Wildcard Rules Caching Algorithm in Software-Define Networkings

Zhenge Jia
The Australian National University

Abstract

In software-define networkings, ternary content addressable memory (TCAM) is widely used in OpenFlow switches. TCAM has high lookup speed and is capable to do wildcard matching. However, it is power-hungry and expensive, and it provides only limited size of entries to cache the rules. Rules caching is one of the solutions to the TCAM capacity problem. In rules caching system, the main challenge is the rules dependency problem existing in wildcard rules, which causes the mismatch. In this report, we study the rules caching algorithm to the problem and propose a new wildcard rules caching algorithm to solve the rules dependency problem.

1. Introduction

Software-Defined Networking (SDN) is a new technology to computer networks that enables administrators to control the network service via software-level functionality. In traditional network architectures, switches and routers are responsible to make decisions for packets routing paths and packets delivery. As a result, the switches and routers have to store several specific rules. Besides, network administrators are responsible to set up and manage each switch and router in the network, which is hard to manage and error prone. SDN is capable of separating the control plane from the data plane and a controller could control the routing path and rule replacement. This feature enables administrators to manage the network packets delivery much easier. The function of switches and routers in SDN is to deliver packets without making decisions.

In an SDN, the most important component is its controller. There are many available controller platforms designed by different organizations. Among these SDN controllers, the most commonly used one is OpenFlow [1], which enables administrators to control the network by flow table. There are three fields in each flow table entries, namely 'match', 'action' and 'counters'. The field ‘match’ is used for setting the packets to match the corresponding rules. It is usually made up of source and
destination IP addresses. The wildcard could be applied to any bit within those addresses to match more patterns. If the packets could not match any rule, the message will be sent to controller and the controller will let the switch install new rule. As for field ‘action’, it determines where the packet is supposed to go or what action the matched packets will take. The field ‘counter’ is used to store the statistical data for packets, and the controller has access to utilize these data.

In modern hardware SDN switches, the rules in the flow table form are stored in Ternary content-addressable memory (TCAM) [2, 3], which is a high-speed memory for searching certain contents. The word “Ternary” means that TCAM can store not only the binary bits but also the input 'X'. The 'X' refers to the wildcard state, which can be representing either 0 or 1. This feature enables TCAM to perform broader searcher based on pattern matching [4]. However, the size of TCAM is limited in switches because TCAM is expensive and power-hungry. A typical TCAM would have 400 times the cost and 100 times the power consumption of traditional switches [5]. The replacement process of data stored in TCAM is slow and usually could have 40 to 50 rule replacements in one second [5]. As a result, methods to take full advantage of TCSM’s limited size and reduce replacement occurrences to improve efficiency are crucial.

According to the previous works, there are three popular approaches to minimize the effects of the TCAM capacity problem, which are packet classification compression [6], rule distribution along traffic route [7] and rules caching [5, 8, 9]. In packet classification compression technology, it merges combinable rules into a new wildcard rule and generates semantically equivalent smaller packet classification. In rule distribution along traffic route, it distributes the rules depending on routing policies and reduces the size of large flow tables. In rule caching system, the key idea is to cache the ‘important’ rules into TCAM. The definition of ‘important’ is varied on different algorithm in rule caching system. In this report, we focus on the rule caching system.

In the rule caching system, different priorities are assigned to the rules to avoid conflicts because overlapping may exist in the field ‘match’ for the wildcard rules [5]. For the OpenFlow switch, the rule with highest priority will be applied first if there is a set of rules matching the packets in the cache. However, the rules dependency problem may cause issues within the wildcard rules caching system [10]. For example, if there are two overlapped wildcard rules, \( R1 \) and \( R2 \), whose field ‘match’ values are ‘000’ and ‘00*’ respectively. The input ‘*’ represent the wildcard bit and it can be either ‘0’ or ‘1’. That means \( R2 \) can match with ‘000’ and ‘001’, which overlaps with \( R1 \) in field ‘match’. Suppose the priority of \( R1 \) is higher than \( R2 \). In TCAM, if we only cache the lower priority rule, the packets matching \( R1 \) would instead match \( R2 \) and cause the incorrect matching problem. To solve
this problem, extra space of cache need to be applied [10].

In this report, we study TCAM in SDN switches, the rules caching system and the rules dependency problem. We focus on improving the efficiency of using TCAM: giving a set of rules, select part of these rules to store in TCAM so that the hit ratio of the packets matching the rules is maximized.

In this report, we make the following contribution. We first propose a wildcard rules caching algorithm which is based on the dependency DAG of rules. The algorithm stores the popular used rules with less extra cache cost and more weight. We also discuss how this new wildcard rules caching algorithm performs better than the algorithm in [10] under some cases. Finally we illustrate the simulation environment and process.

The rest of this report is organized as follows. Section II reviews related works. Section III introduces the system model and the definition of the problem. Section IV proposes the new wildcard rules caching algorithm and the Section V introduces the simulation environment and process. A conclusion and future works are drawn in Section VI.

2. Literature Review

In this section we introduce two papers related to the rules caching in software-define networkings. The idea of rule caching is to cache the ‘important’ rules [10]. The dependency among the rules is the key in the wildcard rules caching algorithm. It leads to an inefficient rules caching, which causes the high cost of TCAM.

In first related paper [5], a DAG graph representing the rule dependency and the cover-set concept are introduced. Besides, the wildcard rules caching algorithm which is based on the cover-set is proposed. In the second related paper [10], the wildcard rules caching algorithm is based on the layer-by-layer calculation using the DAG graph proposed in the paper [5].

2.1 Rules Dependency DAG and Cover-set

A mathematical model on rules dependency is proposed in [5]. There are two kinds of dependency in wildcard rules, direct and indirect dependency. As for direct dependency, it means that there is an intersection between two wildcard rules in the field ‘match’. When the rule is selected to be stored in
TCAM, other rules with direct dependency to it are supposed to be cached in TCAM as well. The indirect dependency means that there are two rules depending on the same rule. These two rules have an indirect dependency even though they do not intersect with each other in the field ‘match’.

Besides, an algorithm generating the DAG is proposed in [5] to represent the rule dependency. In a rule dependency DAG, the rule with higher priority is on the higher level and the arrow pointing from the rule to its parent rule represents that the rule is fully or partially overlapping the parent rule in the field ‘match’. The construction process is to capture all dependencies among the rules one by one. From the dependency DAG, if one rule is going to be cached in TCAM, then all the descendants of this rule are supposed to be stored in TCAM to maintain the dependencies relationship. This DAG graph is the base of the wildcard rules caching algorithm.

Based on the DAG dependency graph, the cover-set concept is proposed to solve the rule dependency problem. The cover-set concept splices the dependency chain by creating a new rule in flow table to cover dependencies of low-weight rules [5]. Long chains of dependent rules are spliced by creating cover-set for the rules. In other words, cover-set concept is to find immediate descendants of each low-priority rule and store these descendants in the software switch with their dependencies. The field ‘action’ of the cover-set is to forward the packets to the place where the rule is stored. The aim of creating cover-set is to avoid caching the rule with low weight and high priority. It could also keep the integrality of rule dependencies and improve the efficiency of using TCAM with limited size.

There is a wildcard rule caching algorithm proposed in [5], which is based on the cover-set concept. In this algorithm, the selection of cached rules proceeds greedily: it selects the rule with the highest contribution and cache the rule with the cover-sets to its descendants. The process repeats until there is no available TCAM entry.

2.2 Wildcard Rules Caching Algorithm with Accumulative Contribution

In the rule dependency DAG, there are three terms that are frequently used. The term weight represents the volume of traffic matching the rule. This value is supposed to be collected by the SDN controller. The term cost represents the number of TCAM entries used to store the rule and its descendants and cover-sets. The last term is contribution. It is the ratio between weight and cost. This value is used to determine the rule that is supposed to be cached to gain higher hit ratio. The individual contribution represents the contribution value of caching one rule with its dependent descendants and cover-sets. The accumulative contribution represents the contribution value of caching the
The author in [10] points out the shortcoming of the wildcard rules caching algorithm proposed in [5] and presents a new wildcard rules caching algorithm based on accumulative contribution. The new algorithm is utilizing the rules dependency DAG and considering the situation where the combinations of the rules to be cached at one time.

As for the shortcoming, the wildcard rules caching algorithm in [5] only considers the contribution of the individual rule, and caches only one rule with its cover-set in each iteration. For example, in Fig. 1(a), it denotes the weight and the dependency of each rule. Let us assume the size of TCAM is three. Then, if we apply the wildcard rules caching algorithm in [5], the rule R6 will be picked in the first round because the contribution of it is 40/2, which is the highest. The denominator ‘2’ represents the total cost to cache R6, which is the cost of caching R6 itself and its cover-set rule. Then for the next round, the algorithm will pick R4 with contribution 10/1 because R4 replaces the TCAM entry for the cover-set when caching R6 and cost 1 TCAM entry for its cover-set. After these two rounds, TCAM is occupied with three rules, and the total contribution is (10+40)/3 or simply 50/3. The Fig. 1(b) denotes the result from cover-set caching algorithm. However, as shown in Fig.1 (c), if we select to cache R5 and R3 with its cover set, the total contribution would be (35+35)/3, which is higher than the result obtained from the cover-set caching algorithm. The reason why the algorithm could not get the best result is that the algorithm proceeds greedily and it caches only one rule with the highest individual contribution in each round. In the end, it could get the local optimal result but that might not be the global optimal. The cover-set caching algorithm only considers the contribution value of an individual rule.

Then, a new wildcard rule caching algorithm is proposed in [10] which considers the contribution of wildcard rules.
value of the combination of the rules. The main idea of this algorithm is to calculate the accumulative contribution for each rule layer by layer. The algorithm separates the layers in rules dependency DAG first. The accumulated contribution value for the un-cached rule with each rule in the same layer is calculated. Then, the combination of the wildcard rules with the maximal contribution value will be selected and this process will not stop until there is no available TCAM entry. There are two phases in this algorithm. The first is to select the combination of rules for one un-cached rule. The second is the overall wildcard rules caching algorithm achieved by applying the algorithm in the first phase.

In the simulation, the benchmark ClassBench [12] is used to generate the policy and the packets header files to trace. In the analysis, the improvement of hit ratio using new wildcard rule caching algorithm is outstanding compared with the cover-set caching algorithm. However, the new wildcard rules caching algorithm takes more time in the preparation section.

3 Preliminaries

In this section we first describe the cover-set concept and rules dependency DAG model. We then explain the wildcard rules caching algorithm in [10] and propose the argument about the disadvantages of this algorithm.

3.1 Cover-set and Rule Dependency DAG Model

Initially, we have a wildcard rules set for rules caching, and we construct a DAG dependency graph [2] for the wildcard rules set before applying the wildcard rules caching algorithm. From TABLE 1, we can view the example rules in flow table form. Each rule consists of five fields, which are rule number, match, action, priority and weight. In field ‘match’, it could be represented as source IP address. In Fig. 2(a), it shows the dependency DAG graph for R1 to R6 constructed based on the TABLE 1.

The arrow in the DAG represents dependency. For instance, since R4 has higher priority comparing to R6, and they overlap in the field ‘match’, there is an arrow pointing from R4 to R6 representing their dependency. The Fig. 2(b) shows rules caching with cover-set. Suppose we choose R4, R5 and R6 to be stored in TCAM. Then there will be one cover-set rule for R3, which is represented as R3*. The function of cover-set R3* acts like a pointer that points to the address of R3 in software switch or secondary memory. The format of R3* is shown in TABLE 2, which is a new rule created by the SDN switches. When the packets hit R3*, the switch or router will send the packets to the place where
the rule is stored to match the rule.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Match</th>
<th>Action</th>
<th>Priority</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>000</td>
<td>Forward to port 1</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>R2</td>
<td>00*</td>
<td>Forward to port 3</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>R3</td>
<td>0**</td>
<td>Forward to port 4</td>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>R4</td>
<td>110</td>
<td>Drop</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>R5</td>
<td>10*</td>
<td>Forward to port 2</td>
<td>2</td>
<td>80</td>
</tr>
<tr>
<td>R6</td>
<td>**0</td>
<td>Forward to port 5</td>
<td>1</td>
<td>90</td>
</tr>
</tbody>
</table>

**TABLE 1:** Example rules input in flow table.

![Fig. 2](image)

(a) Rules dependency graph  
(b) Rules caching

<table>
<thead>
<tr>
<th>Rule</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3*</td>
<td>0**</td>
<td>Forward to Software Switch</td>
</tr>
</tbody>
</table>

**TABLE 2:** Cover-set Rule

### 3.2 Existing Wildcard Rule Caching Algorithm

In the paper [10], the wildcard rule caching algorithm is using the layer-by-layer calculation based on the rule dependency DAG. There are two sub-algorithms. The first is to calculate the accumulative contribution for one given un-cached rule and select the combination of rules with maximal contribution value. The second part is the overall wildcard rules caching algorithm achieved by applying the first sub-algorithm.

In the first sub-algorithm, it forms the new layer with the rules dependent on the selected rules. In the layer, each rule is going to be added into the selected rules set to calculate the accumulative
contribution. The order of the rules to be added is dependent on their individual contribution, which is ranked from the highest to the lowest. After adding all candidate rules on the same layer to the rule set, the algorithm will select the combination of rules with highest accumulative contribution value. Next, the algorithm will repeat the process until there are not enough TCAM entries to cache.

For example, in Fig. 3, the algorithm starts with \( R9 \), so the rule set only contained \( R9 \) and the contribution value is the individual contribution value of \( R9 \). Then, it will gather children of \( R9 \), which are \( R6, R7 \) and \( R8 \), and calculate the individual contribution value of each, followed by sorting them from the highest to the lowest. These children form the new layer. On the new layer, the algorithm will start by adding the rule with the highest individual contribution to the rule set to calculate the accumulative contribution. Eventually, every rule will be added into the rule set with a descending order. After calculating the accumulative contribution of all the rules on the same layer, the algorithm will select one combination of rules that has the highest accumulative contribution, in this example it consists of \( R6 \) to \( R9 \). Then, if the size of combination does not overflow TCAM, the algorithm will repeat the same process to form a new layer and calculate the accumulative contribution.

![Diagram of rules dependency DAG](image)

Fig. 3: The example rules dependency DAG in [10]

### 3.3 Argument about the Weakness

From the wildcard rule caching algorithm [10], the layer-by-layer calculation would not give the combination of rules with highest accumulative contribution value in some cases, because this algorithm would use up the available TCAM entries before it reaches the rules with high contribution but on the higher level. Besides, the algorithm uses three constants to limit the number of candidate rules in each layer, the number of layers and the amount of chosen un-cached rules respectively. These constants would affect the result and should be modified to fit different input rules set.
4. Wildcard Rule Caching Algorithm

In this section, we propose a new wildcard rule caching algorithm that surmounts the shortcoming of the layer-by-layer wildcard rules caching algorithm [10].

The new wildcard rules caching algorithm breaks through the layer-by-layer calculation. There are two parts in this algorithm, which are similar to the wildcard rules caching algorithm in [10]. The first part is to calculate the accumulative contribution value for the given un-cached rule and select the combination with the highest contribution. The second part is to repeat the algorithm in the first part until there is not enough space in TCAM. The new wildcard rules caching algorithm makes changes in the first part to surmount the shortcomings for the algorithm in [10].

The main idea of the new wildcard rules caching algorithm is to select the rule with highest individual contribution value from the candidate rules list, add it into the rules set, and update the candidate rules list with the rules dependent on the selected one.

We use Fig. 4 to illustrate how to select combination of rules by this new wildcard rule caching algorithm. Let us assume the size of TCAM is 7. In the Fig. 4(a), it denotes the weight and dependency relationship for the input rules set.

Suppose we start with $R9$ and the contribution of $R9$ is $9/4$. The first step is to check whether the cost of caching $R9$ overflows the available TCAM entries. If not, the candidate rules list ($List$) will store all rules dependent on $R9$ and add $R9$ into the rules set ($set$). If the cost of caching $R9$ overflows the TCAM size, the algorithm will be interrupted and return, hence the current maximum accumulative contribution value ($CMACV$) would be the individual contribution value for $R9$. The result rules set ($ruleSet$), which should be returned in the end, contains the combination of the chosen rules with maximal accumulative contribution value.

Then, from the $List$, we will select the rule with the highest individual contribution. From the Fig. 4(a), the contributions of $R6$, $R7$ and $R8$ are $2/1$, $3/2$ and $8/1$ respectively. Therefore, $R8$ should be chosen. The next step is to calculate the accumulative contribution value ($ACV$) of $R8$ together with the rules $set$. Here, we have $R9$ in $set$. For $R9$ we need to cache itself with 3 entries for its cover-set, and the total $cost$ should be 4 TCAM entries. Then, if $R8$ is also cached in TCAM, it can replace the cover-set pointing to itself and add a new cover set pointing to $R5$. So the value of $cost$ is 1. In the end, 5 entries of TCAM is enough to cache the $R9$ and $R8$. The result is shown in Fig. 4(b). Then, $List$ will remove $R8$ and update the rules dependent on $R8$. At this stage, $List$ includes all rules dependent
on R8 and R9. Besides, if ACV is larger than CMACV, CMACV will be updated by the value of ACV, and R8 is added into ruleSet. The available number of TCAM entries will decrease from 5, which is 2 at this stage.

Next, the algorithm picks up the rule with the highest individual contribution value from the candidate List. In List, there are R5, R6 and R7. The individual contribution values for them are 4/2, 1/1 and 3/2 respectively. As a result, R5 is selected. The total cost of caching R5, R8 and R9 is 7, including four cover sets pointing to R2, R3, R6 and R7 and three rules R5, R8 and R9. The ACV of this combination is 21/7. Then, set adds R5 into its domain and List adds the rules dependent on R5 and removes R5 itself. However, in this round, the value of CMACV and rulesSet will not be updated because the ACV of the combination of R5, R8 and R9 is less than the CMACV. Following that, after repeating the same process, we will get the result, which is shown in Fig.4 (c). The CMACV here is 30/7, and the combination of the rules consists of five rules and two cover-set rules.

![Fig. 4](image)

(a) Rules set  (b) Rules set containing R8 and R9  (c) Final result

![Fig. 5](image)

(a) Rules set  (b) Result under algorithm in [2]  (c) Result under the new algorithm
If we apply the wildcard rule caching algorithm proposed in [10] to the same dependency DAG shown in Fig.4 (a), the result will be the rules caching combination shown in Fig.5 (b). The layer-by-layer calculation will pick all rules dependent on R9 in the first round, which occupied 6 entries in TCAM. It is because of the fact that the algorithm in [10] calculates the ACV by adding each candidate rule within the same layer. Then, since there is only one entry left, the algorithm could only select R3, which costs 1 in TCAM. In the end, the CMACV of this combination of the rules is 24/7. However, the CMACV of the result, which is shown in Fig. 5(c), gained by the new algorithm is 30/7. The difference is due to that the new algorithm breaks through the layer-by-layer calculation and could reach the rule with higher level before consuming all of the entries in TCAM. The candidate list forms by the rules dependent on the selected rules and is updated once a new rule is selected.

Algorithm 1 illustrates that the algorithm calculates the maximal accumulative contribution of one given un-cached rule. The input for Algorithm 1 is the given rule and the number of available TCAM entries. CMACV denotes the current maximal accumulative contribution value, List represents the candidate list that contains rules, and set represents the combination of the selected rules. If the total
cost does not overflow the available TCAM size, the CMAVC is initialized with the contribution of the input rule and List contains the rules dependent on the input rule. The set contains the input rule only (Lines 2-5). If the cost overflows the TCAM, the algorithm will be interrupted and return (Line 6-8).

Then, Algorithm 1 will get into a loop that calculates the ACV, which denotes the accumulative contribution value. With each iteration, the rule with the highest individual contribution is selected and denoted as the variable combineOne. The cost of selecting this rule is calculated and stored in the variable combineCost (Line 11-12). Then, if the total cost is less than the available number of TCAM entries, the ACV will be calculated and the selected rule will be added into set (Line 15-16). Then, the candidate list List removes the selected rule and adds other rules dependent on it (Line 17-19). In the end, if the ACV is larger than the CMACV, the CMACV will be updated as well as ruleSet (Line 23-25).

Assume there are \( N \) rules in a packet policy, the time complexity of calculating the individual contribution for each rule is \( O(N) \). Then, the process that selects the rule with the highest individual contribution among the candidate list takes \( O(N) \) time. The time complexity of combining the candidate rule is \( O(N) \). Then, there are at most \( N \) rules going to be cached into TCAM. Therefore, the total time complexity is \( O(N^2 + N^2) \), which is \( O(N^2) \).

5. Simulations

In this section, we will introduce the simulation of the proposed wildcard rule caching algorithm. The first section is about the experiment environment and introduction to ClassBench [11], followed by the demonstration of the experimental process.

5.1 Simulation Environment

Since we cannot grasp the real data traffic, the simulation is the method taken to verify the algorithm. The first simulation environment setup is the Mininet [12] with OpenFlow [1] controller. Mininet is a network emulator [12]. It could form a computing network on a laptop. It also vitalizes the hardware components such as switches, routers and hosts into software. On this platform, users could decide the topology of network and the number of switches applied. The OpenFlow controller in the Mininet allows us to have a SDN network in the realistic virtual network environment. However, since we need realistic policy rules, the Mininet platform could not generate the policy by itself [12], and therefore, the Mininet is abandoned.
The next attempt is to use ClassBench [11], a tool used for benchmarking packet classification algorithm and devices. It includes two generators, namely the filter set generator and the trace generator [11]. The filter set generator is used to generate synthetic filter sets that simulate the real filter. The trace generator is used to produce the packets headers to exercise the synthetic filter set. With the filter set generator, we could generate the synthetic rule policy with the set number of rules and dependency.

### 5.2 Experiment Process

There are three main processes in this experiment. The first step is to transform the generated policy into the SDN flow table form. Then, the flow table is supposed to be transformed into the rules dependency DAG, and the wildcard rules caching algorithm is applied to the DAG to select the rules that will be cached in TCAM. The final step is to use the trace generator in ClassBench [11] to generate the trace file containing packets header. With this trace file, the hit ratio could be calculated to verify the new wildcard rules caching algorithm.

Since the benchmark ClassBench could not generate the policy in SDN flow table form, the transferring process is required. As it is shown in Fig. 6, the generated rules consist of five fields, which are source IP address, destination IP address, source port, destination port and protocol number. The wildcard rules could be transformed by getting the complemented code of source and destination IP addresses with the subnet mask in these generated rules. Then, the traffic volume of each rule is set by its field space size [10]. The number of packets matching the rule is proportional to its filed space size, although the range may vary. After getting the values in match and weight filed, the priority of each rule is following a descending order based on the values.

The second step is to generate the rules dependency DAG and apply the new wildcard rules caching algorithm to select the rules to cache. The output of the second step is the list of rules that are selected.

In the final step, the trace file generated by the ClassBench is used to calculate the hit ratio. As shown in Fig. 7, there are five fields in each record, which are the same to the rules generated by the ClassBench. Then, the packets header record should be tested if it matches the rules in the list generated in the second step. The number of the matched packets should be saved. The hit ratio is calculated by the number of matched packets divided by the total number of packets.
6. Conclusion

In the report, we have described the rules dependency problem in software-define networks [5] and the wildcard rule caching algorithm [10] that limits the negative effects of the problem. Besides, the concepts of TCAM [4] and cover-set [5] have been introduced. We have also derived a new wildcard rules caching algorithm to solve the problem so that the rules with high contribution value on the high level are not reachable. We had also proposed a simulation method using ClassBench [11].

Since the new wildcard rules caching algorithm is tested only by the example inputs without evaluated by the simulations, the future work includes: (1) implement the simulation and test the new algorithm using ClassBench [11], (2) utilize the SDN test-bed to grasp the real SDN policy and test the algorithm on hardware level, and (3) investigate other wildcard rules caching algorithms to improve the hit ratio.
References


INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniID: u5433077
SURNAME: Jia
FIRST NAMES: Zhenge
PROJECT SUPERVISOR (may be external): Professor Weifa Liang
COURSE SUPERVISOR (a RSCS academic): Professor Weifa Liang
COURSE CODE, TITLE AND UNIT: COMP 3710 Topics in Computer Science, 6

SEMESTER [ ] S1 [ ] S2 YEAR: 2016

PROJECT TITLE:
Rule Caching in Software Define Networking

LEARNING OBJECTIVES:
1. Literature review of SDN and WLANs.
2. Virtualization of base station (VBS).
3. Algorithm design for deployment of VBS.

PROJECT DESCRIPTION:

The project will study VBS based SDNS to introduce programmability in wireless local area networks (WLANs). It takes advantage of a light virtual access point (LVAP) to simplify client management including authentication, authorisation, load balance, etc. Under the premises, the aim of the project is to explore new methods in scheduling of resources to motivate the network throughput.
ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style: _________ Research report (e.g. research report, software description...)</td>
<td>45%</td>
<td>Week 13 (27th May)</td>
<td>Ramesh Sankaranarayana</td>
</tr>
<tr>
<td>Artefact: name kind: _________ study design, experiment and result (e.g. software, user interface, robot...)</td>
<td>45%</td>
<td>Week 13 (27th May)</td>
<td>Weifa Liang</td>
</tr>
<tr>
<td>Presentation: Oral presentation of research findings</td>
<td>10%</td>
<td>Week 13 (24th May)</td>
<td>Weifa Liang</td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):
1. Weekly meeting with supervisor (0.5-1 hours)
2. Weekly attendance at algorithm research group meetings

STUDENT DECLARATION: I agree to fulfil the above defined contract:

………………………………………………….. ………………………
Signature Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

………………………………………………….. ………………………
Signature Date

REQUIRED DEPARTMENT RESOURCES:

SECTION C (Course coordinator approval)

………………………………………………….. ………………………
Signature Date

SECTION D (Projects coordinator approval)

………………………………………………….. ………………………
Signature Date