2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS) | 978-1-6654-7177-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICDCS54860.2022.00044

ADA: Arithmetic Operations with Adaptive TCAM Population in Programmable Switches

Mojtaba Malekpourshahraki
University of Illinois at Chicago
Chicago, IL, USA
mmalek3@uic.edu

Brent E. Stephens
University of Utah
Salt Lake City, UT
brent@cs.utah.edu

Balajee Vamanan
University of Illinois at Chicago
Chicago, IL, USA
bvamanan@uic.edu

***Abstract*—In-network applications, such as congestion control, load-balancing, and policy enforcement, require complicated arithmetic operations to track networking parameters. Unfortunately, programmable switches that implement protocol independent switch architecture (PISA) support only a limited set of arithmetic operations, such as addition and subtraction, to guarantee high packet throughput. Existing work addresses this problem by implementing unsupported operations (e.g., multiplication) using TCAM match-action tables; they use wildcards to match over a range of operand values. However, because TCAM is a scarce resource, operators must make a difficult trade-off between accuracy and TCAM occupancy. This problem leads to large and unpredictable errors, and also limits the applicability of in-network computing to many applications.**

In this paper, we propose *ADA*, a practical, lightweight approach to reduce TCAM entries without sacrificing accuracy by exploiting the value distribution of operands. *ADA* tracks the operands' distribution via a simple binning mechanism to determine the most accessed interval in the domain space of operands and allocates more (or less) entries based on the observed distribution. Our proposed mechanism, (1) saves TCAM space for other applications by aggregating entries that are unused or less popular, and (2) reduces average error by assigning more TCAM entries to intervals with a higher probability of occurrence (and sub-divides these intervals further, if needed). We implement *ADA* on P4 on a 100 Gbps Barefoot Tofino switch and demonstrate its efficacy by deploying it in existing state-of-the-art in-network applications; *ADA* imposes a negligible overhead of less than 2% in the switch data plane and about 5% in the control plane. We further evaluate *ADA* using our C++ and ns-3 simulators over two existing arithmetic-heavy applications (i.e., Nimble and RCP) to demonstrate that *ADA* can achieve performance close to an ideal implementation with unlimited TCAM space.

***Index Terms*—Programmable switches, TCAM, p4lang**

I. Introduction

Recent advances in programmable networks enable operators to offload computation from end-hosts to switches. While protocol independent switch architectures (PISA) enable matching on any arbitrary packet field and perform an associated action, P4 programming language and associated libraries enable programmers to express a wide range of computations at line rate in the switch data plane [1, 2]. Today's PISA Switches use reconfigurable match-action tables that use simple arithmetic logic units (ALU) to support a limited set of operations, such as addition, subtraction, and bit shifts. In addition, RMT switches provide registers that can be used for stateful operations (e.g., counters). Taken together, these features can be used to offload *some* computations to network switches and design efficient in-network applications.

Despite the flexibility and programmability that PISA provides, designing high throughput RMT switches is challenging. Supporting computations at high line rates implies that all per-packet operations must finish in a small number of clock cycles. This limitation makes it prohibitively difficult to design and implement complicated operations in hardware. For instance, multiplication and division require *tens* of clock cycles. Therefore, today's programmable switches (e.g., Tofino [3]) do not *natively* support multiplication and division. Unfortunately, the lack of native support for these operations limits many *important* in-network applications such as RCP [4] and XCP [5]. Table I shows a list of existing work from different areas that require complicated operations in their design.

Existing proposals use lookup tables to implement operations that are not natively supported by the switch hardware. Fortunately, ternary content-addressable memory (TCAM) can be used for implementing lookup tables at line rate [12, 13]. One advantage of using TCAMs is the ability to encode a range of operand values with a single entry using wildcards. This approach is used in a recent paper to implement three sample in-network applications [12]. Another paper, InREC [13], uses this approach to compute a larger set of arithmetic operations. In addition to multiplication and division, InREC supports single-operand operations with real numbers such as radical and logarithm. Both these papers exploit wildcards to map a range a operand values to one entry and populate TCAM tables with predefined ranges of operands along with the associated result of the operation. During lookups, if there are multiple matching entries, the longest prefix match (LPM) is used to resolve conflicts and provide the ultimate answer.

Prior work populates TCAM by using wildcards to represent a set of ranges for each operand. However, they are agnostic of the *distribution* of operand values and use equal-sized ranges. As a result, they suffer from the following three shortcomings: First, populating TCAMs with equal-sized ranges is not optimal when the operands span a long range. For operations that require two or more operands, such representation also leads to a *combinatorial* blowup in the number of TCAM entries. Further, most network parameters

2575-8411/22/$31.00 ©2022 IEEE
DOI 10.1109/ICDCS54860.2022.00044

TABLE I: List of approaches with the in switch arithmetic requirements

Category	Work	Arithmetic		Error propagation	Target
		Multiplication /Division	Floating point		
Congestion Control	RCP [4]	7	0	Yes	Converge to the correct rate
	XCP [5]	0	4	Yes	Converge to the correct rate
	QCN [6]	1	0	Yes	Quantized congestion notification
	s-PERC [7]	1	0	No	Calculate Max-Min fair rates
Load Balancing	Conga [8]	1	0	Yes	Congestion aware load balancing
Measurement	Precision [9]	1	0	No	Heavy hitter detection
Fairness	Nimble [10]	1	0	Yes	Tracking buffer size
	Ether [11]	1	0	No	Providing both fairness and LSTF

cover only a limited range over the domain of operands. For example, a 32-bit counter counting the queue occupancy would never exceed the maximum queue size (e.g., 256KB), so populating the TCAM to cover the entire domain of operand values is often wasteful. Because TCAM is an expensive and scarce resource that is needed for core network functions such as forwarding and switching, equal-sized ranges either waste TCAM space or provide low accuracy. Second, even though wildcard matches can reduce the number of TCAM entries, they cause a large error, especially for larger numbers. Using LPM, larger numbers usually have a shorter matching prefix, which leads to low accuracy. Such large errors may adversely affect the performance of applications, such as rate limiters, which usually deal with large numbers (e.g, $40\,000\,\mathrm{Mbps}$). Finally, in most instances, network parameters are not uniformly distributed over the operand domain. For example, queue size in a DCTCP congestion control scheme is expected to vary between zero and ECN threshold. Our experiments confirm that *many parameters have a narrow operating range and exhibit highly-skewed distributions*. This key insight motivated us to design a system that populates TCAMs using the knowledge of the operand range and distribution, and dynamically adapts over time.

We present *ADA* to address the problem of optimally populating TCAM tables for implementing operations that are not natively supported by today's programmable switches. The *key* idea behind *ADA* is to learn the *operating range* and *distribution* of operand values in their domain, and use this knowledge to populate the tables to achieve an optimal trade-off between accuracy and table size. Because learning the distribution happens over a relatively long duration, it is done in the slower and flexible *control plane*. However, the learned distribution is used to populate the TCAM tables in the data plane to be used for performing operations at line speed. *ADA* includes a lightweight monitoring system that learns the distribution of the operand values in the control plane and updates TCAM entries according to the distribution. Our evaluations, including a real implementation, demonstrate that *ADA* dramatically reduces the average error and error propagation for recursive/iterative functions, and can save substantial TCAM space without loss of performance. We make the following contributions:

- An adaptive binning algorithm to learn the distribution (i.e., PDF) of variables. The algorithm enables us to intelligently populate TCAM and save space for other applications.
- A lightweight P4-friendly implementation of the monitoring algorithm to adaptively detect the PDF of the operands via variable binning without any sampling or packet resubmit.
- A TCAM entry selection algorithm to generate an optimal lookup table that is cognizant of the distribution of operand values (i.e., more entries for intervals with higher probability of occurrence) to minimize overall average error.
- Real implementation of *ADA* on P4 in our testbed and simulations to evaluate the accuracy and overhead of our system in two real network applications as well as a comparison to an *ideal* implementation that uses unlimited TCAM space.

In summary, *ADA* enables the feasibility of in-network applications with complicated operations in today's programmable switches using a small TCAM footprint while also minimizing error.

The rest of the paper is organized as follows: Section II provides experimental evidence and motivates the need for adaptive TCAM population. Section III presents our proposal and the design details. Section V shows our experimental results and key findings. Section VI discusses related work. Section VII concludes our paper with closing remarks.

II. Motivation

PISA architecture has three components: (i) parser, (ii) match unit, (iii) action unit. When a packet arrives at the switch port, it is first processed by a programmable parser. The parser extracts desirable fields from the packet header and forwards it to an array of pipeline stages. Each stage contains a matching unit to perform either an exact match or a longest prefix match on a subset of header fields. Match units in RMT are designed to match on arbitrary header fields (i.e., table widths and depths can vary subject only to physical capacity limits) and the header format is fully customizable. If the header matches a condition, action units perform the specified operation as expressed in the P4 language. Allowed actions include modifying any field of the packet header and forwarding the packet to the traffic manager or to directly send it to the deparser/outgoing ports. Action units are also powered by Arithmetic Logic Units (ALU) to do *some* operations such as addition, subtraction, shift bits, and hash functions.

A. Switch Limitations

RMT switches guarantee high throughput (e.g., 12.8 Tbps on Tofino 2 and 25.6 Tbps on Tofino 3) by keeping the pipeline stages simple and highly parallelized. This simplification introduces three main challenges for in-network applications: (i) limited set of operations: RMT ALUs support only a small set of basic arithmetic (i.e., addition, subtraction) and logical (i.e., bit shifts) operations. Other operations such as multiplication and divisions are not supported. (ii) limited support for branches: Because branches are inherently complex to pipeline, PISA switches support only a limited number of branches. Loops are not supported since they may need several accesses to the memory (i.e., require several clock cycles). (iii) no sharing of memory between stages: To operate the memory in each stage at high speed, PISA switches keep the memory in each stage isolated and stages cannot access memory of other stages.

Network applications typically monitor and track network state (e.g., queue occupancy) and generate new signals based on algorithms that often involve complicated operations. Table I lists important applications that address various aspects of networking and shows that these applications require operations (e.g., multiplication) that are not supported by today's switches. Because of the aforementioned limitations of branching and memory accesses, we cannot emulate these operations using existing operations (i.e., emulating multiplication as a series of additions).

Programmable switches support a limited number of TCAM tables to lookup values at line rate. TCAMs can be used as lookup tables to emulate arithmetic operations as proposed in previous papers [12, 13]. Authors in [12] provide a list of building blocks to address RMT issues and implement existing applications in programmable switches by populating TCAM tables with logarithmic values and a reverse logarithm TCAM population to lookup the result (e.g., for multiplication and division). Similarly, InREC [13] benefits from using TCAMs as lookup tables and provides limited floating point support to emulate more complicated operations such as radical and logarithm.

However, existing TCAM lookup mechanisms have three main shortcomings: larger error for larger numbers, error propagation, and large table size.

Large error for larger values: Existing TCAM population mechanisms use a wildcard match in the form of $0^p1(0|1)^s\times^r$, where s is the number of significant bits to match over, and $p+s+r$ represents the number of bits in the operand value. Each entry represents a group of values, and the group size increases with r (i.e., more least significant bits are ignored using wildcards). In this case, one TCAM entry represents a group (range) of operand values. Consequently, for large values of r, the average error is large as they approximate a large set of values using a single number. For instance, when calculating 4-bit multiplication with one significant bit i.e., $(s = 1)$ and when the median value is used to represent the entry (similar to the method used in [10]), the worst-case error to lookup the result of X^2 is 8% for $X = 4$ and 35% for $X = 8$.

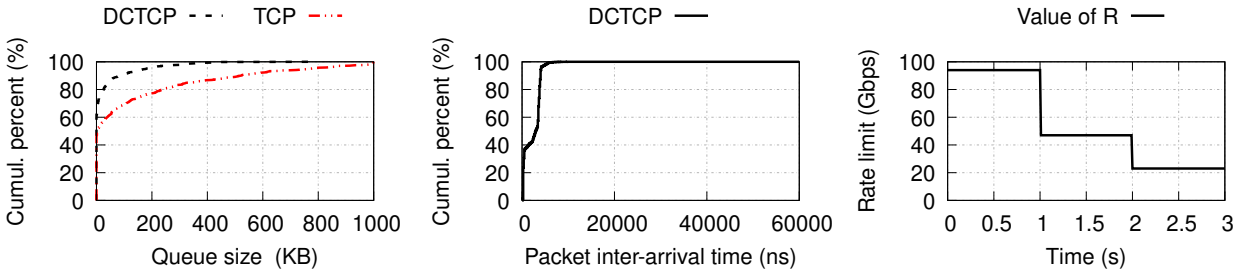
Error propagation: A small error may cause big problems in applications that perform iterative operations and error quickly adds up. Unfortunately, iterative behavior is quite common in network applications as they often perform stateful operations. For example, exponential averaging is commonly used in congestion control or to average out noise in estimations. While large errors are a problem, they can be catastrophic in applications that rely on some notion of convergence to a steady state behavior—the accumulated error may become large enough to affect convergence and compromise the stability of the system. For example, errors in congestion estimation may lead to a congestion control algorithm not converging to the desired optimal operating point and would result in under/over-utilization and/or fairness problems.

Large table sizes: Despite using wildcard matches and limiting the variables' bounds, existing TCAM population mechanisms still require large TCAM capacity. When an application calculates the result of an operation, if one or both of the operands dynamically change, the TCAM must have at least one entry to return the result for every combination of operand values. Unfortunately, many switches support only tens–hundreds of entries as TCAMs are a scarce resource needed for core network functions such as forwarding and packet classification. Thus, existing TCAM population schemes are forced to use fewer but wildcard entries at the cost of accuracy.

B. Opportunity studies

Ideally, all possible combination of the operand values is necessary for accurately emulating any arithmetic operation; however, enumerating all combinations of operand values will require prohibitively large TCAMs and is not feasible. Therefore, we make *two key observations* that enable us to drastically reduce the number of TCAM entries.

First, we observe that many important network parameters are *not* uniformly distributed and their distribution is highly skewed in the common case (i.e., more opportunity to reduce space). *Queue size* is an important network parameter that is used in many applications (e.g., many congestion control algorithms). To illustrate this phenomenon, we observe queue sizes in a realistic scenario. We set up a simple ns3 [14] data center simulation with 128-node 3-tier fat-tree topology. Severs generate random all-to-all traffic consisting of short flows (1-16 KB) and long flows (64 MB). We study the behavior of both TCP Cubic and DCTCP. Figure 1a shows the queue size at one of the ports of an edge switch (i.e., we observed similar behavior at other ports/switches). This graph clearly shows that queue sizes exhibit a skewed distribution: queue size is less than $200\,\mathrm{KB}$ for 80% and 95% of the time, in TCP Cubic and DCTCP, respectively. Therefore, if we were to use queue size in any computation, existing TCAM population schemes would needlessly waste space and/or achieve poor accuracy.



(a) Queue size in DCTCP (b) Packet inter-arrival time (c) Nimble rate

Fig. 1: Variable behaviour in different situations. (a) and (b) Cumulative distribution function (CDF) of two network parameters. (c) Change in sending rate in a rate limier

Packet *inter-arrival time* is another important network parameter that is used in a number of network applications (e.g., rate limiters, token bucket, congestion control). Therefore, we performed measurements to study this parameter. We measured packet inter-arrival times in a simple dumbbell topology with 100 Gbps links with rate limiters (the actual limit rate does not matter to these experiments). We changes the rate three times during the evaluation and each time we set it to half of the previous value. Figure 1b shows the CDF of inter-arrival times. Despite the change in rate limiter parameters, packet inter-arrivals are largely constrained to a narrow range of 120 ns–360 ns most of the time. This is not surprising because several past studies have also observed the exponential nature of packet inter-arrival times.

Second, in addition to how the values are distributed, many network parameters are *range bound* and their working range is typically *much smaller* than the domain of the variable (e.g., TTL values in IP packets). Thus, if we can estimate their working range with reasonable confidence, then we can populate the TCAM accordingly to minimize space as well as to improve accuracy. To illustrate this, we ran the same experiment with Nimble again but this time we track only the rate limit values. Figure 1c shows the result of this experiment. We ran the traffic for one second with the rate limit set to the line rate (i.e., 94 Gbps), and after one second we cut the rate in half. As shown in this figure, for the time interval 0 s–1 s, the rate is always 94 Gbps. This means that the TCAM always looks up the operand value of 94 Gbps for estimating the amount of enqueued bytes (i.e., $bytes\_enqueued = rate\_limit \times \delta T$). After one second, we change the rate to half (47 Gbps) and the TCAM looks up only 47 Gbps. As you can see, instead of populating the TCAM with all possible operand values, if we can estimate the working range (e.g., in the control plane) and populate accordingly, we can save TCAM space and also improve accuracy.

We build upon these key observations to design *ADA*. which we describe in the following section.

III. Design

ADA is an adaptive, P4 friendly, feedback-based system for efficiently populating TCAM entries to minimize average error. Figure 2 shows our overall architecture.

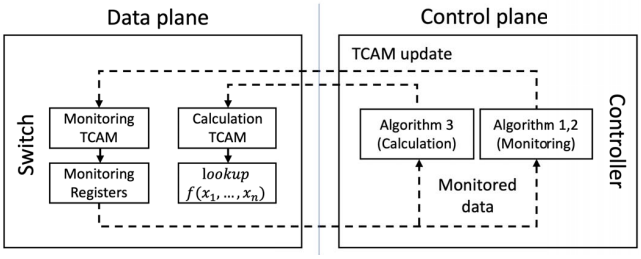


Fig. 2: The architecture of *ADA* for both data and control plane.

ADA has components in both control and data plane. We split our design into control and data plane components to optimize for dual goals of speed and optimality. The data plane component is a lightweight monitoring system for *recording* the frequency of occurrence for a range of operand values. To record this information, we use a small TCAM to match on intervals of operand values (using wildcard entries) and we increment a counter upon a match; we have one counter per interval to record hits and use registers to store these counter values. The control plane component reads the monitored statistics (i.e., register values) to infer the operating range and distribution of operands, and it uses this information to populate TCAM entries (to be used during lookups). For instance, intervals with more hits get more TCAM entries (i.e., finer granularity). The control plane component also includes an algorithm to fine tune the granularity of monitoring. That is, if a certain range has a high probability of occurrence, we can divide the range into two sub-ranges to monitor at a finer granularity and vice versa. We could not implement these sophisticated algorithms in data plane because of the lack of adequate support for branching and limited programmability in data plane [9]. Thus, by monitoring operand values in data plane, *ADA* is able to respond faster to changing dynamics; by using sophisticated algorithms to process the monitored data and improve the quality of monitored data, *ADA* is able to achieve high efficiency.

A. ADA in data plane

An efficient TCAM population requires knowing the distribution (i.e., PDF) of operands in real time. The PDF

provides two important pieces of information: the range and the distribution of the variables. *ADA* uses a monitoring system to record the histogram of operand values (i.e., find a discrete PDF of variables). The operand value is divided into smaller intervals (bins) and *ADA* assigns one register to a bin to count the number of hits when the value falls within the bounds of the interval.

Programmable switches provide limited in-stage branching, which constrains the implementation [9]. For instance, for a variable with 32 bits, PDF can be obtained by dividing the entire interval into 100k bins and assigning the TCAM space according to the frequency of hits in each bin. This logic, however, is not implementable in a P4 program due to the limitations on branching. To address this problem, we use wildcard matching in TCAMs. Each wildcard entry represents a smaller interval (bin) and there is a corresponding register associated with this bin. When a value matches an entry, the corresponding register is incremented. Figure 3 shows the general overview of this design in which $v_1, v_2, ..., v_n$ are target variables, and $r_1, r_2, .., r_n$ are tracking registers. e_i are TCAM entries for matching the values v_i. Each TCAM entry represents a bin and each bin has a separate register assigned. If a value matches an entry in TCAM (e.g., e_i), *ADA* increases the corresponding register (r_i) by one to indicate that another hit is observed in this bin. Note that, by using wildcards and longest prefix match, we can track hits in a sub-interval within an interval and so on (i.e., deeper than just one level).

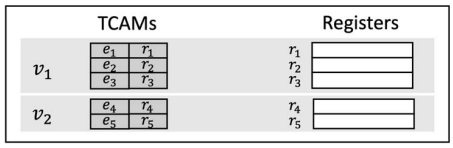
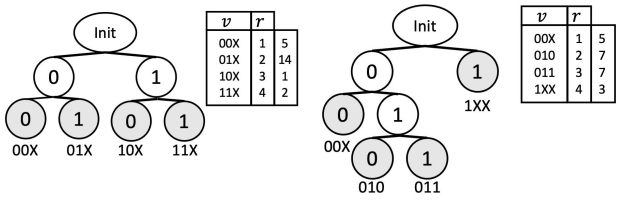


Fig. 3: Binning mechanism in data plane *ADA*

1) Binning abstraction: ADA has an adaptive monitoring system in the data plane to capture the distribution of operand values. To efficiently capture the distribution, *ADA* uses a *trie* data structure, implemented using TCAMs, to record hits to various sub-intervals (bins). Figure 4 shows two examples for an operand with four bins. In this figure, each leaf (shown as nodes with shadow) represents a bin. A binning trie is always a binary tree and starts with the root node (init). Each node has a reference to the left and the right child and a value that shows the number of hits. All hit values are initialized to zero. The path from the root to the leaf represents the TCAM entry. For example, in Figure 4a, for 3-bit operands, the bins are 00× (0–1), 01× (2–3), 10× (4–5), and 11× (6–7). Similarly, in Figure 4b, the bins are 00× (0–1), 010 (2), 011 (3), and 1×× (4–7). Note how Figure 4b has non-uniform intervals (bins). *ADA* defines a register for each leaf node in the binning abstraction model as it is shown in Figure 3. When the value matches one of the leaves, TCAM returns the register id (column shown as r) and increases the corresponding value.

2) Binning formation: Initially, *ADA* generates a trie with the same wildcard length (i.e., all intervals of the same size).



(a) Binning abstraction in *ADA* (b) Extended binary abstraction

Fig. 4: Starting point and extended binary abstraction in *ADA*.

Algorithm 1: Initialization binning tree

Definitions:
  $\mathbf{b}$: Number of significant bits;
  $\mathbf{bit(i)}$: Convert i to the the binary number;
Input:
  $\mathbf{M}$: Number of available entries;
  $\mathbf{s}$: Number of bits in operands;
Output:
  $\mathbf{\Gamma}$: Value set of leaves (ordered set)
  $\mathbf{T}$: Monitoring trie (binary tree)
Initialization:
  $\Gamma = T = \phi$;
Function `binning_table_init()`**:**
    $b = log(M)$;
    $V = \{\forall\ i \mid i \in [0, 2^b - 1]\}$
    for $\forall i \in V$ **do**
      $i \xleftarrow{s-b} \times$; `// Left shift with wildcard (×)`
      if $i \notin \Gamma$ **then**
        $\Gamma.add(i)$; `// populate TCAM entries`
        $n = Node(bit(i))$; `// create a new node (word) from bits of the number` i
        $n.value = 1$; `// initial value to 1`
        $T.add(n)$; `// add a new word to to T`
      end
    end
end

The number of wildcards depends on the number of available entries and is determined by the network operator based on TCAM capacity. If the total available entries for monitoring TCAM is M, the initial value for the significant bits is $b = log(M)$. For example, if there are only four entries, the TCAM entries would be 00X, 01X, 10X, and 11X for 3 bit operands (i.e., 2 significant bits). Algorithm 1 shows the initialization of binning TCAM table. The only input to the algorithm is available TCAM entries, M, and the length of the operand (in bits), s. s is also the maximum possible depth of the trie. The algorithm has a set of leaves (Γ) and the trie (T). This algorithm finds the set of leaves and builds the trie based on that. Figure 4a shows an example trie that this algorithm generates for 3-bit operands ($s = 3$) and one wildcard match ($b = s - 1 = 2$). The corresponding bin for each leaf (shown in gray) is shown in the figure and they are all at the same level.

3) Adaptive binning update: The initial trie divides the operand space into equal-sized intervals and it measures the number of hits in each bin; however, if a small sub-interval generates most of the hits, zooming in on that small sub-interval will help in capturing the distribution precisely. For instance, in Figure 4a, the majority of hits are in bin $01\times$ but the monitoring system has no insight into the distribution of this bin. To remedy this problem, we propose an algorithm (Algorithm 2) to selectively grow the tree based on hits. To keep the number of required entries fixed, *ADA* eliminates the bin (trie node) with the smallest hit and breaks the bin with the maximum number of hits into two sub bins. To avoid frequent changes in the trie structure, we use a threshold ($th_{balance}$) to identify when *ADA* needs to modify the trie (see line 16). Figure 4b is the result of this transition from Figure 4a. In this figure, node $01\times$ is divided into two smaller bins 010 and 011 while $10\times$ and $11\times$ were merged into one bin as $1\times\times$. Thus, our algorithm modifies the original trie to better capture the distribution of operand values without wasting TCAM space.

B. ADA in control plane

1) TCAM population: By proportionally allocating TCAM entries to intervals (bins) based on their frequency, *ADA* minimizes the average error. We include a control plane algorithm for performing this allocation. The algorithm performs a top-down traversal of the trie and allocates entries to the left and right sub-tree based on hits.

Algorithm 3 shows the TCAM population procedure. The controller first reads the number of hits in each bin from the data plane and calculates the aggregated hits for each node in the trie (i.e., the sum of hits of all nodes below this node). Then, *ADA* assigns the available entries to each node in proportion to the aggregated hits of the subtree rooted at that node. At the end of the procedure, we know the number of entries for each bin (leaf). For example, based on Figure 4b, the bin representing $1\times\times$ will get $3/(5+7+7+3) = 14\%$ of entries.

Finally, we use a recursive function *TCAMpopulation* to generate the final TCAM population. In each interval, we assign the TCAMs based on the simple mechanism shown in [10]. This function is simply replaceable with the logarithmic approach proposed in [12]. Note that each bin is independently used in these approaches to populate the TCAM based on the assigned number of entries. If there is no frequency data for any node in the binning tree, the algorithm results in an equal share of the entries for the entire sub-tree.

2) Trie expansion: ADA uses two main TCAM tables: Monitoring TCAM and calculation TCAM. Monitoring TCAM is used to model the trie for each variable, whereas the calculation TCAM is the main lookup TCAM that the P4 program uses to fetch the result of the operation. Both of these tables use the same TCAM hardware from the switch which is limited in the number of entries. If the distribution of the hits for a given variable is uniform, it is better to assign fewer entries to the monitoring TCAM, whereas if the distribution is skewed, then it is better to assign more entries to the monitoring TCAM.

Algorithm 2: Adaptive binning tree modification in TCAM population

```
1  ] Definitions:
2    int max(i) : Maximum value for wildcard i;
3    int min(i) : Minimum value for wildcard i;
4    Node getMin(i) : Item with minimum value in i;
5    Node leaves(T) : Fetch the leaves of the Tree T;
6  Input:
7    T : Monitoring tree
8    adjust_threshold : Expanding trie threshold
9    hit_threshold : Dividing monitoring node threshold
10 Output:
11   T Updated optimized binning table
12 Function receivedQuery():
       /* Increase the number of monitoring TCAM */
13     if change in depth ≥ th_expansion then
14       devideHighHitNode(T);
15     end
       /* Balance the tree before generate the new
          TCAM population                          */
16     if (getMax(T)−getMin(T))/getMax(T) ≥ th_balance then
17       removeLowHitNode(T);
18       devideHighHitNode(T);
19     end
       /* Generate TCAM entries                    */
20     return inorderTraverse(T);
21 end
22 Function removeLowHitNode(T):
23     Γ = leaves(T);
24     list P; // Create parent list
25     for i = 0; Γ.size() < i; i++ do
26       if Γ[i].parent == Γ[i+1].parent then
27         T[i].parent.hits =
            T[i].parent.hits + T[i+1].parent.hits
           P.add(T[i].parent)
28       end
29     end
30     p = getMin(P);
31     p.left = null;
32     p.right = null;
33 end
34 Function devideHighHitNode(T):
35     Γ = leaves(T);
36     n = max_{∀i∈Γ}(i)
37     m = max(n) − min(n); // Find distance
        between max and min in a wildcard
38     m = m/2 + 1; // Find the most significant ×
39     n.left = Node(m ∨ x); // Unwrap left node
40     m = m ⊕ ∼0;
41     n.right = Node(m ⊕ x); // Unwrap right node
42 end
```

To address this problem, *ADA* uses the trie depth as an indicator to detect the type of the variable. If the depth is increasing at each iteration, the value distribution is skewed as Algorithm 2 deepens the tree. In this case, the monitoring TCAM must increase. We use $th_{expansion}$ to identify when *ADA* needs to expand the monitoring TCAM by adding new entries (see line 13 in Algorithm 2). On the other hand, if the trie depth is not increasing, this indicates that the monitoring TCAM size is suitable for tracking the variable or the distribution follows a uniform PDF. In this case, we do not add any new entry but *ADA* might still adjust the table due to the change in the variable behavior. However, *ADA* does not decrease the size of the tree.

IV. Methodology

To perform a comprehensive evaluation, we implement *ADA* in three different platforms: First, we develop our C++ simulator to evaluate the proposed algorithms (Algorithms 1 and 2) without any interfering networking parameters. In this experiment, we generate numbers based on random variables with different distributions and run our binning algorithms in *ADA* to show they converge to the PDF of the random variables. In addition, we test the adaptation mechanism in algorithm 2 by choosing a very small significant bit for the initial trie. Finally, we study error propagation and the effect of error propagation on two simple applications.

Second, we implement *ADA* in P4 [15] on a commercial PISA switch in our local cluster. We ran *ADA* as a part of the P4 program on a Barefoot Tofino [16] Wedge 100BF-32X Ethernet switch with a line rate of 100 $Gbps$. The cluster has three servers forming a star topology. Each server has an 8-core/16-thread Intel Xeon 1.80 GHz CPU and 64 GB of memory. These servers run Ubuntu 18.04, and they use 100 $Gbps$ Mellanox ConnectX-5 [17] NIC to connect to one port of the switch. We code the data plane part of *ADA* as a P4 program and we implement the control plane part using a gRPC-based client to dynamically populate TCAM tables on the switch.

Finally, we implement *ADA* in ns3 simulator [14], to measure the performance of *ADA* on a large-scale topology. We implement *ADA* in ns3 switching module and set up a leaf-spine topology with 400 servers and 20 ToR switches. In our experiments, all connections (host-to-switch and switch-to-switch) are connected using 100 $Gbps$ links with a link delay of 1 μs. We ran three experiments with TCP (baseline), RCP [4], and Nimble [10] with precise (ideal, without TCAM lookups) calculation and with *ADA* (using our TCAM population and lookup) to demonstrate that *ADA* performs close to the ideal approach. In experiments with Nimble, we use DCTCP senders.

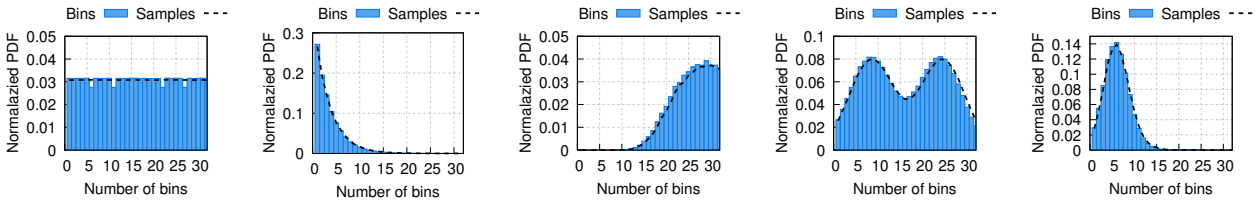
In our experiments, we used switches with a buffer capacity of 400 KB per port. We also set our expansion threshold in algorithm 2 as $th_{expansion} = 2$—the trie expands if the depth of the three increases by more than two. We also set the balance threshold as $th_{balance} = 20\%$. This means that *ADA* balances the trie if the ratio of the minimum and the maximum hits is more than 20%.

Algorithm 3: Operation TCAM population

Definitions:
  T = binning tree;
  $b.left$ = left child of b;
  $b.right$ = right child of b;
  $b.val$ = entry that b represent in trie; // e.g., 01×
  M = number of available entries;
Initialization:
  $\forall b \in B, w(b) = 0.5$;
Output: Optimized table L
Function `main()`**:**
  $updateFreq(T.root)$; // update all parents' frequencies
  for $\forall\ b \in T$ **do**
    $w(b.right) = \frac{f(b.right.value)}{f(b.value)}$;
    $w(b.left) = \frac{f(b.left.value)}{f(b.value)}$;
  end
  TCAMpopulation$(B.root.left, M)$;
  TCAMpopulation$(B.root.right, M)$;
return
Function `updateFreq(`b`)`**:**
  if $b\ is\ a\ leaf$ **then**
    return $b.value$;
  end
  $b.value =$
   $updateFreq(b.left) + updateFreq(b.right)$
  return $b.value$;
end
Function `populateTable(`b`,`M`)`**:**
  $w_{right} = \frac{f(b.right.value)}{f(b.value)}$;
  $w_{left} = \frac{f(b.left.value)}{f(b.value)}$;
  $populateTable(b.left, M \times w_{left})$
  $populateTable(b.right, M \times w_{right})$
  if $b\ is\ a\ leaf$ **then**
    $b = min(b \mid 2^b(\frac{s-b}{2}) = M)$
    $V = \{i | i \in [2^b, 2^{b+1} - 1]]\}$
    $T = \{i | i \in [0, 2^b - 1]\}$
    while $\forall i \in V$ **do**
      for j=*1;*$s \leq j$;*j++* **do**
        $i \overleftarrow{\ }^{j}\ x$; // Left shift with don't care (x) as input
        $T.add(i)$
      end
    end
  end
end

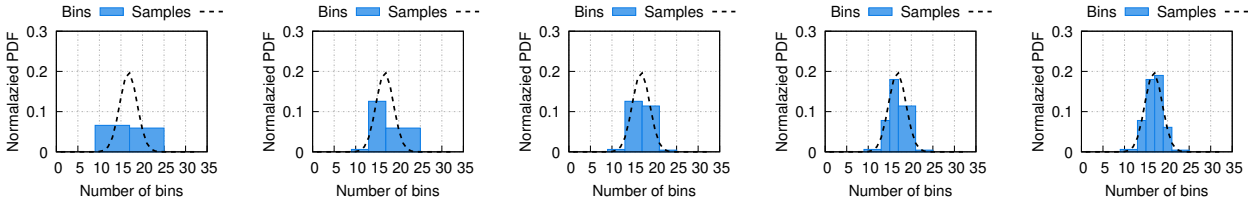
V. Evaluation

In this section, we first evaluate *ADA* with regard to the algorithm accuracy and convergence using our simulator to



(a) Uniform dist. (b) Exponential dist. (c) Fisher F dist. (d) Two Gaussian dist. (e) Gaussian + Fisher F

Fig. 5: Convergence of *ADA* to different distributions after monitoring system reaches to a steady state.



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4 (e) Iteration 5

Fig. 6: Iterations in algorithm 2 for $b = 1$.

show that *ADA* adapts with any operand distribution (i.e., PDF). Second, we provide the result of our testbed with a barefoot switch [16] to demonstrate the performance and scalability of both our control and data plane mechanisms. Finally, we evaluate *ADA* over a large scale datacenter with a leaf-spine topology.

A. Network independent C++ simulator

We developed a C++ simulator to determine the accuracy of our proposed algorithm without any networking parameters. Our simulator generates different PDFs and uses binning algorithm proposed in algorithms 1 and 2 to converge to the original distribution. We use dotted lines to show the distribution and the boxes to show bins.

1) Accuracy and integrity: To show that *ADA* can model a wide range of distributions, we set up our simulation to generate various distributions. Figure 5 shows the results of Algorithms 1 and 2 for creating the bins after reaching to a steady sate (until the condition at line 13 in Algorithm 2 is not satisfied). We use a bin size of 2000 and we use a 32 bits integer value with the domain of $[0, 650000]$. Figure 5a shows a Uniform distribution with normalized value of 0.03. This figure shows that the bins can estimate the uniform distribution accurately. Uniform distribution is common in parameters with the same probability of hits across the variable. Figure 5b shows the result of an exponential distribution with $\lambda = 10$ with the same bin size and variable range. This experiment can model the variable with the heavy hits on the lower numbers and fewer hits when the value is higher. An example of a variable that can form an exponential distribution is queue size. Figure 5c shows a Fisher F distribution with parameters $d_1 = 100$ and $d_2 = 20$. This distribution is used to model values with a heavy-tail hit.

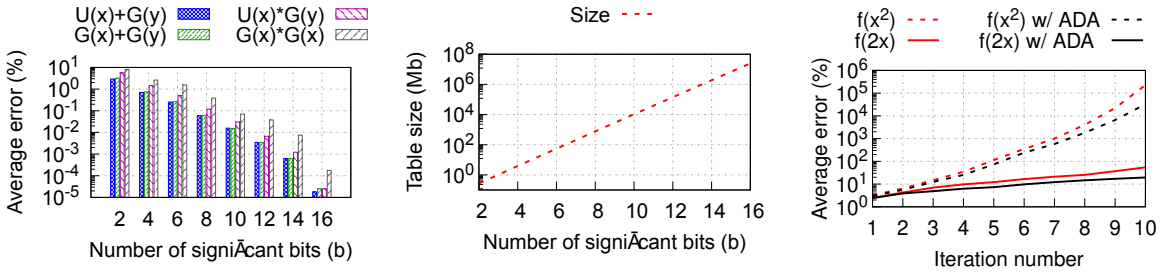
To evaluate *ADA* with more complicated scenarios, we model two other combined distributions. Figure 5d shows the sum of two independent Gaussian distributions $(G1 + G2)$, with parameters $G1(16000, 10000)$ and $G2(48000, 10000)$ with same variance $\sigma = 10000$ but different mean $\mu_1 = 16000, \mu_2 = 48000$ on a range of $[0, 650000]$. The result of $G1 + G2$ has two picks and *ADA* can accurately distribute the bins to model the targeted distribution. Similarly, in Figure 5e, we use a sum of an exponential with $\lambda = 10$ and a Gaussian distribution ($\sigma = 10000$, $\mu_1 = 16000$). The results shows that the algorithm 2 can model combined and random distribution.

2) Adaptive increment: As we mentioned in section III-B2, trie starts with a default value for b and *ADA* increases the number of entries by adding more nodes to the trie if the distribution is skewed. To test this procedure, we use a Gaussian distribution with a median of 4000 and a variance of 32500 as a random generator and we used 2000 as bin size. Figure 6 shows this transition from $b = 1$ to the next five iterations. Initially, there are two bins in Figure 6a, since we start with $b = 1$. In next iteration in Figure 6b, bigger bins divides into two bins. Iterations continue to the fifth iteration where there are a total of 6 bins.

As we expect, bins can represent the original distribution of the variable while the initial trie (Figure 6a) is not able to capture the character of the Gaussian distribution correctly. Bins are completely matched after the fourth iteration which shows that our algorithm in 2 can successfully increase the TCAM space, assigned to the monitoring so that the bins match the PDF of the variable. Note that here iteration is considered a change in trie and not the iteration of the algorithm, 3, and we did not limit the expansion of the trie to find the convergence point.

3) Error analysis: ADA decreases the error propagation and the average error. In this section, we first study an experiment to show the effect of the significant bit (b) on average error. Second, we compare the average error of *ADA* to the existing state-of-the-art algorithms for the TCAM population. All charts in this section are drawn on a logarithmic scale.

Increasing the number of significant bits in populating



(a) Average error vs significant bit. (b) Table size vs significant bit. (c) Average error propagation.

Fig. 7: (a) Average error on increasing significant bit for Gaussian and uniform PDFs. (b) Table size on increasing the size of the significant bit. (c) Average error propagation for two functions x^2 and $2x$. Y-axes is in logarithmic scale.

TCAMs in algorithm 3 increases the function accuracy; however, a large significant bit may need a large TCAM size. To see the effect of error, we use a Gaussian distribution with a median of 4000 and variance of 32500 and Uniform distribution in the range of $[0, 650000]$ for our two variables, and we performed a sum and a multiplication of over two variables. For instance, $U(x) + G(y)$ represents a sum of two variables with x with Gaussian and y with Uniform distribution. Figure 7a shows the result of the average error with different values with significant bits. In all cases, when the significant bit increases, the error reduces significantly. When both variables are Gaussian the error is the maximum regardless of the number of significant bits.

Similarly, Figure 7b, shows the required table size for varying numbers of the significant bits (s). When s increases, the size of the table prohibitively increases, and the table size increases exponentially by the number of significant bits.

4) Error Propagation: Most in-network algorithms are iterative and the results of computation are fed back until some form of convergence is met. In such cases, even small errors can quickly accumulate, leading to unacceptably large errors. To understand the effect of error propagation, we run an experiment comparing two functions $f(x) = 2x$ and $f(x) = x^2$. In the experiment, we assign the computed value (e.g., $2x$) back to x and iterate 10 times. The effect is a recursive computation (i.e., f(f(f(...(x))))). Similar to the previous experiment, we use the same Gaussian distribution with a median of 10 and a variance of 100 for variable x. Figure 7c shows the result of our experiment. This figure shows that the nature of the function plays a critical role in error propagation; higher order functions tend to suffer more. As expected, x^2 is prone to more error propagation compared to $2x$. At the end of 10 iterations, x^2 shows 10813% and 70482% errors for the $f(x)$ without *ADA* and $f(x)$ with *ADA* respectively, while these errors for $2x$ is only 7% and 21%. This shows that the error propagation depends on the function itself more than the population mechanism.

B. Testbed Experiments

In this section, we demonstrate the feasibility of the implementation of *ADA* in hardware. We also show that *ADA* is crucial for in-network applications especially when the number of TCAM entries is limited. Finally, we show the overhead of *ADA* in the programmable data plane and control plane in terms of the number of stages and number of reads and writes that the control plane.

1) Experiment with limited entries: We start by evaluating an in-network rate limiter, Nimble [10], that limits the sending rate of traffic classes. Nimble requires multiplication of rate to the packet inter-arrival time. Once the rate is determined, the control plane populates the TCAM table based on the rate. If the rate changes at any time, the entire TCAM must be updated from the control plane again. This prevents approaches like Nimble from changing the rate from the data plane (e.g., to design a work-conserving decentralized rate limiter).

To demonstrate this problem, we set up an experiment with a single flow between two machines. In this experiment, we use *iperf3* to generate traffic from a client to a server at a full line rate, and we enabled DCTCP on both client and server. These experiments use 16 parallel iperf3 connections to fully utilize the link. We also set the rate limiter to 24 $Gbps$. After 3 ms we change the Nimble setting to limit the flow to 12 $Gbps$. We use a total of 128 entries for approximate multiplication and 12 entries for the monitoring. We ran the experiment with Nimble without a TCAM update from the control plane (In the proposal, the controller updates TCAMs when it changes the rate), and Nimble with *ADA* (i.e., includes TCAM update). In Nimble with *ADA*, we implement only monitoring for the rate variable. Figure 8 shows the result of this experiment. When the rate changes, Nimble generates the result of the multiplication with an extremely large error, which causes Nimble to drop packets incorrectly. On the other hand, Nimble with *ADA* detects the new value in rate after a few iterations and updates the TCAM from the control plane. This experiment shows that applications with *dynamic* arithmetic requirements can perform well using *ADA* without incurring high TCAM overhead.

2) Scalability Analysis: In this section we analyze the overhead of *ADA* on programmable switches and control plane.

Control plane delay *ADA* uses feedback from the data plane to adaptively converge to the correct configuration of the monitoring TCAM and optimal population in calculation TCAM. However, this process is not instantaneous and requires some time to converge. In addition, reading a trie and

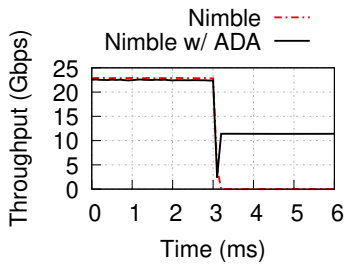


Fig. 8: Nimble Throughput with and without using ADA

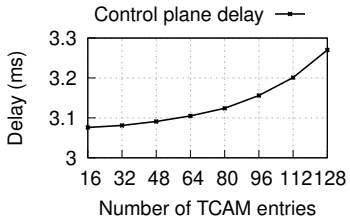


Fig. 9: Delay of control plane to update TCAMs.

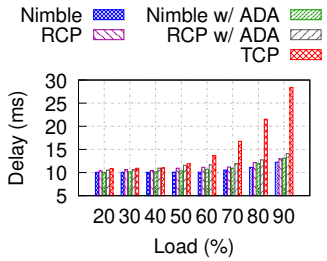


Fig. 10: FCT of short flows in a large scale data center.

updating the calculated TCAM entries from the control plane adds some delay to the the application that runs over *ADA*. If the behavior of the application changes rapidly, *ADA* needs to change the TCAM population quickly to provide good performance.

We measured the delay of our control plane program in generating the optimal TCAM entries upon a new change in the network state. We ran Nimble at line rate ($95Gbps$) first and then after 3 seconds we cut the rate to half. We varied the number of TCAM entries from 16 to 128 (an increase of 16 entries in each experiment). Figure 9 shows the delay that *ADA* takes to converge to the optimal TCAM population for varying number of TCAM entries. The delay for 128 entries is about $3.15\ ms$ which is sufficient for many in-network applications.

Switch overhead To study the overhead of the *ADA* on the switch and the control plane, we analyze our monitoring system for the TCAM population. We populate the table using the result of the operations. The first column in Table II presents an overview of the resources that *ADA* needs in terms of the number of RMT stages. $ADA(\Delta T)$ means we only use *ADA* for ΔT, $ADA(R)$ means we only use *ADA* for variable R and $ADA(\Delta T, R)$ means *ADA* is used for both variables. This table shows that *ADA* needs fewer stages when only one variable needs to be monitored.

We also measured the average read and write requests that our control plane program sends to the switch. We set up Nimble to work at line rate ($95\ Gpbt$) and we reduced the sending rate to half. We start with 8 entries for monitoring the values (trie). Table II shows the number of reads and writes during this evaluation. The average number of reads is higher than 8 for both $ADA(R)$ and $ADA(\Delta T)$ and higher than 16 for $ADA(\Delta T, R)$ because adaptively increasing the monitoring entries increases the number of the register that *ADA* needs to read. The number of reads for $ADA(R)$ is more compared to $ADA(\Delta T)$ because ΔT is less skewed and the hits are more spread across the value range. The average number of writes for $ADA(\Delta T, R)$ is the most since both variables need periodic updates. Similar to reads, we see the number of writes is always more for $ADA(R)$ than $ADA(\Delta T)$ because R is more skewed.

C. Large scale simulation

To show that *ADA* works as expected in reducing the error *at scale*, we implement the complete version of *ADA* for both

TABLE II: Resource usage of *ADA* and control plain overhead.

Variables	No. of stages	No. of reads	No. of writes
$ADA(R)$	2	12.32	73.31
$ADA(\Delta T)$	2	9.27	32.63
$ADA(\Delta T, R)$	3	24.54	98.43

RCP and Nimble in *ns3* simulator and evaluate it for a large network. We used a leaf-spine topology with 10 spine and 20 leaf switches and 400 servers. All links are $100\ Gbps$ and RTT of the longest path with 4 hops is 80 µs. We used two types of flows: long flows which are $1024\ KB$ and short flows which vary from $16 - 64\ KB$. The workload consists of a mix of 80% of short flows and 20% of long flows, and an incast traffic with the average fan-in degree of 32. The generated traffic is based on a typical heavy-tailed flow distribution and we vary the network load from 20% to 80%. Figure 10 shows the flow completion time (FCT) of all short flows in the network for TCP (baseline), and Nimble, and RCP, with and without *ADA* (uses exact/ideal computation). We clearly observe that short flow FCTs achieve similar delay using *ADA* in both RCP and Nimble as they would in an idealized system that always produces 100% accurate results. This experiment shows that *ADA*'s performance is close to ideal while requiring a small number of TCAM entries.

VI. Related Works

Sharma *et al.* provide building blocks to address the lack of complicated arithmetic in ALU [12]. The authors provide a formulation to populate the TCAM entries using the total number of bits in variables and desired accuracy. This approach, however, does not consider any variable limit or variable distribution to populate the table. Similar to the formula in [12], Nimble [10] provides an algorithm to calculate the TCAM population. This algorithm also does not consider the variables' range and distribution.

PRECISION [9] is a heavy hitter detection algorithm that uses probabilistic re-circulation to detect elephant flows on programmable switches. This approach requires calculation of Mean Square Error (MSE), and the authors used the same technique from [12] to populate the TCAM table for their calculation. Similar to [12] this approach also does not exploit variable range or distribution to populate the table.

InREC enables programmable switches to support in-network real-value operations [13]. The main motivation of InREC is to offload CPU-demanding operations to pro-

grammable switches and to reducing the end-host load. InREC provides a tree abstraction for arbitrary formulation to reduce repetitive calculations. Authors also considered the variables' bounds, but they do not consider variable distribution. This approach requires large TCAMs, which is costly.

Our proposal, *ADA*, is complimentary of existing works in this problem space. *ADA* learns the operands' range and distribution, and populates the TCAM accordingly. Existing TCAM population schemes such as logarithmic population and naive population can be used in conjunction with our system. In all of our experiments for Algorithm 3, we used a naive population scheme similar to [12].

VII. Conclusion

Today's programmable switches have the potential to improve existing applications and enable new applications by providing customized packet processing in the network. However, realizing their potential requires us to sidestep some of the architectural bottlenecks in these switches. The lack of support for common arithmetic operations, such as multiplication and division, is a major limitation for several important in-network applications. While using TCAMs for lookup tables to realize these operations is a good first step, TCAMs are also a scarce resource.

In this paper, we introduced *ADA* that exploits the operands' range and distribution to *drastically* reduce the number of TCAM entries while minimizing error. Further, *ADA* dynamically adapts to changes in the operands' range and distribution. Lastly, by exploiting the strengths of both control and data plane, *ADA* is able to scale well without incurring high overheads. As programmable switches become more mainstream, approaches such as *ADA* will be needed to expand the applicability of programmable switches to a wider set of applications.

References

[1] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[2] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[3] Barefoot. Barefoot Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html#tofino, 2017.

[4] Nandita Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. Citeseer, 2008.

[5] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, 2002.

[6] Mohammad Alizadeh, Berk Atikoglu, Abdul Kabbani, Ashvin Lakshmikantha, Rong Pan, Balaji Prabhakar, and Mick Seaman. Data center transport mechanisms: Congestion control theory and ieee standardization. In *2008 46th Annual Allerton Conference on Communication, Control, and Computing*, pages 1270–1277. IEEE, 2008.

[7] Lavanya Jose, Stephen Ibanez, Mohammad Alizadeh, and Nick McKeown. A distributed algorithm to calculate max-min fair rates without per-flow state. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–42, 2019.

[8] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al. Conga: Distributed congestion-aware load balancing for datacenters. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 503–514, 2014.

[9] Ran Ben-Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. Efficient measurement on programmable switches using probabilistic recirculation. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 313–323. IEEE, 2018.

[10] Vineeth Sagar Thapeta, Komal Shinde, Mojtaba Malekpourshahraki, Darius Grassi, Balajee Vamanan, and Brent E Stephens. Nimble: Scalable tcp-friendly programmable in-network rate-limiting. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, pages 27–40, 2021.

[11] Mojtaba Malekpourshahraki, Brent Stephens, and Balajee Vamanan. Ether: providing both interactive service and fairness in multi-tenant datacenters. In *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*, pages 50–56, 2019.

[12] Naveen Kr Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 67–82, 2017.

[13] Matthews Jose, Kahina Lazri, Jérôme François, and Olivier Festor. Inrec: In-network real number computation. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 358–366. IEEE, 2021.

[14] NS-3 network simulator. http://www.nsnam.org/.

[15] P4 Language Consortium. https://p4.org/.

[16] Barefoot. Barefoot Tofino. https://www.barefootnetworks.com/technology/#tofino, 2017.

[17] Mellanox Technologies. ConnectX-5 EN Card. http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_EN_Card.pdf.