at 1.333 GHz, has 512 KB private L1 cache, and a L2 cache of about 30 MB (aggregate for all cores) that is kept fully coherent among cores via a directory tag mechanism. The computing cores, caches, and memory controllers are connected through a

high bandwidth bidirectional ring interconnect (352 GB/s). The Intel Phi is also equipped with 16 GB of GDDR5 of main memory. The combination of the hyperthreading and the high-bandwidth memory is efficient for hiding memory accesses latency that is important for memory-intensive applications.

The MIC architecture combines features of general-purpose CPUs and many-core processors or accelerators to provide an easy to program and high-performance computing environment [Jeffers and Reinders, 2013]. It is based on a x86 instruction set and supports traditional parallel and communication programming models, such as OpenMP (Open Multi-Processing), Pthreads (POSIX Threads Programming), MPI (Message Passing Interface), etc. It runs applications in *native mode* or *offload mode*. In native mode, the user directly accesses the coprocessor via SSH connections to run the application within it. This is possible because Phi runs a simplified operating system. In offload mode, the programmer selects parts of the application, usually annotating the source code with specific pragmas, to be automatically transferred and executed on the coprocessor.

2.2. Previous work

The trie/tree-based is a popular class of IP lookup algorithms [Ruiz-Sanchez et al., 2001], and finding the LPM in these algorithms usually consists of sequentially traversing a sequence of nodes. Therefore, these algorithms strive to reduce the number of required memory accesses as a means to speed up the lookup process. For instance, in order to achieve that, the Multi-Index Hybrid Trie (MIHT) [Lin et al., 2014] employs space-efficient data structures, such as B^+ trees and Priority Tries [Lim et al., 2010]. Recently, the use of compressed trie data structures has also been proposed [Rétvári et al., 2013, Asai and Ohara, 2015]. Nevertheless, trie/tree-based schemes commonly share the characteristic of being memory-intensive. Another interesting class of algorithms for IP lookup is based on Bloom filters [Dharmapurikar et al., 2006, Lim et al., 2014]. These algorithms are compute-intensive and may require many hash calculations during each lookup/store operation. Hashing is used within the Bloom filters as a means to avoid unnecessary memory accesses to hash tables (where the data is actually stored).

A wide range of hardware architectures has been used to implement IP lookup algorithms, including CPU, FPGA, GPU and many-cores [Yang et al., 2015]. In [Ni et al., 2015], a Parallel Bloom Filter (PBF) was implemented in the Intel Phi coprocessor. PBF was proposed to reduce synchronization overhead and improve cache locality in many-core platforms. In this work, we also implement an algorithm that uses Bloom filters in the Intel Phi specialized for IP lookup and, as such, our approach is different both in the algorithmic and implementation levels. First, PBF uses locks in the Bloom filters to avoid data races in concurrent update/lookup operations and enforces sequential consistency by reordering the responses after processing the requests in parallel. Our approach, in the other hand, is built on top of the ideas of [Dharmapurikar et al., 2006] and includes several optimizations targeting its efficient execution on the Intel Phi for IP lookup. Therefore, we have designed a Bloom filters based LPM algorithm specifically optimized for performing the IP lookup task for both IPv4 and IPv6. As will be presented in the experimental evaluation, these optimizations are crucial to attain high performance.

3. Bloom Filters for IP Lookup

The use of Bloom filters coupled with hash tables for computing IP Lookups has been proposed in [Dharmapurikar et al., 2006]. The standard algorithm has been developed

using 32 pairs of Bloom filters and hash tables for IPv4 lookups, whereas this number would increase to 64 in the case of IPv6. For the sake of simplicity, in the remaining of this section, we describe the algorithm in the context of IPv4, while modification for its efficient execution with IPv6 are presented in the next section.

A Bloom filter is an efficient data structure for membership queries with tunable false positive errors [Bloom, 1970] widely used for web caching, intrusion detection, content based routing and LPM [Dharmapurikar et al., 2006]. In essence, a Bloom filter consists of a bit-vector used to represent a set of values. A Bloom filter is programmed by computing hash functions on each element it stores, and by setting the corresponding indices in the bit-vector. Further, to check if a particular value is in the set, the same hash functions are computed on the input value and bits in the bit-vector structure addressed by the hash values are verified. The value is said to be contained in the set with a given probability only if all bits are set. The standard Bloom filter does not support removal of elements from the set, because it does not control the case in which a bit is set multiple times due to the insertion of different values whose hashes collide. This is addressed with the use of Counting Bloom Filters [Fan et al., 2000], which associates a counter with each entry of the bit-vector to store the number of times a given bit-vector entry was set or unset. We have chosen to implement this approach because it provides a fair comparison with other algorithms that also feature dynamic forwarding tables.

The lookup operations in the standard Bloom filters algorithm are executed within multiple sets of filters and hash tables — one for each possible IP prefix length. As network addresses in IPv4 are 32-bit long, they require the algorithm to employ 32 Bloom filters with their respective 32 hash tables. Each hash table stores their corresponding [prefix, next hop] pairs and any other relevant routing information, such as metric, interface, etc. If a default route exists, it is stored in a separate field in the forwarding table data structure. Let $F = \{(f_1, t_1), (f_2, t_2), \ldots, (f_{32}, t_{32})\}$ be the set of Bloom filters (f_i) and associated hash tables (t_i) that form an IPv4 forwarding table, where (f_1, t_1) corresponds to the data structures that store 1-bit long prefixes, (f_2, t_2) corresponds to the data structures that store 2-bit long prefixes, and so on. In addition, let $len(f_i)$ be the length of the bit-vector of the i-th Bloom filter, where $1 \leq i \leq 32$. The forwarding table construction is as follows. For every network prefix p of length l to be stored, k hash functions are computed, yielding k hash values: $H = \{h_1, h_2, \ldots h_k\}$. The algorithm uses H to set the k bits corresponding to the indices $I = \{h_i \bmod len(f_l) \mid 1 \leq i \leq 32\}$ in the bit-vector of the Bloom filter f_l. It also increments the corresponding counters in the array of counters of f_l.

The lookup process is similar to the insertion. Given an input destination address DA, the algorithm first extracts its segments or prefixes. Let $S_{DA} = \{s_1, s_2, \ldots, s_{32}\}$ be the set of all the segments of a particular address DA, where s_i is the segment corresponding to the first $1 \leq i \leq 32$ bits of DA. For each $s_i \in S_{DA}$, k hash functions are computed, yielding k hash values for each segment: $H = \{(h_1, h_2, \ldots h_k)_1, (h_1, h_2, \ldots h_k)_2, \ldots, (h_1, h_2, \ldots h_k)_{32}\}$. The element $H'_i \in H$ is used to query the Bloom filter $f_i \in F$. The process is as follows: the algorithm checks the k bits in the bit-vector of f_i using the indices $I = \{h_j \bmod len(f_i) \mid h_j \in H'_i, 1 \leq j \leq k \text{ and } 1 \leq i \leq 32\}$. The result of this process is a *match vector* $M = \{m_1, m_2, \ldots, m_{32}\}$ containing the answers of each Bloom filter, i.e., each $m_i \in M$ indicates whether a match occurred or not in f_i. The match vector M is used to query the associated hash tables. The

search begins by sequentially performing queries to the associated hash tables by traversing M backwards, i.e., starting in m_{32}. This is because we are interested in the LPM. If the algorithm finds the next hop (a true match) for a given DA in the pair (f_i, t_i), it is the LPM. As Bloom filters may produce false-positives but never false negatives, when a filter does not match a segment, i.e., $m_i \in M$ indicates a mismatch, the algorithm can safely skip to the next Bloom filter f_{i-1} (if $i \geq 2$) without touching its associated hash table t_i. This process continues until the LPM is found or all pairs (f_i, t_i) are unsuccessfully searched. Please, note that false-positives will only lead to extra hash table searches, and the actual result of the algorithm will be the same regardless of that ratio.

4. Bloom Filters Optimizations and Parallelization

This section describes the optimizations implemented in the standard Bloom filters IP lookup algorithm, as well as its parallelization targeting the Intel Phi. The CPU parallel versions employed similar parallelization strategies, but differ with respect to the instruction-level parallelism that used auto-vectorization. The baseline implementation on which our work is built incorporates the following optimizations: the use of an array of counters to allow FIB updates, asymmetric memory allocation proposed in [Dharmapurikar et al., 2006], and Controlled Prefix Expansion (CPE) [Venkatachary and Varghese, 1998] to reduce the number of required data structures.

4.1. Optimizing the Hash Calculations

Hashing quality is a central performance aspect of the algorithm because it is closely related to the efficiency of Bloom filters and hash tables. In the Bloom filters data structure, it affects both the false positive ratio (FPR) and the amount of memory required. With respect to the associated hash tables, the better the quality of the hash, less collisions are likely to happen and, as a consequence, the lookup process will also be faster.

In order to improve the algorithm performance we have (i) accelerated the hash calculations with the use of instruction-level parallelism or vectorization, as discussed in detail in Section 4.4; (ii) reduced the cost of hashing by combining the output of two hash calculations to generate more hashes; and, (iii) implemented and evaluated the reuse of hash values to minimize the overall hash calculations. The reuse affects both the lookup and update operations. The generation of extra hashes was performed through the use of a well-known technique that consists of using a simple linear combination of the output of two hash functions $h_1(x)$ and $h_2(x)$ to derive additional hash functions in the form $g_i(x) = h_1(x) + i \times h_2(x)$. This technique results in much faster hash calculations and can be effectively applied in the Bloom filters and related data structures, such as hash tables, without affecting the asymptotic false positive probabilities [Kirsch and Mitzenmacher, 2008]. We have also proposed the reuse of one of the hashes calculated to search or store a key in the Bloom filter to address its associated hash table. This avoids the calculation of another hash whenever a hash table is visited.

4.2. A New Dynamic Programming CPE (DPCPE) for Efficient IPv6 Lookup

Another crucial optimization we have implemented for IPv6 is the use of CPE to reduce the number of required sets of Bloom filters and hash tables in the algorithm. This technique consists of expanding every prefix of a shorter length to multiple, equivalent, prefixes of a greater length, so that the number of distinct prefix lengths and, consequently,

filters and hash tables, is reduced. In IPv4, we used CPE to expand prefixes into two groups: $G_1 \in [21–24]$ and $G_2 \in [25–32]$. After the CPE, G_1 has only 24-bit prefixes and G_2 has only 32-bit prefixes, and two sets of Bloom filters and hash tables are allocated to store these prefixes. A Direct Lookup Array (DLA) is allocated to store the next hops of the remaining prefixes, whose lengths are ≤ 20 bits, using the prefixes themselves as the indices. Except for the DLA, the lookup process is identical to the standard Bloom filters (discussed in Section 3). The algorithm sequentially searches the two sets of Bloom filters and hash tables (starting from G_2, since it stores the longest prefixes) and, if the LPM is not found, the next hop stored in the DLA position indexed by the first 20 bits of the input address is returned (it may be the default route). This way we are able to bound the worst-case lookup scenario to two queries (G_1, G_2) and one memory access (DLA), as detailed in [Dharmapurikar et al., 2006]. Note that the trade-off of CPE is faster search on the cost of increased memory footprint, as shown in Table 2.

The previous work on IP Lookup using Bloom filters [Dharmapurikar et al., 2006] has reported that this technique would not be efficient for IPv6. This is because of the "strides" between hierarchical boundaries of IPv6 addresses, which would result in a very high use of memory after expansion. However, we have proposed and implemented an algorithm based on dynamic programming (DPCPE) that groups prefix lengths and performs the expansion with a limited additional memory demand.

The DPCPE algorithm works as follows. Let $L = \{l_1, l_2, \ldots, l_{64}\}$ be the prefix distribution of a IPv6 forwarding table, where l_i is the number of unique prefixes of length i (in bits). Given a desired number of expansion levels n, the algorithm uses dynamic programming to compute the set of lengths to be used in order to minimize the total number of prefixes in the resulting forwarding table. DPCPE always starts by picking the length 64, since it is the largest prefix length for IPv6 and, as such, its inclusion is required for correctness (i.e. every IPv6 prefix can, *theoretically*, be expanded to one or more 64-bit prefixes). Let $S = \{64\}$ represent the initial set of resulting lengths and $C = \{1, 2, \ldots, 63\}$ represent the initial set of candidate lengths. While $|S| < n$, the algorithm iteratively removes an element $l \in C$ and inserts it into S. In each iteration, the length l is selected by mapping a cost function f over all possible sets of lengths and choosing the length associated with the smaller cost. For instance, in the second iteration (assuming $n \geq 2$), f is mapped over the set $Q = \{\{l, 64\} \mid l \in C\}$ and the value l from the set that resulted in the minimum cost is selected. The cost function f takes as input L and a set of expansion levels $Q' \in Q$. It then computes the resulting number of prefixes after expanding L to Q'. The (maximum) number of prefixes, resulting from expanding a prefix of length $l_i \in L$ to $q \in Q'$ (such that, $i < q$), is defined as $2^{q-i} \times l_i$. Note that f does not take into account the problem of prefix capture [Venkatachary and Varghese, 1998], which happens whenever a prefix is expanded to one or more existing prefixes in the database. In this case, the existing longer prefix "captures" the expanded one, which is ignored. Therefore, although DPCPE is not guaranteed to return the optimal solution, it usually returns solutions that work better in practice for the Bloom filters algorithm than directly using the database with no preprocessing, as presented in Section 5.5.

4.3. Thread-Level Parallelism (TLP)

Due to its regular data structures, the Bloom filters algorithm exposes multiple opportunities for parallelism. For instance, in [Dharmapurikar et al., 2006] it was suggested a parallel search over the two sets of Bloom filters/hash tables and the DLA (associated with

the different prefix lengths) for a given input address, which is mentioned to be appropriate for hardware implementations. In this strategy, a final pass is performed to verify if a match occur in any of these data structures and to select the next hop. The same approach could be used for a software-based parallelization by dispatching a thread to search each data structure. However, IPv4 prefix databases have the well-known characteristic that prefixes are not uniformly distributed in the range of valid prefix lengths and, as a consequence, it is more likely that a match occurs to prefixes within lengths that concentrate most of the addresses, i.e., the set of Bloom filter and hash table that stores 24-bit prefixes. Therefore, computing all Bloom filters in parallel may not be efficient because, most of the times, the results from the data structures associated with prefix lengths smaller or greater than 24 bits will not be used. Instead, it is more compute efficient to sequentially query the Bloom filters and the DLA. The other option for TLP, which is used in our approach, is to perform the parallel lookup computation for multiple addresses by assigning one or multiple addresses to each computing thread available. In this way, we can carry out the processing of each address using the most compute efficient algorithm, while we are still able to improve the system throughput by computing the lookup for multiple addresses concurrently. This is possible because the processing of addresses are independent and, as such, there is no synchronization across the computation performed for different addresses. The implementation of the parallelization at this level employed the Open Multi-Processing API (OpenMP) [OpenMP, 2016], which was used to annotate the main algorithm loop that iterates over the input addresses to find their next hops. The specific OpenMP settings used, which led to the better results, were the *dynamic* scheduler and *chunk size* of one.

4.4. Instruction-Level Parallelism (ILP)

The use of ILP is important to take full advantage of the Intel Phi, which is equipped with a 512-bit vector processing unit (see Section 2.1). We used its SIMD instructions to efficiently compute the hash values for multiple input addresses at the same time. The ILP optimization focused on the hashing calculations because it is the most compute intensive stage of the algorithm. The original work [Dharmapurikar et al., 2003] and previous implementations of algorithms employing Bloom filters to the LPM problem [Lim et al., 2014, Ni et al., 2015] do not discuss their decisions and reasons on the hash functions used. Thus, we have decided to implement, vectorize, and evaluate three hash functions: MurmurHash3 [Appleby, 2011] (Murmur), Knuth's multiplicative method [Knuth, 1998] (Knuth), and a hash function named to here as H2 [Mueller, 2006]. Murmur is widely used in the context of Bloom filters, but its original version takes as input a variable-length string. In order to improve its efficiency, we have derived versions of it specialized to work on 32-bit (for IPv4) and 64-bit (for IPv6) integer keys. Knuth is a simple hash function of the form: $h(x) = x \times c \bmod 2^l$, where c should be a multiplier in the order of the hash size 2^l that has no common factors with it. H2 takes as input a key and mixes its bits using a series of bitwise operations, as shown in Algorithm 1. Although simple, the H2 hash function has been shown to be effective in practice [Mueller, 2006].

The hash implementations employed the low-level Intel® Intrinsics API [Intel, 2015] to perform a manual vectorization of all the hash functions. We have also evaluated the use of automatic vectorization available with the Intel® C Compiler, but the manually generated code has proved to be more efficient.

Algorithm 1: Definition of the H2 hash function.

Input : x {A 32-bit unsigned integer key}.
Output: The computed hash value.

1 $x := ((x \gg 16 \oplus x) \times$ 0x45d9f3b
2 $x := ((x \gg 16 \oplus x) \times$ 0x45d9f3b
3 $x := ((x \gg 16 \oplus x)$
4 **return** x

5. Performance Evaluation

This section evaluates the performance of our optimized Bloom filters algorithm both for IPv4 and IPv6. Our evaluations include the analysis of parameter impacts to the performance of the algorithm — namely, the hash functions, FPR and CPE.

5.1. Experimental Setup and Databases

The experiments were performed in a machine equipped with a dual socket Intel® Xeon E5-2640v3 CPU (16 CPU cores with Hyper-Threading), 64 GB of main memory, an Intel® Xeon Phi™ 7120P coprocessor (described in Section 2.1), and CentOS 7 operating system. The source codes were developed using C11 and compiled with the Intel® C Compiler 16.0.3 for both the CPU and the Intel Phi using the -O3 optimization flag.

Database	Location	Originally			After CPE		
		≤ 20	$21-24$	$25-32$	$=20$	$=24$	$=32$
AS65000	-	104,283	516,699	1625	1,048,576	971,555	113,397
DE-CIX	Frankfurt	102,984	535,074	9287	1,048,576	1,007,513	209,488
LINX	London	100,331	519,503	354	1,048,576	982,940	19,863
MSK-IX	Moscow	102,555	528,728	9529	1,048,576	1,004,073	203,360
NYIIX	New York	102,085	528,455	3637	1,048,576	1,000,128	151,391
PTTMetro-SP	Sao Paulo	103,733	544,703	4095	1,048,576	1,024,899	147,201

Table 1. Characteristics of the IPv4 prefix datasets used.

We used 6 real prefix databases for IPv4, whose characteristics are summarized in Table 1. The first database, AS65000, was obtained from [BGP Potaroo, 2016], whereas the remaining databases were downloaded from [RIPE NCC, 2016]. Table 1 presents the amount of addresses in each database and the total number of prefixes before and after performing the CPE to group them into sets of 24-bit and 32-bit long prefixes.

For IPv6, we use the AS65000-V6 database collected from [BGP Potaroo, 2016]. IPv6 is still not widely used and this database has only 31,645 prefixes, which are distributed in 34 distinct prefix lengths. Table 2 shows the effects of applying our DPCPE on the AS65000-V6 database. Note that, for this database, we can not use less than 3 expansion levels, since the amount of memory required becomes prohibitively large. Also, although the algorithm ignores the prefix capture problem (Section 4.2) when computing the levels, its estimates are very close to the actual results.

5.2. The Effect of the Hash Function and False Positive Ratio

The false positive ratio (FPR) is a key aspect for the effectiveness of a Bloom filter because it affects the memory requirements and the number of hash calculations per lookup.

Resulting database	Desired number of distinct lengths	Expansion levels suggested by the algorithm	Estimated number of prefixes	Actual number of prefixes
CPE8	8	$\{24, 29, 33, 38, 40, 44, 48, 64\}$	61,208	60,261
CPE7	7	$\{29, 33, 38, 40, 44, 48, 64\}$	77,700	76,638
CPE6	6	$\{29, 33, 38, 44, 48, 64\}$	104,625	103,094
CPE5	5	$\{33, 38, 44, 48, 64\}$	385,200	380,217
CPE4	4	$\{33, 44, 48, 64\}$	1,185,300	1,166,135
CPE3	3	$\{44, 48, 64\}$	650,417,961	—

Table 2. Results of performing CPE in the AS65000-V6 database. There are a total of 34 distinct lengths and 31,645 unique IPv6 prefixes in AS65000-V6.

We highlight that the FPR does not affect the results of the algorithm, but only the number of times that a value is informed to be in the associated hash table by a Bloom filter without being. When this occurs, the algorithm will unsuccessfully search in the hash table. Probing a hash table consists in traversing a linked list, which may become expensive as the FPR increases. On the other hand, a very low FPR requires a larger number of hash calculations and a high memory utilization. The FPR is determined by three main parameters: the number n of entries stored in the filter, the size m of the filter, and the number k of hash functions used to store/query the filters. Given a n value, the values m and k can be derived as detailed in [Bloom, 1970] to attain a desired FPR.

The trade-off between increasing the hash calculations and the application memory footprint in order to avoid the extra cost of a false positive is complex. Therefore, we have evaluated it experimentally by measuring the performance in million lookups per second (Mlps) of various FPRs and hash function configurations. Hash functions are used in the Bloom filters algorithms for querying the Bloom filters and to address the hash tables associated to each filter. As such, we are able to use combinations of hash functions to compute the multiple hashes within a Bloom filter or the single hash that address a particular hash table. The hash functions used were presented in Section 4, and we employ the AS65000 prefix database and an input address dataset with 2^{26} random addresses.

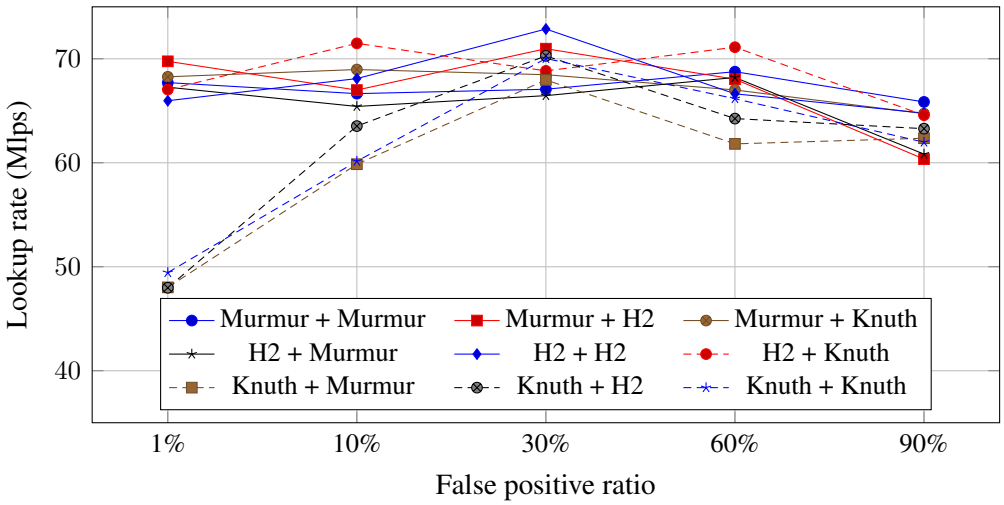

Figure 1. Performance of multiple hash functions and FPRs using 244 threads in the Intel Phi. The "Murmur + H2" entry means Murmur was used within the Bloom filters and H2 was used to address the hash tables.

The results presented in Figure 1 shows that the performance of the application is strongly affected by the FPR and hash functions. As presented, the use of Knuth resulted in a lower average performance as compared to other methods. The reason for the observed results is that this hash function preserves divisibility, e.g., if integer keys are all divisible by 2 or by 4, their hash values will also be. This is a problem in Bloom filters or hash tables in general, where many values will address the same bits in the bit-vector and only a half or a quarter of the buckets will end up being used, respectively. On the other hand, Murmur and H2 are more sophisticated functions that provide better statistical distributions, hence all the configurations using any combination of them attained similar lookup rates. However, the best average performance was reached with 30% of FPR, where the results are less scattered for all hash functions. Furthermore, the best performance was attained when H2 was used in both stages of the algorithm. This occurs, in part, because we are able to reuse the hash calculated to probe the Bloom filter to address the hash table and, as a consequence, hash calculations are saved. Therefore, we use the configuration of 30% of FPR and H2 for both stages in the remaining experiments.

5.3. The Impact of the Input Addresses (Querying) Characteristics to Performance

In order to investigate the effects of the input addresses on the performance, we performed the lookups using pseudo-random generated datasets containing 2^{26} IP addresses with different matching ratios and the AS65000 prefix database. We call *matching ratio* the relation between the number of addresses that matches at least one prefix in the database and the total number of addresses, thus a matching ratio of 80% implies that 20% of the input addresses do not match any prefix in the database and, as such, end up being forwarded to the default route. This evaluation intended to vary the characteristics of the input data and evaluate the algorithms under different configurations.

We ensure that a given address has the same probability to match any prefix stored in the database, and we also filter out all the IETF/IANA reserved IP addresses. Please note that a workload for forwarding could include other characteristics, such as the arrival of packets in bursts. We use a randomized input because it may be considered the worst-case scenario and it is the most commonly method used in previous works. The lookup rates obtained for the *bloomfwd* algorithm on both the CPU and on the Intel Phi are shown in Figure 2.

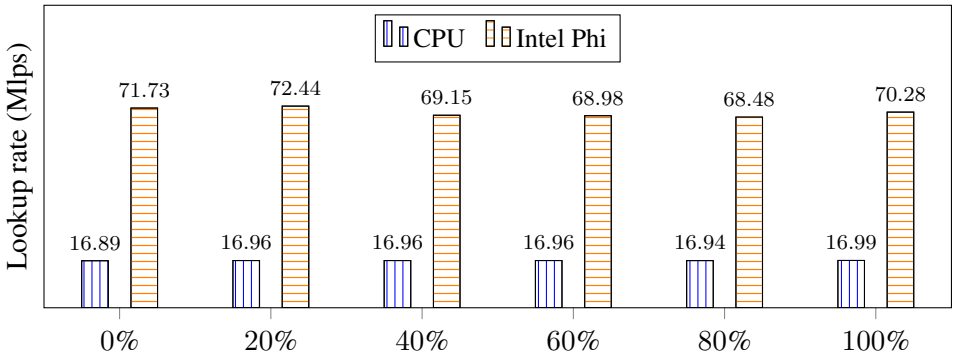

Figure 2. Performance of as matching ratio is varied. The entry 80% means that this percentage of the addresses match with equal probability a prefix in the forwarding table, while 20% of them end up in the default route.

As presented in Figure 2, the input querying dataset has little impact in the overall performance of the application. The reason for that is that the case of an address matching

some prefix in the database *is not necessarily faster* than the case where the address end up in the default route, and vice-versa. For example, consider an address that does not match any prefix in the forwarding table. If no false positives occur, i.e., the two Bloom filters correctly answer not to look in their associated hash tables, the search quickly finishes with one additional memory access to the DLA. However, if for another prefix a false positive occurs in the first, but not in the second Bloom filter, or if there are plenty of values stored in the searched hash table buckets, then this case of matching will likely be *slower* than the former. As such, the speed and good statistical distribution of the hash function to control the FPR and also minimize the number of collisions in the hash tables is an important aspect to limit the variations in the performance as a result of datasets characteristics.

5.4. Scalability and Performance Evaluation: Bloom filters vs. MIHT

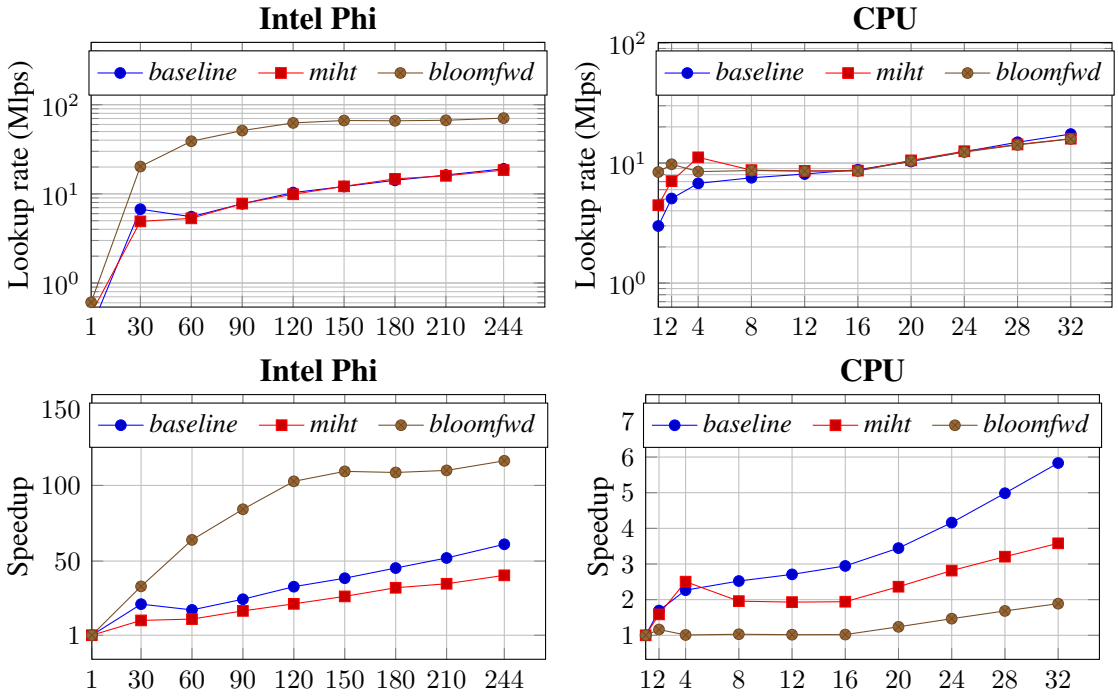

Figure 3. Lookup rate and scalability of the IPv4 algorithms on Intel Phi and CPU using the AS65000 prefix database and the 80% address dataset.

In this section, we evaluate the performance gains of our optimized Bloom filters algorithm for IPv4, which we refer here to as *bloomfwd*, as the number of computing cores used is increased on the Intel Phi and on the CPU. We compare *bloomfwd* to a *baseline* implementation of the Bloom filters algorithm, introduced in Section 4.1, and to the Multi-Index Hybrid Trie (MIHT). The difference between *baseline* and *bloomfwd* is the hash function, i.e., *baseline* uses the standard C *rand* function with no vectorization [Dharmapurikar et al., 2006, Lin et al., 2014]. The MIHT is a state-of-the-art software-based algorithm that has outperformed several other popular IP lookup algorithms [Lim et al., 2014]. Our implementation of MIHT for IPv4 (*miht*) was optimally tuned according to the parameters suggested in the original work for the (16,16)-MIHT. The speedups for the IPv6 dataset are similar and were omitted because of space limitations.

The evaluations used the AS65000 database and an address dataset with 80% of matching ratio. The lookup rates (in log scale) and speedups for both algorithms and processors are presented in Figure 3. As shown, the performance of *miht* ($\approx$ 0.46 Mlps) is better than *baseline* ($\approx$ 0.31 Mlps) for the sequential execution on the Intel Phi. However, as the number of computing threads used increases, the performance gap reduces quickly due to the better scalability of the Bloom filters approach. For instance, the maximum speedup of *baseline* as compared to its sequential counterpart is about 61$\times$, whereas *miht* attains a speedup of up to 40$\times$ when compared to its sequential version. The *bloomfwd*, on the other hand, is the fastest algorithm on a single core and is still able to attain better scalability on the Intel Phi (116$\times$). Also, it is at least 3.7$\times$ faster than the other algorithms. Finally, the difference between the lookup rates of *bloomfwd* and *baseline* highlights the importance of the use of vectorization and the hash function choice to performance.

The analysis of the CPU results show that all algorithms attained very similar lookup rates at scale in a multithread setup, though they attained different speedups. We attribute the similar performance of the algorithms on the CPU to the fact that the memory bandwidth of this processor is much smaller than that of Intel Phi, which limits the scalability of the solutions that are memory-intensive. However, note that in the sequential execution, *bloomfwd* was the fastest, followed by *miht* and then *baseline*. This result further reinforces the importance of the hash function choice to performance and the effectiveness of the H2 hash function.

5.5. Performance of the Bloom filters IP Lookup on IPv4 and IPv6

This section evaluates the best version of the Bloom filter algorithms on IPv4 and IPv6 prefix datasets in the Intel Phi. First, we present the performance for the 5 remaining IPv4 prefix databases presented in Table 1 using the querying input dataset with 80% of matching ratio. The results, presented in Figure 4, show that the performance gains of our *bloomfwd* as compared to the *miht* is about 4$\times$ regardless of the dataset used.

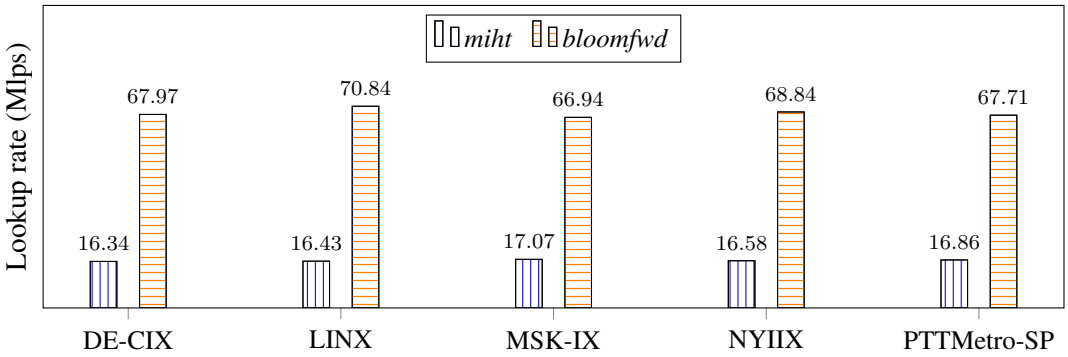

Figure 4. Performance of *bloomfwd* and *miht* for 5 IPv4 prefix datasets.

We further evaluate the performance of the Bloom filters algorithm for IPv6, and the impact of using our DPCPE algorithm. Therefore, we compare the lookup rates of our implementation (*bloomfwd-v6*) with the corresponding version of MIHT for IPv6 (*miht-v6*). The *miht-v6* is equivalent to the (32,32)-MIHT [Lim et al., 2014].

Figure 5 shows the results for the AS65000-V6 with and without the use of DPCPE for multiple expansion levels and 2^{26} random input addresses. As presented, the performance of the *bloomfwd-v6* algorithm is greatly improved by the use of our DPCPE, and the expansion with 7 levels is about 5.9$\times$ faster than the performance without CPE.

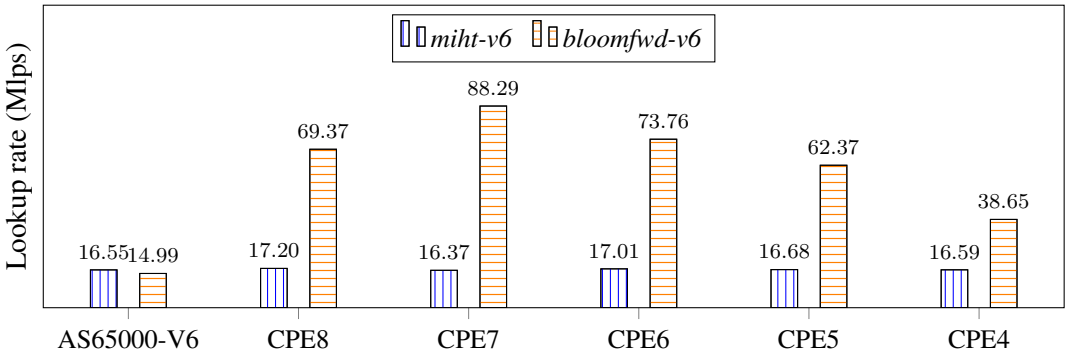

Figure 5. Performance of *bloomfwd* and *miht* for the AS65000-V6 prefix dataset.

This version is also 5.3× faster than the *mith-v6* algorithm. If the expansion value is reduced, however, the performance of the algorithm degrades because of the higher memory demands and search cost. The performance of *miht-v6* is similar in all cases because, in the MIHT, routes are grouped by keys in Priority Tries (PTs), rather than prefix lengths (as in the Bloom filters approach). In other words, the number of distinct prefix lengths in the forwarding table does not directly affect the performance of MIHT.

6. Conclusions and Future Directions

In this work, we have implemented and evaluated the performance of state-of-the-art algorithms for IP lookup (MIHT and Bloom filters approach) in multi-/many-core systems, which is a core operation for efficient packet forwarding in routers. The MIHT is known to be a very efficient sequential algorithm [Lin et al., 2014]. However, it is also very irregular, which typically leads to reduced opportunities for optimized execution on parallel systems. The baseline Bloom filters algorithm, on the other hand, is a more compute intensive and regular algorithm with a less efficient sequential version. Nevertheless, it offers more opportunities for optimizations, for instance, due to SIMD instructions, and it is more scalable with respect to the number of computing cores used. As presented in the results section, our optimized Bloom filters algorithm was able to compute the next hop for 2^{26} IPv4 and IPv6 input addresses, respectively, in a rate of 70.84 Mlps and 88.29 Mlps. Also, the speedup of 116.2× obtained on Intel Phi motivates the use of the proposed techniques, along with many-core technologies, in the construction of efficient software routers. The CPU versions of the algorithms attained good performance but its low memory bandwidth limits the scalability of the algorithms. The good scalability of the Bloom filters approach shows that it may be a better option for devices with a large number of computing cores.

As a future work, we intend to improve the performance of our Bloom filters algorithm by using the CPU and the Intel Phi cooperatively to perform the lookups. We also plan to integrate our optimized algorithm in a complete software router, such as Click or Open vSwitch (OvS).

References

Appleby, A. (2011). MurmurHash3 Hash Function. `https://code.google.com/p/smhasher/wiki/MurmurHash3`.

Asai, H. and Ohara, Y. (2015). Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 57–70. ACM.

BGP Potaroo (2016). BGP Potaroo. http://bgp.potaroo.net/.

Bloom, B. H. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426.

Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. E. (2003). Longest prefix matching using bloom filters. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 201–212. ACM.

Dharmapurikar, S., Krishnamurthy, P., and Taylor, D. E. (2006). Longest Prefix Matching Using Bloom Filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409.

Fan, L., Cao, P., Almeida, J., and Broder, A. Z. (2000). Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293.

Intel (2015). Intel Intrinsics Guide. urlhttps://software.intel.com/sites/landingpage/IntrinsicsGuide/.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi coprocessor high-performance programming*. Elsevier Waltham (Mass.), Amsterdam, Boston (Mass.).

Kirsch, A. and Mitzenmacher, M. (2008). Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures & Algorithms*, 33(2):187–218.

Knuth, D. E. (1998). *The Art of Computer Programming: Sorting and Searching*, volume 3. Pearson Education.

Lim, H., Lim, K., Lee, N., and Park, K.-H. (2014). On Adding Bloom Filters to Longest Prefix Matching Algorithms. *IEEE Transactions on Computers*, 63(2):411–423.

Lim, H., Yim, C., and Swartzlander Jr, E. E. (2010). Priority Tries for IP Address Lookup. *IEEE Transactions on Computers*, 59(6):784–794.

Lin, C.-H., Hsu, C.-Y., and Hsieh, S.-Y. (2014). A Multi-Index Hybrid Trie for Lookup and Updates. *IEEE Transactions on Parallel and Distributed Systems*, 25(10):2486–2498.

Mueller, T. (2006). H2 Database Engine. http://h2database.com.

Ni, S., Guo, R., Liao, X., and Jin, H. (2015). Parallel Bloom Filter on Xeon Phi Many-Core Processors. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 388–405. Springer.

OpenMP (2016). OpenMP API for Parallel Programming, Version 4.0. http://openmp.org/wp/.

Rétvári, G., Tapolcai, J., Kőrösi, A., Majdán, A., and Heszberger, Z. (2013). Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 111–122. ACM.

RIPE NCC (2016). RIPE Network Coordination Centre. http://data.ris.ripe.net/.

Ruiz-Sanchez, M., Biersack, E. W., Dabbous, W., et al. (2001). Survey and Taxonomy of IP Address Lookup Algorithms. *IEEE Network*, 15(2):8–23.

Venkatachary, S. and Varghese, G. (1998). Faster IP Lookups using Controlled Prefix Expansion. *PERFORMANCE EVALUATION REVIEW*, 26:1–10.

Yang, T., Xie, G., Li, Y., Fu, Q., Liu, A. X., Li, Q., and Mathy, L. (2015). Guarantee IP Lookup Performance with FIB Explosion. *ACM SIGCOMM Computer Communication Review*, 44(4):39–50.