uc3m | Universidad Carlos III de Madrid

e-Archivo

This is a postprint version of the following published document:

Ullah, A., Reviriego, P., Akram, A. & Siraj, M. N. (2021). Switch-Based High Cardinality Node Detection. *IEEE Embedded Systems Letters, 13*(4), 190–193.

DOI: 10.1109/les.2021.3062155

© 2021, IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2021.3062155, IEEE Embedded Systems Letters


# Switch based High Cardinality Node Detection

Anees Ullah[1], Pedro Reviriego[2], Adeel Akram[3] and Malik Najmus Siraj[4]

***Abstract*—The detection of supernodes with high cardinality is of interest for network monitoring and security. Existing schemes for supernode detection rely on data structures that are independent of the switching functions. This means that for each packet that traverses the switch, both the switching table and the supernode detection structure have to be checked which requires significant memory bandwidth. This can create a bottleneck and reduce the speed of the switch, especially for software implementations. In this letter, a scheme that performs supernode detection as part of Ethernet switching and does not require additional memory accesses nor separated data structures is presented. The scheme has been implemented and compared with existing methods. The results show that the proposed scheme can reliably identify supernodes while providing a speed up of more than 15% when compared with existing solutions.**

***Index Terms*—Network Monitoring, Supernodes, Anomaly Detection, Cardinality estimation.**

## I. Introduction

Identifying nodes (that typically correspond to end user devices or servers) on a network that have a large cardinality, also known as supernodes, is of interest to detect a number of threats or attacks [1]. For example, network scanning, worm spreading or Distributed Denial of Service (DDoS) attacks can be detected using cardinality-based mechanisms [2], [3], [4]. A number of algorithms have been proposed to efficiently detect supernodes, for example [5], [6], [7], [8]. The authors in [5] have designed a special data structure called "snare" that can capture supernodes and can be updated with high-speed and has a low memory-footprint. Instead, the authors in [6] have proposed a solution for a decentralized network measurement scenario to avoid massive data accumulation and processing in centralized elements. The use of Bloom filters has also been proposed to detect supernodes [7] and existing cardinality estimation algorithms have also been generalized to estimate cardinality for a set of nodes using the same data structure [8].

Although those structures can efficiently identify supernodes, all of them require to access the memory for each packet. This creates an issue for high speed switches that need to process a large number of packets per second. For example, a 400 Gb/s Ethernet port can receive close to one billion packets per second if they have the smallest size and close to 100 million packets per second for average size packets. For each of them, memory needs to be accessed to check the switching table which already puts pressure on the memory. This means that the additional per packet memory accesses to detect supernodes can reduce the packet processing capacity and speed. Therefore, it is of interest to design schemes in which both switching and supernode detection can be performed sharing the memory accesses.

In this letter, a scheme that integrates supernode detection with switching is presented and evaluated. By using a single data structure for both functions, the need for additional memory accesses is eliminated, leading to significant gains in speed that in our initial evaluation are larger than 15%. This has been achieved by leveraging the existing data structures, in particular cuckoo hashing [9], that are widely used in high-speed Ethernet switches. The main contributions of this letter are first: to show that supernode detection can be integrated with switching and second: to demonstrate that an integrated structure can reliably detect supernodes and provide significant gains in speed.

## II. Preliminaries

This section discusses the necessary background for understanding of the proposed idea i.e. the cardinality estimation and the cuckoo hashing.

### A. *Cardinality Estimation*

Over the years, researchers have developed efficient high-speed data structures and algorithms for cardinality estimation with low-memory footprint. The seminal work was done by the authors in FM85 [10]. The algorithm is based on counting the number of leading zeros in a bit pattern. Their observation was that if a hash function maps elements (having duplicate entries) to integers, where each integer is a binary string of "$M$" bits, uniformly distributed over the range $0, 1, ..., 2^M - 1$ , the bit pattern $0^k1$ will appear with probability $2^{-(k+1)}$ and if recorded for each element, can serve for cardinality estimation. The occurrence of each $0^k1$ pattern called the rank of that element is used as an index into a bitmap of length "$M$". It is initialized to all zeros and set to one for each rank index calculated from the hash of input element. After all the elements are scanned the cardinality is estimated from the least significant zero bit position as $2^R/0.77351$, the constant in the denominator is the statistical correction factor. The simpler version of FM85 [10] algorithm has a high variance which was improved with stochastic averaging often termed as Probabilistic Counting with Stochastic Averaging (PCSA). The idea is to reduce the number of hash functions computations required for averaging with multiple bitmaps. This is achieved by splitting a single hash function into remainder "$r$" and quotient "$q$" (calculated from the hashed value through division and remainder operation with "$m$" i.e number of simulated hash functions) used to select the corresponding bitmap and the computation of rank respectively. The rank for all is then calculated by taking the left-most

[1]Anees Ullah is with University of Engineering and Technology, Peshawar, Pakistan.
[2]Pedro Reviriego is with Universidad Carlos III of Madrid, 28911 Leganés, Madrid, Spain.
[3]Adeel Akram is with University of Engineering and Technology, Taxila, Pakistan.
[4]Malik Najmus Siraj is with Sir Syed CASE Institute of Technology, Islamabad, Pakistan.
This work is supported in part by Higher Education Commission (HEC) Pakistan and the Ministry of Planning, Development and Special Initiatives under National Centre for Cyber Security. The work of Pedro Reviriego was supported by the ACHILLES project PID2019-104207RB-I00 and the Go2Edge network RED2018-102585-T funded by the Spanish Ministry of Science and Innovation and by the Madrid Community research project TAPIR-CM grant no. P2018/TCS-4496.

1943-0663 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

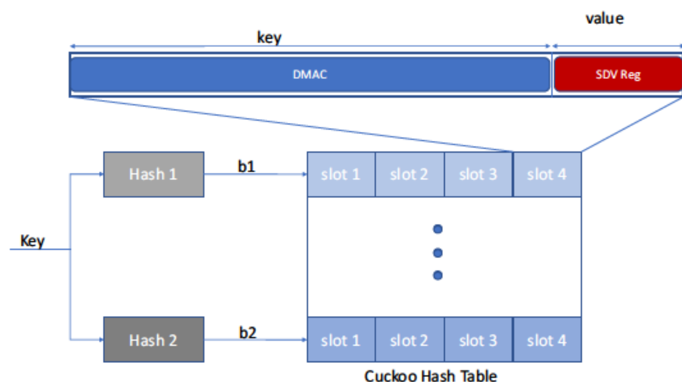


Fig. 1. Original Cuckoo Hash Table with proposed Table entry modification

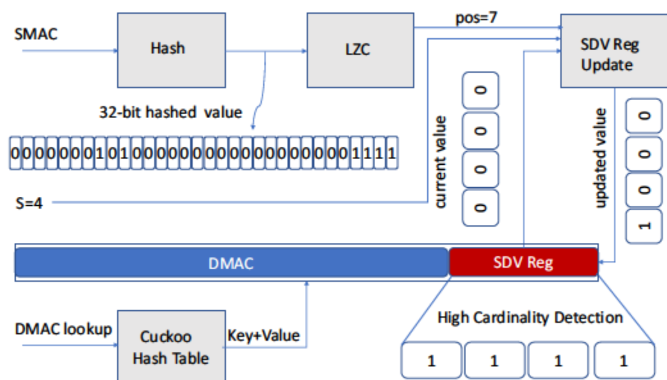


Fig. 2. Proposed switch-based High-cardinality detection: Update and High Cardinality Detection

position of zero in each bitmap and dividing it by the number of bitmaps i.e. $\bar{R} = \frac{1}{m}\sum_{k=0}^{m-1} R_k$. The final cardinality is then estimated by $\frac{m}{0.77351}2^{\bar{R}}$. Both the simple FM and the PCSA suffer from large storage requirements for big data applications. The loglog family of algorithms include loglog [11] and hyper-loglog [12]. Further, optimizations of hyperloglog are proposed in [13], [14], [15]. Unlike, the probabilistic counters the loglog family requires less memory storage i.e. in comparison to a bitmap of size "$N$" (N is number of unique elements) it requires a counter of size $log_2(log_2(N))$. This is achieved by storing only the value of rank in the counters. For the scanning process, each element is hashed by a single function. The first "$p$" bits are used to select the counter and the rest of "$M-p$" bits are used for calculating the rank "$r$". The selected counter is updated with the value of that is maximum between the computed rank "$r$" and the already stored counter value. After scanning, the evaluation function computes final rank "$R$" through arithmetic mean in loglog [11] or through harmonic mean as suggested in hyperloglog [12]. In this work, we have used a vector that is similar to FM85 for high-cardinality estimation as will be discussed in the proposed method section.

### B. Cuckoo Hashing

A hash table is a dictionary data structure stored as an array of length "$m$" where each entry is called a bucket and indexed by a key in the range $0, 1, ..., (m-1)$. Each element during lookup and update operations passes through a hash function which can result in mapping two elements to the same bucket. A cuckoo hash table uses two hash functions instead of a single hash function to minimize collisions. To insert a new elements into a cuckoo hash table, the element is hashed by two hash functions "$h1$" and "$h2$" resulting in two possible bucket positions "$b1$" and "$b2$". If one of these buckets is empty the element is inserted into that bucket. Otherwise, a random bucket is chosen to store the element while moving the already stored element to an alternative bucket position. The procedure is repeated until an empty bucket is found or the total number of tries are exhausted in which case the hash table is considered too full to insert a new element. The lookup procedure always takes a constant amount of time as it consists of computing the hashes "$h1$" and "$h2$" and checking "$b1$" and "$b2$" for the element. A second technique that is used to reduce the collisions in cuckoo hash tables is multi-way set associativity. This means that each bucket is further divided into slots. Without using set-associativity a cuckoo hash table would encounter un-resolvable collision when the table becomes 50% full. A typical value for slots is four per bucket as illustrated in Figure 1. Such a table achieves a space utilization of more than 95%. Therefore, our proposed methodology uses a a (2,4) cuckoo hash table i.e. two buckets and four slots per bucket. In particular, we have adapted the open-source implementation in [16] for our purposes.

## III. Proposed Scheme

As discussed in the introduction, the objective is to perform supernode detection in an Ethernet switch as part of the lookup for the destination MAC address that is done for each packet in an Ethernet switch. This lookup consists of an exact match of the packet's destination MAC address and those stored in the switch table that has been traditionally done using Content Addressable Memories (CAMs) [17]. However, in the last decade, hashing schemes and in particular cuckoo hashing have been widely adopted for Ethernet switching as they also perform the lookup in one memory access cycle but using standard memories that are much simpler [9], [18]. In the following, we consider that the switch uses cuckoo hashing to implement lookups. In particular, a (2,4) cuckoo hash table which consists of two hash functions generating two bucket addresses for each update/lookup and four slots per bucket and is based on [16]. The proposed methodology consists of two parts. One is the data structure used for cardinality estimation and its integration with the data structure of cuckoo hash table as presented in Figure 1. The second is the update and detection logic for cardinality and its seamless integration with the cuckoo switch look-up and update procedure as illustrated in Figure 2.

### A. Supernode Detection Vector (SDV)

Supernode Detection Vector (SDV) is a bitmap data structure that holds the cardinality estimation. SDV consists of a few bits typically four, as increasing the number of bits in the SDV improves the detection only marginally while introduces additional memory overhead. As we are using set-associativity in our cuckoo hash table, each slot in a bucket is extended to include the SDV as an additional value field as illustrated in Figure 1. The inclusion of SDV does not affect the update or lookup operation of the cuckoo hash table even in the case of collision and subsequent bucket eviction and alternative bucket search the operation is the same. Thus, SDV is non-intrusive

1943-0663 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

and does not require data structure changes to the table entries. This makes the SDV independent of the implementation details and can be easily integrated with existing cuckoo libraries.

**Algorithm 1:** Programmable Offset based SDV update

**Input:** Leading zeros $pos$, Programmable Offset $s$, Size of SDV $sdvsize$
**Output:** Return bool value Detected
**if** $pos \quad s+1>0$ **then**
  **if** $pos \quad s<sdvsize$ **then**
    $sdv[pos \quad s]=1;$
  **else**
    $sdv[sdvsize \quad 1]=1;$
  **end**
**end**
$Detected=sdv.all.set();$

### B. Lookup and Update

When a packet for destination MAC address "$DMAC$" arrives in the Ethernet switch, a lookup is done on the cuckoo hash tables to determine the output port on which to send the packet. If the "$DMAC$" is not found, it is added to the table through cuckoo insert procedures and the corresponding SDV register in Figure 2, is initialized to all zeros. In case the lookup is successful we need to update the SDV register from the corresponding source MAC address "$SMAC$". This is shown with an example in Figure 2. First the current value of SDV register is retrieved through a lookup operation with "$DMAC$" as a key. The example shows that the current SDV register holds "0000". This means the the SDV register size is 4 bits. The "$SMAC$" for this "$DMAC$" pair is passed through a hash function. This hashed value is then passed through a Leading Zero Count block which return the number of leading zero count. For example, for the hashed value "00000001010000000000000000001111" the LZC block returns a position "*pos*" equal to 7 which represents the count of leading zeros in this case. This block is followed by the SDV Reg update block, which receives the "*pos*", the programmable offset parameter "*S*" and current value of SDV register. The programmable offset "*S*" is a very important parameter of the update process in the proposed scheme. It allows the SDV to adjust itself for detection of arbitrary cardinalities as will be discussed in the evaluation section. Based on this information, i.e. "$pos=7$" and "$S=4$", as shown in Figure 2, the current value of "0000" is modified to "1000" based on $(pos-S)=3$. This is illustrated with more details in Algorithm 1. The algorithm takes as input the "*pos*", "*S*" and maximum size of SDV register. Then, based on the difference of "*pos*" and "*S*" appropriately shifts the SDV storage index to automatically fall within bounds of the SDV register size. This programmable offsetting is what makes the proposed method scalable for high cardinalities even by using only a few bits for SDV register. The modified SDV value is then stored in the table. This process is repeated each time a "$DMAC$" is looked up. The SDV register acts as a high cardinality estimator and when all the bits of the SDV register become non-zero, a high cardinality event is detected. It can be noted from the update and lookup operations that the cardinality estimation neither requires a separate data structure nor additional memory accesses. Therefore, it is both memory foot-print wise and processing speed wise efficient.

### C. High Cardinality Estimation Probability

Let us consider the probability of detecting a supernode when using $V$ bits for the vector that starts at bit $S$ as a function of the supernode cardinality $C$. For the node to be detected, hash values with their first one on positions $S,S+1,..,S+V-2$ and at least a value larger than $S+V-2$ have to appear. The probability of having values with their first one at position $S+i$ is given by:

$$P(S+i)=1-\left(\frac{2^{S+i}-1}{2^{S+i}}\right)^{C} \quad (1)$$

and thus the probability of having all of them would be approximately:

$$P_{sn}\approx\left(1-\left(\frac{2^{S+V \quad 2}-1}{2^{S+V \quad 2}}\right)^{C}\right)\cdot\prod_{i=0}^{V \quad 2}\left(1-\left(\frac{2^{S+i}-1}{2^{S+i}}\right)^{C}\right) \quad (2)$$

where the first term corresponds to the probability of having values with at least $S+V-1$ zeros. This equation allows us to estimate the probability of detection as a function of the cardinality when $S$ and $V$ are fixed and will be used to check the experimental results presented in the following section.

## IV. Evaluation

This section presents the experimental results of the proposed method for supernode detection and discusses its use in practical network scenarios. The proposed scheme was implemented in C++ and using the open-source libcuckoo library [16] as a starting point. As already explained, the library uses 4-way set associativity which means that each of the buckets holds four slots. In our case, we have added a 4-bit SDV to the entries of our cuckoo hash table. The programmable offset parameter "*S*" was set to 4, 8 and 12 bits for different test cases. The cardinalities "*C*" are generated for each case of "*S*" as an array of 10 elements populated as $2^{S \quad 4},2^{S \quad 3},...,2^{S+5}$. Each data point i.e. a pair of "*S*" and "*C*" is evaluated for 1000 iterations and the number of times the SDV becomes all ones is logged. This would correspond to the number of supernode detections. Note that during these experiments, synthetic MAC addresses were generated for source and destination MACs. They were formatted as different SMACs and a single DMAC vectors respectively to capture the high-cardinality scenario. The size of these test vectors depends on the cardinality of the current data point. The following subsections present the results for detection effectiveness and computational overhead.

### A. Detection effectiveness

The results in terms of number of detections over the 1000 runs are presented in Figure 3 that also contains "$Theoretical$" supernode detections calculated through equation 2 from section III-C. It can be seen from Figure 3 that the probability of detection increases in a similar fashion for S=4, S=8 and S=12. This illustrates that the proposed method is able to estimate high-cardinalities with the programmable offset handling of the SDV register. As for the practical use of the proposed scheme, assuming that supernode cardinality is much larger than that of normal nodes, for example it is 16x larger, the proposed method can reliably detect supernodes while triggering a low number of false positives. As an example if supernodes have cardinalities larger than 8000 and normal nodes below 500, then setting "*S*"

1943-0663 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

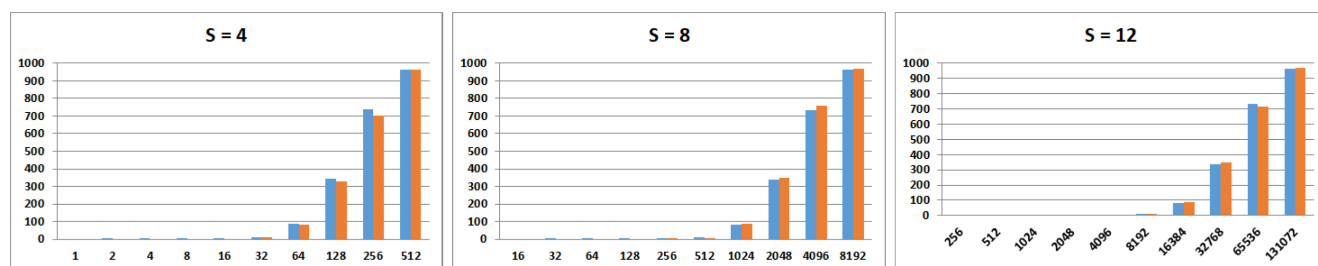


Fig. 3. Number of detections theoretical (blue) and simulated (orange) over 1000 runs for $S = 4$ (left), $S = 8$ (middle) and $S = 12$ (right) and different cardinalities.

equal to eight would detect supernodes in 96% of the cases while triggering false positives in less than 1% of the cases. This shows the effectiveness of the proposed scheme to detect supernodes. Similar results are obtained for smaller (512 and 32) and larger (131072 and 8192) cardinality ranges when "$S$" is equal to four and twelve respectively. This illustrates how the proposed scheme can be adjusted to detect supernodes of different cardinalities by adjusting $S$. In more detail, detection is reliable for cardinalities larger than $2^{S+5}$ when using an SDV of four bits (in general, $2^{S+b+1}$ for an SDV of $b$ bits).

### B. Overhead

The proposed scheme introduces overheads in terms for memory and computation effort when compared to a plain cuckoo hash based switching that does not implement supernode detection. In terms of memory footprint, the overhead is small, four bits per entry in the Ethernet FIB table and there is no overhead in terms of memory bandwidth as the SDV bits are read when reading the FIB entry to forward the frame. The proposed scheme does require additional computing effort to update and check the SDV bits. To assess the overhead introduced by those operations, the execution time needed to perform a lookup in the FIB has been measured for both, the plain cuckoo hash FIB and with our proposed extension for supernode detection when running on an Intel Core i7 processor. In more detail, the average over one million operations was logged and found to be 1.56 and 1.78 microseconds respectively. This corresponds to an overhead of approximately 14% which would be acceptable in many practical settings. In fact, to put this result on perspective the plain cuckoo switching [16] combined with independent virtual hyperloglog [8] was also tested and used as reference. This combination would provide the same functionality with the cuckoo hash doing the switching and the virtual hyperloglog the supernode detection. In this case the average time required to process a frame was 2.05 microseconds which implies an overhead of 31%, more than twice that of the proposed scheme which clearly shows the benefits of the proposed scheme.

## V. Conclusions and Future Work

In this letter, for the first time a scheme that jointly performs switching and supernode detection has been presented and evaluated. This significantly reduces the number of memory accesses needed per packet. The proposed scheme has been shown to effectively detect supernodes while providing a significant speed up in processing time. The presented approach is particularly suitable for soft switches that are becoming increasingly important with the wide spread acceptance of the Software Defined Networking (SDN) paradigm. The next step will be the implementation of the scheme in a packet processing framework like Vector Packet Processing (VPP) and its evaluation with packet traces. The integration of other cardinality estimators with switching is also an interesting area for future work

## References

[1] W. Chen, Y. Liu and Y. Guan,"Cardinality change-based early detection of large-scale cyber-attacks," in Proceedings of IEEE INFOCOM, 2013.
[2] Y. Chabchoub, R. Chiky, and B. Dogan, "How can sliding Hyperloglog and EWMA detect port scan attacks in IP traffic?" in the EURASIP Journal on Information Security, pp. 1-11, 2014.
[3] M. Cai, K. Hwang, J. Pan and C. Papadopoulos, "WormShield: Fast Worm Signature Generation with Distributed Fingerprint Aggregation," in IEEE Transactions on Dependable and Secure Computing, vol. 4, no. 2, pp. 88-104, April-June 2007.
[4] H. Jiang, S. Chen, H. Hu and M. Zhang, "Superpoint-based detection against distributed denial of service (DDoS) flooding attacks," in Proceedings of the 21st IEEE International Workshop on Local and Metropolitan Area Networks, Beijing, 2015.
[5] L. Zheng, D. Liu, W. Liu, Z. Liu, Z. Li and T. Wu,"A Data Streaming Algorithm for Detection of Superpoints with Small Memory Consumption," in IEEE Communications Letters, vol. 21, no. 5, pp. 1067-1070, May 2017.
[6] Y. Liu, W. Chen, and Y. Guan, "Identifying high-cardinality hosts from network-wide traffic measurements," IEEE Transactions on Dependable and Secure Computing, vol. 13, pp. 547–558, Sep. 2016.
[7] W. Liu, W. Qu, J. Gong and K. Li, "Detection of Superpoints Using a Vector Bloom Filter," in IEEE Transactions on Information Forensics and Security, vol. 11, no. 3, pp. 514-527, March 2016.
[8] Q. Xiao, S. Chen, M. Chen, and Y. Ling, "Hyper-Compact Virtual Estimators for Big Network Data Based on Register Sharing," in Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2015.
[9] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance Ethernet forwarding with CuckooSwitch," In Proceedings of the ninth ACM conference on Emerging networking experiments and technologies (CoNEXT), pp. 97–108, 2013.
[10] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications,"in Journal of Computer and System Sciences, 31(2):182–209, 1985
[11] M. Durand and P. Flajolet, "Loglog Counting of Large Cardinalities (Extended Abstract)," In Proceedings of ESA 2003.
[12] P. Flajolet, E. Fusy, O. Gandouet, and et al.,"Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in Proceedings of the International Conference on Analysis of Algorithms (AOFA), 2007.
[13] O. Ertl, "New cardinality estimation algorithms for HyperLogLog sketches," arXiv preprint arXiv:1702.01284, 2017.
[14] D. Ting, "Approximate Distinct Counts for Billions of Datasets," in Proceedings of SIGMOD, 2019.
[15] Q. Xiao, Y. Zhou, and S. Chen, "Better with fewer bits: Improving the performance of cardinality estimation of large data streams," in Proceedings of IEEE INFOCOM, 2017.
[16] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman, "Algorithmic improvements for fast concurrent cuckoo hashing," in Proceedings of ACM Eurosys 2014.
[17] R. Seifert and J. Edwards, "The All-New Switch Book: The Complete Guide to LAN Switching Technology," Wiley, 2008.
[18] P. Reviriego, S. Pontarelli and G. Levy, "Improving energy efficiency of Ethernet switching with modular Cuckoo hashing," in Proceeding of the IEEE Online Conference on Green Communications (OnlineGreenComm), 2015.

1943-0663 (c) 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.