NeuroLPM - Scaling Longest Prefix Match Hardware with Neural Networks

Alon Rashelbach
Technion
Israel
alonrs@campus.technion.ac.il

Igor de Paula
Technion
Israel
igordptx@gmail.com

Mark Silberstein
Technion
Israel
mark@ee.technion.ac.il

ABSTRACT

Longest Prefix Match engines (LPM) are broadly used in computer systems and especially in modern network devices such as Network Interface Cards (NICs), switches and routers. However, existing LPM hardware fails to scale to millions of rules required by modern systems, is often optimized for specific applications, and thus is performance-sensitive to the structure of LPM rules.

We describe *NeuroLPM*, a new architecture for multi-purpose LPM hardware that replaces queries in traditional memory-intensive trie- and hash-table data structures with inference in a lightweight Neural Network-based model, called RQRMI. NeuroLPM scales to millions of rules under small on-die SRAM budget and achieves stable, rule-structure-agnostic performance, allowing its use in a variety of applications. We solve several unique challenges when implementing RQRMI inference in hardware, including minimizing the amount of floating point computations while maintaining query correctness, and scaling the rule-set size while ensuring small, deterministic off-chip memory bandwidth.

We prototype NeuroLPM in Verilog and evaluate it on real-world packet forwarding rule-sets and network traces. NeuroLPM offers substantial scalability benefits without any application-specific optimizations. For example, it is the only algorithm that can serve a 950K-large rule-set at an average of 196M queries per second with 4.5MB of SRAM, only within 2% of the best-case throughput of the state-of-the-art Tree Bitmap and SAIL on smaller rule-sets. With 2MB of SRAM, it reduces the DRAM bandwidth per query, the dominant performance factor, by up to 9× and 3× compared to the state-of-the-art.

CCS CONCEPTS

• **Hardware** → **Networking hardware**; • **Networks** → **Packet classification**; • **Computing methodologies** → *Machine learning*.

ACM Reference Format:
Alon Rashelbach, Igor de Paula, and Mark Silberstein. 2023. NeuroLPM - Scaling Longest Prefix Match Hardware with Neural Networks. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3623769

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada*, https://doi.org/10.1145/3613424.3623769.

1 INTRODUCTION

Longest Prefix Matching (LPM) is a versatile primitive with a variety of applications. Given a *rule-set*, where each rule is a binary vector in the form of `prefix:wildcard`, and a query, LPM finds a matching rule with the longest prefix shared with the query. Traditionally being the core mechanism in network systems, e.g., routing [18, 27] and policy-based routing [53], LPM has also been used in load balancing [35], string matching [9] and clustering [23].

Despite years of research and pervasive deployment, LPM engines struggle to keep up with the increasingly challenging requirements of modern applications, as systems must support large, million-scale LPM rule-sets. For example, data center packet forwarding workloads need to handle millions of rules, and the rule-sets are likely to grow [25, 48, 49, 60].

Unfortunately, efficient scaling to large rule-sets is hard. Ternary Content Addressable Memory (TCAM), the go-to solution for implementing hardware LPM [6, 11, 28], does not scale beyond a few thousands of rules due to its high power consumption and large silicon area per rule. State-of-the-art hardware-oriented algorithms, such as SAIL [78] and Tree Bitmap [20], can handle any number of rules, but their performance drops significantly at scale: their data structures spill out of on-chip memory, and the resulting DRAM accesses are largely random, leading to poor cache efficiency and high DRAM access overheads. LPM engines deployed in Network Interface Cards (NICs) suffer from similar issues [36]. Purely software solutions running on a CPU [40, 79] are too slow, and also suffer from poor scaling.

Additionally, today's systems mostly support 32-bit rules and cannot be easily extended to use higher bit-width rules, e.g., 128-bit for IPv6 addresses. Wide rules cause a dramatic increase in space requirements of their internal data structures. For example higher bit-width increases the depth of tries, which in turn leads to an exponential growth in the space requirements as they use sparse bitmaps in the nodes. Thus, they cannot fit in SRAM and require a high number of dependent DRAM accesses during the query, which deteriorates performance significantly. A common solution [7] is to exploit the application-specific structure of the rules leveraging low diversity of the most-significant bits, which then serve as a directory for the lower-bits LPMs. However, they cannot handle more general rules.

At the same time, there is a need for a general, multi-purpose LPM engine that can handle diverse rule-sets for a variety of applications, both in networking and beyond. For example, LPM can be used in string pattern search for network security applications [9], regular expression matching [47], and fast data clustering [23]. The rule-sets from these applications do not have the domain-specific

properties of packet-forwarding rules, i.e., small entropy of target actions associated with each rule [79], or a skewed distribution of highly popular prefix lengths. Unfortunately, existing approaches [17, 20, 24, 27, 34, 78, 79], leverage these very properties to scale to more rules. As a result, their performance is sensitive to the structure of the rules, as they suffer from a worst-case exponential blow-up of their internal data structures. For example, only a few thousand rules are enough to inflate the active working set of Tree Bitmap [20] to tens of MBs, resulting in unacceptable performance.

In this work, we introduce **NeuroLPM**, a multi-purpose LPM hardware engine that can efficiently handle millions of rules with a modest on-chip memory budget, is robust to the rules' characteristics, allows bit-width scaling, and thus suitable for use in diverse LPM applications. A key to NeuroLPM's advantages is the recent *learned data structures* [39] that it employs instead of traditional tries and hash tables. Our work builds upon a *Range-Query Recursive Model Index* (RQRMI) data structure [56] which learns LPM rules by training a model, and then can quickly find the matching rule for a given input via inference. RQRMI lookups are *precise*, i.e., they are guaranteed to produce the same results as the traditional algorithm. The RQRMI model is lightweight: it is built of a hierarchy of tiny Neural Nets and its inference requires only a handful of arithmetic operations. Thus, the LPM query is performed via compute-intensive cache-friendly inference instead of memory-intensive cache-inefficient trie traversal or hash table lookup.

RQRMI offers several important benefits. First, it can scale to larger rule-sets as it is more space-efficient than the traditional approaches: the model is small (8KB in our case) and only requires storing the rule-set without any auxiliary data structures. Second, its memory footprint is insensitive to the properties of the rules, and always scales linearly with the rule-set size. Third, extending RQRMI to support rules with higher bit-width only requires linear scaling of the width of the arithmetic units used to perform model inference, but does not pose higher memory demands, nor it changes the underlying hardware architecture. Last, RQRMI allows fast batched updates to the rule-sets [58] through lightweight hardware-friendly training algorithm.

Using RQRMI in an LPM engine poses several challenges.

Representing LPM rules as non-overlapping integer ranges. RQRMI can learn a set of non-overlapping integer ranges (i.e., $\{[1-3],[5-9]\}$), but LPM rules do overlap by definition: the goal of LPM engine is to find the best match among many matching (overlapping) rules. The solution suggested in the original RQRMI paper [56] would be highly inefficient. We devise an algorithm to convert LPM rules into non-overlapping ranges with low space overheads, leveraging the prior work on the binary search for LPM [41, 71].

Minimizing floating-point computations without violating correctness. RQRMI inference assumes floating-point arithmetic which is slow and costly in terms of power and area. A common solution is to apply quantization [26, 70, 74]. While quantized inference for neural nets is well-understood, RQRMI is different: ensuring the correctness of the queries requires *analytically estimating the model accuracy during training for all possible inputs*, which assumes no loss of precision during inference and thus does not tolerate quantization. In other words, naive attempts to quantize RQRMI inference break its correctness guarantees. We develop a method to reduce the number of floating-point operations from 26 to 4 per inference without compromising query correctness, thus obviating the need for quantization.

Scaling to off-chip memory. The original RQRMI model becomes inefficient when the rule-set itself exceeds the size of the fast on-chip memory and spills into DRAM. That is because the required DRAM bandwidth to access the spilled data turns out to be prohibitive and non-deterministic, nullifying RQRMI's algorithmic advantages. We develop an approach to split the model data structures into on-chip and off-chip which allows scaling with the number of rules and also leaves sufficient on-chip memory for caching DRAM accesses. This approach enables fine-grained performance tuning depending on the workload and cache architecture.

We prototype NeuroLPM in Verilog with on-chip memory, as well as in a software emulator with DRAM and on-chip caching. We evaluate it on real network traces and ten, million-scale production packet forwarding rule-sets. NeuroLPM shows up to 9× lower off-chip access rate and up to 5× lower bandwidth compared to the state-of-the-art. It is also the only solution that can serve these rule-sets with 4.5MB of SRAM in a DRAM-less configuration. NeuroLPM hardware engine achieves stable throughput of 196M queries/second at 100MHz, which is nearly identical to the *best-case* performance of the state-of-the-art SAIL LPM engine. We further show that updates can be as fast as 100msec on x86 CPUs and 500msec on NVIDIA BlueField-2 [51] SmartNIC.

In summary, we make the following contributions.

- We show the first use of RQRMI learned data structures in hardware as a replacement for traditional memory-intensive data structures;
- We leverage RQRMI to build a multi-purpose LPM engine, NeuroLPM, and solve several algorithmic and design challenges associated with its hardware implementation, including LPM conversion to ranges, minimization of the floating-point operations and efficient scaling to off-chip memory;
- We show, via a comprehensive evaluation, that NeuroLPM offers significant scalability and performance advantages.

2 BACKGROUND

2.1 Longest Prefix Matching

An LPM rule consists of b bits, where the least significant bits are replaced by zero or more wildcards, e.g., 5-bit rules r_1 =`001**` and r_2 =`00***`. The non-wildcard bits are called a *prefix*. For a binary string of b bits, LPM finds the *matching rule* with the longest prefix. In the example, for input `00111`, the matching rule is r_1 as it matches more fixed bits than r_2.

2.2 RQRMI algorithm

Range-Query Recursive Model Index (RQRMI) is a learned data structure for range matching [56]. It uses a collection of independent 3-layer Neural Networks (Multi-Layer Perceptrons), each with one input, one output, and eight fully-connected perceptrons with ReLU activation function. We defer the description of RQRMI to §5.2.

The model is first *trained* to learn the distribution of *sorted, non-overlapping* integer ranges $S=\{R_0,R_1,...,R_{n-1}\}$ in memory. We

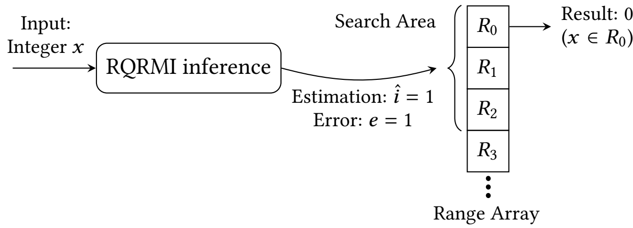


Figure 1: Range matching via RQRMI model inference [56].

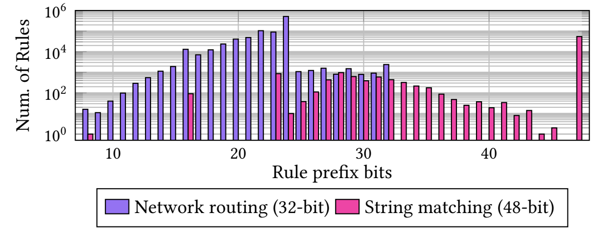


Figure 2: Real-world LPM rules prefix distribution for the network routing and string matching applications.

call S a *Range Array.* A query (Figure 1) is performed in two steps: (Step 1) The input x is fed into the RQRMI model which outputs the *estimate* of the index of the matching range ($\hat{i}$) in S and an upper bound on the prediction error (e) computed during training; (Step 2), a *secondary search* over a subarray of S located between ($\hat{i}-e$, $\hat{i}+e$) outputs the true index (if there is a match). In the example, $\hat{i} = 1$, $e = 1$, and the matching range is at index $i = 0$. The RQRMI training takes about a second on a single CPU core for a range array of hundreds of thousands of ranges [58].

3 THE MULTI-PURPOSE LPM ENGINE

We first present several classes of applications that benefit from a high-performance LPM hardware engine. We derive the requirements for a multi-purpose LPM engine and explain the limitations of the state-of-the-art methods.

3.1 LPM applications

App 1: Routing. Packet routing is performed in hardware by network routers based on the packet's destination IP address and the installed forwarding tables that hold LPM rules. An *action* associated with a rule specifies the router port to forward the packet to its next hop. The number of forwarding rules is constantly growing, straining the available table capacity [27, 54].

App 2: Policy-based routing. Data centers use network virtualization to manage their complex multi-tenant networks [15, 21]. Virtual switches such as Open vSwitch allow specifying complex packet steering policies [2, 45, 53], and commonly leverage LPM and exact match hardware offloads in NICs to speed up the processing. Open vSwitch often uses *multiple* chained rule tables, so the matching is done sequentially table by table, resulting in multiple queries per packet, and requires low, predictable latency per query to allow high throughput with small buffering. Thus, the query latency should be a small fraction of the best-case packet latency of today's production NICs (a few μs) [36].

App 3: Clustering. K-means clustering is broadly used in streaming applications to group elements into subgroups according to the proximity to predefined centroids. Some applications, such as Denial-Of-Service attack defence [5], must perform clustering at network speed. Recent work [23] transforms the clustering logic into LPM, where centroids are encoded as rules and groups are encoded as actions. Thus, the rule action might be any large integer and is not limited to 8 bits as assumed by popular LPM algorithms [78, 79].

App 4: String pattern matching. String pattern matching is used in a variety of performance-sensitive systems to scan data streams for string patterns. For example, Snort [63] and ClamAV [13] are open-source Network Intrusion Detection systems (NIDS) that perform pattern matching on incoming packets to block malware. Prior work [47] shown the utility of LPM for pattern matching. Further, these tools often use the Aho-Corasick algorithm [3] for dictionary search, which transforms a set of strings into a deterministic finite automation (DFA), and in turn can be efficiently described using LPM rules [9]. Importantly, the bit-width of the rules, the length of the prefixes and their total number are determined by the size and diversity of the dictionary. These do not match the assumptions made by standard packet forwarding LPM engines, hence rendering their optimizations ineffective. For example, most rules in IP-routing have the prefix of 24 bits [78], while string matching rules for Snort signatures have a different prefix distribution, as shown in Figure 2.

App 5: Load balancing. Load balancing is commonly used in network systems for providing higher throughput or reliability, as in the case of limited link capacities or hardware resources [77]. Efficient approaches to load balancing use LPMs for splitting the traffic [35, 61]. Specifically, these algorithms approximate load-balancing weights using LPM rules and achieving high accuracy requires high rule capacity.

3.2 Multi-purpose LPM requirements

We derive the following requirements from the surveyed applications.

R1: Scalability. All the applications require efficient support for large million-scale rule-sets.

R2: Robustness. As the rule structure depends on the application, no assumptions should be made about it in a multi-purpose LPM design (see string matching).

R3: High throughput, bounded latency. LPM throughput is the primary performance goal, especially as network rates approach 800 Gbps. In addition, the query latency should be bounded to avoid large buffers in chained multi-query workloads (e.g., in policy-based routing).

3.3 State-of-the-art hardware LPM engines

Existing LPM engines do not satisfy these requirements, as we explain below.

Ternary Content Addressable Memory (TCAMs) are broadly used in hardware designs [46]. The largest commodity TCAMs hold up to 256 tables with up to 1K rules per table, each consisting of up to 160 ternary bits [67]. TCAMs are fast, but the number of rules per table a TCAM may hold is over two orders of magnitude smaller [44] than our target rule-set size. Further, TCAMs consume high power and area, with at least 2× transistors per bit compared to SRAM [4, 43]. Their dynamic power consumption is an order of magnitude higher than that of SRAM, limiting their use [43, 81].

Tuple Space Search (TSS) [64] is used in both software (e.g., by Open vSwitch [53]) and hardware (e.g., by NVIDIA's NICs) [36]. It splits the rules into several hash-tables according to the length of their prefixes. Each table contains rules with prefixes of the same length, resulting in the worst case of 32 tables for 32-bit LPM rules. Unfortunately, increasing the number of tables causes severe performance degradation. For example, in NVIDIA's NICs, using 4 and 16 tables degrades the throughput by up to 2.5× and 7.5× respectively [36]. Worse, different applications use many more tables, e.g., string matching rules for intrusion detection [63] require over 26 tables, and forwarding rules span over 24 tables (Figure 2). Thus, as TSS is sensitive to rule prefix distribution, it is inefficient in such applications.

SAIL [34, 78] is a hardware-oriented LPM algorithm optimized for network routing. The rules are divided into three tables: for rules with up to 16-bit prefix (stored on-chip), up to 24-bits (on-chip), and for the rest (off-chip). A query traverses the tables from the first to the last and returns the first match. If no match is found in the second table, it retrieves the pointer to the third one in DRAM, and then performs another DRAM access to find the rule. These accesses are dependent, and with only 4 bytes per access. Overall, SAIL cannot scale to a larger bit-width, and its performance is highly sensitive to the rule prefix length distribution.

Tree Bitmap [20] is a hardware-oriented algorithm used in commercial NICs [1]. Rules are represented as a trie. The trie nodes are aggregated into 64-byte chunks, each representing the same prefix and including 511 bit-nodes in the same subtree of depth 8. Thus, 32-bit rules require a 4-level tree. A query traverses the trie from the root as usual, thus visiting up to four chunks, some of which are stored in off-chip memory. The traversal involves random reads of 64-byte chunks, with a poor spatial locality, and, consequently, low performance even in the presence of caches. Thus, the algorithm performance heavily depends on the prefix distribution. Further, for rules with a larger bit-width, the number of dependent, off-chip accesses grows linearly with the width, which causes further performance degradation.

Hybrid approaches combine LPM and exact match. Exact match (EM) offloading is supported in many modern NICs, including NVIDIA's Connect-X family [36] or Intel's IPUs [33]. Using these for LPM matching involves *expanding* LPM rules with longer prefixes to several EM entries while keeping the smaller prefixes in other data structures. For example, the rule `01*` gets expanded to two EM entries `010` and `011`. Since the expansion grows exponentially with the number of wildcard bits, EM rules often reside in off-chip memory to free valuable on-chip capacity for LPM data-structure. Unfortunately, this technique is not scalable by design, creates unacceptable off-chip bandwidth with multiple tables [36], and is thus highly sensitive to rule prefix distribution.

[1]We cannot disclose the vendor due to NDA.

3.4 Support for updates

Any LPM engine is required to support modifications to its installed rules. However, there is significant variability in the application requirements.

Network routing, the most demanding application, must accommodate updates that involve thousands of rules per second [32]. The update delay is usually within hundreds of milliseconds in modern devices [30, 36].

Other applications do not require frequent updates. For example, strings in Network Intrusion Detection Systems do not change too often. Similarly, in load-balancing systems, the rule update rate depends on the server churn [35], but it rarely exceeds a few dozen per second.

In this work, we primarily focus on the *LPM query performance* while striving to maintain the update performance on par with existing network devices. We achieve this goal by leveraging unique trade-offs offered by RQRMI (§6.5).

4 NEUROLPM

NeuroLPM overcomes the limitations of the existing LPM engines by leveraging an RQRMI learned data structure to index the rules, thus sidestepping inherent weaknesses of traditional tries and hash-tables.

NeuroLPM operation comprises an offline rule-set preparation stage and online query execution. The former is usually not time-critical, so we currently implement it in software. The latter is on the critical path and thus implemented in hardware. When the rules need to be updated, the rule-set preparation is invoked again on the modified rule-set which contains both old and new rules (more details in §6.5).

The rule-set preparation stage includes:

(1) *Conversion of LPM rules* into a sorted array of non-overlapping integer ranges, called *range array.* (§5.1).
(2) *Bucketization (optional)* is performed if the range array does not fit in on-chip SRAM. It merges every k neighbouring ranges into a wider range, forming a k-times smaller *bucket directory.* The bucket directory is always stored in SRAM, whereas the corresponding *bucket array* with the original ranges is stored in DRAM. An index b_i in the bucket directory corresponds to the bucket in the bucket array at offset $b_i \times k$. Bucketization is described in §7.
(3) *Training* the RQRMI model to learn the locations of SRAM-resident ranges in *RQ Array.* RQ Array is either a range array (if one fits SRAM) or a bucket directory.

The NeuroLPM *hardware query engine* is shown in Figure 3. It is connected to DRAM via a cache. A query enters the *RQRMI inference* module to produce index $\hat{i}$; the *Secondary Search* module searches the RQ Array from $\hat{i} - e$ to $\hat{i} + e$, where e is the model's error bound for that input, and outputs the index of the matching range. If the RQ Array stores the original range array, this is the final output. Otherwise, the index is used to find the respective bucket in DRAM. The *Bucket Reader* retrieves the original ranges from the bucket

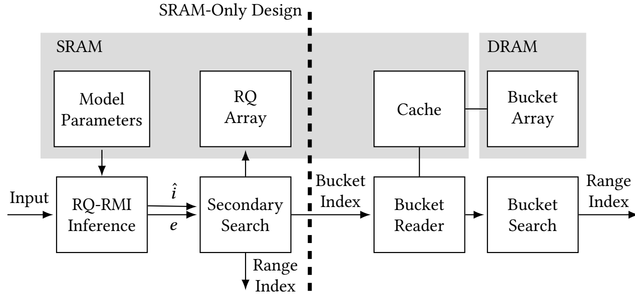


Figure 3: NeuroLPM overview.

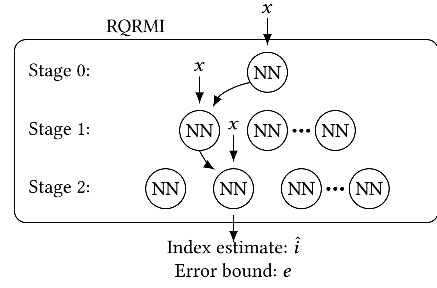


Figure 4: The RQRMI model [56] (see §5.2.1).

and forwards them to the *Bucket Search*, which produces the index of the matching range in the range array.

We discuss all the components in detail next.

5 ALGORITHMIC CHALLENGES

We discuss two challenges: how to learn LPM rules using RQRMI, and how to minimize the use of floating point operations without breaking query correctness.

5.1 Converting LPM rules into ranges

Out-of-the-box, RQRMI cannot be efficiently applied to general LPM rules. A single RQRMI model can learn only non-overlapping rules, but this is not the case for LPM. For example, 5-bit rules r_0 =`1000*` and r_1 =`100**` overlap over the whole range of r_0. The original RQRMI paper [56] suggests splitting the rule-set into multiple subsets of non-overlapping rules and learning each subset in a separate model, choosing the final result from their outputs. However, while this approach works well for the original packet classification use cases, it becomes too expensive both compute- and memory-wise for LPM rule-sets, because it requires evaluation of multiple RQRMI models for a single query. For example, the evaluated rule-sets (§10.1) require between 7 to 12 RQRMI models, which would be too expensive in terms of hardware resources and DRAM memory bandwidth requirements.

Instead, we directly represent LPM rules as ranges, similar to prior work [41, 71]. For the two rules in the example above, the resulting ranges are `10000-10001` for r_0, and `10010-10011` for r_1.

This transformation can be done efficiently in $O(|Rules|)$ steps and results in at most $2 \times |Rules|$ ranges [41], though in practice we observe the expansion of 18% on average (§10.5). The algorithm resembles a balanced bracket checking [73].

The algorithm works as follows.

(1) Add a null rule for the whole input domain, if necessary.
(2) Sort the rules by their lower bounds, then sort those in a tie by upper bounds. We say that a range is *open* when its upper bound is not yet determined.
(3) In increasing order, for each lower bound m_l of rule m, push m onto a stack. Denote the rule at the top of the stack as t. If there is an open range r^i, close it by assigning the upper bound $r^i_u = t_l$. Then open a new range r^{i+1} with the lower bound $r^{i+1}_l = t_l$, and assign t as the matching rule for that range.
(4) Symmetrically, for each upper bound m_u of m, close the current open range r^i by assigning $r^i_u = m_u$. Remove the rule from the top of the stack. Open a new range r^{i+1} such that $r^{i+1}_l = m_u$ and assign t as the rule for that range.

A range is stored using 32 bits (for 32-bit rules), i.e., only its lower bound, as the method covers all the input domain.

5.2 Reducing floating-point operations

To help understand our design, we first explain the RQRMI model in more detail. We refer to the original paper [56] for a more elaborate description.

5.2.1 Background: RQRMI. The RQRMI model architecture we use in NeuroLPM is depicted in Figure 4. It consists of three *stages*. Each stage includes a predefined number of *independent* Neural Nets (NNs), which we call submodels.

The first stage always consists of a single submodel. During RQRMI inference, only a single submodel gets evaluated in each stage. All evaluated submodels are fed with the *same* input x, normalized using an affine transformation determined at training time. The output of a submodel in the internal stages is the index of the submodel used in the subsequent stage. The output of the submodel in the last stage is the output of the model. In addition, the final output includes the error bound e for the submodel calculated at training time.

Each submodel is a Multi-Layer Perceptron (MLP) with a single input, a single output, and a single hidden layer with eight perceptrons with a ReLU activation function. In total, each submodel requires 26 floating point (FP32) operations to compute: 8 multiply-adds, 8 multiplies, 8 sums, and 2 multiplies for input normalization and output scaling.

The model learns the association between the ranges and their locations in memory. It is trained stage by stage, submodel by submodel, using a uniform sampling of the elements in the input domain, and the respective index of their rules in the rule array. Each subsequent stage is trained by sampling from a smaller input domain compared to the previous stage, hence producing more accurate submodels.

The submodels in the final stage are trained to output the index of the matching range within the range array. During the training of each submodel, the algorithm *analytically computes the bound on the prediction error* of each submodel for any possible input. Determining this bound is crucial to guarantee the correctness of

the query. In the end, the training process outputs all the weights of the submodels and their respective bounds on the prediction error.

5.2.2 Quantization challenge. The RQRMI inference requires 26 FP32 operations, which is too expensive both in terms of hardware resources and latency. One common solution is quantization, i.e., replacing the FP32 arithmetics with integer arithmetics [74], which is much faster and more resource-efficient.

Unfortunately, quantization introduces computation errors that make analytical calculation of the submodel prediction error impractical. Specifically, each original submodel is a piece-wise linear function as it is computed as a sum of eight ReLU functions of the submodel's input. This piece-wise linearity is key to efficiently computing the upper bound on the prediction error for each linear segment, with only a handful of *vantage points* (see the details in [57]). With quantization, i.e., when all the operations are performed in lower bit-width or using fixed point arithmetic, these segments cease to be linear and monotonic, due to the quantization error. As a result, computing the submodel prediction error requires sweeping over the entire input domain of a submodel, which is clearly impractical (consider the input domain of 64-bit integers, for example). Without knowing the prediction error, it is impossible to guarantee query correctness.

To reiterate, the problem is not the quantization of neural net inference, rather, the fact that the change in the output due to quantization dramatically complicates the analytical calculation of the submodel prediction error, and this challenge is unique to RQRMI.

Solution: lookup instead of quantization. We observe that each submodel is a piece-wise linear function with at most nine linear segments in the form of $A_i x + B_i$ ($i = 1, \ldots, 9$; x is the input). Coefficients A_i and B_i are a sum of the respective weights and biases of the subset of non-zero ReLU functions on segment i. Thus, we can compute A_i and B_i offline in full precision for each segment. Thus, given x we can find the respective coefficients, and the submodel output would require a single MAC operation, affording computing it in full precision as well.

For example, consider a submodel with two neurons in the second layer, with weights 1 and 2, and biases 1 and 0 respectively. Thus, the output of the submodel $y = 2 \cdot ReLU(x+1) + ReLU(2x)$. The first ReLU gets non-zero output for $x > -1$, and the second for $x > 0$. Therefore, the submodel can be divided into three linear sections: (1) $x < -1$, in which $y = 0$, i.e., $A_0 = B_0 = 0$; (2) $-1 \leq x < 0$, in which $y = 2(x+1)$, i.e., $A_1 = 2$, $B_1 = 2$; and (3) $x \geq 0$, in which $y = 4x + 2$, i.e., $A_2 = 4$, $B_2 = 2$.

This logic can be easily implemented using a lookup table as follows. We compute the boundaries of the linear segments X_i in the original submodel and save them in a table, with the respective A_i and B_i. During inference on x, we search for matching segment j for which $X_j \leq x$ and then compute $A_j x + B_j$. Notably, this technique results with *the same output* as the original submodel so query correctness is preserved.

Our FPGA-based prototype reveals that the lookup-table design is 4× faster compared to implementing RQRMI inference in full precision.

6 SRAM-ONLY DESIGN

We first focus on the NeuroLPM hardware design assuming that the rule-set fits in SRAM, but later relax this assumption.

6.1 RQRMI inference design

The lean structure of the RQRMI model facilitates efficient implementation of inference in hardware. We first develop a pipelined module for performing the inference of a single submodel (Figure 5b), and then connect three such modules in pipeline, according to the number of stages in the RQRMI model which we use for LPM (Figure 5a, left).

The submodel parameters are retrieved from the per-stage parameter buffer in SRAM according to the submodel index within the RQRMI stage. Input normalization and output scaling are performed according to the model parameters. The output of the module is either the index of the submodel in the next RQRMI stage, or, for a leaf submodel, the final result of the whole model. The submodel error bound is determined during training, and is passed as part of the output.

6.2 Secondary search

The Secondary Search module (Figure 5a, right) searches for the matching range in the RQ Array segment $[\hat{i} - e; \hat{i} + e]$ in SRAM, where $\hat{i}$ is the RQRMI predicted index and e is the error bound. The error may range from tens to hundreds of indexes, so we use binary search (recall that RQ Array is sorted).

We considered two alternatives: (1) a pipelined design where each stage performs a single access to the RQ Array, with a total of $\lceil \log e \rceil$ number of stages; or (2) multiple Finite State Machines (FSM) each performing the whole search to completion. We note that achieving high throughput with multiple FSMs assuming enough input parallelism is a valid approach as we strive to support many independent queries (R3 in §3.2). We chose the FSM option for its simplicity.

We use multiple banks to connect the FSMs to SRAM to minimize memory stalls. Each FSM can access any bank via a crossbar. The RQ Array entries are distributed across memory banks in a round-robin manner, and the number of banks is a power of two to allow efficient bank indexing.

To determine the number of FSMs, we observe that the Secondary Search module must sustain one query per cycle on average. Thus, the number of FSMs should be no less than the number of memory accesses to the RQ Array by a single FSM (assuming one access per cycle from each FSM), which in turn is determined by the error bound of the RQRMI model. On the other hand, the memory subsystem must sustain the demand from all the FSMs, and its throughput depends on the number of banks and bank conflicts. Next, we develop an expression to compute the throughput of the memory subsystem under bank conflicts.

6.2.1 Memory throughput vs. banks vs. FSMs. Assume that each FSM issues one memory request per cycle, and this request is independent of the previous request by the same FSM. There are k FSMs and m banks, $m \leq k$ where an FSM may access any bank uniformly at random. The system runs at maximum throughput when no banks are idle. Therefore, the average throughput in terms

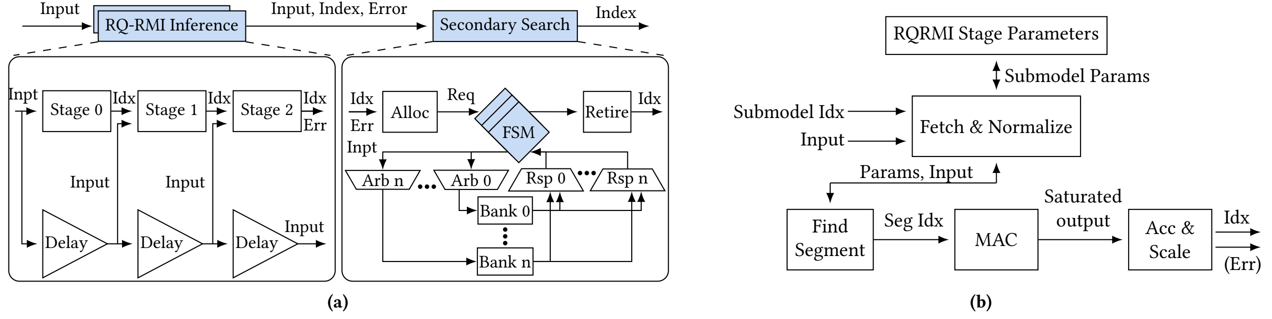
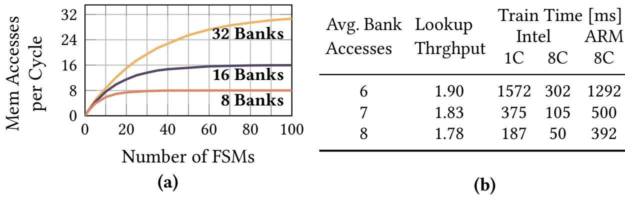


Figure 5: (a) SRAM-only NeuroLPM pipeline. (b) The RQRMI submodel inference module.



Avg. Bank Accesses	Lookup Thrghput	Train Time [ms] Intel 1C	Intel 8C	ARM 8C
6	1.90	1572	302	1292
7	1.83	375	105	500
8	1.78	187	50	392

(b)

Figure 6: (a) The theoretical average throughput of the memory subsystem vs. the number of FSMs. (b) Training time and its effect on end-to-end lookup throughput (packets/cycle).

of the number of memory requests per cycle is equal to the expected number of banks accessed per cycle. To model the throughput as a function of the number of FSMs, we observe that this is equivalent to computing the expected number of distinct birthdays for k people while having m days in a year. This is given by $T = m \cdot (1 - (\frac{m-1}{m})^k)$ and depicted in Figure 6a.

This expression assumes that the accesses by the same FSM are independent, which is not the case if the FSM's access is delayed due to a conflict. Instead, the expression gives an upper bound on the throughput as it overestimates the number of FSMs that can choose a bank in each cycle.

We now can determine the number of FSMs and memory banks as follows. Assume that an average prediction error of a submodel is 512. Thus, the number of binary search steps is log(512), i.e., the average number of memory accesses per RQRMI query is 9. Thus, to achieve the throughput of one query per cycle in the Secondary Search, the memory subsystem must sustain 9 memory accesses per cycle on average. According to Figure 6a, 16 banks and 10 FSMs may serve only about 8 accesses on average, which is not enough. However, 16 FSMs allow the system to serve about 10 memory accesses per cycle on average, which is sufficient to achieve the target throughput.

Inversely, this analysis shows that to achieve the highest throughput with 16 banks and 16 FSMs, the RQRMI model must be trained with error bound e such that $\lceil \log e \rceil \leq 10$.

6.3 SRAM-only pipeline

The complete design supports multiple RQRMI modules to feed the Secondary Search FSMs. This is necessary due to the stochastic nature of the Secondary Search. Specifically, the submodel error bound (and, consequently, the number of memory accesses per search) varies from submodel to submodel, creating an uneven load between the FSMs. Further, the number of bank conflicts varies and depends on the input queries. As a result, when Secondary Search runs without conflicts and with lower errors, FSMs might get underutilized and the RQRMI module becomes the bottleneck.

NeuroLPM enables the addition of inference engines by augmenting the FSM job allocator to accept two queries and allocate them to two FSMs at once. When only one FSM is available, one engine is stalled in a round-robin fashion.

6.4 Scaling the bit-width

The design so far assumed 32-bit rules and inputs. However, scaling the bit-width to 64 or 128-bit requires no architectural changes. It requires increasing the bit-width of the MAC units for inference, and the width of each SRAM bank. If these modifications decrease the throughput of the RQRMI module, more such modules can be added to feed the Secondary Search. Indeed, the NeuroLPM software prototype scales to 128-bit rules (IPv6) just by using wider data types.

Notably, the rule bitwidth has no impact on the number of memory accesses. This is in stark contrast with other LPM engines where the number of accesses grows linearly with the bitwidth, e.g., due to changing the height of a trie, or the number of accessed hash tables.

6.5 LPM rule updates

Update types. Rule updates can be categorized into three types: action modification, rule deletion, and new rule insertion. The first two do not require retraining the RQRMI model, and involve traversal of the RQ-array to mark relevant entries as invalid or modifying their associated action. Inserting new rules is described as follows.

Training time. Rule insertions require full retraining, i.e., the time to train the whole rule-set determines the time to commit the new rules, regardless of how many rules are inserted.

The training algorithm introduced in prior work [58] takes 1.5 seconds to train on a representative 870K rule-set (§10) using a single x86 core. However, it can be sped up significantly by exploring

the *tradeoff between the training time and query performance, and via parallelization.*

Background: training. At a high level, a model is trained layer by layer, where each submodel (Neural Network, NN, in Figure 4) in a layer is trained independently using a standard back-propagation via gradient descent by sampling from its input domain. The training ends when the attained error bound, e, is below the target value.

A shorter training might result in lower model accuracy and higher e, hence, a longer secondary search and slower lookup. This tradeoff is not linear, however. First, the number of samples used to train an NN can be reduced by about 3× compared to [56], without any measurable effect on e. Second, the training time is dominated by a few "straggler" submodels which take longer to converge to the desired e. However, if faster training is needed, having a small portion of submodels with higher e has a negligible impact on the lookup performance (about 3.5%), yet can shorten the training by up to 4×. This is because most of the lookups are performed via NNs with small e, and because the query time increases as $\log(e)$.

The training time improvements are shown in (Figure 6b). Together with parallelization, i.e., by training each submodel independently, the same 870K rule-set is trained on eight x86 cores in 105ms (50ms) with 3.5% (7%) slower lookups (see §10 for setup details).

If NeuroLPM is deployed on a SmartNIC, training on eight ARM A72 cores on the NVIDIA BlueField-2 [51], without using the host (x86) CPUs, takes only 500ms (392ms).

Last, we note that training each submodel requires about 50K arithmetic operations and is entirely compute-bound. Given that RQRMI has three layers, the critical (sequential) path comprises 150K operations. Thus, dedicated hardware may enable training at a millisecond scale.

To conclude, RQRMI models offer novel tradeoffs that allow training about a million rules on commodity hardware within hundreds of milliseconds. According to [36], this update rate is on par with existing SmartNICs.

Atomicity. An atomic transition from the old rule-set to a new one is often desirable, with minimum downtime. It implies that both the old and the new versions of the rules must reside concurrently in SRAM to avoid stalling the engine. The downside of this solution is its low SRAM utilization. In NeuroLPM, however, any free SRAM is used as a cache. Thus, placing the updated rules in SRAM momentarily reduces the cache size and increases the DRAM bandwidth without affecting the overall throughput.

Limitations. NeuroLPM targets applications in which rule insertions originate in the control plane and take the order of hundreds of milliseconds. For example, updates in Cloud routing tables take about 180ms [16, 22]. On the other hand, NeuroLPM is ill-suited for applications that require frequent, atomic rule insertions, such as the *Reconfigurable Match-Action Table* (RMT) [8] or the disaggregated RMT (dRMT) [12] architectures used by programmable switches. In these architectures, data-driven updates take a few microseconds to complete on TCAM hardware [31].

Nonetheless, this limitation can be side-stepped using a delta buffer for accommodating incremental updates. For example, a small TCAM with 10K entries can support 33K (100K) updates per second, given that RQRMI training takes 300ms (100ms). Note that this approach is realistic as NVIDIA production switches make use of 10K-entry TCAMs for similar purposes [59].

7 SCALING TO EXTERNAL MEMORY

In million-scale LPM rule-sets, the size of the range array may exceed the available SRAM capacity.

One could use SRAM as a cache, and store the RQ Array in DRAM. Unfortunately, the worst-case performance is unacceptable. Secondary Search over DRAM would result in a prohibitively large number of DRAM accesses per query, i.e., eight accesses on average in the evaluated workloads (§10.3). Dependent memory accesses and variable DRAM latency deteriorate the throughput, as observed in production NICs (§3.3). Furthermore, these are 4-byte accesses (for 32-bit rules) to non-adjacent indexes, so serving them would incur high bandwidth overheads.

For example, in the packet forwarding use case, handling minimum-size packets at 200 Gbps would require about 200 Gbps of DRAM bandwidth, even assuming an 8-byte access granularity. The memory bandwidth is a scarce resource, i.e., NVIDIA's BlueField provides only 140 Gbps to DRAM [50], and it is shared between all applications. Moreover, NICs are often DRAM-less and instead use the host's DRAM via PCIe bus, where both the latency and the bandwidth are significantly worse.

Last, the DRAM bandwidth requirements in this solution are dictated by the RQRMI error bound, and cannot be set according to system constraints.

7.1 Avoiding secondary search in DRAM

We compress the range array such that the compressed ranges, called *bucket directory*, can be placed in SRAM, while the uncompressed ranges, called *bucket array*, are stored in DRAM. We call this process *bucketization.* Adjacent ranges in the original range array are grouped together into equal-sized *buckets* each with k ranges. A bucket directory is formed by merging the ranges in the same bucket and forming new bucket ranges that subsume the original ones. The size of the bucket directory is exactly $[bucketsize]\times$ smaller than the original range array. All inputs that match the ranges in the bucket also match the respective bucket range. Thus, given the index of the bucket range, finding the bucket in the bucket array is easy, akin to finding a memory frame in virtual memory.

Example. Given a range array of [1-3],[4-5],[6-10],[11-15]. Bucketization with the buckets of size 2 results in a bucket directory array of [1-5],[6-15]. For input 9, the matching range [6-15] is located in the bucket directory at offset 1. To find the actual matching range, one compares the two ranges in the bucket at offset 2 of the range array.

With bucketization, the RQRMI query is first performed on the bucket directory. When the respective bucket range is found, the bucket rules are fetched from DRAM at once. Thus, the per-query DRAM bandwidth is *no longer a function of the error bound* and determined by bucket size. Moreover, the bucket size can be adjusted to suit other constraints, such as DRAM memory bandwidth.

An optimized version does not store additional ranges in RQ array, rather it uses every k^{th} range in the range array as a range boundary in the bucket directory. Thus, in a bucket of size k, one

range already resides in SRAM hence only $k - 1$ ranges need to be fetched from DRAM.

8 DISCUSSION

Rule-set scaling. Bucketization is key to scaling to large rule-sets. For example, with 4MB of SRAM as in NVIDIA ConnectX6-Dx [55], bucket size of 8 (32 bytes) and half of SRAM allocated for cache, NeuroLPM can support up to four million ranges. Importantly, the memory requirements grow linearly with the size of the rule-set, and cannot spike unexpectedly due to problematic rule-set characteristics. Scaling further within the same SRAM is possible by doubling the bucket size, at the expense of doubling the worst-case DRAM bandwidth as we explain below.

DRAM bandwidth vs. cache size. Bucketization offers a new design trade-off between the cache size and the worst-case DRAM bandwidth. Given a fixed SRAM budget, larger buckets result in a smaller bucket directory and more SRAM space for the cache. On the other hand, large buckets increase the worst-case DRAM bandwidth and assume higher spatial access locality to ranges to use the ranges in the same bucket. We evaluate this trade-off in our application in §10.2.

Exact match cache. Exact Match Cache (EMC) stores the matched rules instead of individual memory accesses. This approach, however, is inefficient [69]. Furthermore, it is complementary to ours, so we do not consider it here.

RQRMI vs. full binary search. At a high level, RQRMI can be seen as a way to speed up a simple binary search. Specifically, it reduces the asymptotic complexity of binary search in a sorted array of size n from $O(\log n)$ to $O(\log e)$, where e is RQRMI error boud. In the data sets we use to evaluate NeuroLPM, RQRMI reduces the number of required memory accesses per query by more than 2× compared to a full binary search.

Scaling the number of rules. NeuroLPM introduces a unique tradeoff between the lookup performance, DRAM bandwidth, and training time. More rules can be accommodated with the *same* model (and the same training time), but with larger buckets (i.e., higher k), so lookups would consume more DRAM bandwidth. However, if updates are infrequent and longer training is acceptable, one may increase the model size to maintain the same DRAM bandwidth and the same throughput. Then, the training time can be shortened but at the expense of slight throughput degradation.

We illustrate this tradeoff by experimenting with a rule-set 4.5× larger (3.9 million LPM rules) than a representative 870K rule-set we use in §10. Under the same model configuration, the lookup throughput degrades by 12% but training is only 1.6× longer. Doubling the number of submodels increases the training by an additional 2×, but regains the throughput within 2% from the throughput of the 870K rule-set. Doubling the DRAM bandwidth, i.e., by using double-sized buckets, keeps the lookup throughput as with the 870K rule-set but takes 22% longer to train.

In contrast, state-of-the-art LPM engines do not offer such flexibility, as they either cannot handle larger rule-sets at all, or suffer from significant performance degradation due to a large number of DRAM accesses, e.g., two-three DRAM accesses per packet in the worst case for SAIL and Tree Bitmap, compared to one access for NeuroLPM (§10.2).

The effect of RQRMI size on the training time. Increasing the RQRMI model size by adding NNs may reduce the number of "straggler" submodels and shorten its training time, even though more submodels must be trained (§6.5). However, establishing a correlation between the two is challenging due to the reliance of "straggler" submodels on the range distribution. For example, a single submodel suffices to approximate rule sets with uniformly distributed ranges, so adding submodels would only increase the training duration. Therefore, we prefer to use smaller RQRMI models and absorb the high error bound of problematic submodels, if such exists, in the secondary search phase. In practice, our experiments (§10) demonstrate that an RQRMI model with 1, 4, and 64 submodels per stage, respectively, achieves good performance for all evaluated rule sets.

Deployment options. NeuroLPM is naturally suitable for NICs and network middle boxes, but we also envision its use in switches and routers, by deploying multiple replicas. Such architecture can achieve multi-Tbit bandwidth by leveraging parallelism as ports can be processed in parallel, as has been considered earlier in Tree Bitmap [20].

9 IMPLEMENTATION

We implement the RTL of the SRAM-only NeuroLPM pipeline (See Figure 3) in Verilog in Xilinx Vivado. The implementation closely follows the described design.

RQRMI inference. We implement the 3-staged RQRMI model configuration described in the design. The implementation is fully pipelined and is designed to produce one output per cycle. For each stage, we implement the lookup-table (LUT) approach described in §5.2.2. It utilizes FP32 operations implemented using Vivado's IP blocks.

Secondary search. A query enters a simple scheduler that allocates it to an idle secondary search FSM. When two RQRMI pipelines are connected, and there is more than one input per cycle, the scheduler finds the available FSMs and chooses which input engine to stall. First, the allocator uses pop-count to find the available FSMs. If at least two are available, it allocates them for execution. Otherwise, it stalls one or both input units using the round-robin policy.

All FSMs are connected to memory banks via a crossbar. Each bank has a round-robin arbiter to resolve bank conflicts. Blocked FSM waits for the next read cycle to try again.

Our design supports one or two RQRMI pipelines, 8, 16, or 32 memory banks, and 8, 16, 32, 48, 64, and 96 FSMs. Doubling the number of banks and FSMs further resulted in prohibitively high resource utilization, i.e., 55% FPGA LUT utilization for 64 banks, 3.6× more than with 32 banks.

Software. We implement all parts of the NeuroLPM design in software to allow the evaluation of the DRAM version for large rule sets.

10 EVALUATION

We seek to answer the following questions: (1) comparison to the state-of-the-art; (2) evaulation design trade-off; and (3) resource consumption.

10.1 Methodology

We focus on packet forwarding. We obtain ten large real-world LPM rule-sets from RIPE [1] (four rule-sets of about 870K each), Stanford [80] (two of about 180K rules), and RouteViews [52] (four from 850K to 950K rules each). These rule-sets are then converted using the Zebra FIB converter [2] such that all duplicate rules are removed.

We use a CAIDA packet trace from Equinix data center in Chicago [10], similar to prior works [56, 58]. We modify the traces for each rule-set to allow the evaluation of the system with realistic query locality according to the methodology used by prior works [56].

We note that the graphs presented below include a single rule-set from each source, as the other rule-sets from the same source behave almost identically.

NeuroLPM configuration. NeuroLPM uses 32-byte buckets. As a result, the worst-case DRAM bandwidth is 88 Gbps when serving minimum-size packets at 200 Gbps. However, thanks to caching the effective bandwidth is only a small fraction of that (as discussed next). The RQRMI model consists of three stages, with 1, 4, and 64 submodels per stage respectively. The total model size is 8KB. We report the numbers for RQRMI models with 6 average bank accesses.

Baselines. We compare NeuroLPM against the state-of-the-art SAIL and Tree Bitmap hardware-oriented LPM algorithms (See §3.3). We do not evaluate TCAM as it is limited to much smaller rule-sets.

10.2 DRAM access rate per query

We measure the DRAM access rate and bandwidth of these algorithms by running their software emulation. The measurements are performed as follows. We emulate a cache (2-way associative LRU, with 32 bytes cache lines), and instrument the algorithms to access their internal data structures used for matching via the cache. Thus, the Tree Bitmap algorithm accesses the trie chunks of 64 bytes, SAIL reads two-byte indexes of the 3rd-level table and one-byte action indexes, and NeuroLPM reads bucket-size buckets with the ranges, four bytes per range. The rest of the data are assumed to be statically allocated in SRAM, so accessing them does not contribute to the cache statistics. The effective cache size is determined by the SRAM size set in the experiment, minus the size of the data structures stored statically in SRAM according to the design of the algorithm. We measure the cache miss rate per query and use it to derive the DRAM bandwidth per query according to the maximum between the effective size of memory accesses and the cache line size.

Such an approach provides a fair comparison of the cache efficiency and DRAM access rates at the algorithmic level, independent of the hardware implementation. We note that caching has not been considered in the context of baseline algorithms before.

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

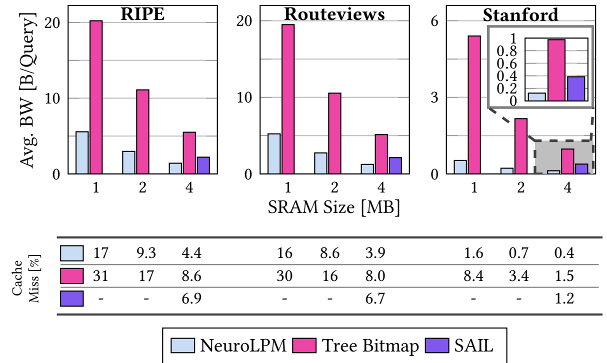


Figure 7: Average DRAM bandwidth per query vs. SRAM size. SAIL requires at least 2.4MB of SRAM to run. Lower is better.

We run a trace with 10M packets (i.e., 10M queries) on all ten rule-sets. As Figure 7 shows, NeuroLPM outperforms all other algorithms for all the rule-sets. Specifically, it shows up to 5× and 3× miss rate reduction over Tree Bitmap and SAIL, respectively, for the Stanford rule-set, and up to 4× and 1.7× DRAM bandwidth reduction for the Routeviews rule-set. These effects are particularly pronounced for smaller SRAM sizes, where SAIL cannot run at all as it statically allocates 2.25MB of SRAM.

With 4.5MB SRAM (not shown in the figure), NeuroLPM is the only one that allows serving queries *from SRAM alone*. In contrast, SAIL and Tree Bitmap access DRAM, e.g., their miss rate for the RIPE rule-set is 4.3% and 1.5% respectively.

We claim that this experiment is indicative of the NeuroLPM performance and scalability advantages in a complete system. The algorithms achieve the best-case throughput, i.e., about 200 Mpps (Million packets per second) for SAIL, when serving queries from SRAM (more details about throughput below). However, their performance is dominated by DRAM accesses once they need to scale to larger and more diverse rule-sets. This is why DRAM access savings of NeuroLPM directly translate into system performance benefits.

Worst-case DRAM access rate. The worst-case number of memory accesses per query is an important metric to evaluate LPM engines for networking applications, as abrupt changes in traffic might occur, rendering caching ineffective [69].

The worst-case scenario for Tree Bitmap is when all trie nodes but the root are in DRAM. Trie traversal involves dependent memory accesses, and latency hiding requires large buffers to avoid stalls, and designs must be built for such a worst-case.

SAIL and Tree Bitmap require two and three accesses in the worst case, whereas NeuroLPM needs only **one** access. This deterministic behavior is a clear advantage over the alternatives.

10.3 Hardware performance

We use Intel Quartus ModelSim Logic Simulator to estimate the latency and throughput of the NeuroLPM hardware implementation without caching (the SRAM-only design in Figure 3). We use a trace with 10M packets and report the results of a representative rule-set from each public archive.

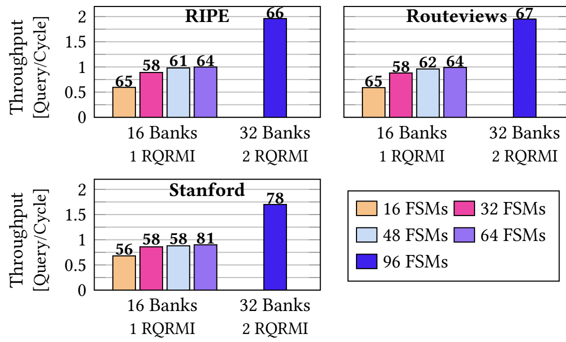


Figure 8: End-to-end performance of the hardware prototype (SRAM-only). Higher is better. The average latency in cycles (for 100MHz synthesis) is annotated on top of each bar.

Throughput and latency. The NeuroLPM prototype with 96 FSMs, 32 banks and two RQRMI modules achieves the average throughput of 196Mpps at 100MHz. For comparison, SAIL reaches 200Mpps at 200MHz. RQRMI inference completes in 22 cycles, and the end-to-end query latency is dominated by the Secondary Search (between 35-55 cycles depending on the configuration and the trace).

These results show that NeuroLPM can serve multiple dependent queries at a total latency of a few hundred cycles on average. This translates into a μsecond packet latency at 100MHz for multiple dependent queries in a sequence but can be further improved with a higher frequency design.

Design-space exploration. We evaluate the system performance under different hardware configurations while varying the number of memory banks (16-32) and search FSMs (16-96). The graphs do not show the inferior configurations: (1) when the number of FSMs is smaller than the number of memory banks, and (2) when the number of memory banks is 8, which leads to the memory bottleneck due to bank conflicts.

Figure 8 shows the throughput (bars), and the average latency (annotations) of the evaluated configurations for the representative rule-sets. For a single RQRMI module, the 16:48 (#banks:#FSMs) configuration yields the best throughput. The results also highlight the throughput-latency trade-off: increasing the number of FSMs while the memory is the bottleneck, without increasing the number of banks, improves throughput but affects latency. Figure 9 confirms this result.

Furthermore, Figure 8 demonstrates that when we keep the FSM bank ratio constant but double the bank size we can double the throughput by adding another RQRMI inference engine. This approach offers substantial improvement to scalability as the inference stage requires only 8KB of memory and 1.5% of the board's DSPs.

10.4 Resource consumption

We synthesize the SRAM-only prototype using Vivado Studio. For comparison, we also implement and synthesize the SAIL algorithm (§3.3). We target Xilinx Kintex UltraScale+ xcku 5p-ffve1760-3-e FPGA. While our goal is a comparative analysis of the resource consumption, this specific device is a viable platform for network applications that can benefit from LPM engines [75].

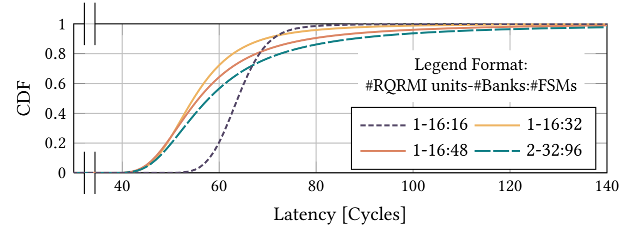


Figure 9: End-to-end query latency.

	NeuroLPM (16 Banks:48 FSMs)	NeuroLPM (32 Banks:96 FSMs)	SAIL
LUT	10165 (1.9%)	81862 (15.6%)	600 (0.1%)
FlipFlop	2194 (0.2%)	11899 (1.13%)	757 (0.07%)
DSP	30 (1.52%)	60 (3.04%)	0
BRAM	120 (12.1%)	120 (12.1%)	546 (55%)

Table 1: FPGA resource consumption.

For NeuroLPM, we use the best-performing configuration with 16/32 banks, RQRMI with FP32 arithmetic, and either 48 or 96 FSMs. The BRAM size (about 540KB) is sufficient to hold the RQ Array for all the evaluated rule-sets with 32-byte buckets. For SAIL, we allocate BRAMs to hold its 16- and 24-bit tables, which results in 2439KB.

Table 1 shows that SAIL consumes fewer LUTs and flip-flops than NeuroLPM, but almost three times more BRAM. Note that LUTs and FlipLops are usually abundant on FPGAs, whereas BRAM is a scarce resource. As expected, SAIL does not use DSPs whereas NeuroLPM uses them for RQRMI inference.

Alternatively, if we assume the same amount of BRAM for both NeuroLPM and SAIL, we can instantiate 4× more NeuroLPM engines using that memory budget, while having an extra 279KB available for cache. Assuming one RQRMI module, 16 banks and 48 FSMs per NeuroLPM instance, four replicas of NeuroLPM can reach 400Mpps average-case throughput at 100MHz, 2× better than SAIL at 200MHz, only utilizing additional 6% and 5% LUT and DSP, respectively, compared to a single replica.

In conclusion, NeuroLPM's compute logic requires more resources on die compared to SAIL, but this is compensated by a much smaller memory footprint, and can thus be leveraged to improve throughput by instantiating more NeuroLPM engines.

Power. LPM accelerators require large SRAM for their internal data structures and the cache. SRAM logic dominates the power and area of the design. Indeed, the static power estimate by the Vivado analyzer shows that 72% is consumed by BRAM. Therefore, the overall power consumption of SAIL and NeuroLPM is proportional to their respective use of SRAM. Given the same amount of SRAM for both, their power consumption is likely to be similar.

10.5 Analysis

LPM-to-ranges conversion overheads. We measure the space costs of converting LPM to non-overlapping ranges. We observe

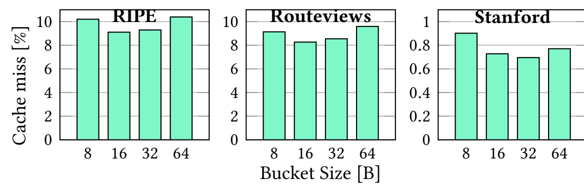


Figure 10: NeuroLPM cache miss rate for 2MB SRAM for different bucket sizes. Lower is better.

that each rule-set produces 18% more ranges than the rules on average across all rule-sets. The largest rule-set of 948K is transformed into 1.05M ranges. In the worst case (Stanford rule-set), we observe 32% more rules.

Impact of bucket size on cache hit rate. Bucket size is an important parameter that facilitates tuning NeuroLPM to system requirements such as cache line size and DRAM bandwidth. Additionally, large buckets effectively result in a smaller RQ Array in SRAM, freeing space for the cache. However, larger buckets have lower spatial locality, so larger cache lines reduce the *effective* cache space.

Figure 10 shows the effect of bucket size on cache hit rate with different rule-sets, with 2MB SRAM shared by the cache and RQ Array. In this experiment (only), the cache line size is the bucket size. The miss rate reduces when we increase the bucket size up to 32B, but then it grows again.

This result is expected. For example, with 8-byte buckets (bucket size of 3) and 1M ranges as in Routeviews rule-set, the RQ Array is 1.3MB and leaves about 700K for the cache. Increasing the bucket size to 16 bytes, increases the cache to 1.2MB, but also doubles the cache line size. Given the poor spatial locality of accesses in the same cache line, the cache miss rate does not improve.

11 RELATED WORK

Learned data structures. Learned data structures have been applied to databases, key-value stores, and virtual software switches [14, 19, 37–39, 42, 56, 58, 68], and show performance benefits by trading memory accesses for ML inference. To the best of our knowledge, our work is the first to consider and solve the challenges of using them for this purpose in hardware, such as quantization, bank organization, and extension to DRAM.

Packet classification. The closest to our work is NuevoMatch [56], which uses RQRMI for compressing the data structure to fit the CPU cache. NeuroLPM builds on RQRMI, but it extends it to use for LPM and designs a hardware engine that implements it efficiently.

Machine-learning in the data-path. ML have been applied in the performance-critical parts of flash devices [29], RDMA key-value stores [72], programmable switches [76], NICs [62, 65, 66], and virtual switches [58]. To the best of our knowledge, ours is the first work that applies ML to LPM hardware.

Quantized inference of Neural Nets. Quantization of neural nets has been explored in the context of reducing DRAM bandwidth and improving the inference throughput [26, 74]. This work is fundamentally different: we reduce the number of floating point operations required for the RQRMI model inference, thus obviating the use of quantization altogether.

12 CONCLUSION

This paper discusses a new approach for designing hardware LPM engines using RQRMI learned data structure. From the algorithmic perspective, this is the first use of RQRMI for LPM, and its robustness is leveraged to input distributions, scalability with the number of rules, and easy extension to larger bit-width. From the hardware perspective, this is the first design of learned data structures in hardware that replaces tries. Our comprehensive evaluation shows scalability advantages of NeuroLPM over state-of-the-art alternatives. We hope that this work will open a new avenue for using learned data structures in hardware architectures.

13 ACKNOWLEDGEMENTS

We thank our anonymous shepherd and the reviewers of MICRO'23 for their helpful comments and feedback. We gratefully acknowledge generous support from Intel and Israel Science Foundation (Grant 1998/22).

REFERENCES

[1] 2022. RIPE NCC RIPE NETWORK AND COORDINATION CENTER. https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.
[2] A Linux Foundation Collaborative Project. 2021. Open vSwitch. https://www.openvswitch.org/.
[3] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (1975), 333–340.
[4] Mohammad J. Akhbarizadeh, Mehrdad Nourani, Deepak S. Vijayasarathi, and T. Balsara. 2006. A nonredundant ternary CAM circuit for network search engines. *IEEE Trans. Very Large Scale Integr. Syst.* 14, 3 (2006), 268–278.
[5] Albert Gran Alcoz, Martin Strohmeier, Vincent Lenders, and Laurent Vanbever. 2022. Aggregate-based congestion control for pulse-wave DDoS defense. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.
[6] Michiel Appelman and Maikel de Boer. 2012. Performance analysis of OpenFlow hardware. *University of Amsterdam, Tech. Rep* (2012), 2011–2012.
[7] Barefoot Networks. 2019. Algorithmic longest prefix matching in programmable switch. https://patents.google.com/patent/US10511532B2/en.
[8] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando A. Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.
[9] Anat Bremler-Barr, David Hay, and Yaron Koral. 2014. CompactDFA: Scalable Pattern Matching Using Longest Prefix Match Solutions. *IEEE/ACM Trans. Netw. (TON)* 22, 2 (2014), 415–428.
[10] CAIDA. 2021. Center for Applied Internet Data Analysis based at the University of California's San Diego Supercomputer Center. https://www.caida.org/.
[11] Dibei Chen, Zhaoshi Li, Tianzhu Xiong, Zhiwei Liu, Jun Yang, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. CATCAM: Constant-time Alteration Ternary CAM with Scalable In-Memory Architecture. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[12] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.
[13] ClamAV. 2022. ClamAV - Open Source Antivirus Engine for detecting trojans, viruses, malware and other malicious threats. https://www.clamav.net/.
[14] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[15] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud

Network Virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[16] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[17] Mikael Degermark, Andrej Brodnik, Svante Carlsson, and Stephen Pink. 1997. Small Forwarding Tables for Fast Routing Lookups. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[18] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. 2006. Longest prefix matching using bloom filters. *IEEE/ACM Trans. Netw. (TON)* 14, 2 (2006), 397–409.

[19] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *ACM International Conference on Management of Data (SIGMOD)*.

[20] Will Eatherton, George Varghese, and Zubin Dittia. 2004. Tree bitmap: hardware/software IP lookups with incremental updates. *ACM SIGCOMM Computer Communication Review (CCR)* 34, 2 (2004), 97–122.

[21] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[22] Daniel Firestone, Andrew Putnam, Sambrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian M. Caulfield, Eric S. Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert G. Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[23] Roy Friedman, Or Goaz, and Ori Rottenstreich. 2021. Clustreams: Data Plane Clustering. In *The ACM SIGCOMM Symposium on SDN Research (SOSR)*.

[24] Kaustubh Gadkari, M. Lawrence Weikum, Daniel Massey, and Christos Papadopoulos. 2016. Pragmatic router FIB caching. *Comput. Commun.* 84 (2016), 52–62.

[25] Igor Gashinsky. 2011. Datacenter scalability panel. *North American Network Operators Group (NANOG)* 52 (2011).

[26] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *CoRR* abs/2103.13630 (2021).

[27] Garegin Grigoryan, Yaoqing Liu, and Minseok Kwon. 2020. PFCA: A Programmable FIB Caching Architecture. *IEEE/ACM Trans. Netw. (TON)* 28, 4 (2020), 1872–1884.

[28] Qing Guo, Xiaochen Guo, Yuxin Bai, and Engin Ipek. 2011. A resistive TCAM accelerator for data-intensive computing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[29] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[30] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. 2015. Measuring control plane latency in SDN-enabled switches. In *The ACM SIGCOMM Symposium on SDN Research (SOSR)*.

[31] Peng He, Wenyuan Zhang, Hongtao Guan, Kavé Salamatian, and Gaogang Xie. 2018. Partial Order Theory for Fast TCAM Updates. *IEEE/ACM Trans. Netw. (TON)* 26, 1 (2018), 217–230.

[32] Thomas Holterbach, Stefano Vissicchio, Alberto Dainotti, and Laurent Vanbever. 2017. SWIFT: Predictive Fast Reroute. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[33] Intel. 2021. Intel IPU's. https://www.intel.com/content/www/us/en/products/network-io/smartnic.html.

[34] Md Iftakharul Islam and Javed I. Khan. 2019. SAIL Based FIB Lookup in a Programmable Pipeline Based Linux Router. In *IEEE International Conference on High Performance Switching and Routing (HPSR)*.

[35] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. 2015. Efficient traffic splitting on commodity switches. In *ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*.

[36] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostic, and Gerald Q. Maguire Jr. 2021. What You Need to Know About (Smart) Network Interface Cards. In *Passive and Active Measurement (PAM)*.

[37] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *ACM International Conference on Management of Data (SIGMOD)*.

[38] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A learned database system. In *Biennial Conference on Innovative Data Systems Research (CIDR)*.

[39] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *ACM International Conference on Management of Data (SIGMOD)*.

[40] Minseok Kwon, Krishna Prasad Neupane, John Marshall, and M. Mustafa Rafique. 2020. CuVPP: Filter-based Longest Prefix Matching in Software Data Planes. In *IEEE International Conference on Cluster Computing (CLUSTER)*.

[41] Butler W. Lampson, Venkatachary Srinivasan, and George Varghese. 1999. IP lookups using multiway and multicolumn search. *IEEE/ACM Trans. Netw. (TON)* 7, 3 (1999), 324–334.

[42] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. 2019. A Scalable Learned Index Scheme in Storage Systems. *CoRR* abs/1905.06256 (2019).

[43] Alex X. Liu, Chad R. Meiners, and Eric Torng. 2016. Packet Classification Using Binary Content Addressable Memory. *IEEE/ACM Trans. Netw. (TON)* 24, 3 (2016), 1295–1307.

[44] Rick McGeer and Praveen Yalagandula. 2009. Minimizing Rulesets for TCAM Implementation. In *IEEE International Conference on Computer Communications (INFOCOM)*.

[45] Nick McKeown, Thomas E. Anderson, Hari Balakrishnan, Guru M. Parulkar, Larry L. Peterson, Jennifer Rexford, Scott Shenker, and Jonathan S. Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review (CCR)* 38, 2 (2008), 69–74.

[46] Chad R. Meiners, Alex X. Liu, Eric Torng, and Jignesh Patel. 2011. Split: Optimizing Space, Power, and Throughput for TCAM-Based Classification. In *ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*.

[47] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. 2010. Fast Regular Expression Matching Using Small TCAMs for Network Intrusion Detection and Prevention Systems. In *USENIX Security Symposium*.

[48] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. 2009. PortLand: a scalable fault-tolerant layer 2 data center network fabric. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[49] Thomas Narten, Manish Karir, and Ian Foo. 2013. Address Resolution Problems in Large Data Center Networks. *RFC* 6820 (2013), 1–17.

[50] NVIDIA. 2020. NVIDIA Mellanox BlueField SmartNIC for Ethernet. https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[51] NVIDIA. 2021. NVIDIA BLUEFIELD-2 DPU. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf.

[52] University of Oregon. 2022. University of Oregon Route Views Project. https://www.routeviews.org/routeviews/.

[53] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The Design and Implementation of Open vSwitch. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[54] Danny Pinto. 2021. What will happen when the routing table hits 1024k? https://blog.apnic.net/2021/03/03/what-will-happen-when-the-routing-table-hits-1024k/.

[55] Boris Pismenny, Liran Liss, Adam Morrison, and Dan Tsafrir. 2022. The benefits of general-purpose on-NIC memory. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[56] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2020. A Computational Approach to Packet Classification. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.

[57] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2022. A Computational Approach to Packet Classification. *IEEE/ACM Trans. Netw. (TON)* 30, 3 (2022), 1073–1087.

[58] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2022. Scaling Open vSwitch with a Computational Cache. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[59] Pedro Reviriego, Gil Levy, Matty Kadosh, and Salvatore Pontarelli. 2022. Algorithmic TCAMs: Implementing Packet Classification Algorithms in Hardware. *IEEE Commun. Mag.* 60, 9 (2022), 60–66.

[60] Ori Rottenstreich, Marat Radan, Yuval Cassuto, Isaac Keslassy, Carmi Arad, Tal Mizrahi, Yoram Revah, and Avinatan Hassidim. 2014. Compressing Forwarding Tables for Datacenter Scalability. *IEEE Journal on Selected Areas in Communications (JSAC)* 32, 1 (2014), 138–151.

[61] Yaniv Sadeh, Ori Rottenstreich, and Haim Kaplan. 2022. Minimal Total Deviation in TCAM Load Balancing. In *IEEE International Conference on Computer Communications (INFOCOM)*.

[62] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Hamed Haddadi, Gianni Antichi, and Roberto Bifulco. 2020. Running Neural Networks on the NIC. *arXiv:2009.02353* (2020).

[63] Snort. 2022. Snort Open Source Intrusion Prevention System. https://www.snort.org/.
[64] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet Classification Using Tuple Space Search. In *ACM Special Interest Group on Data Communication (SIGCOMM)*.
[65] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: a data plane architecture for per-packet ML. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[66] Tushar Swamy, Annus Zulfiqar, Luigi Nardi, Muhammad Shahbaz, and Kunle Olukotun. 2023. Homunculus: Auto-Generating Efficient Data-Plane ML Pipelines for Datacenter Networks. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[67] Synopsis. 2022. DesignWare Ternary Content-Addressable Memory Compilers. https://www.synopsys.com/dw/ipdir.php?ds=dwc_tcam_memory_compilers.
[68] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*.
[69] Marcel Waldvogel. 2000. Fast longest prefix matching: algorithms, analysis, and applications. *Ph.D. dissertation, ETH Zürich* (2000).
[70] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. 2018. Training Deep Neural Networks with 8-bit Floating Point Numbers. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.
[71] Priyank Ramesh Warkhede, Subhash Suri, and George Varghese. 2004. Multiway range trees: scalable IP lookup with fast updates. *Comput. Networks* 44, 3 (2004), 289–303.
[72] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[73] Weisstein, Eric W. 2022. Dyck Path. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/DyckPath.html.
[74] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *CoRR* abs/2004.09602 (2020).
[75] Xilinix. [n. d.]. Xilinix Ultrascale+ datasheet. https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale.html.
[76] Zhaoqi Xiong and Noa Zilberman. 2019. Do Switches Dream of Machine Learning?: Toward In-Network Classification. In *ACM Workshop on Hot Topics in Networks (HotNets)*.
[77] Minxian Xu, Wenhong Tian, and Rajkumar Buyya. 2017. A survey on load balancing algorithms for virtual machines placement in cloud computing. *Concurr. Comput. Pract. Exp.* 29, 12 (2017).
[78] Tong Yang, Gaogang Xie, Alex X. Liu, Qiaobin Fu, Yanbiao Li, Xiaoming Li, and Laurent Mathy. 2018. Constant IP Lookup With FIB Explosion. *IEEE/ACM Trans. Netw. (TON)* 26, 4 (2018), 1821–1836.
[79] Marko Zec and Miljenko Mikuc. 2017. Pushing the envelope: Beyond two billion IP routing lookups per second on commodity CPUs. In *IEEE International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*.
[80] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. 2014. Automatic Test Packet Generation. *IEEE/ACM Trans. Netw. (TON)* 22, 2 (2014), 554–566.
[81] Kai Zheng, Chengchen Hu, Hongbin Lu, and Bin Liu. 2006. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM Trans. Netw. (TON)* 14, 4 (2006), 863–875.