USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

PlanB: Efficient Software IPv6 Lookup with Linearized B^+-Tree

Zhihao Zhang, *Alibaba Cloud, NICE Lab, XMU, and Tsinghua University;* Lanzheng Liu, Chen Chen, and Huiba Li, *Alibaba Cloud;* Jiwu Shu, *Tsinghua University;* Windsor Hsu, *Alibaba Cloud;* Yiming Zhang, *NICE Lab, SJTU and NICE Lab, XMU*

https://www.usenix.org/conference/nsdi26/presentation/zhang-zhihao

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

PlanB: Efficient Software IPv6 Lookup with Linearized B^+-Tree

Zhihao Zhang[1,3,4], Lanzheng Liu[1], Chen Chen[1], Huiba Li[1]*, Jiwu Shu[4], Windsor Hsu[1], Yiming Zhang[2,3]
[1]*Alibaba Cloud* [2]*NICE Lab, SJTU* [3]*NICE Lab, XMU* [4]*Tsinghua University*

Abstract

IP lookup via Longest Prefix Match (LPM) is critical for packet forwarding. Unfortunately, conventional lookup algorithms are inefficient for IPv6 Forwarding Information Bases (FIBs), which are characterized by a set of long prefixes with diverse lengths. We observe that LPM inherently represents a two-dimensional (2D) search problem over both prefix values and prefix lengths, but existing algorithms mostly treat LPM as two separate levels of one-dimensional (1D) searches, causing poor lookup performance and high memory overhead.

This paper presents PlanB, a novel scheme for high-speed IPv6 lookup. We transform the 2D LPM into an equivalent 1D search problem over elementary intervals, unifying the search across prefix value and lengths. We then adapt the flat-array B-tree structure to the needs of LPM to propose *linearized B^+-tree*, based on which we introduce an efficient search algorithm tailored to the properties of the transformed space. To maximize performance, we integrate PlanB with vectorization, batching, branch-free logic, and loop unrolling to fully exploit CPU parallelism. Extensive evaluation shows that PlanB achieves single-core performance of 390 Million Lookups Per Sec (MLPS) with real-world IPv6 FIBs on AMD processor, and scales to full-12-core performance of 3.4 Billion Lookups Per Sec (BLPS). This is 1.6×∼14× higher than state-of-the-art software-based schemes (PopTrie, CP-Trie, Neurotrie and HBS).

1 Introduction

The Internet's transition to IPv6 is accelerating, driven by the exhaustion of IPv4 addresses and the increasing connectivity demands of IoT (Internet of Things) and mobile networking. The transition fuels unprecedented growth in global IPv6 traffic and routing table sizes, with the latter projected to exceed 300,000 prefixes by 2030 [1, 2, 14]. This trend is evident across the globe. Google reports that over 45% of its users now access services via IPv6 [45], and adoption rates in countries like the United States, Germany, and India have exceeded 57% [6]. The transition is particularly pronounced in countries facing IPv4 address shortages. For instance, China now has over 822 million IPv6 users [12], with IPv6 constituting over 31% of its national network traffic [21]. Moreover, it coincides with a broader shift in networking towards Network Function Virtualization (NFV) [27, 34, 38, 48, 51]. The reliance of these architectures on software packet processing makes the efficiency of IP lookups especially critical. The rapid adoption of IPv6, coupled with longer prefixes, larger routing tables, and increasing traffic volumes, places significant pressure on existing lookup mechanisms.

Hardware-based IP lookup solutions, leveraging components like TCAMs [7, 9, 36, 40, 52], FPGAs [30, 46, 47], and ASICs [22, 23], offer high throughput but face significant limitations. These specialized components are prohibitively expensive and consume significant power, and their limited capacity is insufficient for today's rapidly growing IPv6 routing tables [9, 26, 36, 59], making them impractical for large-scale or cost-sensitive deployments [4, 5, 33]. Furthermore, specialized hardware lacks flexibility required by modern paradigms like NFV [18, 38, 51], which depend on software for rapid service deployment and management. As a result, there is a trend towards implementing routers in software on commodity servers [15, 20, 41]. Software-based IPv6 lookup solutions are more cost-effective and flexible for cloud deployments.

However, software-based solutions face a long-standing challenge: IP lookup remains a critical performance bottleneck. The problem is particularly severe for IPv6, where the characteristics of Forwarding Information Bases (FIBs) significantly degrade router performance. Specifically, IPv6 FIBs contain a wide range of prefix lengths [11, 35, 39, 43, 58], with typical backbone entries spanning from /32 to /64, increasing both computational overhead and memory footprint. Compared to IPv4 [25, 31], this results in a 3×–10× higher lookup cost for state-of-the-art algorithms and a 10×–50× increase in memory footprint.

Existing software-based IP lookup schemes, which strug-

*Huiba Li (huiba.lhb@alibaba-inc.com) is the corresponding author.

gle to meet the dual demands of high speed and low memory footprint for IPv6 [23, 31, 60], fall into two main categories. First, trie-based schemes, such as SAIL [56, 57] and Poptrie [3], construct a multi-bit trie to traverse the prefix space. These methods use fixed strides (e.g., 6 bits per level) or fixed partitioning schemes to reduce memory accesses. Second, hash-based schemes, such as HBS [25] and HHR [31], perform a binary search (BS) over the set of unique prefix lengths, using hash tables at each step to check for a prefix match.

IP lookup determines the forwarding path by selecting the longest prefix match (LPM) for a given destination address. The performance of existing lookup schemes is limited by a key problem: they treat LPM, an inherently two-dimensional (2D) search problem across prefix values and lengths, as two separate one-dimensional (1D) searches. This separation is inefficient for large IPv6 FIBs. The problem is further compounded by memory hierarchy constraints. To achieve high throughput, the lookup structure must fit into the fast but capacity-limited SRAM (i.e., CPU caches) [3, 57, 58], rather than being stored in the larger but slower DRAM. As a result, efficient IPv6 lookup schemes need to be cache-friendly.

In this paper, we present PlanB, a novel IPv6 lookup scheme designed to achieve high lookup performance on commodity hardware. PlanB is distinguished by its integration of a 2D-to-1D dimensional reduction with a pointer-less, cache-aligned linearized B^+-tree layout. This combination specifically enables the data-parallel vectorization that standard trie-based structures cannot support. PlanB addresses the inefficiencies of existing methods in three aspects.

First, PlanB transforms the 2D search problem of LPM into a 1D search problem. PlanB reframes this by representing each prefix/prefix_length entry as a range [start_address, end_address]. PlanB partitions the entire address space into a set of non-overlapping, elementary intervals. This transformation elegantly converts the complex 2D problem into a simple 1D search on the sorted start addresses of these intervals.

Second, to efficiently solve the resulting 1D search problem, PlanB proposes linearized B^+-tree, which adapts the flat-array B-tree structure [10, 28, 53, 54] to the specific needs of LPM and maps the entire B^+-tree into a single, contiguous array thus being inherently pointer-less. Based on linearized B^+-tree, PlanB's search algorithm navigates the tree by computing parent-child relationships using simple arithmetic on array indices, and employs a lower-bound binary search [1] at each node to find the appropriate child to traverse next. The linearized B^+-tree eliminates pointer chasing, increases data density, and aligns nodes with cache lines.

Third, PlanB implements the search algorithm with a set of optimizations. It leverages Single Instruction Multiple Data (SIMD) instructions (e.g., AVX-512) and batching to enable simultaneous comparisons of the target address against multiple keys within a B^+-tree node. The search is further designed to be branch-free: by replacing conditional statements with bitwise operations and masks, PlanB eliminates costly branch misprediction penalties and maintains a more efficient CPU pipeline. In addition, compile-time loop unrolling reduces loop-control overhead, further accelerating the search.

By combining these techniques, PlanB delivers high-speed IPv6 lookups on commodity hardware. Our evaluation using real-world IPv6 FIBs on a 12-core AMD processor shows that PlanB achieves 390 Million Lookups Per Sec (MLPS) on a single core. The throughput scales linearly with the core count, reaching 3.4 Billion Lookups Per Sec (BLPS) across all cores, significantly outperforming existing state-of-the-art software solutions. We further test PlanB with synthetic FIBs containing up to one million prefixes, showing that PlanB always achieves high performance. Crucially, PlanB makes no distribution assumptions, and its effectiveness on both real-world and synthetic datasets confirms its robustness. Furthermore, PlanB demonstrates near constant performance under fully random traffic. This resilience stems from a robust adaptability to dynamic workloads that inherently lack exploitable temporal locality. Because search latency remains strictly bounded and independent of traffic predictability, the system sustains optimal throughput even during severe traffic bursts. For various IPv6 FIBs, PlanB is 1.6∼14 times faster than state-of-the-art software-based schemes, PopTrie, CP-Trie, Neurotrie and HBS. Additionally, it reduces memory overhead by 56.4%∼92.5%.

2 Background and Motivation

2.1 IPv6 Prefix Length and Distribution

For every packet it forwards, a router performs an IP lookup to determine the correct next hop. This operation queries a FIB, which contains a set of (prefix, next_hop) rules. Given a packet's destination address, the router must find the rule with the longest matching prefix [36, 57]. This matching rule dictates where the packet should be sent next. While the LPM principle is the same for both IPv4 and IPv6, the shift to IPv6 introduces significant challenges that fundamentally alter the performance landscape of lookup algorithms.

To understand the challenges of IPv6 routing, we collect seven large real-world backbone network datasets from RIPE [1] and seven backbone datasets from RouteViews [2] to analyze prefix length and distributions. The RIPE datasets are obtained from the RRC00 collector at 8:00 AM, covering the first day of August for each year from 2019 to 2025. The RouteViews datasets are collected from multiple geographic regions, including Africa, Asia, North and South America, Australia, and Europe. All datasets are processed using the Zebra FIB converter [2] and bgpdump [3] to remove duplicate rules, ensuring they can be directly utilized by our evaluation

[1] The lower-bound binary search finds the first element in a sorted array that is not less than the given target value.

[2] https://github.com/rfc1036/zebra-dump-parser

[3] https://github.com/RIPE-NCC/bgpdump

Table 1: The details of collected FIBs from RIPE and RouteViews.

Name	Time	Physical Location	IP Prefix Distribution (Top 5)	# of Prefixes
rrc00-19	20190801	Amsterdam, NL	48(46.52%),32(17.13%),44(6.13%),40(5.40%),36(3.97%)	76342
rrc00-20	20200801	/	48(47.87%),32(15.52%),44(6.14%),40(5.31%),36(4.15%)	95504
rrc00-21	20210801	/	48(41.82%),32(14.59%),44(8.73%),40(6.66%),64(4.81%)	143823
rrc00-22	20220801	/	48(47.28%),32(13.31%),44(8.32%),40(7.73%),36(3.76%)	168572
rrc00-23	20230801	/	48(46.39%),32(11.83%),44(9.09%),40(7.89%),36(3.65%)	199801
rrc00-24	20240801	/	48(45.27%),32(11.19%),44(9.40%),40(9.01%),36(3.64%)	226222
rrc00-25	20250801	/	48(44.55%),32(11.00%),40(10.11%),44(9.69%),36(3.89%)	235466
rv1	20250801	Johannesburg, SA	48(44.78%),32(11.07%),40(10.00%),44(9.47%),36(3.76%)	235038
rv2	20250801	Singapore, SG	48(44.93%),32(10.92%),40(9.85%),44(9.56%),36(3.71%)	238498
rv3	20250801	Tokyo, JP	48(46.03%),32(11.42%),40(10.30%),44(9.19%),36(3.83%)	226893
rv4	20250801	New York, USA	48(44.88%),32(11.30%),40(10.17%),44(9.69%),36(3.82%)	229417
rv5	20250801	Rio de Janeiro, BR	48(45.37%),32(11.26%),40(9.36%),44(9.24%),36(3.86%)	227833
rv6	20250801	Sydney, AU	48(44.92%),32(11.21%),40(10.15%),44(9.53%),36(3.80%)	230672
rv7	20250801	Frankfurt, DE	48(44.56%),32(11.02%),40(10.02%),44(9.49%),36(3.82%)	234916

framework. A detailed summary of these datasets is provided in Table 1. Our analysis of these datasets reveals several key characteristics of IPv6 FIBs that motivate the need for a new lookup algorithm.

Rapid FIB Growth and Increasing Granularity. The IPv6 routing table is not only growing but is doing so at a dramatic pace as shown in Table 1. The number of prefixes in the RIPE dataset increases from 76,342 in 2019 to a projected 236,466 in 2025—a more than threefold increase in just six years. This rapid scaling demands a lookup algorithm with low memory overhead and high throughput. More importantly, the structure of the FIB is evolving. While the proportion of /48 prefixes remains high, we observe a significant trend towards de-aggregation. From 2019 to 2025, the share of aggregated /32 prefixes drops from 17.13% to 11.00%. Concurrently, the shares of more specific, longer prefixes rise; for instance, /40 prefixes nearly double their representation from 5.4% to 10.11%. This shift towards finer granularity deepens the lookup problem, as algorithms must efficiently handle a growing number of longer prefixes.

Highly Skewed and Concentrated Distribution. A defining feature of IPv6 FIBs is their highly skewed prefix-length distribution. IPv6 routing tables are dominated by a few specific prefix lengths. Across all 14 datasets, /48 is the most common prefix length, consistently accounting for 41-48% of all entries. The prevalence of these long prefixes makes traditional bit-wise trie traversal inefficient, as it requires navigating deep data structures. The 2025 RouteViews data shows that this skewed distribution is remarkably consistent across the globe. From Johannesburg to Tokyo, the total prefix count varies by less than 6%, and the proportions of dominant prefix lengths are nearly identical. For example, the share of /48 prefixes only varies between 44.5% and 46.0%. Moreover, Our additional observation is the concentration of prefixes. Across all datasets, just five prefix lengths, /32, /36, /40, /44, and /48, consistently comprise approximately 80% of the entire FIB. For example, in the rrc00-25 and rv1 datasets, these five lengths account for 79.2% and 79.1% of prefixes, respectively.

The combination of fast table growth, skewed prefix lengths, and global uniformity presents key requirements: (i) Algorithms should be designed around the reality that nearly half of lookups are against /48s, (ii) Lookup performance should be insensitive to the table expanding by hundreds of thousands of entries within a few years, and (iii) Since distributions are stable both temporally and geographically, an adaptive algorithm tuned to the observed mix of longer prefixes will remain effective globally and into the near future.

2.2 Software-based Lookup Methods

Software-based IP lookup schemes can be mainly divided into trie-based [3, 56, 57] and hash-based [24, 25, 42] methods.

Trie-based methods construct a tree where the path from the root represents an IP prefix. While effective for IPv4, their performance on IPv6 is often hampered by the increased address length. SAIL [56, 57] separates the lookup process into finding the prefix length and the next-hop, and it splits prefixes by length (e.g., ≤ 24 vs. > 24). By expanding prefixes and using large bitmap arrays, SAIL achieves a constant, low number of memory accesses for most IPv4 traffic. Its primary drawback is a massive memory footprint. For IPv6 which prefix lengths are typically less than 64 bits in backbone rounters, the tatol memory size of all bitmaps in SAIL is $\sum_{i=0}^{64} 2^i = 4EB$. While it works for the shorter prefixes of IPv4, its memory requirement explodes exponentially with prefix length, rendering it unusable for the larger address space of IPv6. The memory footprint of SAIL exceeds the typical CPU cache size, requiring relatively slow DRAM access in case of cache misses. Moreover, the performance of SAIL relies on the destination IP address locality of the traffic pattern

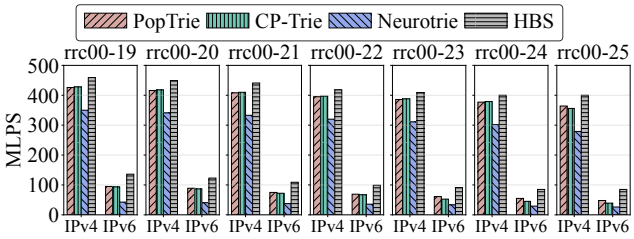

Figure 1: Lookup performance of four software schemes.

PopTrie [3] is a state-of-the-art trie-based algorithm that builds a compressed 64-ary trie. It achieves a small memory footprint and high lookup speed for IPv4 by using bitmaps and the popcnt CPU instruction to quickly identify child nodes. However, its fixed 6-bit stride is a critical limitation for IPv6. A 48-bit prefix requires traversing up to 8 levels in the worst case, leading to numerous memory accesses that diminish cache efficiency and degrade performance. CP-Trie [23] extends PopTrie by incorporating a cumulative popcount, which allows for longer fixed strides while still using a single popcnt instruction per step. This reduces the trie depth compared to PopTrie, resulting in fewer memory accesses. Nonetheless, because its stride is still fixed, it cannot adapt its structure to the specific prefix distribution of a given FIB, making it suboptimal for the diverse IPv6 landscape. Neurotrie [60] introduces an adaptive trie structure that supports arbitrary strides at each node. It uses deep reinforcement learning (DRL) to determine the optimal stride for each node, aiming to minimize trie depth while respecting a memory budget. Its main weakness is the construction process, which is computationally intensive and time-consuming, requiring significant training.

Hash-based methods partition prefixes by length into separate hash tables. The classic Binary Search on Prefix Lengths (BS) scheme and its derivatives [24, 25, 42] organize these hash tables into a Binary Search Tree (BST) to guide the lookup. HBS (Heuristic Binary Search) [25] is a modern BS-family scheme designed for IPv6. It enhances the classic BS approach with several key techniques. First, it employs a heuristic search that uses pre-computed information about marker origins to prune the search space within the BST, reducing the number of hash probes. Second, it uses tree rotation to dynamically adjust the BST structure in response to changes in prefix distribution, ensuring the tree remains optimized without full reconstruction. However, its worst-case lookup time is not as tightly bounded as in trie-based schemes, and its performance depends on the hash function and the dynamic shape of the BST.

We implement and benchmark several state-of-the-art software-based IP lookup methods. We then evaluate these methods on the Intel server described in Section 4. We use the real-world IPv6 FIBs detailed in Table 1 for our benchmarks. Fig. 1 presents the lookup performance in Million Lookups Per Second (MLPS) of PopTrie, CP-Trie, Neurotrie and HBS. A stark and immediate observation is the dramatic performance degradation from IPv4 to IPv6 across all tested methods. For instance, in the rrc00-19 dataset, the lookup speed for the schemes drops by over 75% when transitioning from IPv4 to IPv6. For both schemes, the lookup cost for IPv6 is significantly higher (3× to 10×) than for IPv4. This highlights the fundamental challenge posed by the IPv6's address space, which increases structure size and processing overhead, thereby reducing cache efficiency and overall throughput.

Furthermore, the figure reveals a consistent decline in performance for all algorithms as the FIB size increases from 2019 to 2025. For example, the IPv6 lookup speed of HBS drops by 37.5%, from 136 MLPS on the rrc00-19 dataset to 85 MLPS on the rrc00-25 dataset. This trend underscores the scalability challenge that all software-based lookup schemes face as routing tables continue to grow. Moreover, The memory hierarchy is a critical factor in determining lookup performance. Once the FIB size exceeds the CPU's combined cache capacity (36 MB on our test platform), all algorithms suffer noticeable slowdowns due to the increased cost of off-chip DRAM accesses. Thus, an effective lookup data structure must not only fit within fast but limited on-chip SRAM, but also be highly cache-efficient.

3 Design and Implementation

As established in Section 2, existing software-based IPv6 lookup schemes fail to scale with the size and complexity of modern FIBs. To overcome these limitations, we propose PlanB, a new framework that reformulates the LPM as a one-dimensional search problem over elementary intervals, resolved efficiently using a linearized B^+-tree.

In this section, we first outline the design principles of PlanB, then describe its core components and optimizations in detail. We assume PlanB is used solely to obtain the FIB index for next-hop selection during IP forwarding. The actual routes are maintained separately in a routing table (RIB), such as a radix [29] or Patricia trie [3], allowing aggressive compression of routes that share the same next hop.

3.1 From 2D Prefixes to 1D Intervals

PlanB transforms the LPM into a one-dimensional search problem. This transformation is performed as a pre-computation process each time the FIB is updated. The process involves two key steps.

Range Conversion. Each forwarding rule, defined by a prefix P and a prefix length L, is converted into a closed interval $[start, end]$ on the 128-bit unsigned integer space. The *start* address is simply the prefix P itself, while the *end* address is calculated as $P + (2^{128-L} - 1)$. For example, the prefix 2001:0db8::/32 is converted to the range [2001:0db8::, 2001:0db8:ffff:ffff:ffff:ffff:ffff:ffff]. This conversion maps all prefixes, regardless of their length, into a uniform format of address ranges. As visualized in Fig. 2, this can be conceptualized in two dimensions: the x-axis represents the entire IPv6

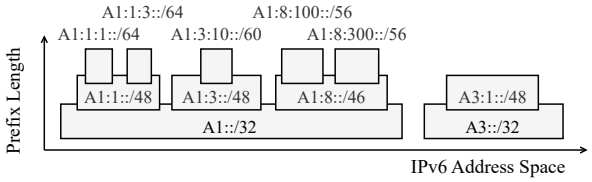

Figure 2: Visualization of IPv6 prefixes as prioritized address ranges. The horizontal and vertical axes represent the IPv6 address space and prefix length (priority), respectively.

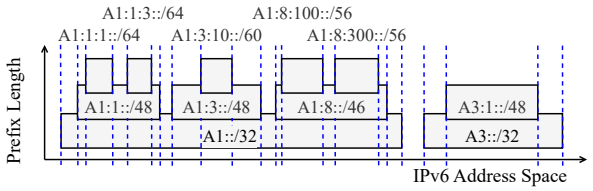

Figure 3: Partitioning the address space into non-overlapping elementary intervals using the start and end points of prefix ranges as boundaries (vertical dashed lines).

address space, and the y-axis represents priority. Each rule becomes a rectangle whose width corresponds to its address range and whose height corresponds to its prefix length (priority). Consequently, more specific routes (longer prefixes) are represented as narrower, higher-priority rectangles that visually overlay the wider, lower-priority rectangles of less specific routes.

Interval Partitioning. Next, the *start* and *end* addresses from all converted ranges are collected. These boundary points are sorted and used to partition the 128-bit IPv6 address space into a set of disjoint, contiguous elementary intervals. This partitioning process is deterministic and requires no tunable parameters, as the partitions are derived directly from the prefix boundaries. As illustrated by the vertical dashed lines in Fig. 3, these boundaries divide the address space into non-overlapping segments. For a FIB containing N prefixes, this process generates $2N$ boundary points (N start points and N end points), resulting in at most $2N+1$ elementary intervals.

This procedure constructs a canonical, one-dimensional data structure from the FIB, implemented as a sorted array of elementary intervals. Each element in the array associates a best-matching route (next-hop) pre-calculated along with the partitioning. The elements that are not covered by any prefix are associated with the default route. Consequently, a lookup for a destination address D is reduced to a standard predecessor search: finding the interval in the sorted list with the largest $interval\_start$ value that is less than or equal to D. This reduction of two-dimensional LPM to a one-dimensional search problem is fundamental to PlanB's efficiency. The 1D transformation is distribution-agnostic as it operates on the boundaries of prefix ranges, not the prefixes themselves. The critical insight of this partitioning is that all addresses within a single elementary interval are covered by the same set of original prefix ranges. Due to the LPM rule, a single, highest-priority route (i.e., the one with the longest prefix length) is the definitive best match for all addresses in that interval.

3.2 1D Search with Linearized B^+-Tree

To efficiently support high-speed lookups, PlanB proposes linearized B^+-tree, which extends the flat-array layout originally developed for standard B-tree structures [28].

Flat Array Layout of B-Tree. A B-tree is a balanced tree structure in which each node contains multiple keys and children [13]. In the flat array layout of B-tree [28], the data are

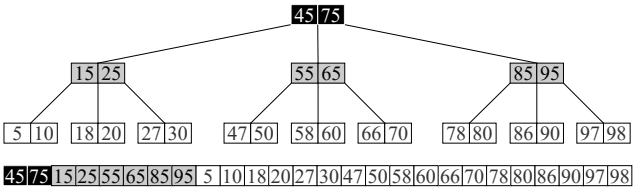

Figure 4: An example of the flat array layout of a 3-ary B-tree.

conceptually organized as a complete binary search tree. The values of the nodes in this virtual tree are stored in an array in the order they appear during a left-to-right breadth-first traversal (Fig. 4). However, a key limitation of this approach is that the search process must continuously check if the target key has been found at each internal node. These intermediate checks introduce conditional branches, which incur lookup overhead and complicate acceleration through batching.

Linearized B^+-Tree. PlanB proposes linearized B^+-tree to overcome this limitation. This variant stores all actual keys reside in the leaf nodes. The internal nodes contain only discriminating keys that guide the search toward the target leaf. These keys correspond to the start and end addresses of elementary intervals that are transformed from the original prefixes. The tree also stores all keys in a single flat array, where the internal nodes are laid out first in breadth-first order, followed by all the leaf nodes. As shown in the blue box in Fig. 5, leaf nodes replicate the keys from internal nodes.

Search on Linearized B^+-Tree. The search algorithm traverses the structure from the root to a leaf node within the flat array, as detailed in Algorithm 1. In the search process, the parent-child relationship in the linearized B^+-Tree can be computed arithmetically from array indices. To locate the keys of a child node during traversal of the internal nodes, we use its parent's level d (with the root at $d=0$) and the parent's 0-indexed position j within that level. The calculation involves two steps:

- Find the starting index of the child's level. The total number of keys in all internal node levels preceding the child's level $(d+1)$ is the sum of a geometric series. With k keys per node and a tree arity of b $(=k+1)$, the starting index of level $d+1$:

$$\text{level\_start}(d+1) = k \times \frac{b^{(d+1)}-1}{b-1}$$

- Find the child's offset within that level. Each node has b children, so the j-th node at level d has its children starting

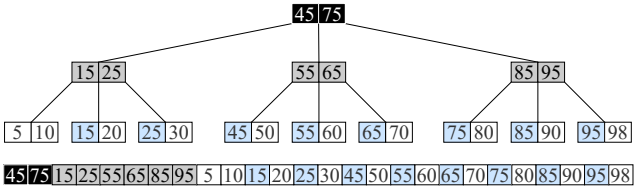

Figure 5: An example of a 3-ary linearized B^+-tree.

at offset $j \times b$ within level $d+1$. Therefore, the starting index of the keys for a specific child c (where c is from 0 to $b-1$) of the j-th node at level d is given by:

$$\text{child\_start}(d, j, c) = \text{level\_start}(d+1) + (j \times b + c) \times k$$

Parameter Tuning. We tune b and k for cache efficiency, sizing each node to fit a 64-byte cache line (common on x86 CPUs). PlanB uses 64-bit keys (most significant bits of 128-bit interval starts), allowing $k = 8$ sorted keys per node. This yields a B^+-tree of order 9 (8 keys, $b = 9$ children), forming a 9-ary tree where nodes align perfectly with cache lines. For instance, the root node's keys occupy indices 0-7 of the array, and the keys of its nine children are stored contiguously starting from index 8. The design is generalizable: an architecture with 128-byte cache lines, such as Apple Silicon [19], would use 16-key nodes to form a 17-ary tree.

The search on the linearized B^+-tree always proceeds down to a leaf node, making the lookup process deterministic. A tree with d internal levels has a fixed search depth of $d+1$, meaning the worst-case lookup cost is strictly bounded by the tree's height. This property is crucial for minimizing tail latency by avoiding the performance outliers common in trie- or hash-based approaches.

The tree's capacity scales with its height. A B^+-tree with d levels of internal nodes can have up to b^d leaf nodes. With each leaf holding k keys, the total capacity is $k \times b^d$. For example, a tree with 5 non-leaf levels has $\sum_{i=0}^{4} 9^i = 6,560$ internal nodes, indexing $9^5 = 59,049$ leaf nodes. The number of internal nodes is approximately a fraction of $1/(b-1) = 1/8 = 12.5\%$ of the number of leaf nodes. With 8 keys per leaf, the structure supports more than 472,392 elementary intervals, which meets the needs of current FIBs. In this configuration, the leaf nodes (M_{leaf}) occupy ~3.6MB of memory for 8-byte keys, and the memory footprint of the internal nodes is $M_{internal} \approx M_{leaf}/(b-1) \approx 0.45$MB. Consequently, each FIB entry consumes an average of 18 bytes, comprising 16 bytes for the leaf node and 2 bytes for internal node overhead. A tree with 6 internal levels can index over 4.2 million elementary intervals, sufficient for future FIBs with over 1M prefixes. For this larger structure, each lookup completes in a predictable 7 accesses. In this case, the memory consumption increases to ~32MB for the leaf nodes and 4MB for the internal nodes. The total size of ~36MB remains well within the L3 cache available on modern server-class CPUs.

Maintaining Structural Integrity. To guarantee efficient arithmetic indexing and search, PlanB maintains the structural integrity of a complete b-ary tree. This is achieved by mandating that all nodes have a fixed, full layout. All nodes on non-leaf levels must be full; partially filled nodes are padded with sentinel values (i.e., 0xFFFFFFFFFFFFFFFF) to achieve this. For example, if a node with eight key slots has only five valid entries, the remaining three are filled with sentinels. Leaf nodes are also padded, but PlanB applies a memory optimization by concentrating sentinel values in the rightmost nodes of each level in a bottom-up manner. This approach eliminates the need to allocate memory space for trailing leaf nodes that would otherwise contain only sentinels. Although internal nodes consisting entirely of sentinel keys still occupy memory, they are excluded from the search process and are never accessed, thereby avoiding CPU cache loads. Padding preserves the property that child locations can be derived solely through index arithmetic, eliminating the need for additional metadata such as node sizes or linked pointers. In practice, PlanB uses a sentinel key value for padding, which guarantees correct search behavior while preventing misrouting.

Algorithm 1 Search on Linearized B^+-tree

Input: `prefix`: The target IPv6 prefix. `key[]`: The flat array representing the linearized B^+-tree. `d`: The depth of internal nodes in the tree. `k`: keys per node.
Output: `final_idx`: The index in the leaf-node region of `key[]` for the matching elementary interval.
b ← k +1
node_off_in_level ← 0
for i ← 0 **to** $d-1$ **do**
    level_start ← k × $\left(\frac{b^i-1}{b-1}\right)$
    node_start ← level_start + node_off_in_level × k
    node_idx ← `lower_bound_binary_search`(key, node_start, k, prefix)
    node_off_in_level ← node_off_in_level × b + node_idx
end for
leaf_start ← k × $\left(\frac{b^d-1}{b-1}\right)$
leaf_node_start ← leaf_start + node_off_in_level × k
final_idx ← `lower_bound_binary_search`(key, leaf_node_start, k, prefix) + leaf_node_start
return final_idx

Separation of Keys and Values. The B^+-tree itself is represented in a single, dense array containing only the sorted 64-bit interval start addresses (the keys). The corresponding next-hop information (the values) is stored in a separate array. The two arrays are associated implicitly by their indices: the value corresponding to the key at index i in the leaf-node region of the key array is located at $values[i - leaf\_start]$, where $leaf\_start$ denotes the first index of the leaf nodes in the key array. During a lookup, the traversal algorithm only needs to access the key array to find the correct index. Once the search completes and the index is identified, the next-hop is retrieved in a single memory access.

3.3 Vectorization and SIMD Acceleration

PlanB exploits data-level parallelism through modern CPU SIMD extensions, particularly AVX-512 [16]. The proposed

Listing 1 Vectorized Search on Linearized B$^+$-tree.

```
1  // k: keys per node, key: B+-tree keys array
2  long LBTree::vectorized_search(uint64_t prefix) {
3      long i = 0,c = -1, d = tree_depth;
4      __m512i vx = _mm512_set1_epi64(prefix);
5      do {
6          i = (k + 1) * i + (c + 1) * k;
7          __m512i node = _mm512_load_si512(&key[i]);
8          c = _mm512_cmpge_epu64_mask(vx, node);
9          c = __builtin_popcount(c);
10     } while (--d);
11     return i+c;
12 }
```

Listing 2 Assembly Code Snippet of The Vectorized Search.

```
1  <+00>: vpbroadcastq %rsi, %zmm0
2  <+06>: movl   0x8(%rdi), %ecx
3  <+09>: movq   (%rdi), %rsi
4  <+12>: orq    $-1, %rdx
5  <+16>: xorl   %eax, %eax
6  <+18>: leaq   (%rax,%rax,8), %rax
7  <+22>: leaq   0x8(%rax,%rdx,8), %rax
8  <+27>: vpcmpnltuq (%rsi,%rax,8), %zmm0, %k0
9  <+35>: kmovb  %k0, %edx
10 <+39>: popcntl %edx, %edx
11 <+43>: decq   %rcx
12 <+46>: jne    <+18>
13 <+48>: addq   %rdx, %rax
14 <+51>: retq
```

design is portable across diverse SIMD architectures, such as AVX2 [16], SSE [37], and ARM NEON [50]. The contiguous, cache-line-aligned layout of our linearized B^+-tree is critical for enabling efficient SIMD operations. Instead of performing a sequence of scalar comparisons, PlanB's search algorithm executes the in-node search using a few highly efficient vector instructions. Listing 1 shows this algorithm, and Listing 2 shows the corresponding assembly code generated by the compiler and disassembled using `gdb`. The process for a node containing 8 keys on an AVX-512 capable CPU is as follows:

(i) **Broadcast.** The 64-bit target address is broadcast into all 8 slots of a 512-bit vector register (line 4, Listing 1).

(ii) **Vector Load.** The 8 sorted 64-bit keys of a B^+-tree node are loaded from memory into a second 512-bit SIMD register (line 7, Listing 1).

(iii) **Parallel Comparison.** A single SIMD instruction is executed to compare the target address against all 8 keys in the node simultaneously (line 8, Listing 1). Certain compilers can fuse this operation with the preceding load, generating a single assembly instruction such as `vpcmpnltuq` (line 8, Listing 2).

(iv) **Child Traversal.** The comparison results are used to determine the index of the next child node to visit (line 9 & 6 in Listing 1). The process then repeats from Step (ii) at the next tree level until a leaf node is reached.

By replacing a loop of scalar operations with a short sequence of powerful vector instructions, this SIMD-accelerated approach dramatically reduces the number of CPU cycles required to traverse each node in the B^+-tree. For example, in a B^+-tree of height 7, a lookup requires only seven vector loads and comparisons, substantially fewer than the dozens of scalar comparisons performed in a conventional binary search. On platforms with different SIMD widths, like AVX2 (256-bit) or ARM NEON (128-bit), the principle remains identical. The in-node search is simply composed of multiple vector operations to cover all keys in the node (e.g., two 256-bit operations for an 8-key node), demonstrating the flexibility and hardware-agnostic nature of our approach.

3.4 Batching

PlanB processes lookups in batches rather than handling packets one by one. Processing addresses individually underutilizes the CPU, as the unavoidable latency of key instructions forces the pipeline to wait. Batching mitigates this by enabling the processor to work on multiple lookups in parallel, hiding latency and maximizing the use of its execution units.

First, the complex SIMD instructions central to PlanB's node search, such as `vpcmpnltuq` (line 8, Listing 2), exhibit multi-cycle latency. In a single-lookup model, the CPU would issue the comparison and then stall, waiting for the results before it could execute the subsequent instruction. By processing a batch of lookups, PlanB provides the CPU with independent work. This pipelined execution of instructions from different lookups keeps the processor's resources constantly engaged, effectively hiding instruction latency and sustaining a high rate of completed operations.

Second, after a SIMD comparison, the intermediate results must be transferred from vector units into general-purpose units before determining the correct child index. Although efficient, this transfer still introduces overhead. By treating a bundle of addresses as a batch, PlanB amortizes this per-lookup cost, executing transfer and post-processing instructions in a way that maximizes throughput across the entire batch rather than paying the cost repeatedly for each lookup.

Our DPDK implementation (Section 3.7) forms batches directly from the bursts of packets retrieved from the NIC's RX ring. This approach maximizes throughput by processing available packets immediately, up to the hardware vector width, without incurring any artificial latency for batch formation. Batching transforms the lookup from a series of latency bound, sequential operations into a highly parallel, throughput oriented pipeline. Such a design synergizes with our other optimizations to fully exploit the instruction level parallelism of modern CPUs.

3.5 Branch-Free Traversal and Loop Unrolling

A standard B^+-tree traversal involves a sequence of comparisons within each node to find the correct child branch. Implemented naively with if-else statements, this process creates conditional branches in the instruction stream. For unpre-

dictable input traffic, these branches are highly susceptible to misprediction by the CPU's branch predictor. A single misprediction incurs a significant performance penalty, forcing the CPU to flush its speculative execution pipeline and restart from the correct path, which can cost dozens of cycles and severely degrade lookup throughput. PlanB's traversal algorithm is designed to be entirely branch-free. Instead of conditional jumps, we employ specialized CPU instructions that manipulate data based on comparison outcomes without altering the control flow. We leverage two key types of branch-free instructions: vectorized comparisons, which produce a bitmask, and conditional moves.

First, we can replace a sequence of scalar comparisons with a single vectorized comparison as described in Section 3.3. This operation compares the target address against all keys in a node simultaneously using an SIMD instruction. The result of this parallel comparison is an 8-bit mask, where each bit corresponds to the outcome of one of the 64-bit lane comparisons. Next, we apply a population count (`popcnt`) instruction to this mask (line 9 in Listing 1 and line 10 in Listing 2), The resulting count directly provides the index of the child pointer to traverse next. This single instruction replaces an entire loop of bit checks, making the process of finding the correct branch from the comparison result extremely fast and branch-free.

Alternatively, in scenarios where vector instructions are not used, the target address is compared against a node's keys, setting flags in the CPU's status register. PlanB then use these flags to conditionally update the index to the next node without executing a jump. Both techniques are integral to PlanB's design, as they replace control-flow dependencies with data-flow dependencies, which modern out-of-order CPUs can execute far more efficiently.

To further optimize performance, we fully unroll the nested traversal loops (per-tree-level search $\times$ per-batch lookup). Each iteration consists of only a handful of instructions, including one vector comparison, one move-mask, a single `ctz`, and an arithmetic calculation for indexing the child node. Because the linearized B^+-tree is shallow (seven levels are sufficient to index over one million intervals) and lookup batches are typically small (tens of addresses per core), the total number of operations per lookup remains bounded and predictable. This makes complete unrolling feasible at compile time without code-size explosion or instruction-cache pressure. The resulting straight-line instruction sequence eliminates all loop-control overhead and maximizes the compiler's ability to reorder instructions aggressively, hide instruction latencies, and issue operations in parallel.

3.6 Dynamic FIB Update

Handling dynamic FIB updates without compromising lookup performance is a critical requirement for practical routing systems. PlanB adopts a batch-oriented, rebuild-and-swap model that cleanly separates updates from lookups. The update process is triggered for a batch of one or more prefix changes (insertions, deletions, or modifications). Instead of altering the live data structure, PlanB performs the update in the background by following these steps:

- **Batch Modify.** Updates are accumulated into a secondary prefix list maintained in a seperate memory region, preferably by other CPU cores. Multiple pending changes can be coalesced into a single batch.
- **Complete Rebuild.** The entire pre-computation process, including range conversion, interval partitioning, and linearized B^+-tree construction, is executed on the prefix list as detailed in Algorithm 2. The result is a brand-new lookup structure that is compact and consistent.
- **Atomic Swap.** Once fully built, the system performs a single atomic pointer update to redirect future lookups to the new structure. Threads in the middle of a lookup continue using the old structure until they complete, after which it can be safely deallocated.

This strategy provides several benefits. First, the lookup path remains strictly contention-free. Since lookups continue to operate on the stable, existing data structure while the new one is built in the background, they suffer no performance degradation or stalls from update activity. No locks or other synchronization primitives are needed in the forwarding fast

Algorithm 2 Linearized B^+-tree Construction from FIB

1: **Input:** A FIB $F = \{f_1, f_2, ...\}$. Each entry f is $(f.\text{prefix},\ f.\text{length},\ f.\text{next\_hop})$.
2: **Tree parameters:** depths d, keys per node k, arity $b = k+1$.
3: **Output:** Linearized B^+-tree of keys array *key* and next-hop array *value*.
4: Let E be an empty list of tuples $(address, next\_hop)$.
5: $E \leftarrow \{(f.\text{prefix},\ f.\text{next\_hop}), (f.\text{prefix} + 2^{64-f.\text{length}},\ -1), ... \mid f \in F\}$
6: Sort E lexicographically by $(address, length)$
7: Let S be an empty stack for next-hops
8: **for** each element $e \in E$ **do**
9: **if** e.next_hop ≥ 0 **then**
10: S.push(e.next_hop)
11: **else**
12: S.pop()
13: **if** S is not empty **then**
14: e.next_hop $\leftarrow S$.top()
15: **end if**
16: **end if**
17: **end for**
18: $L_{\text{start}} \leftarrow level\_start(d-1)$
19: **for** $i \leftarrow 0$ to $|E|-1$ **do**
20: $key[L_{\text{start}} + i] \leftarrow E[i]$.prefix
21: $value[i] \leftarrow E[i]$.next_hop
22: **end for**
23: Fill remaining entries in the leaf level with sentinels.
24: **for** $l \leftarrow d-1$ to 1 **do**
25: $p_{\text{start}} \leftarrow level\_start(l-1)$
26: $c_{\text{start}} \leftarrow level\_start(l)$
27: **for** $i \leftarrow 0$ to $b^l - 1$ in steps of $k \times (k+1)$ **do**
28: **for** $j \leftarrow 1$ to k **do**
29: $key[p_{\text{start}}] \leftarrow key[c_{\text{start}} + i + k \times j]$
30: $p_{\text{start}} \leftarrow p_{\text{start}} + 1$
31: **end for**
32: **end for**
33: **end for**
34: **return** $(key, value)$

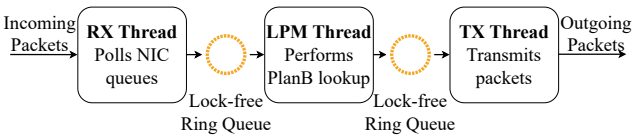

Figure 6: PlanB's RX-LPM-TX module within DPDK.

path. Second, the transition is seamless. Once the new structure is ready, a single atomic pointer swap redirects all new lookups. This guarantees that every lookup operation completes correctly without being affected by the update. Third, by batching updates and performing a full rebuild, PlanB amortizes the update cost and efficiently handles high-volume or bursty route changes. This makes its performance predictable and robust, regardless of the update pattern.

The trade-off is a temporary increase in memory usage to hold both the old and new structures. However, this overhead is significantly mitigated by PlanB's inherent compactness. PlanB's lean design ensures that even during an update, the total memory consumption is less likely to evict other critical data from the CPU cache. Once all in-flight lookups on the old structure complete, it is safely deallocated and has no further impact on the system.

3.7 Implementation

We implemented a prototype software router for PlanB using the Data Plane Development Kit (DPDK) [17, 32]. Our implementation leverages thread specialization and batch processing pipelines to meet three key requirements: (i) fully exploiting PlanB's SIMD-accelerated search, (ii) sustaining line-rate forwarding with large, realistic IPv6 FIBs, while being robust to anticipated growth in FIB size and complexity, and (iii) scaling efficiently across multi-core processors with minimal coordination overhead.

Thread Pipeline. To prevent other packet processing tasks from degrading LPM cache hit rate, we extend the DPDK performance-thread RX-TX model [32] by introducing a dedicated lookup stage between packet reception and transmission. In this RX-LPM-TX model (illustrated in Fig. 6), RX threads poll NIC queues and enqueue bursts of packets into per-queue software rings. Lookup (LPM) threads dequeue those bursts, execute PlanB's B^+-tree search to resolve next-hop indices, and annotate packets with egress metadata. Finally, TX threads batch, format, and transmit the annotated packets to the appropriate NIC port. This separation enables independent scaling of each stage: RX threads scale with NIC queues, LPM threads scale with the cost of IPv6 lookups, and TX threads scale with output load.

Cache Residency. Modern multi-core CPUs typically feature private L1/L2 caches for each core and a shared L3 cache. This L3 cache can be either fully shared across all cores or partitioned by groups of cores (e.g., the CCDs/CCXs in AMD Zen5 architecture). For architectures with such group-shared L3 caches, PlanB leverages the hardware topology to ensure cache isolation. It pins the LPM threads to cores in one cache group, while confining the RX/TX and management threads to cores in a separate group. This strategy prevents the LPM thread's L3 cache slice from being evicted by other system activities, thereby preserving its cache residency and maintaining high, predictable performance.

Design Trade-offs. A key trade-off of our three-stage RX-LPM-TX pipeline is a slight increase in per-packet latency over a conventional RX-TX model. However, for IPv6 workloads where lookup cost dominates, this overhead is offset by reduced contention, improved SIMD batching, and better load distribution. For less demanding workloads, such as those with smaller IPv4 or IPv6 FIBs, PlanB supports a fused two-stage RX-(LPM+TX) configuration. In this model, the lookup and transmission logic are combined into a single thread, eliminating the extra ring crossing while still leveraging cache-friendly, SIMD-accelerated lookups. Moreover, the number of LPM+TX threads can be tuned independently of the RX threads. This flexibility allows PlanB to be optimized for different scenarios: a three-stage pipeline for lookup-bound workloads and a two-stage pipeline for I/O-bound ones.

4 Evaluation

We have implemented PlanB in C++ as a user-space application on Linux. Our evaluation is conducted on two hardware platforms. We first employ two servers, each featuring a 24-core Intel Xeon 8331C processor (3GHz turbo, 36MB L3 cache) and 1TB of DDR5 RAM. The servers are connected to a 400Gbps Ethernet switch and runs Ubuntu 22.04 with Linux kernel 5.15.0-143-generic. The second platform is a commodity system equipped with an AMD Ryzen 9 370HX mobile processor. The processor features performance cores (Zen5, up to 5.1GHz) and efficiency cores (Zen5c, up to 3.3GHz). The system is configured with 128GB DDR5 RAM and runs the same OS and kernel version as the server.

Evaluated Schemes. We compare PlanB against four state-of-the-art software-based IP lookup schemes: PopTrie [3], CP-Trie [23], Neurotrie [60] (specifically the Neurotrie-S variant) and HBS [25]. The schemes are implemented in C++, integrated with DPDK for packet I/O, and evaluated using the same environment. To ensure fairness, we use a uniform DPDK version, identical DPDK configurations, and compile all code with g++ 11.4.0 using the -O3 optimization flag on both platforms. All evaluated results are averaged over multiple runs to ensure statistical significance.

Datasets. We use rrc00-19–rrc00-25 in Table 1 to test, as the prefix distribution and numbers of rv1–rv7 is similar to rrc00-25. To assess scalability, we generate four synthetic FIBs (synth-1 to synth-4) with 250K, 500K, 750K, and 1M prefixes, respectively. The prefix length distribution of these synthetic FIBs is modeled after the rrc00-25 dataset to ensure realism. For evaluating lookup performance, we generate a test trace for each FIB. Each trace contains a number of lookups equal to 100 times the number of prefixes in the corresponding FIB. These lookups are generated by randomly

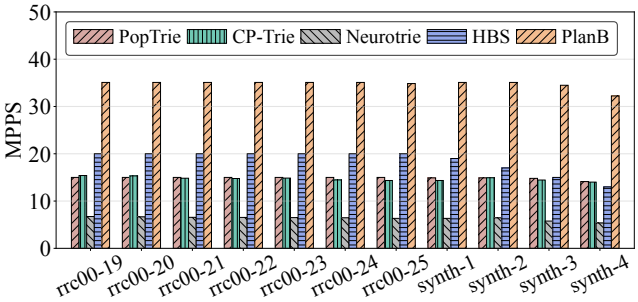

Figure 7: System throughput for DPDK-based L3 forwarding.

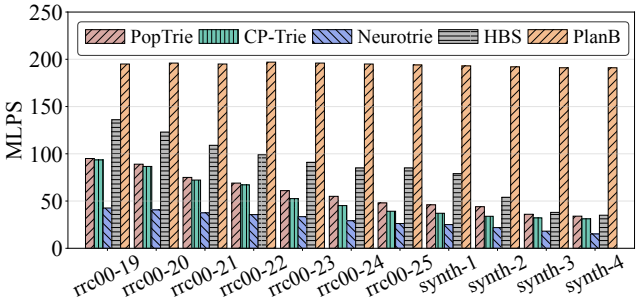

Figure 8: Lookup speed with single core on Intel CPU.

sampling prefixes from the FIB and then creating a destination IP address that matches each sampled prefix. For update evaluation, we collecte one-hour update traces for rrc00 between 8:00 AM and 9:00 AM for every year in the dataset [1]. Moreover, in the trace-driven benchmarks, we observe that loading trace records (64-bit prefixes) from memory significantly degrades overall performance. To address this, we prefetch these records into spare vector registers, which will be reused across subsequent iterations.

Performance Metrics. We evaluate PlanB and other systems across the following key performance dimensions:

- System Throughput: The end-to-end forwarding rate of the DPDK-based L3 application for 64-byte packets, measured in Million Packets Per Second (MPPS).
- Lookup Speed: The raw performance of the IP lookup module, measured as a microbenchmark. We report this in MLPS and BLPS.
- Memory Overhead: The total memory consumed by the lookup data structure, measured in Megabytes (MB).
- Update Overhead: The time required to perform batch updates of prefixes, measured in microseconds (us).

To understand the contributions of various design choices, we further conduct an ablation study by selectively disabling key designs in PlanB and measuring the resulting performance impact. Each experiment are repeated 20 times to ensure reliability, with the average value of each metric are reported.

4.1 System Throughput

We first evaluate the end-to-end system throughput using PlanB-enabled DPDK for back-to-back 64-byte packet forwarding. Fig. 7 shows the results. Across all evaluated FIBs, PlanB achieves a system throughput that is 2.3×∼2.8× faster than PopTrie, 2.3×∼2.9× faster than CP-Trie, 4.7×∼5.8× faster than Neurotrie, and 1.7×∼2.5× faster than HBS. As the number of prefixes in the routing tables increases, the performance gap widens, highlighting PlanB's superior scalability. Whereas trie-based schemes such as PopTrie, CP-Trie, and Neurotrie are constrained by costly memory indirection and conditional logic, PlanB avoids these overheads by transforming the lookup into a 1D search. This is achieved by partitioning the address space into elementary intervals and performing an efficient search on a linearized B^+-tree. This dense, contiguous layout with vectorized search and branch-free logic, enables PlanB to fully leverage modern CPU architectures, delivering high throughput even with large and complex FIBs. Moreover, PlanB is integrated into a dedicated lookup stage between packet reception and transmission in the DPDK pipeline, ensuring that its high lookup speed translates into tangible end-to-end forwarding performance.

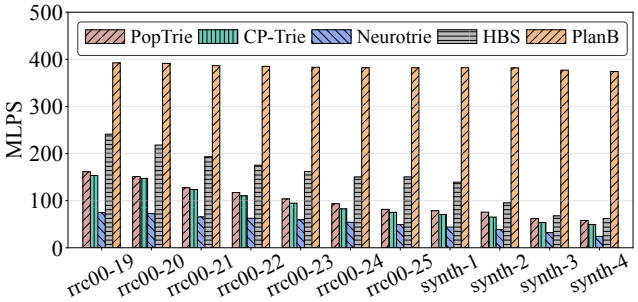

Figure 9: Lookup speed with single core on AMD CPU.

4.2 Lookup Speed

We measure the lookup speed in MLPS/BLPS on both single-core and multi-core configurations with Intel and AMD CPUs to assess their absolute performance and scalability.

Single-Core Performance. Figs. 8 and 9 illustrate the single-core lookup performance across a range of real-world and synthetic IPv6 FIBs. On the Intel server, PlanB consistently delivers a lookup speed between 191 and 197 MLPS. This performance represents a speedup of 2×∼5.6× over PopTrie 2.1×∼5.8× over CP-Trie, 4.3×∼9.6× over Neurotrie, and 1.4×∼5.6× over HBS. The performance advantage of PlanB is even more pronounced on the AMD mobile processor, where it achieves a remarkable 374∼393 MLPS. This represents a speedup of 2.3×∼6.7× over PopTrie, 2.6×∼7× over CP-Trie, 5.3×∼12.5× over Neurotrie and 1.6×∼6.3× over HBS. The significant performance improvement of PlanB stems from its fundamental design. By converting LPM to a 1D search, PlanB executes a fixed, small number of highly optimized steps to find the result. The performance of trie-based schemes degrades with longer prefixes that require deeper trie traversal. While Neurotrie attempts to mitigate this with a DRL-optimized shallow trie, its real-

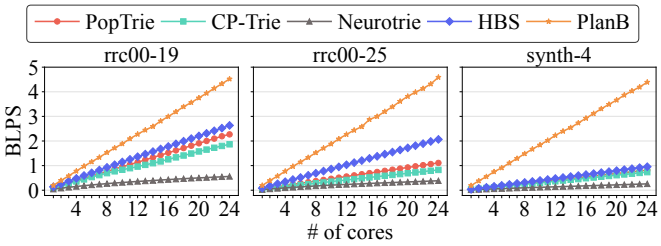

Figure 10: Lookup speed with multi-cores on Intel CPU.

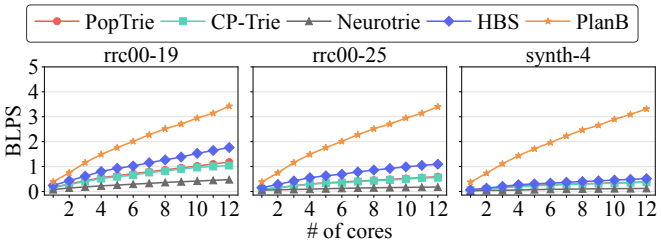

Figure 11: Lookup speed with multi-cores on AMD CPU.

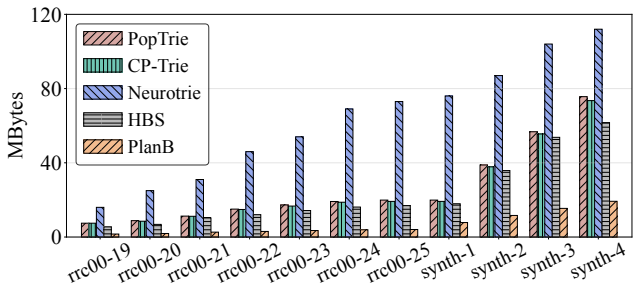

Figure 12: Memory overhead.

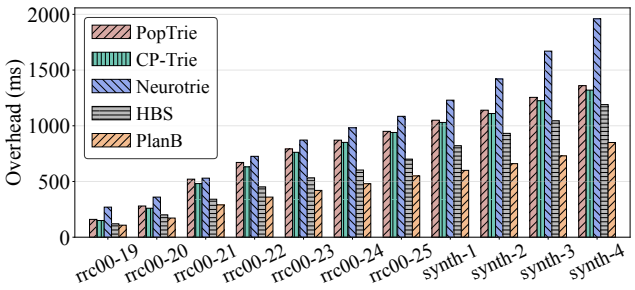

Figure 13: Update overhead.

world performance is bottlenecked by high per-node overhead. Each step in its lookup requires chasing pointers between separate memory structures and executing complex scalar logic. HBS's performance is sensitive to the number of distinct prefix lengths and the efficiency of its hash function.

Multi-Core Scalability. Figs. 10 and 11 show the aggregated lookup throughput as the number of active cores increases. PlanB demonstrates near-linear scalability on both CPU architectures, a critical attribute for modern multi-core packet processing engines. On the 24-core Intel server, PlanB's throughput scales to a peak of 4.6 BLPS. This is 2×∼5.6× higher than PopTrie, 2.4×∼6× higher than CP-Trie, 8.1×∼14× higher than Neurotrie and 1.7×∼4.8× higher than HBS at maximum core counts. On the 12-core AMD mobile processor, PlanB achieves up to 3.4 BLPS, outperforming PopTrie by 2.9×∼8.7×, CP-Trie by 3.3×∼9.0×, Neurotrie by 7.2×∼18.3×, and HBS by 1.8×∼6.7×. PlanB's batching and branch-free logic hide instruction latencies and avoid misprediction penalties, which plague PopTrie, CP-Trie and Neurotrie with their pointer-based, conditional traversals and HBS's heuristic searches. Loop unrolling further optimizes the shallow tree depth (typically 6–7 levels), enabling linear multi-core scaling.

4.3 Memory Overhead

As established in our design principles, maintaining cache residency is critical for high-speed lookups. Fig. 12 quantifies PlanB's advantage in this regard, shows the memory overhead for various schemes across different FIBs. Our evaluation shows that PlanB reduces the memory overhead by 60.8%∼79.6% compared to PopTrie, 59.3%∼79.7% compared to CP-Trie, 82.8%∼92.5% compared to Neurotrie, and 56.4%∼75.9% compared to HBS. This dramatic reduction allows PlanB's data structure to comfortably fit within the L3 cache of modern CPUs, even for FIBs containing up to one million prefixes, which is a key factor in its high throughput.

By transforming prefixes into a set of elementary intervals, PlanB ensures that the size of its primary data structure scales linearly with the number of unique prefix boundaries (at most $2N+1$ for N prefixes). This contrasts sharply with the overheads of competing methods: trie-based schemes suffer from prefix expansion, particularly for long IPv6 prefixes, while HBS requires substantial extra space to maintain low hash collision rates. PlanB handles significantly larger FIBs before its performance is degraded by the memory hierarchy. While other schemes would face a performance collapse due to DRAM latency, PlanB maintains cache residency, ensuring high throughput and superior scalability.

4.4 Update Overhead

While prioritizing lookup performance, PlanB also maintains a highly efficient update mechanism. By employing a batch-oriented, rebuild-and-swap strategy, PlanB achieves a lower amortized update overhead than competing schemes. As illustrated in Fig. 13, our evaluation across real-world FIBs shows that PlanB's update process is consistently faster, reducing the average update overhead by 32.5%∼44.6% compared to PopTrie, 28%∼41.5% compared to CP-Trie, 49.3%∼60% compared to Neurotrie, and 10.2%∼21.5% compared to HBS. Neurotrie incurs high update overhead because rebuilding its structure requires executing a DRL-based inference process. Moreover, for a FIB with 1 million prefixes (synth-4), PlanB's total rebuild time is just 850 ms. This quantification of the worst-case reconstruction overhead for large-scale datasets confirms that our rebuild-and-swap model remains practical and efficient even as routing tables continue to grow.

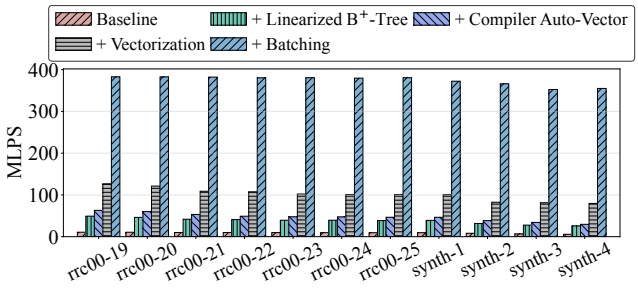

Figure 14: Ablation study of PlanB's design components.

4.5 Ablation Study of Design Components

To quantify the performance contributions of PlanB's design, we conduct an ablation study on our AMD platform, measuring single-core lookup speed as we progressively enable optimizations. Fig. 14 illustrates the results across.

Our **Baseline**, a naive implementation using 1D interval transformation with standard C++ binary search (`std::lower_bound`) on a sorted array, achieves only 5.8∼10.7 MLPS due to poor cache locality and branch mispredictions. Replacing this with the **Linearized B^+-Tree and Branch-Free scalar logic** yields a speedup of 3.8×∼4.6× over the baseline. We evaluate these two components together, as compiler optimizations (-O3) automatically generate branch free code for a simple scalar loop. Moreover, the sequential layout of the linearized B^+-tree inherently enables the subsequent zero-overhead vectorization and batching optimizations, underscoring its foundational role in the system. Enabling **Compiler Auto-Vectorization** for the in-node search provides another 1.15×∼1.3× speedup. This gain demonstrates the limitations of automatic compiler optimizations and serves as a reference for our manual approach.

Introducing AVX-512 **Vectorization** using intrinsics provides the next major boost, outperforming the auto-vectorized version by 2×∼2.7×. This improvement comes from replacing the sequential in-node search with a single, parallel SIMD comparison. Finally, incorporating **Batching** achieves PlanB's full performance, delivering an additional speedup of 3×∼4.5×. Batching effectively hides SIMD instruction latency by pipelining independent lookups, thus maximizing CPU utilization. This study clearly demonstrates that each component of PlanB's design contributes synergistically to its state-of-the-art performance.

5 Discussion and Related Work

Implementation on Diverse Hardware. PlanB adapts easily to varied platforms. Without 512-bit SIMD, wide comparisons simply decompose into narrower SIMD or scalar operations to build the bitmask. For hardware implementations on FPGAs or ASICs, the linearized B^+-tree can reside in on-chip addressable SRAM. The necessary operations, simple aligned loads and vector to scalar comparisons, are efficient and do not require a dedicated vector unit. Moreover, a hybrid software and hardware implementation can isolate the data plane from update overhead. A control plane processor can rebuild the lookup structure in software, while the data plane uses a hardware accelerator with the resulting linearized B^+-tree stored in on-chip memory for high speed forwarding.

Near Constant Performance. PlanB is highly resilient to unpredictable routing patterns and malicious attacks. Unlike traditional traversal mechanisms that suffer severe performance degradation or potential router failure under random traffic due to overwhelmed pipelines and frequent cache misses, PlanB guarantees a fixed, shallow search depth. By sustaining high cache-hit rates and branch-free execution, PlanB reliably maintains near constant throughput even under extreme dynamic workloads and security threats.

Multiway Range Trees. Multiway Range Trees (MRTs) [49, 55] are a foundational IP lookup data structure that partitions the address space into elementary intervals. MRTs are typically implemented as pointer-based B-trees. Traversing these trees involves pointer chasing, which leads to memory indirection and can cause cache misses and pipeline stalls. Moreover, MRT lookups are complex, requiring resolution of candidate prefixes stored in nodes along the root-to-leaf path to determine the longest match.

Hybrid CAM and RAM Architectures. Hybrid architectures leverage Content Addressable Memory (CAM) for fast, parallel lookups alongside high-density RAM. For instance, CRAM [8] demonstrates that partitioning data structures across TCAM and SRAM supports databases exceeding individual memory capacities. Similarly, PtCAM [44] accelerates name-prefix matching by storing compact Patricia trie bit-patterns in TCAM for the initial search, while offloading the final prefix verification to RAM.

6 Conclusion

We present PlanB, a high-performance IPv6 lookup framework that recasts the LPM problem as a one-dimensional search problem over elementary intervals. PlanB traverses this simplified search space using linearized B^+-tree and a search algorithm aggressively optimized with SIMD instructions, batching, branch-free logic and loop unrolling. These techniques fully exploit CPU parallelism to deliver high throughput. Evaluation results show that PlanB substantially outperforms current state-of-the-art software lookup solutions across a wide range of FIBs and hardware, from mobile CPUs to server-grade processors. PlanB has been open-sourced at https://github.com/nicexlab/planb.

Acknowledgments

We thank our shepherd, Prof. Jingxian Wang, and the anonymous reviewers for their valuable comments and suggestions. Zhihao Zhang and Yiming Zhang are the co-primary authors. This work is supported by the National Natural Science Foundation of China (grant no. 62441220).

References

[1] RIPE RIS is a BGP routing data collection platform. https://ris.ripe.net/docs/route-collectors/.

[2] University of Oregon RouteViews Project. https://www.routeviews.org/routeviews/collectors/.

[3] Hirochika Asai and Yasuhiro Ohara. Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, pages 57–70, 2015.

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

[5] Sukanya Bhowmik, Muhammad Adnan Tariq, Alexander Balogh, and Kurt Rothermel. Addressing tcam limitations of software-defined networks for content-based routing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems (DEBS'17)*, pages 100–111, 2017.

[6] IPv6 Capable Rate by Country. https://stats.labs.apnic.net/ipv6.

[7] Martin Casado, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Rethinking packet forwarding hardware. In *HotNets*, pages 1–6, 2008.

[8] Robert Chang, Pradeep Dogga, Andy Fingerhut, Victor Rios, and George Varghese. Scaling ip lookup to large databases using the cram lens. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, NSDI '25, USA, 2025. USENIX Association.

[9] Dibei Chen, Zhaoshi Li, Tianzhu Xiong, Zhiwei Liu, Jun Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. Catcam: Constant-time alteration ternary cam with scalable in-memory architecture. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'20)*, pages 342–355, 2020.

[10] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. utree: a persistent b+-tree with low tail latency. *Proc. VLDB Endow.*, pages 2634–2648, July 2020.

[11] Daguo Cheng, Lin He, Chentian Wei, Qilei Yin, Boran Jin, Zhaoan Wang, Xiaoteng Pan, Sixu Zhou, Ying Liu, Shenglin Zhang, Fuchao Tan, and Wenmao Liu. Luori: Active probing and evaluation of internet-wide ipv6 fully responsive prefixes. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[12] China Internet Network Information Center (CNNIC). The 55th statistical report on china's internet development. https://www.cnnic.com.cn/IDR/ReportDownloads/202505/P020250514564119130448.pdf.

[13] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[14] IPv6 BGP Table Data. https://bgp.potaroo.net/v6/as2.0/index.html.

[15] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, pages 15–28, 2009.

[16] Intel Architecture Instruction Set Extensions and Future Features Programming Reference. https://www.intel.com/content/www/us/en/content-details/671368/intel-architecture-instruction-set-extensions-programming-reference.html.

[17] Marco Faltelli, Giacomo Belocchi, Francesco Quaglia, Salvatore Pontarelli, and Giuseppe Bianchi. Metronome: Adaptive and precise intermittent packet retrieval in dpdk. *IEEE/ACM Transactions on Networking*, 31(3):979–993, 2023.

[18] Andrew D. Ferguson, Steve Gribble, Chi-Yao Hong, Charles Killian, Waqar Mohsin, Henrik Muehe, Joon Ong, Leon Poutievski, Arjun Singh, Lorenzo Vicisano, Richard Alimi, Shawn Shuoshuo Chen, Mike Conley, Subhasree Mandal, Karthik Nagaraj, Kondapa Naidu Bollineni, Amr Sabaa, Shidong Zhang, Min Zhu, and Amin Vahdat. Orion: Google's Software-Defined networking control plane. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 83–98, 2021.

[19] Apple Silicon CPU Optimization Guide. https://developer.apple.com/documentation/apple-silicon/cpu-optimization-guide.

[20] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, pages 15–26, 2013.

[21] IPv6 Development in China. https://www.cac.gov.cn/2025-08/1/c_1755590302116970.htm.

[22] Md Iftakharul Islam and Javed I Khan. C2rtl: A high-level synthesis system for ip lookup and packet classification. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR'21)*, pages 1–8, 2021.

[23] Md Iftakharul Islam and Javed I Khan. Cp-trie: Cumulative popcount based trie for ipv6 routing table lookup in software and asic. In *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR'21)*, pages 1–8, 2021.

[24] Donghong Jiang, Yanbiao Li, Yuxuan Chen, Jing Hu, Yi Huang, and Gaogang Xie. Heuristic binary search: Adaptive and fast ipv6 route lookup with incremental updates. In *Proceedings of the 7th Asia-Pacific Workshop on Networking (APNet'23)*, pages 47–53, 2023.

[25] Donghong Jiang, Yanbiao Li, Yuxuan Chen, Jing Hu, Yi Huang, and Gaogang Xie. Heuristic binary search: Adaptive and fast ipv6 route lookup with incremental prefix updates. *IEEE Transactions on Networking*, pages 554–569, 2025.

[26] W. Jiang, Q. Wang, and V. K. Prasanna. Beyond tcams: An sram-based parallel multi-pipeline architecture for terabit ip lookup. In *Proceedings of the IEEE INFOCOM 2008*, pages 1786–1794, 2008.

[27] Murad Kablan, Blake Caldwell, Richard Han, Hani Jamjoom, and Eric Keller. Stateless network functions. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, pages 49–54, 2015.

[28] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *ACM J. Exp. Algorithmics*, 22, May 2017.

[29] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE'13)*, pages 38–49, 2013.

[30] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. Dhl: Enabling flexible software network functions with fpga acceleration. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18)*, pages 1–11, 2018.

[31] Yuqiang Li, Fan Zhou, Xiaoxiang Zhu, and Jianming Liao. An ipv6 routing lookup algorithm based on hash table and hot. In *2022 5th International Conference on Information Communication and Signal Processing (ICICSP)*, pages 397–402, 2022.

[32] DPDK Performance Thread Model. https://doc.dpdk.org/guides-16.04/sample_app_ug/performance_thread.html.

[33] Jeffrey C. Mogul and John Wilkes. Physical deployability matters. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks (HotNets'23)*, pages 9–17, 2023.

[34] Md S. Q. Zulkar Nine, Tevfik Kosar, Muhammed Fatih Bulut, and Jinho Hwang. Greennfv: Energy-efficient network function virtualization with service level agreement constraints. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'23)*, 2023.

[35] Maxime Piraux, Tom Barbette, Nicolas Rybowski, Louis Navarre, Thomas Alfroy, Cristel Pelsser, François Michel, and Olivier Bonaventure. The multiple roles that ipv6 addresses can play in today's internet. *SIGCOMM Comput. Commun. Rev.*, pages 10–18, 2022.

[36] Alon Rashelbach, Igor de Paula, and Mark Silberstein. Neurolpm - scaling longest prefix match hardware with neural networks. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, pages 886–899, 2023.

[37] Intel SSE4 Programming Reference. https://www.intel.com/content/dam/develop/external/us/en/documents/d9156103-705230.pdf.

[38] Michael Reininger, Arushi Arora, Stephen Herwig, Nicholas Francino, Jayson Hurst, Christina Garman, and Dave Levin. Bento: safely bringing network function virtualization to tor. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 821–835, 2021.

[39] Erik Rye and Dave Levin. Ipv6 hitlists at scale: Be careful what you wish for. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 904–916, 2023.

[40] Yaniv Sadeh, Ori Rottenstreich, and Haim Kaplan. How much tcam do we need for splitting traffic? In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 169–175, 2021.

[41] Hua Shao, Xiaoliang Wang, Yuanwei Lu, Yanbo Yu, Shengli Zheng, and Youjian Zhao. Accessing cloud with disaggregated Software-Defined router. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*, pages 1–14, 2021.

[42] Tong Shen, Xian Yu, Gaogang Xie, and Dafang Zhang. High-performance ipv6 lookup with real-time updates using hierarchical-balanced search tree. In *2018 IEEE Global Communications Conference (GLOBECOM'18)*, pages 1–7, 2018.

[43] Guanglei Song, Jiahai Yang, Lin He, Zhiliang Wang, Guo Li, Chenxin Duan, Yaozhong Liu, and Zhongxiang Sun. AddrMiner: A comprehensive global active IPv6 address discovery system. In *2022 USENIX Annual Technical Conference (ATC'22)*, pages 309–326, 2022.

[44] Tian Song, Tianlong Li, and Yating Yang. Ptcam: Scalable high-speed name prefix lookup using tcam. In *Proceedings of the ACM SIGCOMM 2025 Conference*, SIGCOMM '25, pages 707–719, 2025.

[45] Google IPv6 Statistics. https://www.google.com/intl/en/ipv6/statistics.html.

[46] Thibaut Stimpfling, Normand Belanger, JM Pierre Langlois, and Yvon Savaria. Ship: A scalable high-performance ipv6 lookup algorithm that exploits prefix characteristics. *IEEE/ACM Transactions on Networking*, 27(4):1529–1542, 2019.

[47] Thibaut Stimpfling, J.M. Pierre Langlois, Normand Bélanger, and Yvon Savaria. A low-latency memory-efficient ipv6 lookup engine implemented on fpga using high-level synthesis. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'18)*, pages 402–411, 2018.

[48] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. Nfp: Enabling network function parallelism in nfv. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17)*, pages 43–56, 2017.

[49] Subhash Suri, G. Varghese, and P.R. Warkhede. Multiway range trees: scalable ip lookup with fast updates. In *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, volume 3, pages 1610–1614, 2001.

[50] ARM NEON Technology. https://www.arm.com/technologies/neon.

[51] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in network function virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 283–297, 2018.

[52] Ying Wan, Haoyu Song, Yang Xu, Yilun Wang, Tian Pan, Chuwen Zhang, Yi Wang, and Bin Liu. T-cache: Efficient policy-based forwarding using small tcam. *IEEE/ACM Transactions on Networking*, pages 2693–2708, 2021.

[53] Jing Wang, Qing Wang, Yuhao Zhang, and Jiwu Shu. Deft: A Scalable Tree Index for Disaggregated Memory. In *Proceedings of the Twentieth European Conference on Computer Systems (EuroSys'25)*, pages 886–901, 2025.

[54] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD'22)*, pages 1033–1048, 2022.

[55] Priyank Warkhede, Subhash Suri, and George Varghese. Multiway range trees: scalable ip lookup with fast updates. *Comput. Netw.*, 44(3):289–303, February 2004.

[56] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. Guarantee ip lookup performance with fib explosion. In *Proceedings of ACM Conference on SIGCOMM (SIGCOMM'14)*, pages 39–50, 2014.

[57] Tong Yang, Gaogang Xie, Alex X. Liu, Qiaobin Fu, Yanbiao Li, Xiaoming Li, and Laurent Mathy. Constant ip lookup with fib explosion. *IEEE/ACM Transactions on Networking*, pages 1821–1836, 2018.

[58] Xinyi Zhang, Zhiyuan Xu, Huaiyi Zhao, Yanbiao Li, and Gaogang Xie. Tar: Traffic adaptive ipv6 routing lookup scheme. In *Proceedings of the 8th Asia-Pacific Workshop on Networking (APNet'24)*, pages 135–141, 2024.

[59] Kai Zheng, Chengchen Hu, Hongbin Lu, and Bin Liu. A tcam-based distributed parallel ip lookup scheme and performance analysis. *IEEE/ACM Transactions on Networking*, 14(4):863–875, 2006.

[60] Yuxi Zhu, Hao Chen, Yuan Yang, Mingwei Xu, Yuxuan Zhang, Chenyi Liu, and Jianping Wu. Fast software ipv6 lookup with neurotrie. *IEEE/ACM Transactions on Networking*, pages 4040–4055, 2024.