Foresight Indexing: Accelerating B+tree Index with Programmable Switches on the Network Path

Feiyu Wang*, Qiuheng Yin†, Yixin Zhang†, Tong Yang*
* School of Computer Science, Peking University, Beijing, China
† School of Software & Microelectronics, Peking University, Beijing, China
wangfeiyu@pku.edu.cn, yinqiuheng@stu.pku.edu.cn, yxzh@alumni.pku.edu.cn, yangtong@pku.edu.cn

***Abstract*—The B+tree indexing scheme is widely employed in file systems and database systems. Modern data centers often adopt a disaggregated architecture, where compute servers and storage servers are deployed on separate machines connected via a network. Storage servers typically rely on B+tree-based indexing. However, with the rapid growth in network bandwidth, the bottleneck has shifted from the network to the indexing, challenging the scalability of these systems. In this paper, we present FOREIN, a novel architecture that offloads part of the storage indexing process to programmable switches within the network path. Specifically, we make three key contributions. First, we develop the PREFIXCOVER algorithm, which converts a B+tree query into a longest prefix match query. This transformation allows partial deployment of B+tree operations in the switches at line rate. Second, we propose a greedy algorithm that dynamically adapts the PREFIXCOVER algorithm to the resource constraints of programmable switches. Third, we design a data plane leveraging programmable switch capabilities, ensuring consistency between servers and switches with minimal overhead and minimal device modifications. We implement FOREIN on a testbed and conduct extensive experiments. Results demonstrate that FOREIN improves the throughput of B+tree-based storage servers by an average of 1.2 times. The source code is publicly available on GitHub [1].**

***Index Terms*—B+tree index, LPM, in-network computing, programmable switches**

I. Introduction

The B+tree [2], [3], [4], [5] is a widely-used efficient index scheme, particularly in disaggregated data centers, where compute and storage servers are connected via a network. With cloud computing and faster networks, the performance bottleneck has shifted from networking to indexing [6], [7], [8]. As bandwidth scales, CPUs struggle to handle B+tree operations at line rate. While technologies like Remote Direct Memory Access (RDMA) help reduce CPU overhead [9], [10], indexing remains a critical bottleneck since the overhead of executing B+tree operations and managing memory access makes achieving high performance difficult. In particular, even when the entire index resides in memory, each lookup still incurs multiple dependent memory accesses starting from the root. Several studies have explored ways to mitigate the CPU bottleneck and enhance system performance, including Memcached [11], SwitchKV [12], NetCache [13], etc. Although these hardware-based solutions offer substantial speed improvements, their high cost remains a barrier to large-scale deployment.

This work was supported in part by the National Key Research and Development Program of China under Grant 2024YFB2906601, in part by the National Natural Science Foundation of China (NSFC) under Grant 62372009.
Corresponding author: Tong Yang (yangtong@pku.edu.cn).

The emergence of programmable switch ASICs has made it possible to customize network functions and offload selected workloads from servers into the network itself [13], [14], [15], [16], [17], [18]. In particular, offloading the B+tree index to programmable switches is a promising direction for accelerating data center operations. Beyond programmable switches, many commodity switches and routers can be configured for simple user-defined packet processing, yet their hardware resources—especially longest prefix match (LPM) tables—are often underutilized. With the growing adoption of rack-scale disaggregated storage architectures [19], top-of-rack (ToR) switches are in a unique position to offload computation from storage servers, as their routing logic is relatively simple and stable. By leveraging both the programmability and idle resources of these switches, it becomes feasible to push part of the indexing workload directly into the network fabric. Furthermore, the inherent hierarchical structure of the B+tree naturally decomposes its operations into independent subtasks, making it particularly amenable to partial offloading. Importantly, partial offloading enables a design point between two extremes: caching-based approaches that remain purely endpoint-centric, and fully in-network designs that embed data structures directly into switches. This middle ground allows the network to provide lightweight traversal foresight without replicating or relocating the B+tree itself. In summary, motivations for offloading the B+tree index to the network are:

- **Indexing Bottleneck.** Index lookups are a major performance bottleneck in large-scale storage systems, especially for read-dominant and skewed workloads where a small fraction of keys accounts for most accesses.
- **Programmable Switches.** Modern switches support flexible, programmable packet processing, enabling new in-network computing paradigms.
- **Spare LPM Resources.** Many ToR switches have significant unused LPM capacity, which can be repurposed for indexing tasks.
- **Layered B+tree Structure.** The balanced, hierarchical organization of the B+tree allows its traversal to be decomposed and partially executed within the switch.

We present FOREIN (**Fore**sight **In**dex), a B+tree-based

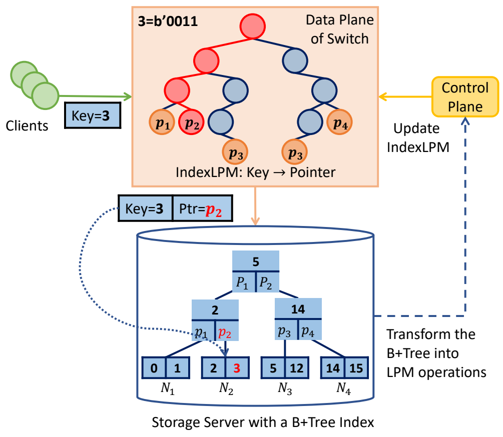


Fig. 1. Overview of FOREIN. FOREIN utilizes PREFIXCOVER to transform B+tree operations into LPM lookups, leveraging INDEXLPM in the switch's data plane, which is managed by the control plane. The INDEXLPM maps each query key to a pointer that indicates the location of a specific B+tree node along the network path. For example, consider a client issuing a query with key $= 3$. The switch appends its pointer p_2 in the packet. Upon receiving the packet, the storage server initiates the B+tree lookup directly from the node N_2 as indicated by p_2, instead of the root node.

indexing scheme that leverages programmable switches to offload selected B+tree traversal steps, rather than relocating the index itself, thereby accelerating server-side performance. Figure 1 provides an overview of FOREIN. To enable efficient offloading, we first propose the PREFIXCOVER algorithm, which transforms B+tree operations into LPM lookups. We then introduce a data structure called INDEXLPM to maintain the mapping from B+tree keys to their corresponding data node pointers. Moreover, we propose a greedy algorithm that adapts to the resource constraints, ensuring efficiency across diverse hardware configurations. FOREIN implements a packet-processing pipeline that utilizes INDEXLPM to handle B+tree queries. The pipeline encapsulates each query with a pointer to a node along the key's search path in the original B+tree, allowing the server to begin the search from a deeper data node rather than the root, thus reducing computational overhead. In summary, FOREIN accelerates B+tree indexing by offloading the B+tree into the network path via PREFIXCOVER and INDEXLPM, alleviating the CPU bottleneck in B+tree-based storage systems. We implement FOREIN on a testbed and evaluate its performance, demonstrating an average throughput improvement of 1.2 times for the B+tree index.

In summary, we make the following contributions:

- We propose the PREFIXCOVER algorithm, which transforms B+tree sub-operations into LPM operations and adapts to varying LPM resources available in programmable switches.
- We present FOREIN, a B+tree-based index architecture that leverages programmable switches to accelerate B+tree operations on the network path.
- We implement FOREIN on a testbed and evaluate its performance, demonstrating an average throughput improvement of 1.2 times for the B+tree index scheme.

We introduce the background and motivation of this work in Section II. Section III present the overview of PREFIXCOVER. Section IV details the PREFIXCOVER algorithm and the proposed greedy algorithm for adapting PREFIXCOVER to varying LPM resources in network data planes. The system design of FOREIN is described in Section V. Implementation details are provided in Section VI, and performance evaluations of both PREFIXCOVER and FOREIN are presented in Section VII.

II. BACKGROUND

A. Programmable Switches

The advent of programmable switches, such as Barefoot Tofino [20], [21], [22], [23], [24], has fundamentally transformed modern network infrastructure. Central to this transformation is P4 [25], a protocol-independent language that enables developers to define custom packet processing logic in a hardware-agnostic manner. By targeting any network device that supports the P4 runtime [26], P4 decouples software from hardware constraints and accelerates innovation across the networking stack. Programmable switches implement user-defined network functions directly in hardware, while improving the adaptability of network systems. P4-based switches typically adopt a pipeline of Match-Action Units (MAUs), where each stage processes packets according to rules set by the control plane and compiled from P4 programs. This architecture allows rapid, dynamic updates to the data plane's behavior without physical hardware changes. Beyond basic routing and forwarding, programmable switches drive advances in network monitoring [27], [28], [29], [30], security [31], [32], and application-specific optimizations [33], [34], [35], [36], making them a cornerstone of next-generation, flexible network architectures.

B. B+tree

B+tree is a self-balancing tree data structure. As a widely adopted index structure, it underpins storage systems such as NTFS directories [37], [38], and database indexes in InnoDB (MySQL), SQLite, and SQL Server. A B+tree organizes data into a root, internal nodes, and leaf nodes, with all records stored at the leaves. Each node holds multiple keys—internal nodes guide the search, while leaf nodes store actual records—resulting in a high fan-out and shallow tree depth. Lookup, insertion, and deletion always begin at the root and follow a unique path to a leaf, with the tree rebalancing as needed through node splits and merges. The high fan-out and balanced nature of B+tree enable efficient index traversal across a wide range of storage architectures. As illustrated in Figure 2(a), a key feature of B+tree is its layered and balanced design: all data resides at uniform-depth leaf nodes, and any search follows a deterministic root-to-leaf path. This layered and deterministic structure allows indexing to begin at any intermediate node along the search path, enabling flexible traversal strategies for performance optimization.

III. OVERVIEW OF FOREIN

This paper proposes using programmable switches and their sparse LPM resources to accelerate B+tree-based storage servers by offloading part of the index to the data plane. Figure 1 shows the overview of FOREIN, where server-side B+tree-based key-value stores are enhanced by offloading index operations to programmable switches, reducing CPU load and alleviating indexing bottlenecks. FOREIN extends standard L2/L3 routing with a custom module, INDEXLPM, which supports B+tree index offloading while remaining compatible with existing protocols. The switch maintains INDEXLPM, an extended LPM table mapping keys to pointers, intentionally limiting data-plane logic to LPM lookups for line-rate processing. These entries are installed during initialization by a control-plane algorithm, PREFIXCOVER, which computes and loads the necessary mappings. The FOREIN protocol supports Get, Put, and Delete operations. Upon receiving a packet, the switch performs a INDEXLPM lookup and appends a pointer to a relevant internal or leaf node, bypassing the root and reducing memory access overhead. By leveraging in-network processing, FOREIN effectively mitigates indexing bottlenecks and improves the performance of B+tree-based storage systems.

IV. ALGORITHM: PREFIXCOVER

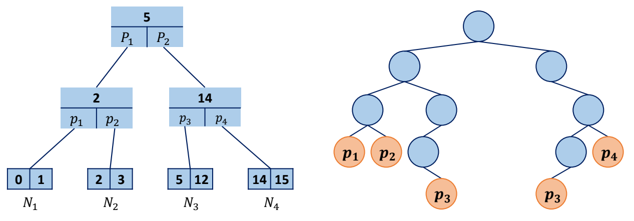
This section introduces PREFIXCOVER, an algorithm for converting B+tree indexes into LPM tries to enable efficient offloading into programmable switch data planes. The key idea is to aggregate the key intervals of B+tree nodes into a minimal set of LPM prefixes, maximizing resource efficiency. A central challenge is the limited and variable LPM capacity of switch data planes. PREFIXCOVER addresses this with two components: (1) a compact interval-to-prefix conversion, and (2) a resource-aware, adaptive selection strategy detailed in Section IV-B.

A. The Basic Version

1) Basics of B+tree: A B+tree is a balanced search tree widely used in storage systems. All data records are stored in the leaf nodes, which are doubly linked for efficient range queries, while internal nodes serve only as navigational keys. Given a query key k, the goal is to rapidly identify the unique leaf node whose interval $[I_{\mathrm{B}}, I_{\mathrm{E}}] = [k_{\min}, k_{\max}]$ contains k. To enable offloading to a switch that supports Longest Prefix Match (LPM), we map each interval to a set of LPM prefixes.

2) Interval-to-Prefix Conversion: Let keys be represented as W-bit unsigned integers. A prefix V/L denotes the set of all keys x whose L most significant bits match those of V, i.e., $(x \,\&\, \mathsf{mask}(L)) = (V \,\&\, \mathsf{mask}(L))$, where $\mathsf{mask}(L)$ is a W-bit mask with the highest L bits set to 1 and the remaining $W - L$ bits set to 0. For example, when $W = 4$, the prefix $5/4$ (binary $0101/4$) matches only the integer 5, whereas $5/3$ (binary $0101/3$) matches the set $\{4, 5\}$.

Given a leaf interval $[I_{\mathrm{B}}, I_{\mathrm{E}}]$, our goal is to cover all integers in this range with the minimal set of such prefixes. The conversion algorithm works as follows: The algorithm computes the minimal set of LPM prefixes that precisely cover a given integer interval $[I_B, I_E]$. It operates iteratively, maintaining a queue of candidate prefixes. At each iteration, the algorithm



(a) Original B+tree. (b) LPM trie.

Fig. 2. Transformation from B+tree intervals to minimal LPM prefix covers.

selects a prefix from the queue and determines whether its value range is entirely contained within the target interval. If so, this prefix is added to the cover set. If not, the prefix is split into its left and right child prefixes by increasing the prefix length, and any child whose range intersects the target interval is enqueued for further examination. This process continues until the queue is empty, at which point the algorithm returns the set of selected prefixes as the smallest possible prefix cover for the interval. The approach effectively decomposes the interval into the largest possible binary-aligned subranges at each step, ensuring both correctness and minimality of the resulting cover.

Concrete Example: Consider node N_3 with the interval $[5, 12]$. Its binary representation is $[0101_2, 1100_2]$. The smallest set of prefixes that cover this range is:

- $5/4$ (0101_2, length 4): covers 5
- $6/3$ (0110_2, length 3): covers from 6 to 7
- $8/2$ (1000_2, length 2): covers from 8 to 11
- $12/4$ (1100_2, length 4): covers 12

Thus, $[5, 12] \rightarrow \{5/4, 6/3, 8/2, 12/4\}$. Each prefix can be seen as a mask rule, just as in network routing.

3) Eliminating Hollow Prefixes: Not every generated prefix corresponds to actual data keys; some are "hollow prefixes" that cover gaps where no real key exists (e.g., in a sparse key space). To avoid unnecessary LPM entries, we filter out all hollow prefixes, retaining only those that map to actual data. For the above example, if only keys 5 and 12 exist, $6/3$ and $8/2$ are hollow and discarded. Only $5/4$ and $12/4$ are kept and installed as LPM entries in the switch, each acting as a pointer to the corresponding leaf node. This optimization greatly reduces resource usage and mitigates the risk of prefix explosion under limited TCAM capacity.

4) Summary: By mapping each leaf interval in the B+tree to a minimal set of LPM prefixes and discarding hollow prefixes, we can efficiently offload index lookups to programmable switches using minimal hardware resources. Figure 2 visualizes this conversion.

B. Adapting to Available Resources

Although offloading all leaf nodes of the B+tree to the switch would provide the fastest lookups, this is often infeasible due to the limited LPM table size on programmable switches. Therefore, our goal is to select a subset of nodes to offload, maximizing acceleration while respecting hardware resource constraints.

1) Problem Formulation: Formally, we model the problem as follows:

- Each B+tree node n_i is associated with a *weight* w_i (reflecting its importance, e.g., access frequency) and a *cost* c_i (the number of LPM entries required to offload this node).
- The switch provides a total LPM resource budget M (the maximum number of LPM entries available).
- Let S denote the set of nodes selected for offloading.

If a query's target key falls under a node in S, the switch can redirect the query directly to this node, skipping higher levels of the tree. Otherwise, the query must start from the root. We seek to minimize the **average memory accesses per query** (AMA), defined as:

$$\textbf{AMA} = \sum_{L \in \text{leaves}} w_L D_L,$$

where w_L is the weight of leaf node n_L and D_L is the number of nodes traversed from n_L up to its closest ancestor in S (including both endpoints). If no ancestor of n_L is in S, D_L equals the depth of n_L. The constraint is:

$$\sum_{n_j \in S} c_j \leq M.$$

Thus, the optimization problem is:

$$\min_{S \subseteq \{n_1, n_2, \ldots, n_N\}} \textbf{AMA} \quad \text{s.t.} \quad \sum_{n_j \in S} c_j \leq M.$$

2) Greedy Algorithm: As the above problem is combinatorial and intractable for large trees, we design a greedy algorithm for practical deployment.

1) **Initialization:** Start with $S = \{\text{root}\}$. At this stage, every query must traverse the full height of the tree.
2) **Iterative Selection:** At each step, among all nodes not in S, select the node n^* whose offloading yields the largest reduction in $\textbf{AMA}$.
3) **Update:** Add n^* to S, update the weights and path lengths for affected nodes, and remove n^*'s nearest ancestor (other than root) from S to prevent redundant offloading on the same path.
4) **Termination:** Repeat until adding another node would exceed the resource limit M.

The Algorithm 1 summarizes our approach. This greedy method ensures that, at each step, we make the most effective use of limited LPM resources and incrementally build an offload set providing a substantial reduction in query latency.

V. System Design of ForeIn

This section describes the system design of ForeIn. To integrate PrefixCover with programmable switches, we implement the ForeIn protocol at the application layer, above UDP or TCP (Section V-A). The system workflow is outlined in Section V-B. ForeIn supports three primary operations: Get, Put, and Delete. We detail the Get operation in Section V-C, and discuss dynamic B+tree updates in Section V-D.

Algorithm 1: Resource-Constrained Offloading

```
Input: B+tree $T$ with node weights $w_i$, costs $c_i$,
       resource limit $M$
Output: Selected offload set $S$
1  $S \leftarrow \{\text{root}\}$;
2  $C \leftarrow c_{\text{root}}$;
3  while $C \leq M$ do
4      $n^* \leftarrow$ node (not in $S$) with largest AMA reduction;
5      if $C + c_{n^*} > M$ then
6          break
7      Add $n^*$ to $S$;
8      Remove $n^*$'s nearest ancestor from $S$ (if not root);
9      $C \leftarrow C + c_{n^*}$;
10     Update affected weights and path lengths;
11 return $S$
```

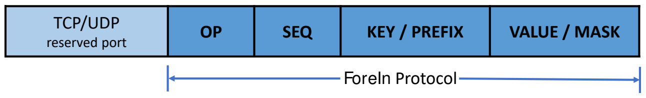


Fig. 3. Packet format of the ForeIn protocol.

A. Network Protocol

In our architecture, the B+tree-based key-value store operates as an application-layer service in a typical client-server scenario. To enable switch-level acceleration, the ForeIn protocol is designed as an application-layer protocol compatible with both UDP and TCP. This ensures seamless deployment in diverse network environments, leveraging UDP's speed or TCP's reliability as needed. For effective packet processing, ForeIn uses a reserved port number so that programmable switches can easily identify and handle its packets.

1) Packet Format: As shown in Figure 3, each ForeIn packet consists of four fields: OP, SEQ, KEY, and VALUE.

- OP: Operation code, indicating Get, Put, Delete, Install, or Reply.
- SEQ: Sequence number, used when transferring large values.
- KEY, VALUE: The query's key and value. For Get and Delete, VALUE is empty.

The reply packet format is identical to the query format, with OP set to Reply. Additionally, for updates between the server and controller, an Install operation allows dynamic modification of the LPM table, with KEY and VALUE indicating the prefix and its length.

2) ForeIn Switches: Integrating ForeIn with existing network stacks requires only minimal switch changes. The main enhancement is recognizing and processing ForeIn protocol packets, while preserving standard L2/L3 forwarding for all other traffic. Upon receiving a packet, the switch checks whether it is a ForeIn query. If so, it performs an LPM lookup on the KEY field, and appends a pointer to the matched B+tree node. Otherwise, the packet is forwarded using standard routing. This design enables the switch to provide "foresight"—by offloading part of the index lookup,

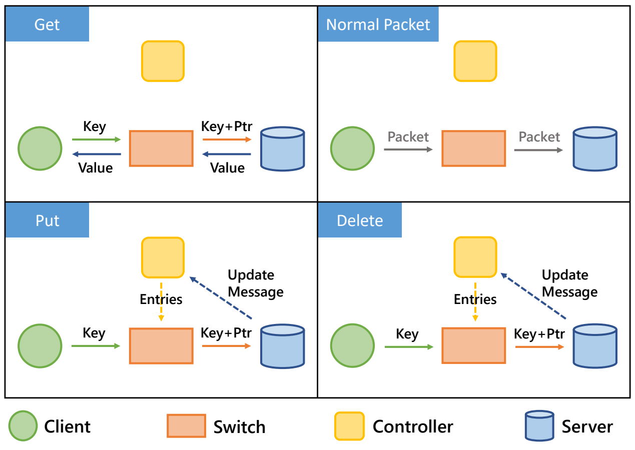


Fig. 4. FOREIN workflow.

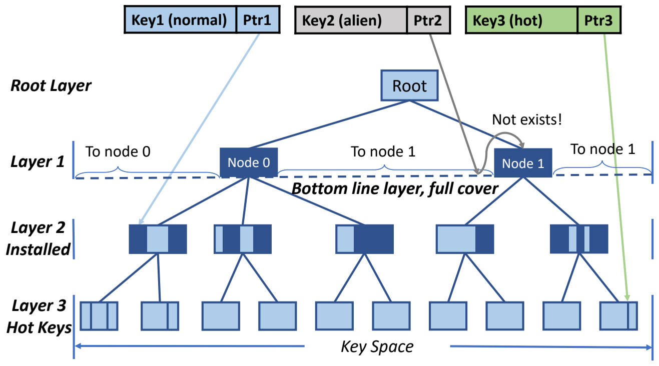


Fig. 5. Lookup process for different types of keys in FOREIN. Dark blue: covered prefixes; Light blue: hollow/uninstalled zones.

the server can directly jump to a deeper node in the B+tree, reducing memory accesses and improving performance.

B. *Workflow of* FOREIN

The overall workflow is illustrated in Figure 4. Processing proceeds as follows:

- If the packet is not a FOREIN packet, it is forwarded normally by the switch using L2/L3 routing.
- If the packet is a FOREIN packet, the switch consults its LPM table:
  - Get: The switch appends a pointer to the relevant B+tree node, and the server uses it to efficiently locate and return the value. The reply (OP set to Reply) is forwarded as a normal packet.
  - Put/Delete: The switch appends the pointer, and the server performs the update on the B+tree. The server then computes if the LPM table needs updating and, if so, notifies the controller, which installs the necessary changes on the switch.

While Figure 4 depicts the server and controller as separate entities, they can be integrated on a single machine for ease of deployment.

C. *Lookup of* FOREIN

When a FOREIN switch receives a query, it appends a pointer indicating the target node where the key is likely to reside. The server then extracts this pointer, accesses the specified node, and performs a key lookup starting from there. If the key exists, INDEXLPM ensures direct access to the correct node. However, due to omitted hollow prefixes, keys that are not in the B+tree ("alien keys") may be routed to nodes where they do not belong. To guarantee correctness and minimize unnecessary lookups, we introduce two complementary methods.

1) Range Check and Fallback: Upon receiving a query, the server checks if the queried key falls within the range of the target node. If so, the lookup proceeds normally. If not (i.e., the key falls into a hollow prefix or is not present), the server performs a fallback: it restarts the search from the root of the B+tree. This mechanism ensures correctness by redirecting alien keys to a standard full traversal, at the cost of a single extra memory access.

2) Bottom-line Guarantee: To further reduce fallback overheads, we propose a "bottom-line guarantee": all nodes from an upper layer (e.g., the first internal layer) are installed in the switch, ensuring complete coverage of the key space at that layer. For any key, if a lower-layer prefix does not match, the query is redirected to the appropriate node at this upper layer instead of the root. Since upper layers have few nodes, this incurs little memory overhead but eliminates unnecessary root traversals for alien keys.

3) Hybrid Policy Example: Figure 5 shows how hybrid policies improve lookup efficiency:

- **Root Layer:** No prefixes installed. Root serves only as a final fallback for alien keys.
- **Layer 1 (Bottom-line):** Prefix covers are created for all nodes, and gaps are merged with adjacent nodes to ensure full key-space coverage. Thus, most alien keys are redirected here, not to the root, for an efficient secondary check (e.g., *key* 2).
- **Layer 2 (Main Index):** The main prefix cover is installed here, omitting hollow prefixes to save space. Valid keys (dark blue) are efficiently directed to their target node (e.g., *key* 1). Alien keys in hollow zones are redirected up to Layer 1.
- **Layer 3 (Hot Keys):** Hot leaf nodes, identified by access frequency, are directly offloaded. Queries for hot keys (e.g., *key* 3) are completed in a single memory access, optimizing for common cases.

This layered approach, combining range checks, bottom-line guarantees, and targeted hot key offloading, ensures both correctness and high efficiency in FOREIN lookups with minimal resource overhead.

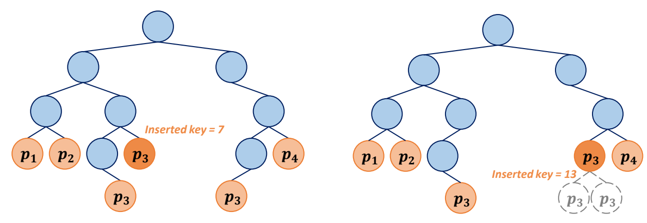
D. *Update and Coherence*

Put and Delete operations in FOREIN may trigger updates to both the B+tree and the switch's INDEXLPM table. To ensure correctness, updates are handled in two coordinated phases:

- **Phase I:** The server updates its local B+tree and generates the new LPM entries required for the switch.
- **Phase II:** The server sends these table entries to the controller, which applies them to the switch data plane. Coherence between the B+tree and IndexLPM must be preserved during this process.

1) Phase I: Update Procedure: During Phase I, the server first modifies the B+tree, then determines the necessary changes to IndexLPM. As summarized in Table I, different scenarios arise based on the operation type and whether the B+tree structure changes.

- **Put:**
  - If the key exists, only the value is updated.
  - For new keys: *(i)* If matching an installed prefix, do nothing. *(ii)* If matching a hollow prefix, install the missing prefix (Figure 6(a)). *(iii)* If the node's key range changes, install new prefixes and perform prefix aggregation if possible (Figure 6(b)). *(iv)* If the insertion causes node splitting, recalculate prefix covers for affected nodes.
- **Delete:**
  - If the key does not exist, no action is taken.
  - If no structural change, logic is similar to insertion (possibly triggers prefix aggregation).
  - If deletion leads to node re-balancing or merging, recalculate prefix covers for affected nodes to maintain consistency.



(a) Install a hollow prefix. (b) Prefix aggregation.

Fig. 6. Examples of updates.

2) Phase II: Coherence Management: Once updates are sent to the controller, their installation in the switch may be delayed. To ensure lookup correctness during this interval, ForeIn employs the following strategies:

- **Hollow prefix installation:** Before installation completes, keys in these ranges are temporarily routed to their parent node, requiring one extra lookup. This guarantees correctness at minimal performance cost.
- **Prefix aggregation:** New shorter prefixes are installed before old ones are removed. The server tracks "invalid" nodes during the transition, ensuring all queries are either handled directly or safely redirected.
- **Prefix cover recalculation (for re-balance/merge/split):** New prefixes are always installed first.
  - *Re-balance:* Keys may be briefly misdirected, but bidirectional links allow correct redirection to sibling nodes.
  - *Merge:* Invalid pointers are tracked, with queries redirected until confirmation of update completion.
  - *Split:* Original and new nodes are adjacent; sibling links enable correct lookup if an outdated pointer is used.

Overall, ForeIn ensures strong coherence and correctness during all update operations, with only minor and temporary overhead.

VI. Implementation

We implement a prototype of ForeIn to validate its functionality and performance. The prototype consists of four main components: the programmable switch data plane, a control plane agent, a B+tree-based server, and a benchmarking client. The server and client run on separate machines connected to different switch ports, with forwarding tables set up to enable communication.

- **Switch Data Plane:** The data plane is programmed in $P4_{16}$ targeting Barefoot Tofino hardware. An LPM table is compiled into the switch; it matches on 8-byte keys in ForeIn protocol packets. For matched keys, the switch attaches an 8-byte pointer to the packet using a "hit" action; unmatched keys receive a default "NoAction." Pointers use 8 bytes to align with the server's x86-64 architecture.
- **Control Plane Agent:** The agent is a Python-based gRPC server serving as the switch control plane. It receives prefix entries and pointers via an RPC interface, translating them into the format required by the Tofino switch. The agent ensures consistency between the switch LPM table and the B+tree database, and supports asynchronous updates for high throughput.
- **Server:** The server is a DPDK application written in C++. It maintains a B+tree for key-value storage, with keys as 8-byte unsigned integers (`uint64_t`), but configurable for other sortable types. The server parses incoming ForeIn protocol packets, processes queries or updates using the B+tree, and sends replies. For Put and Delete operations that change the B+tree structure, the server determines affected prefixes and triggers gRPC calls to the agent to update the LPM table accordingly.
- **Client:** The client is a DPDK application implemented in C. It generates and transmits ForeIn protocol requests to the server and records replies. The client module also tracks packet transmission and reception to measure real-time throughput.

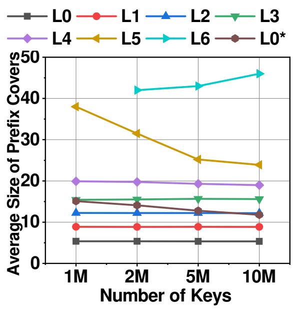
This modular design allows us to evaluate the end-to-end performance of ForeIn in a realistic setup, including control plane coordination and dynamic updates.

A. Deployment Considerations and Limitations

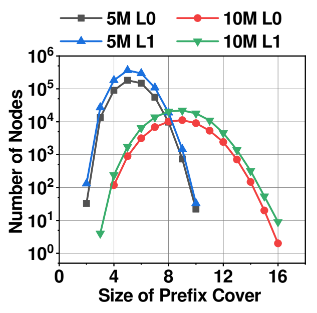
a) Coexistence with Routing Rules.: ForeIn leverages LPM tables that are already widely used for routing and forwarding. In practice, ForeIn operates in a logically separate namespace and does not interfere with existing L2/L3 forwarding rules. Only a fraction of the available LPM entries is allocated to IndexLPM, and the priority of routing rules

TABLE I
DIFFERENT CASES OF UPDATE.

OP	B+tree structure does not change.				B+tree structure changes.	
	Hit an installed prefix.	Hit a hollow prefix	Range change	Nodes re-balance	Nodes merge	Nodes split
Put	Do nothing.	Install the hollow prefix.	Prefix aggregation	N/A	N/A	Re-calculate prefix covers.
Delete	Do nothing.	N/A	Prefix aggregation	Re-calculate prefix covers.	Re-calculate prefix covers.	N/A

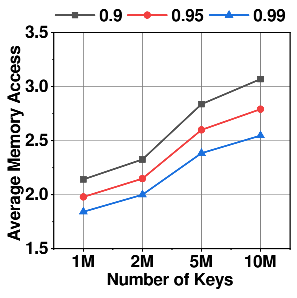


(a) Average prefix cover size of different layers.

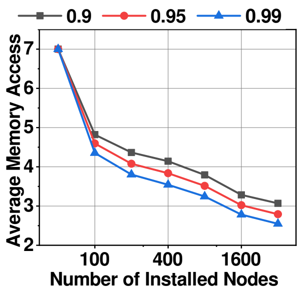


(b) The distribution of prefix cover size.

Fig. 7. LPM resource usage of PREFIXCOVER



(a) AMA vs. dataset size.



(b) AMA vs. # installed nodes.

Fig. 8. AMA Improvement with the proposed greedy algorithm.

remains unaffected. This design allows FOREIN to be incrementally deployed without modifying the switch forwarding pipeline.

b) Multi-Tenancy and Isolation.: In multi-tenant environments, INDEXLPM entries can be scoped per service or per application by reserving disjoint prefix ranges or logical tables. Since FOREIN does not modify packets across tenants and only appends application-layer hints, it preserves isolation guarantees provided by the underlying network.

c) Encrypted Traffic and Protocol Compatibility.: FOREIN operates at the application layer and does not require access to encrypted payloads. The switch only parses a small, explicitly defined header field that is exposed to the network, making FOREIN compatible with encrypted transport protocols and existing security mechanisms.

d) Dynamic Workloads and Applicability.: As shown in Section VII, frequent updates introduce additional control-plane overhead for maintaining prefix consistency. Therefore, FOREIN is most effective for read-dominant workloads with moderate update rates, which are common in key-value serving and database indexing scenarios. For highly write-intensive workloads, the benefits of offloading may be reduced due to update and coherence costs.

VII. EVALUATION

A. Methodology

1) Testbed: We evaluate FOREIN on a local testbed with a programmable 100GbE Barefoot Tofino-based network switch and two server machines (one as a FOREIN server and the other as a client). Each server machine has a 12-core (24-thread) CPU (Intel Xeon Silver 4116 CPU @ 2.10GHz), 32 GB 2400 MHz DDR4 memory, and a 16 MB L3 cache. Each server machine has a 100 Gbps Mellanox NIC connected to the switch. The version of DPDK installed on the server and client is 20.11.8 LTS.

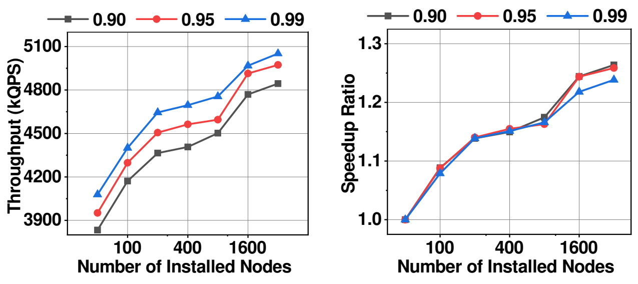
2) Workloads: We generate workloads where keys satisfy the Zipf distribution. The values are not important, so random numbers are used. The number of times each key appears in the workload is its frequency. The proposed greedy algorithm uses these key frequencies to determine whether a node in B+tree should be offloaded, as Zipfian access patterns are commonly observed in production key-value stores and database indices. Zipfian access patterns are widely observed in production key-value stores and database indices, where a small fraction of keys accounts for the majority of requests. We therefore focus on varying the Zipf parameter to evaluate FOREIN under representative skewed workloads. Longer keys or composite keys can be supported by hashing or encoding them into fixed-length prefixes before applying PREFIXCOVER, without changing the algorithm.

3) Metrics: We use queries per second (QPS) as a metric of throughput[1]. The ratio of query throughput with prefixes offloaded in the switch to the throughput of the same system with in-network offloading disabled is called **speedup ratio**. We used the same workload in both cases when calculating the speedup ratio. The speedup ratio directly shows the acceleration effect of FOREIN.

B. Performance of PREFIXCOVER

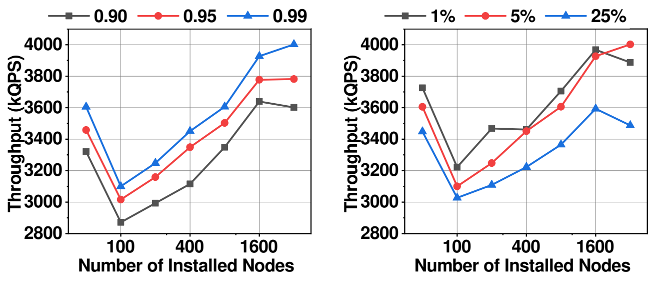
1) The LPM resource: We evaluate the average LPM resource usage per node in different layers of the B+tree, as

[1]kQPS stands for 1000 of queries per second



(a) Throughput vs. skewness. (b) Speedup Ratio vs. skewness.

Fig. 9. Performance of FOREIN under static workloads.



(a) Throughput vs. skewness. (b) Throughput vs. dataset size.

Fig. 10. Performance of FOREIN under dynamic workloads.

indicated by the size of the prefix cover. Figure 7(a) shows the average size of prefix covers for nodes in various layers of the B+tree. The layers are numbered from bottom to top, with the deepest layer labeled as L0 and the root node assigned the highest number, determined by the number of keys. As the dataset size increases, the average prefix cover size decreases across all layers, except for the L6 layer. L6 is the root layer, which consists of only one node and is therefore sensitive to key distribution. Note that L0* refers to the case where the average size of the prefix cover is calculated without discarding hollow prefixes. The results indicate that discarding hollow prefixes can significantly reduce the resource consumption in L0, compared to L0*. This reduction is also observed in other layers, though for simplicity, we omit the data for these layers in the figure. Figure 7(b) illustrates the distribution of prefix cover sizes. The labels indicate the dataset size and the corresponding B+tree layer. The prefix cover sizes of nodes within the same layer approximately follow a normal distribution. Based on this observation, we use the average LPM entries to estimate the cost of offloading a node, as discussed in Section IV-B.

2) *Evaluation of the proposed greedy algorithms:* We evaluate the greedy algorithm designed to adapt to different available LPM resources in Algorithm 1. The experimental results demonstrate that the proposed greedy algorithm effectively handles the number of installed nodes and behaves as expected. Figure 8(a) shows how the Average Memory Access (AMA) varies with the dataset size, while the number of installed nodes is fixed at 3200. As the dataset size increases, the AMA also increases due to the growth in the number of B+tree nodes. Under the same dataset size, more skewed workloads result in lower AMA, as the installed nodes can accommodate more hot keys, thereby reducing memory access. Figure 8(b) illustrates how AMA in B+tree operations varies with the number of nodes installed in the INDEXLPM, under key distributions characterized by different Zipf parameters. The first data points represent AMA without any offloading. The Zipf parameter controls the skewness of the key distribution, with higher values indicating more skewed workloads where a small subset of keys is accessed more frequently. For a moderately skewed distribution (Zipf $= 0.9$), AMA decreases gradually as more nodes are installed, reflecting the fact that a larger proportion of frequently accessed keys are accommodated, thus reducing tree traversal depth. For a highly skewed distribution (Zipf $= 0.99$), AMA decreases sharply because most queries target a small subset of keys, which are resolved with minimal memory access due to their placement in offloaded nodes. The experimental results highlight the effectiveness of FOREIN in optimizing AMA, particularly for highly skewed key distributions.

C. *Performance of* FOREIN

1) *Static Workloads:* We use static workloads in this experiment, where the client only sends Get requests to the server, without any modification requests such as Put or Delete. The B+tree is pre-constructed, and prefix entries are offloaded to the switch during the server's initialization phase. Once initialized, the offloaded prefix entries in the switch remain unchanged throughout the experiment.

A highly skewed workload, where most queries target a small subset of keys, benefits from offloading hot keys to the switch. This reduces the number of memory accesses on the server, significantly improving throughput. The dataset size is set to 1M keys. Figure 9(a) shows the throughput under workloads with different levels of skewness. Since the baseline performance varies, the speedup ratio is also shown in Figure 9(b). As the number of installed nodes increases, the throughput also improves. However, the relative speedup decreases under highly skewed workloads. This is likely due to the local cache of storage servers. In highly skewed workloads, the local cache may improve the baseline performance due to spatial locality, which results in fewer memory accesses. Consequently, the speedup ratio decreases. In summary, the experimental results demonstrate that FOREIN performs better under skewed workloads, with an average speedup ratio of approximately 1.2x.

2) *Handling Dynamics:* In this experiment, dynamic workloads are used. In addition to Get, the client sends Put and Delete requests. The B+tree and the offloaded prefixes may change as the server processes FOREIN protocol packets. To handle updates, the server reloads outdated prefixes and offloads new ones by calling the agent. An update (Put or Delete) is considered successful only after the prefix is successfully offloaded to the switch. Figure 10(a) shows

the performance of FOREIN under workloads with different skewness. When the number of installed nodes is small, FOREIN does not outperform the baseline due to the high overhead of updates and coherence. As the number of installed nodes increases, the benefits of offloading begin to outweigh the overhead, resulting in an average speedup of 1.05x. Figure 10(b) shows the performance of FOREIN under workloads with different dynamic ratios. A higher ratio of dynamic operations leads to lower performance for FOREIN.

Although the observed throughput improvements are moderate, they are achieved without relocating the index, modifying the server-side B+tree structure, or introducing specialized hardware. Even under dynamic conditions, FOREIN does not degrade baseline performance, providing a safe optimization envelope. As such, FOREIN represents a lightweight and orthogonal optimization that incrementally reduces index traversal cost and can be combined with existing acceleration techniques.

D. Hardware Resource Consumption

TABLE II
RESOURCES USED BY FOREIN IN TOFINO.

Resource	Usage	Percentage
Exact Match Input xbar	2	0.13%
Ternary Match Input xbar	40	5.05%
VLIW Instructions	6	1.56%
SRAM	19	1.98%
TCAM	98	34.03%
Hash Bits	10	0.20%

The resource usage of FOREIN, when 3200 nodes are installed, is summarized in Table II. FOREIN primarily utilizes TCAM and Ternary Match Input xbar. With 3,200 nodes installed, approximately 25,000 out of the 73,000 available entries in the LPM table are consumed, accounting for about 34% of the total TCAM resources. Regarding other resources, excluding Stateful ALUs, FOREIN's usage remains below 6%. Overall, the resource consumption of FOREIN is considered moderate, especially considering its performance improvements with a relatively small number of installed nodes. Moreover, with the availability of additional TCAM resources, there is potential for further performance enhancements.

VIII. RELATED WORK

A. In-Network Computing

The rise of programmable switches has enabled extensive research in in-network computing, which exploits in-switch data processing to boost system performance. SwitchKV [12] combines high-performance cache nodes with lightweight backend nodes to balance load, tracking cached keys, and routing requests at line rate. NetCache [13] stores key-value pairs directly in switches, significantly improving throughput and latency, though limited by switch resource constraints. IncBricks [14] introduces a hardware-software co-designed caching fabric with basic computing primitives to enhance cache functionality. DistCache [17] offers a distributed cache with provable load-balancing for large-scale systems. Pegasus [15] adds an in-network coherence directory to better handle skewed workloads through selective key replication. Fat-B+tree [18] embeds part of the B+tree into the hierarchical connected programmable switches in the data center network, enabling in-network traversal of tree nodes. Recent work further explores programmable switches for complex operations, such as transactional processing [39], and in-network aggregation for ML workloads [40]. These studies demonstrate the growing versatility of in-network computing. Our proposed FOREIN system builds on this line of work, but differs in how the switch is involved in the indexing process. Instead of embedding or traversing B+tree nodes in the data plane, FOREIN leverages prefix-based lookups to provide lightweight traversal hints, allowing the server to resume the search from an intermediate node while keeping the original B+tree entirely on the server. This design prioritizes partial offloading and resource awareness under practical switch constraints.

B. Disaggregated Data Centers

Disaggregated architectures are increasingly adopted in modern, cloud-native data centers. Systems like Aurora [6], Socrates [41], and PolarDB [7], [8] decouple compute and storage nodes, enabling multi-tenant databases with elastic, scalable resources built on shared storage pools. Disaggregated memory architectures have also emerged to improve memory utilization at data center scale, with rack-scale designs enhancing overall efficiency. Meanwhile, RDMA-based tree indexing techniques [9], [10], [18] reduce remote access overhead in distributed memory systems. These systems build multi-tenant database services on top of a shared storage pool, achieving improved elasticity and independent scalability for storage and compute resources. Our proposed FOREIN system complements these approaches by accelerating B+tree indexing in disaggregated environments and reducing RDMA operations, particularly for read-dominant and skewed workloads, where repeated index traversal remains a performance bottleneck.

IX. CONCLUSION

In this paper, we exploit programmable data planes and sparse LPM resources to accelerate B+tree-based storage servers. We propose an algorithm, PREFIXCOVER, to partially convert the B+tree index into INDEXLPM and offload selected B+tree traversal steps to programmable data planes. Based on PREFIXCOVER, we design a system called FOREIN, which integrates PREFIXCOVER and enhances the end-to-end performance of B+tree-based storage systems. FOREIN incorporates several optimizations to handle foreign keys and manage B+tree dynamics efficiently. We implement FOREIN on a testbed, and evaluation results demonstrate that FOREIN can deliver up to 1.2 times end-to-end performance improvement compared to the baseline solution. The authors have provided public access to their code at [1].

ACKNOWLEDGMENT

The authors would like to express their gratitude to Dr. Cheng Zhang for his invaluable advice and insightful discussions on this work.

References

[1] F. Wang, Q. Yin, Y. Zhang, and T. Yang, "Foresight indexing source code," https://github.com/foresight-indexing/foresight-indexing-code, 2025, available at GitHub.

[2] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, no. 2, pp. 121–137, 1979.

[3] G. Graefe *et al.*, "Modern b-tree techniques," *Foundations and Trends® in Databases*, vol. 3, no. 4, pp. 203–402, 2011.

[4] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013, pp. 302–313.

[5] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, pp. 1–32, 2013.

[6] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon aurora: Design considerations for high throughput cloud-native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1041–1052.

[7] W. Cao, Y. Zhang, X. Yang, F. Li, S. Wang, Q. Hu, X. Cheng, Z. Chen, Z. Liu, J. Fang *et al.*, "Polardb serverless: A cloud native database for disaggregated data centers," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 2477–2489.

[8] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan *et al.*, "Polardb meets computational storage: Efficiently support analytical workloads in cloud-native relational database." in *FAST*, 2020, pp. 29–41.

[9] Q. Wang, Y. Lu, and J. Shu, "Sherman: A write-optimized distributed b+ tree index on disaggregated memory," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1033–1048.

[10] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska, "Designing distributed tree-based index structures for fast rdma-capable networks," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 741–758.

[11] "Memcached key-value store," https://memcached.org/, 2017.

[12] X. Li, R. Sethi, M. Kaminsky, D. G. Andersen, and M. J. Freedman, "Be fast, cheap and in control with switchkv," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 31–44.

[13] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 121–136.

[14] M. Liu, L. Luo, J. Nelson, L. Ceze, A. Krishnamurthy, and K. Atreya, "Incbricks: Toward in-network computation with an in-network cache," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 795–809.

[15] J. Li, J. Nelson, E. Michael, X. Jin, and D. R. Ports, "Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, 2020, pp. 387–406.

[16] Q. Wang, Y. Lu, E. Xu, J. Li, Y. Chen, and J. Shu, "Concordia: Distributed shared memory with in-network cache coherence." in *FAST*, 2021, pp. 277–292.

[17] Z. Liu, Z. Bai, Z. Liu, X. Li, C. Kim, V. Braverman, X. Jin, and I. Stoica, "Distcache: Provable load balancing for large-scale storage systems with distributed caching." in *FAST*, vol. 19, 2019, pp. 143–157.

[18] Y. Zhao, Y. Li, Z. Xu, T. Yang, K. Yang, L. Chen, X. Yao, and G. Zhang, "Fat-b tree: Fast b tree indexing with in-network memory," 2024.

[19] S. Legtchenko, H. Williams, K. Razavi, A. Donnelly, R. Black, A. Douglas, N. Cheriere, D. Fryer, K. Mast, A. D. Brown *et al.*, "Understanding rack-scale disaggregated storage." *HotStorage*, vol. 17, p. 2, 2017.

[20] O. Michel, R. Bifulco, G. Retvari, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–36, 2021.

[21] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 20–25.

[22] G. Siracusano and R. Bifulco, "In-network neural networks," *arXiv preprint arXiv:1801.05731*, 2018.

[23] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 15–28.

[24] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015, pp. 103–115.

[25] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[26] "P4runtime specification." https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html, 2021.

[27] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho, "Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4," in *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, 2019, pp. 1–6.

[28] J. Geng, J. Yan, Y. Ren, and Y. Zhang, "Design and implementation of network monitoring and scheduling architecture based on p4," in *Proceedings of the 2nd International Conference on Computer Science and Application Engineering*, 2018, pp. 1–6.

[29] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 561–575.

[30] Y. Zhang, Z. Liu, R. Wang, T. Yang, J. Li, R. Miao, P. Liu, R. Zhang, and J. Jiang, "Cocosketch: High-performance sketch-based measurement over arbitrary partial key query," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 207–222.

[31] Y. Zhou, J. Bi, Y. Lin, Y. Wang, D. Zhang, Z. Xi, J. Cao, and C. Sun, "P4tester: Efficient runtime rule fault detection for programmable data planes," in *Proceedings of the International Symposium on Quality of Service*, 2019, pp. 1–10.

[32] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE access*, vol. 9, pp. 87 094–87 155, 2021.

[33] H. Zheng, C. Tian, T. Yang, H. Lin, C. Liu, Z. Zhang, W. Dou, and G. Chen, "Flymon: enabling on-the-fly task reconfiguration for network measurement," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 486–502.

[34] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023.

[35] Y. Zhao, W. Liu, F. Dong, T. Yang, Y. Li, K. Yang, Z. Liu, Z. Jia, and Y. Yang, "P4lru: towards an lru cache entirely in programmable data plane," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 967–980.

[36] K. Yang, Y. Wu, R. Miao, T. Yang, Z. Liu, Z. Xu, R. Qiu, Y. Zhao, H. Lv, Z. Ji *et al.*, "Chamelemon: Shifting measurement attention as network state changes," in *Proceedings of the ACM SIGCOMM 2023 Conference*, 2023, pp. 881–903.

[37] G.-S. Cho, "Ntfs directory index analysis for computer forensics," in *2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2015, pp. 441–446.

[38] "New technology file system," https://www.ntfs.com/.

[39] M. Jasny, L. Thostrup, T. Ziegler, and C. Binnig, "P4db-the case for in-network oltp," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 1375–1389.

[40] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtárik, "Scaling distributed machine learning with {In-Network} aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021, pp. 785–808.

[41] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash *et al.*, "Socrates: The new sql server in the cloud," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1743–1756.