Webology (ISSN: 1735-188X)
Volume 18, Number 5, 2021

Dependency Aware Caching (Dac) For Software Defined Networks

V. Rajkumar[1] , Dr. V. Maniraj[2]

[1]Research Scholar, Dept. of Computer Science, AVVM Sri Pushpam College, Poondi, Thanjavur, (Affiliated to Bharathidasan University),
ORCID ID: 0000-0002-8113-2616

[2]Research Advisor, PG and Research Dept. of Computer Science,
AVVM Sri Pushpam College, Poondi, Thanjavur,
(Affiliated to Bharathidasan University).

Abstract

By utilising software-defined networking (SDN), control applications are granted the ability to establish fine-grained forwarding policies within the switches that sit beneath them. As a direct result of this, the number of wildcard rule patterns that hardware switches are able to implement is restricted due to the price and energy requirements of Ternary Content Addressable Memory (also known as terabytes) (TCAM). To add insult to injury, the hardware switches in question are unable to manage a quick series of rule table updates, thereby compounding the problem. In this investigation, we demonstrate how to combine the most advantageous aspects of hardware and software processing in order to provide programmes the impression of high-speed forwarding, big rule tables, and quick updates. This strategy "caches" the rules that are used the most frequently in a very tiny TCAM, and it relies on software to regulate the relatively little amount of "cache miss" traffic that is produced. Cache-replacement strategies that are now available cannot be implemented in a blind method, however, because of the interdependencies that exist between different rules that share similar patterns. It is advised to "splice" big dependency chains in order to cache smaller sets of rules while still keeping the policy's semantics. This may be done by splitting up the chains into many smaller chains. Experiments that were carried out with our DAC prototype suggested that rule splicing may be an effective method for utilising limited TCAM space while simultaneously reacting quickly to changes in both policy and traffic.

Keywords: SDN, Caching, dependency aware caching, TCAM

Introduction

The implementation of easy packet processing rules in the underlying switches of a Software-Defined Network (SDN) is what is required to achieve the goal of governing the flow of traffic inside the SDN. These rules make it possible to carry out even relatively easy operations, such as forwarding, flooding, altering the headers, or sending packets to the controller. Examples of these actions include: They are so adaptable that they may be used as firewalls, load balancing servers, and network address translators. Any role between the two can be played by a switch that supports software-defined

networking (SDN). Policies that enable fine-grained forwarding are not desired due to the large number of rules that must be implemented in the switches in order for them to function properly.

Each of today's hardware switches comes equipped with a function known as Ternary Content Addressable Memory (TCAM), which is the location where these rules are stored (TCAM). A TCAM is able to perform line rate operations while simultaneously comparing the contents of an incoming packet to the patterns that are contained in all of the rules. On the other hand, the number of rules that may be supported by commodity switches is on the lower end of the thousands or even the tens of thousands range. This is because commodity switches are designed to support just the rules that are absolutely necessary. In the future, TCAMs will still require a fundamental trade-off between the size of the rule table and other considerations, such as cost and power. This will be the case. This compromise will continue to be obligatory. TCAMs have a price tag that is one hundred times greater than ordinary RAM and utilise one hundred times more power than traditional RAM does. Because of the sluggish pace at which rule tables in TCAM can be updated by today's hardware switches, it is possible that a large network with dynamic policies might be hampered by this. The fact that these switches are only capable of managing between 40 and 50 rule-table changes per second presents a potential challenge for such a large network.

It is possible that making changes to the software in order to improve it will appear to be the more appealing option. It is theoretically possible, with commodity servers running software switches, to process packets at a rate of around 40 Gbps while concurrently keeping massive rule tables in memory and, to a lesser degree, in the L1 and L2 caches. This is a theoretically possible scenario. There is a possibility that software switches will be able to update the rule table in real time, and they are able to do it more quickly than hardware switches can by a factor of more than 10. Because it is difficult to permit wildcard rules that match on numerous header fields at the same time, software switches are forced to resort to delayed processing in user space in order to deal with the initial packet of each flow. This is necessary in order to handle the packet. A linear scan is one example of this kind of processing that may be done. As a direct consequence of this, they are unable to compete with the "horsepower" of hardware switches, which are able to process hundreds of gigabits per second of packets (and high port density).

The fact that traffic follows a Zipf distribution, which suggests that the vast majority of it adheres to a confined subset of rules, is fortunate. This allows for greater efficiency. This paves the way for management that is more effective. As a consequence of this, we were able to rely on software switches to manage the remaining amount of the traffic while utilising a TCAM that was less robust to transmit the majority of it. A hardware switch with 800 Gbps throughput and a software packet processor with 40 Gbps throughput alone may be able to easily manage the traffic on a TCAM network that has a "miss rate" of 5 percent. In addition, the majority of rule-table adjustments might be sent to components through the slow-path, whilst very popular rules are only promoted to hardware on extremely few occasions. This is due to the fact that rules that are really popular are only pushed to hardware on an extremely infrequent basis. When hardware and software processing are connected, controller applications will have the appearance of high-speed packet forwarding, massive rule tables, and quick rule modifications.

In addition to TCAM, the design of our digital to analogue converter is built of a sharded collection of software switches (DAC). You have the option of constructing the software switches on separate servers, or as an inherent part of the software agent that controls the hardware switch. SDN

controllers that have not been modified will send their OpenFlow instructions to a CacheMaster module, which is the component that is in charge of DAC implementation. CacheMaster has the potential to adjust rules and query counts while still preserving the integrity of the OpenFlow interface's semantics in its entirety. This ability is one of its many useful features. CacheMaster sends rules to commodity software and hardware switches by utilising the OpenFlow protocol. These switches are regarded to be standard in the industry. In the component for the control plane known as CacheMaster, control sessions are represented by dashed lines, but data forwarding is represented by solid lines. Both types of lines are illustrated in the diagram.

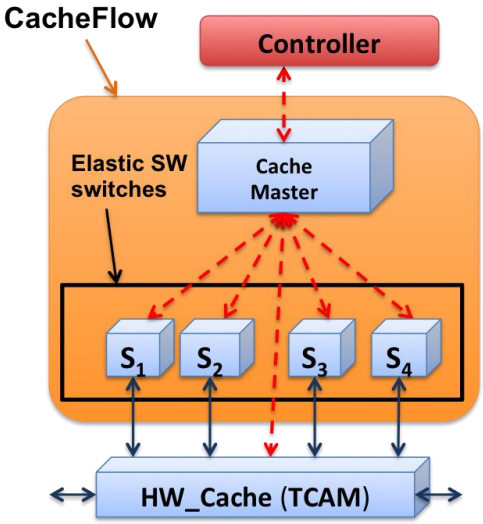


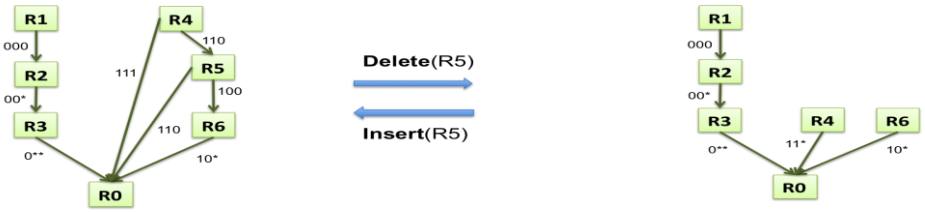
Figure 1: DAC architecture

Rule Dependencies

The OpenFlow policy of a switch is made up of several rules that govern the way in which packets are processed. Each rule possesses its very own one-of-a-kind pattern, priority, collection of actions, and counters. The switch will first determine the matching rules that have the highest priority once a packet is received, after which it will carry out the actions that correspond to those rules, and lastly it will increment the counters. In order for DAC to accomplish this, the collection of rules is segmented into two separate groups, one of which is stored in the TCAM, and the other of which is stored in a software switch. DAC then stores the TCAM with the first group of rules and the software switch with the second group of rules.

Rule	(dst_ip, dst_port)	Ternary Match	Action	Priority	Weight
R1	(10.10.10.10/32, 10)	000	Fwd 1	6	10
R2	(10.10.10.10/32, *)	00*	Fwd 2	5	60
R3	(10.10.0.0/16, *)	0**	Fwd 3	4	30
R4	(11.11.11.11/32, *)	11*	Fwd 4	3	5
R5	(11.11.0.0/16, 10)	1*0	Fwd 5	2	10
R6	(11.11.10.10/32, *)	10*	Fwd 6	1	120

(a) Example rule table



(b) Incremental DAG insert (c) Incremental DAG delete

Figure 2: The process of building the rule dependency graph

To put this another way, if there is no matching rule in the TCAM, the highest-priority rule in the software switch is used as the foundation for the semantics of the DAC. This occurs only if there is no matching rule in the TCAM. Only in the event that the TCAM has no matching rules will this take place. This suggests that putting all of the requirements into practise might not always be possible due to the lack of appropriate procedures. It is possible that packets that should have been handled by rules in the software switch may instead have been processed by a cached rule in the TCAM, which would have resulted in an incorrect answer. This would have been the case if the TCAM had been used to process the packets. It's possible that this has been the situation ever since the TCAM storage restrictions were implemented.

It is not impossible for regulations to be dependent on one another despite the fact that they prioritise distinct things. We are unable to choose the four rules that have the largest total traffic volume since packets that should match R1 (with pattern 000) match R2 (with pattern 00*), and some packets (say, with header 110) that should match R4 (with pattern 1*0) match R5 (also with pattern 1*0). This is due to the fact that certain packets that ought to match R4 (with pattern 1*0) instead match R5 (with pattern 1*0). (this would be R2, R3, R5, and R6) As can be seen in the illustration that comes after this one, each of these guidelines is, in some way, dependent on the others. To put it another way, there is a direct link between the rules R1 and R2, which can be found below. Both of these rules can be found below. R1 ought to be cached in the TCAM in the same way that R5 and R4 are currently being maintained there in order to keep the semantics of the policy intact. This is necessary in order to preserve its integrity.

Rule patterns that overlap one another give strong indication that they are related to one another (e.g., R2 is dependent on R1). When a rule is added to the TCAM, all of the other rules that depend on it should also be moved there. This is called the "transfer of dependencies" rule. On the other hand, even because you are seeking for policy ramifications doesn't mean you will discover all of them just because you are looking for them. If you are looking for patterns that intersect, for example. If this definition is followed, for example, rules R6 and R7 are the only ones that are dependent on rule R5, and they are the only ones that are dependent on rule R4. Because the switch would only have access to R5 and R6 if the TCAM only preserved R5 and R6, the switch would not handle packets correctly (with header 110) that should match R4 because it would only have access to R5 and R6. R6 is dependent on R4 in a roundabout fashion despite the fact that the matches for R4 and R6 do not coincide since R4's match overlaps with R5's. This is because R5's match overlaps with R4's match. We need to establish a very clear description of what it means to rely on something in order to deal with these kinds of scenarios in a way that is acceptable to everyone involved.

Caching Algorithms

When there is a limited amount of space available, the DAC method for inserting rules into a TCAM

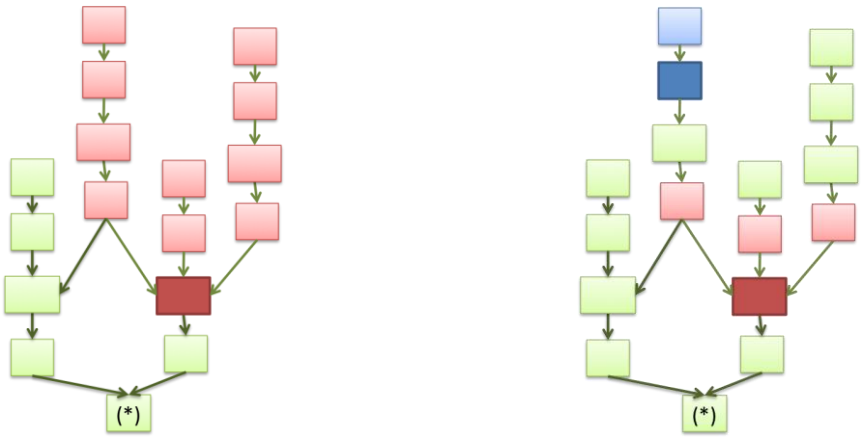
will be discussed in this section of the article. The DAC will choose a subset of the controller's rules and store them in the TCAM. At the same time, it will send the remaining rules to the software switches so that they may be utilised at a later time.

Optimization: NP Hardness

When it comes to finding a solution to the problem of rule caching, one of the things that may be fed into the problem is a dependency network that contains the rules R1 and R2. Each rule has a matching match and action, in addition to a weight wi that specifies the proportion of total traffic that fits the requirements for the rule. This information may be found in the rule's description. As was discussed in the part that came before this one, there are edges of dependency that exist between each pair of rules. The TCAM5 will, if it is completed, provide a list of C rules that are arranged in the order of priority. When processing "hit" packets in accordance with the initial rule table, the goal is to maximise the total of weights for traffic that reaches the TCAM. This is accomplished by increasing the value of each individual weight. This is done in order to allow the TCAM to process a greater volume of communications.

$$\begin{aligned} &\textit{Maximize} && \sum_{i=1}^{n} w_i c_i \\ &\text{subject to} && \sum_{i=1}^{n} c_i \le C;\ c_i \in \{0, 1\} \\ & && c_i - c_j \ge 0 \text{ if } R_i.is\ \ descendant(R_j) \end{aligned}$$

The fact that the optimization issue that has been described has both n and k dimensions contributes to the fact that it is an NP-hard problem. The NP-hard problem may be decomposed into its more manageable subproblem, which is referred to as the densest k-subgraph problem. Despite this, the NP-hard problem is famously difficult to solve. At this point, we are going to draw a comparison between the two distinct methods of deciding how to proceed with the issue. Let's imagine you're in a position where you need to make a choice on caching, but you're not sure what to do. This search should include any C rules from this table that respect directed dependencies, have a cumulative weight of at least W, and take into account directed dependencies. Specifically, this search should look for rules that respect directed dependencies.



(c) Dep. Set Cost (b) Cover Set Cost

Figure 3: Dependent-set vs. cover-set Cost

In order to better understand the problem of maximum coverage within a certain budget, we have modelled it after a PTAS that is extremely greedy. The algorithm selects a set of rules with the assistance of our greedy heuristic in order to maximise the ratio of combined rule weight to combined rule cost (W), and it continues to do so up to the point where the total cost is closer to k. The amount of time required to run this procedure is undefined.

First, the greedy algorithm selects R6 (and its dependent set R4, R5), and then it selects R1. This results in a total cost of four as a consequence of the method's activities, which indicates that the process is greedy. As a direct result of this, the collection of rules R1, R4, R5, and R6 included inside the TCAM is taken into consideration to be the most effective combination. In the business world, this method is more commonly referred to by its technical name, the dependent-set algorithm.

Cover-Set: Splicing Dependency Chains

Rules that have a high weight and are dependent on a large number of rules that carry a low weight might result in significant extra expenditures. The situation that was described before serves as an example of how a firewall that relies on multiple high-priority "deny" rules that match relatively little traffic might have a single low-priority "allow" rule. For the sake of saving only one "accept" rule in the cache, it would be required to store a large number of "deny" rules beforehand. Instead of simply squeezing the old rules into the available space, our method is superior because we have the ability to modify the rules in a variety of ways while preserving the coherence of their semantic meaning. This allows us to make the necessary adjustments without compromising the rules' original intent. By providing a limited number of extra rules that cover an assortment of light rules, we "splice" the dependant chain. Because of this, we are able to transfer the packets that were affected to the software switch.

Updating the TCAM Incrementally

Our caching algorithms choose a new set of cached rules over time in response to the changing traffic distribution that takes place across the rules. This causes the list of cached rules to change over time. As a result of this, it will be essential to upgrade the TCAM on a regular basis utilising the most recent policy cache. Due to the substantial amount of time that is necessary to input TCAM rules, it is not feasible to delete everything that is now stored in the cache and start the procedure from the very beginning. During the process of modifying the cached rules, it is vital to reduce the amount of TCAM churn that takes place as much as possible.

In this particular circumstance, a plain update of the differences will not be effective. It is conceivable for TCAM policy snapshots to have errors during the transition if the cached rules are simply replaced with new ones (while preserving the common set of rules). If this is the case, then why does the process of updating the TCAM rules take so long? Because packets face the danger of being erroneously handled by an incomplete snapshot if the transition is taking place, why?

Commodity Switch as the Cache

Our system employs a Pronto-Pica8 3290 switch that runs on PicOS 2.1.3 and is compatible with OpenFlow. This switch serves as a caching mechanism for the system. During the course of our testing, we discovered that there were a few issues with the switch that needed to be fixed.

Incorrect handling of large rule tables: Within the ASIC ASIC of the switch, there is enough room for a maximum of two thousand OpenFlow rules. There is a limit of 2000 rules that may be saved in

the TCAM; beyond that, any more rules will need to be saved in the software agent instead. As a consequence of this, inappropriate forwarding behaviour may occur if a switch does not adhere to the cross-rule dependencies when it is updating the TCAM. This is because the TCAM contains information that is dependent on other rules. We are able to get around this problem by operating under the presumption that the rule capacity is capped at two thousand rules. This is possible due to the fact that we are unable to make any modifications to the (private) software agent. The precise strategies that have been discussed in this research might be utilised by a software agent in order to address and resolve this issue.

Slow processing of control commands: In order for the switch to query traffic statistics and update the TCAM, the switch needs an excessive amount of time. The amount of time needed to update the TCAM does not grow in a linear way if new rules are introduced or existing ones are eliminated. The addition of the first 500 rules often takes around six seconds, whereas the addition of the subsequent 1500 rules typically takes about two minutes. This is a decent rule of thumb. The utilisation of the switch's central processing unit (CPU) reached one hundred percent as a direct result of this, and as a direct result of this, the switch disconnected its connection to the controller. While we wait for the rules that have been established to become generally stable before we begin to ask the counters on a more regular basis, we make use of counters that are included inside the software switch in order to avoid this from taking place.

Prototype and Evaluation

Along with the assistance of the Ryu controller module, a prototype of a Python DAC was constructed. Because it makes use of OpenFlow, this prototype is able to establish a connection with the switches. DAC offers an interface on the northern side, which enables FlowMods to be transferred to it so that it may then distribute them to the switches. This is possible because DAC can receive FlowMods at this interface. With the exception of rule timeouts, the OpenFlow 1.0 semantics have been implemented by us in the past in a form that is transparent to control applications as well as switches. This implementation was done in the past.

For the purpose of the hardware cache, we make use of an Open vSwitch 2.1.2 multithread software switch. This move is managed by an AMD eight-core server that has six gigabytes of RAM at its disposal. As a consequence of passing TCP packets from one location to another, a Pica8 switch generated a substantial volume of data traffic as a result of its activities.

Cache-hit Rate

Our prototype is judged according on how well it complies with the following three rules, as well as the accompanying packet traces:

A real-world packet trace as well as a synthetic policy are both available for observation by the general public. In addition, an educational campus network routing policy and an associated synthetic packet trace are also available for observation. In addition, an OpenFlow policy that is offered by an Internet eXchange Point is accessible to the general public, and so is the packet trace that is associated with it (IXP). In order to determine whether or not these strategies are effective in terms of the number of cache hits they receive, three different caching techniques were used (dependent-set, cover-set, and mixed-set). Taking the total number of packets that have been transmitted by tcpreplay and subtracting the number of times that the cache has been invalidated allows ifconfig to determine

whether or not a cache hit has occurred. PyPy is a tool that can help speed up Python programming, which means that all of these results are attainable if you use the tool. [Further citation is required] The term "overhead" can also be used to refer to latency. The length of time that elapses after a cache entry has been read before it is retrieved from memory. When the previous CAIDA packet trace was being analysed, the latency was assessed by connecting two more hosts to the switch. This took place during the processing period. In order to process the ping packets in the TCAM, new rules had to be included into the policy. These new rules were triggered by the fact that there was a substantial quantity of traffic. When there was a hit in the cache in both directions, the average round-trip delay was 0.71 milliseconds, but when there was only a miss in the cache in one direction, the average delay was 0.81 milliseconds. When there was only a miss in the cache in one direction, the delay was 0.81 milliseconds. To put this into perspective, when a hardware switch is involved, the 1-way delay of the packets increases by 25 microseconds; hence, the cost of a 1-way cache miss was 100 microseconds.

In the event that an application does not desire to bear the higher costs that are associated with a software transition, it may make a request to the CacheMaster to have its rules placed in the fast path. In this case, the application will not have to worry about the additional costs. This is something that the CacheMaster is able to do by putting a "infinite" amount of significance upon the aforementioned criteria.

Incremental Algorithms

In order to determine whether or not the incremental update methods were effective, we carried out two distinct tests. In the first experiment, we sought to analyse the algorithms that, in response to the addition or subtraction of rules, gradually update the dependency network. The purpose of the second experiment was to assess the algorithms that gradually update the TCAM in response to shifts in the traffic distribution over the course of time.

On top of the 180K rules that are taken into consideration by the Ford routing policy, progressively add or remove constraints. The amount of time it takes to do an incremental deletion is around 3.7 milliseconds, whereas the amount of time it takes to update the dependency tree is approximately 15 milliseconds. At least for a few thousand flowmods, the graphs demonstrate that the total amount of time required is directly proportional to the number of flowmods that are either added or deleted throughout the process. This is the case even if the complete number of flowmods is not known. An incremental deletion is four times quicker than an insert because of the local set of dependent adjustments that take place when a rule is eliminated. An insert, on the other hand, takes twice as long. On the other hand, an insert needs to go through a substantially higher number of branches, beginning at the root, in order to locate the proper location to insert the rule. When compared to an incremental insert, an incremental deletion is four times quicker. In addition, we timed the process of statically generating the graph by incorporating 180K rules, and we discovered that it took around 16 minutes to finish. Therefore, there are about 60000 times quicker incremental versions available for altering dependency graphs as opposed to a static version.

In order to evaluate the possible benefits of applying incremental TCAM update techniques, comparisons were done between the cache-hit rates attained by mixed-set algorithms which utilised the two TCAM update approaches. When the TCAM size becomes close to 2000 rules, the cache-hit rate of the incremental algorithm sees a significant increase in its overall effectiveness. The incremental update has a cache-hit rate of 93% for 2000 rules in the TCAM, whereas the nuclear

update only has a cache-hit rate of 53% for the same amount of rules. This indicates that the incremental update is the superior method. Because it takes so much longer to install more than 1000 rules, the nuclear update process concludes that doing so will result in declining returns. This is due to the fact that adding more than 1000 rules takes so much more time.

Anytime the DAC decides to update the cache, the naïve method of conducting a nuclear update on the TCAM has an impact on the cache-hit rate. This happens whenever the DAC takes the choice to update the cache. Repeating the CAIDA packet trace at a pace of 330k packets per second led to the observation that an increasing number of cache misses were occurring over the course of the test. After a very short amount of time, the incremental update technique was able to effectively acquire a cache-hit rate of 95%. On the other hand, the nuclear update version, which gets rid of all of the old rules and replaces them with the new cache, has a significant number of cache misses while it is in the process of updating the TCAM. This is because the nuclear update version eliminates all of the previous rules and replaces them with the new cache. It takes around two minutes after removing the rules to go back to the high cache-hit rate; but, after installing the new cache, that rate leaps up to 90 percent. The nuclear plant is not a good target for TCAM upgrades since the rate at which it misses caches is highly variable.

Conclusion

DAC is a hybrid switch architecture that incorporates both hardware and software and takes advantage of rule caching. It is feasible to store rules in huge rule tables at a cost that is manageable because to DAC's capabilities. We do not keep or compress individual rules in cache like most other common caching systems do since doing so would prevent us from accurately accounting for rule dependencies (to preserve the per-rule traffic counts). It is preferred to "splice" large chains in order to store smaller sets of rules while still preserving the semantic meaning of the rules that were originally written. This may be accomplished by keeping the original rules' structure. The following four crucial requirements can all be met by our solution: flexibility, elasticity, transparency, and fine-grained rule caching (to enable incremental changes to the rule caching as the policy changes).

References

[1] Interference-Avoid Channel Assignment for Multi-Radio Multi-Channel Wireless Mesh Networks With Hybrid Traffic: Lu Yang;Yujie Li;Shiyan Wang;Haoyue Xiao, IEEE Access_2019.

[2] AI-Assisted Framework for Green-Routing and Load Balancing in Hybrid Software- Defined Networking: Proposal, Challenges and Future Perspective: Richard Etengu;Saw Chin Tan;Lee Ching Kwang;Fouad Mohammed Abbou;Teong Chee Chuah, IEEE Access_2020.

[3] QoS-Aware Hybrid Scheduling for Geographical Zone-Based Resource Allocation in Cellular Vehicle-to-Vehicle Communications: Chun-Yi Wei;Arvin C.-S. Huang;Cheng- Yu Chen;Jen-Yeu Chen, IEEE Communications Letters_2018.

[4] Outage Constrained Power Efficient Design for Downlink NOMA Systems With Partial HARQ: Yanqing Xu;Donghong Cai;Fang Fang;Zhiguo Ding;Chao Shen;Gang Zhu, IEEE Transactions on Communications_2020.

[5] Energy-Efficient Resource Allocation for Time-Varying OFDMA Relay Systems With Hybrid Energy Supplies: Bo Yang;Yanyan Shen;Qiaoni Han;Cailian Chen;Xinping Guan;Weidong Zhang, IEEE Systems Journal_2018.

[6] Channel Estimation and Transmission Strategy for Hybrid mmWave NOMA Systems: Dian Fan;Feifei Gao;Gongpu Wang;Zhangdui Zhong;Arumugam Nallanathan, IEEE Journal of Selected Topics in Signal Processing_2019.
[7] Dynamic Resource Management for LTE-Based Hybrid Access Femtocell Systems: Ying Loong Lee;Jonathan Loo;Teong Chee Chuah, IEEE Systems Journal_2018.
[8] Prediction-Based End-to-End Dynamic Network Slicing in Hybrid Elastic Fiber-Wireless Networks: Shan Yin;Zhan Zhang;Chen Yang;Yaqin Chu;Shanguo Huang, Journal of Lightwave Technology_2021.
[9] Adaptive Traffic Management and Energy Cooperation in Renewable-Energy-Powered Cellular Networks: Hyun-Suk Lee;Jang-Won Lee, IEEE Systems Journal_2020.
[10] Achieving Near-Optimal Traffic Engineering Using a Distributed Algorithm in Hybrid SDN: Cheng Ren;Shiwei Bai;Yu Wang;Yaxin Li, IEEE Access_2020.
[11] Resource Allocation and HARQ Optimization for URLLC Traffic in 5G Wireless Networks: Arjun Anand;Gustavo de Veciana, IEEE Journal on Selected Areas in Communications_2018.
[12] Optimization of MAC Frame Slots and Power in Hybrid VLC/RF Networks: Madiha Amjad;Hassaan Khaliq Qureshi;Syed Ali Hassan;Arsalan Ahmad;Sobia Jangsher, IEEE Access_2020.