An Approach to the Translation of Software-Defined Network Switch Flow Table into Network Processing Unit Assembly Language

Andrei Markoborodov
Lomonosov Moscow State University
Moscow, Russia
amark@lvk.cs.msu.su

Yuliya Skobtsova
Lomonosov Moscow State University
Moscow, Russia
xenerizes@lvk.cs.msu.su

Dmitry Volkanov
Lomonosov Moscow State University
Moscow, Russia
volkanov@asvk.cs.msu.su

***Abstract*—This paper considers the OpenFlow 1.3 switch based on a programmable network processing unit (NPU). OpenFlow switch performs flow entry lookup in a flow table by the values of packet header fields to determine actions to apply to incoming packet (classification).**

In the considered NPU assembly language, lookup operation may be implemented on the basis of search trees. But these trees cannot be directly used for OpenFlow classification because of compared operands width limitation. In this paper, we propose flow table representation designed for easy translation into NPU search trees. Another goal was to create a compact program that fits in NPU memory.

Another NPU limitation requires program updating after each flow table modification. Consequently, the switch must maintain the current flow table state to provide a fast NPU program update. We developed algorithms for incremental update of flow table representation (flow addition and removal).

To evaluate the proposed flow table translation approach, a set of flow tables was translated into NPU assembly language using a simple algorithm (based on related work) and an improved algorithm (our proposal). Evaluation was performed on the NPU simulation model and showed that our approach effectively reduces program size.

***Index Terms*—OpenFlow, network processing unit, flow table, software-defined network**

I. Introduction

Software-Defined Networks (SDN) have been actively developed recently. In SDN network devices, or switches, implement data forwarding plane, when device and data flow management (control plane) is performed by special software — SDN controller, running on a separate server [1]. For interaction between the data plane and the control plane, a special control protocol is used. The OpenFlow 1.3 [2] protocol is one of the most widespread SDN control protocols.

Packet processing in the OpenFlow switch is performed using special processing rules (called flow entries in the OpenFlow protocol) organized in flow tables. SDN controller updates these flow entries by sending OpenFlow messages. To classify incoming packets, OpenFlow switch looks up for the flow entry in the flow table that matches values of corresponding packet header fields.

One of the directions of the SDN technology development are high-performance switches based on programmable network processor units (NPU) [3], which are widely used. NPU is a System-on-a-Chip with architecture specialized for network traffic processing. NPU performs packet header parsing, classification of incoming packets, modification of the packet header and traffic management functions [4]. Programmable NPUs allow us to change packet processing algorithms and distinguishable packet header fields, which is highly valuable in SDN deployments with emerging standards like data centers or 5G [5].

NPU is a specialized device that executes packet processing program loaded into its memory and usually does not make changes to its program itself. The central processing unit (CPU) of the switch implements the interface with the SDN control plane. OpenFlow software in the operating system environment of the CPU provides a connection with the controller and makes changes to the NPU program. Program update requires a special system to translate OpenFlow abstractions into the assembly language of the NPU. This research is devoted to the development of such a system, specifically, its part responsible for packet classification according to the flow table.

The paper has the following structure. Section II describes the main architectural features of the NPU and its assembly language. Section III contains the problem statement of this research. In Section IV, we perform an analysis of related work applied to flow table representation in considered NPU. Section V presents developed data structure and algorithms for translating it into the assembly language and also the data structure updating algorithms. Section VI is devoted to the evaluation of the developed flow table translation approach.

II. NPU architecture

Our research considers a switch with the NPU based on specialized computing cores. This NPU contains a set of parallel packet processing pipelines consisting of the uniform stages that execute binary code loaded into them.

The computing core of the NPU pipeline stage contains a single general-purpose register and a memory area to store the currently processed packet header and associated metadata (such as ingress port identifier). The register is used as an

operand register and a result register. NPU does not contain a separated memory area to store program data. Program data is recorded directly to the binary code of the stage processor instructions. Thus, any change of the data, such as flow removal in OpenFlow, requires a new program to be loaded.

The assembly language of the NPU contains conditional jump instructions that compare the value in the register with the value from the instruction operand. Length of the value should be 64 bits or less. Conditional jump can be made only to the label located in the program below. Therefore, for example, it is impossible to implement loops or return to previously defined packet modifying action.

The program in the assembly language of the NPU can be represented as a finite set of linear instruction blocks connected by jump instructions. The program contains the following main types of linear instruction blocks:

- **Load value.** The sequence of instructions of this block loads a value from the packet header memory area or associated metadata memory into the register.
- **Change register.** The sequence of instructions of this block performs arithmetical or logical operations to change the current value of the register.
- **Search tree.** In the simplest case (exact match search tree) this block contains a set of jump instructions. The search key is an integer value, which length is 64 bits or less. It can also be the longest-prefix match search tree, which additionally requires prefix length for the search key.
- **Apply actions.** The sequence of instructions of this block performs modifying actions on the packet header memory or associated metadata memory, such as changing header field values, pushing tags.

A directed graph of the program can be created from the instruction blocks of the program. In this graph vertices are created for each instruction block. An arc leads from one vertex to another, if the block corresponding to the first vertex has jump instruction to label located in instruction block corresponding to the second vertex. The program graph has no cycles and contains only one vertex without incoming arcs.

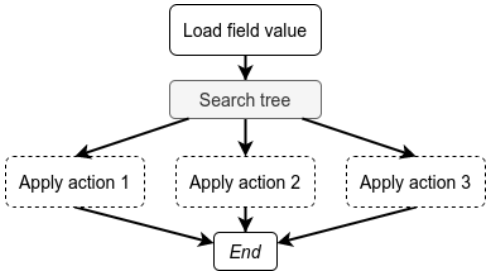

Fig. 1. Example NPU program graph

Figure 1 shows the graph for a simple NPU program which performs classification by the value of one header field and applies one of the three packet modifying actions. The program has one instruction block for loading the value of this field and one block containing the search tree. In the program graph three arcs lead from the vertex of search tree block to three vertices of action blocks to apply.

III. Problem Statement

Consider a switch that operates under the OpenFlow 1.3 protocol, based on the NPU described in Section II. Let R be flow table with flow entries containing only match fields with exact values. The set of flow table match fields is denoted by $I = \{m_1, m_2, \ldots, m_k\}$. Flow entry may specify exact value only for a subset of I allowing any value in other match fields. Let the symbol "$*$" denote any value of the match field. To avoid search ambiguity, each flow entry is marked with priority p.

Our goal is to create a program in the assembly language that is compliant with the graph described in Section II and performs received packet classification by the given flow table R. The program must perform the search for matching flow entry in the flow table that has the highest priority and matches packet header fields.

Additionally, we have to load a new program into the NPU each time flow table contents are changed. Considering the usual frequency of flow table updates, it is advisable to maintain an incrementally updated intermediate representation of the flow table for quick translation after the update.

Thus, the problem is to develop a data structure for translating given flow table R into the program in the assembly language of the considered NPU, which implements a search on this set of flow entries and supports the addition and removal of flow entries.

IV. Related Work

This section provides a brief review of other researches devoted to data structures developed for classification by the flow table or similar multi-field tables.

The papers [6], [7] investigate an approach based on the decomposition of the classification by many fields into several classifications by one field. This approach uses a separate data structure for each match field, such as search trees or hash tables. The search result for one data structure is the Bloom filter [8] or label identifier. To get the classification result for all fields, it is necessary to intersect pairs of separate classification results. As a result, an identifier of the required flow entry is calculated.

This approach has significant limitations in implementation for the considered NPU, including the impossibility of hash function implementation required for Bloom filters and the necessity to store intermediate labels when classification results are intersected.

The papers [9], [10], [11], [12] suggest an approach that uses decision trees. Each vertex of such a tree is associated with a predicate. During the search, the predicate determines the next descendant vertex to continue the search. During passing from vertex to its descendant, the initial set of flow entries decreases and, as a result, turns into a smaller set of

flow entries, among which the desired flow entry is determined by simple enumeration.

This approach also has limitations for implementing in the considered NPU. In the search process, it is required to load the header field values more than once, that can lead to unreasonable expenses for the packet processing time and multiple duplication of instruction blocks for loading the field value.

All considered approaches to the representation of flow tables have limitations and disadvantages for their implementation in the assembly language of considered NPU. Data structures based on decision trees are more suitable for our research problem. However, the disadvantages of such data structures should be eliminated, or the program just will not fit into NPU memory.

V. Proposed Approach

In this section, we describe the developed data structure for representing the flow table and the developed algorithms for flow addition and removal from the data structure and for translating the data structure into a program in the assembly language of the NPU.

A. Data Structure

To represent a flow table with the set of flow entries R, we use a tree with marked vertices and arcs. The following values are associated with each tree vertex, except for leaf vertices:

- Match field from the set of considered fields $I = \{m_1, m_2, \ldots, m_k\}$: the tree root corresponds to the field m_1, the descendants of the root correspond to the field m_2, etc.
- Subset of the flow set R. The tree root corresponds to the whole set R.

TABLE I
Example flow table

Flow	Priority	Field 1	Field 2
F_1	2	0	0
F_2	2	0	1
F_3	2	1	0
F_4	1	1	*

Each tree leaf is associated with a flow entry subset of R sorted in descending priority order. The tree has a depth of k, and all the tree leaves are vertices of the depth k. Table I presents an example flow table with two match fields, consisting of four flow entries F_1, F_2, F_3, F_4. In Figure 2, the data structure constructed for example flow table in Table I is shown.

Consider the tree vertex v, which corresponds to the field m and a flow subset $S \subset R$. Then:

- If M is a set of all possible values of the field m in the flow entries from the set S, including the special value $*$, for each value $f \in M$ the vertex v has a descendant which arc is marked f.
- If the vertex u is a descendant of the vertex v with arc marked f, the flow subset of the vertex u contains only those flow entries from S, which value for the field m is f or $*$.

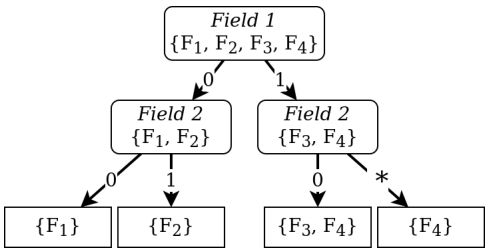

Fig. 2. Data structure constructed from flow entries given in Table I

The developed data structure differs from approaches shown in related work by a fixed order of viewing fields. Vertices having the same depth refer to the same match field. This allows us to load the value of each match field only once in the search process. This order also allowed us to develop an algorithm for translating the data structure into the assembly language of the NPU, which receives the program without duplicating the instruction blocks for loading field value.

B. Flow Addition

Flow addition to the data structure is performed by traversing the tree vertices, starting from the root. When traversing a vertex, a new flow entry is added to its flow subset. Then, traversing continues in vertex along the arc, which is marked with the value of the field specified in the added flow entry. If such an arc is in absent, a new vertex descendant is added.

When traversing a vertex, two special cases require additional actions:

1) The value of the field corresponding to the vertex specified in the added flow entry is $*$. In this case, in addition to the descendant along the arc marked $*$, it is necessary to traverse all other descendants of this vertex.
2) The value of the field corresponding to the vertex in the added flow entry is $f \neq *$, and this vertex has a descendant along the arc marked $*$. Then, in case of adding a new descendant, it is necessary firstly to copy the subtree corresponding to the arc marked $*$ to the subtree along the arc marked f, and then continue the traversal.

C. Flow Removal

Flow removal from the data structure is also performed by traversing the tree vertices, starting from the root. When traversing a vertex, the removed flow entry is deleted from the vertex flow subset, and traversing continues in vertex along the arc, which is marked with the value of the field specified in the removed flow entry.

When traversing a vertex, two special cases require additional actions:

1) The value of the field corresponding to the vertex specified in the removed flow entry is $*$. In this case, in addition to the descendant along the arc marked $*$, it

is necessary to traverse all the other descendants of this vertex.

2) The value of the field corresponding to the vertex in the removed flow entry is $f \neq *$, and this vertex has a descendant along the arc marked $*$. For this case, after traversing the subtree along the arc marked f, it is necessary to compare the subtree along the arc marked f and the subtree along the arc marked $*$. In case of equality, the subtree along the arc f is removed, because it is redundant.

D. Translation into NPU Assembly Language

Proposed data structure can be directly translated into a program in the assembly language of the NPU. The program graph will have a structure similar to a tree, but for each vertex of the tree, except for the leaves, the program graph will contain sequentially connected vertices corresponding to the instruction block of loading the field value, which corresponds to the vertex, and the instruction block of a search tree, for the values that mark the outgoing arcs from the vertex. The tree leaf will correspond to the instruction blocks of actions of the flow entry that has the highest priority in leaf flow set. Listing 1 and Figure 3 show the program and the program graph translated by the direct method from flow table representation presented in Figure 2.

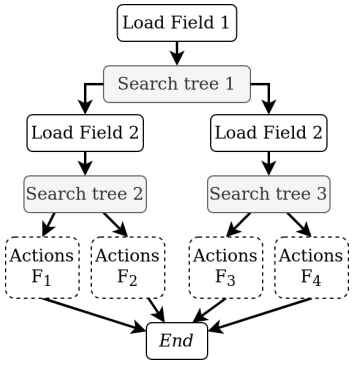

Fig. 3. Program graph obtained by direct translation method

However, with the direct method of translation, the resulting program contains a lot of search trees for similar key sets and duplicating instruction blocks for loading the field value (see duplicating field loading block in Figure 3). To eliminate this drawback, a method for translating the data structure with encoding arcs was developed (method with encoding).

When translating by the method with encoding, tree levels are introduced. Tree level corresponds to the table match field and includes vertices of the same depth. The arcs outgoing from the tree levels vertices are numbered, that is, the code is assigned to each arc. Numbering for each level is independent. Then, for each tree level, a level list, consisting of all pairs *(code of the incoming arc, marker of the outgoing arc)* is formed. For the root vertex, zero is used instead of the incoming arc code. Figure 4 shows arc encodings, tree levels, and level lists for our example data structure.

Listing 1
PROGRAM OBTAINED BY DIRECT TRANSLATION METHOD

```
    <...> // Load Field 1
    tree_in "tree_1"
    j End
L1: // Load Field 2
    tree_in "tree_2"
    j End
L2: <...> // Load Field 2
    tree_in "tree_3"
    j F4
F1: <...> // Actions F1
    j End
F2: <...> // Actions F2
    j End
F3: <...> // Actions F3
    j End
F4: <...> // Actions F4
    j End
End:
```

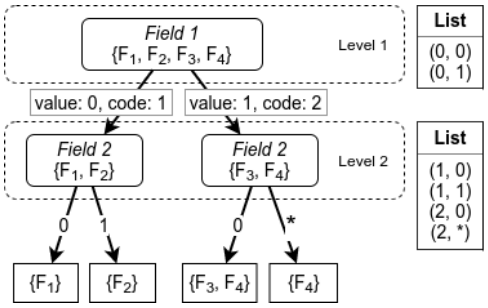

Fig. 4. Flow table representation marked up with tree levels, encoded arcs and level lists

Instruction blocks of the search tree and loading field corresponding to the level are created for each list of pairs. Jumps in the block of the search tree are performed in a special block for loading the code of the outgoing arc, to which the pair corresponds. Then jump is performed to loading the value of the next field. In Figure 5, the resulting program graph, obtained by the method with encoding from marked flow table representation in Figure 4, is shown. In Listing 2, the corresponding program in the assembly language is presented.

Thus, the resulting program contains one loading instruction block for each match field and a fixed number of search trees, one search tree per match field.

VI. EVALUATION

For the developed data structure, we evaluated translation method with encoding in comparison to the direct translation method inspired by approaches from related work.

For the evaluation, we used a simulation model of the NPU pipeline. The simulation model receives a program in the assembly language, translated by one of the methods in our case, and a set of input packets to be processed. As an output the model produces a set of outgoing packets along with statistics, including the amount of memory occupied by the program binary code and the average number of ticks spent on processing the input packets. Before processing

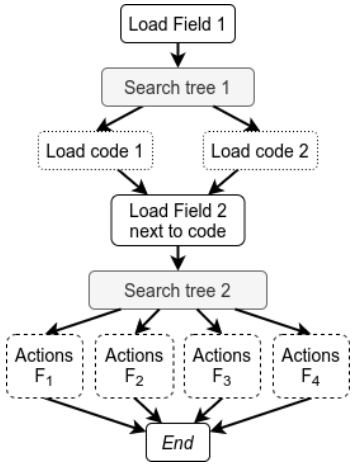

Fig. 5. Program graph obtained by translation method with encoding

Listing 2
PROGRAM OBTAINED BY TRANSLATION METHOD WITH ENCODING

```
    <...> // Load Field 1
    tree_lpm "tree_1"
    j End
L1: loadi 1
    j L3
L2: loadi 2
    j L3
L3: rol FIELD_2_WIDTH
    <...> // Load Field 2
    tree_lpm "tree_2"
    j End
F1: <...> // Actions F1
    j End
F2: <...> // Actions F2
    j End
F3: <...> // Actions F3
    j End
F4: <...> // Actions F4
    j End
End:
```

the packets, the simulation model translates the program in assembly language into binary code, where each instruction has 16 bytes length.

We generated a set of OpenFlow tables with match fields of the data link (L2) and network layer (L3) header fields with different numbers of flow entries. For each table, the evaluated data structure was built and translated into NPU assembly language. For each flow table, one input packet per flow entry was generated.

The dependency between the number of table flow entries and the average number of instructions is presented in Figure 6. The measurements show that the program translated by the direct method takes from 1.2 to 1.5 times more memory than the program translated by the method with encoding.

Figure 7 shows the proportion of the search trees instructions in the program depending on the number of table flow entries. The proposed method with encoding uses memory much more effectively than the direct method, removing from

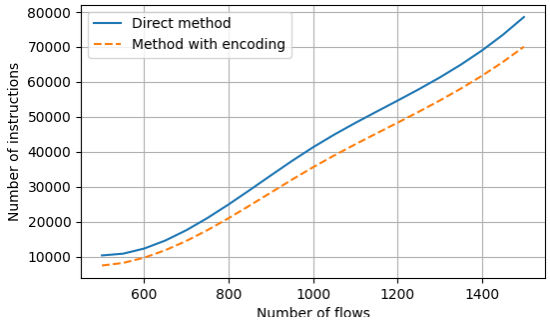

Fig. 6. Average number of instructions for different flow table sizes

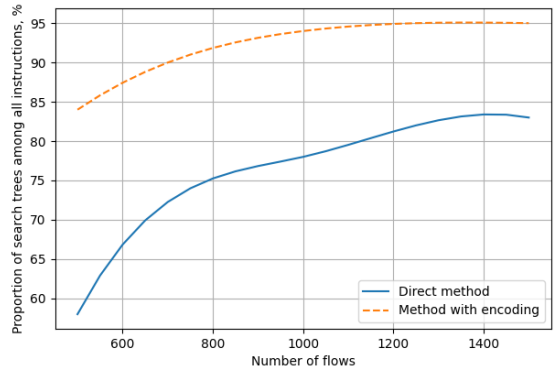

Fig. 7. Proportion of the search trees instructions for different flow table sizes

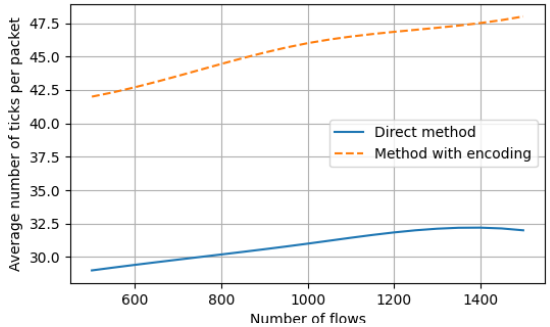

Fig. 8. Average number of ticks per packet for different flow table sizes

15 to 30% duplicating code from the program.

However, the fee for reducing the size of the program is an increase in the number of ticks per packet, which is about 10%. In Figure 8, the dependency between the number of flow entries in the table, and the average number of ticks per packet is rendered. The increase in packet processing time, though, is still within acceptable limits for our NPU and does not lead to unexpected delays or packet drops.

In future research, we are going to determine the dependencies of the evaluated characteristics by the number of match fields in the flow entries. All of the proposed algorithms, including flow addition and removal, will be evaluated in terms of data structure update time.

VII. Conclusion

In our research, we considered the switch based on programmable NPU, which has architectural limitations in memory organization. To use this NPU in the SDN switch operating under the OpenFlow 1.3 protocol, the system for translating flow table into NPU program was developed. The system allows us to get a program for the NPU with acceptable packet processing time, which takes up to 30% less memory comparing to the programs based on data structures in the considered related work. These results are achieved by reducing the duplication of instruction blocks that load the value of the same fields, reducing the program size (in some cases by 1.5 times). In the future, we will consider maskable match fields of the flow entries and examine the effect of the fields parsing order on the resulting program characteristics.

References

[1] Open Networking Foundation. Software-defined networking: the new norm for networks. *ONF white paper*, 2012.

[2] Open Networking Foundation. OpenFlow switch specification version 1.3.0. 2012.

[3] Giladi R. Network processors: architecture, programming, and implementation. *Morgan Kaufmann*, 2008.

[4] Orphanoudakis T., Perissakis S. Embedded multi-core processing for networking. *Embedded Multi-Core Systems*, 2010, pp. 399–463.

[5] Bifulco R.,Rtvri G. A survey on the programmable data plane: abstractions, architectures, and open problems. *IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–7.

[6] Taylor D., Turner J. Scalable packet classification using distributed crossproducting of field labels. 2005.

[7] Kekely M., Korenek J. Packet classification with limited memory resources. *Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 179–183.

[8] Bloom B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970, vol. 13, no. 7, pp. 422–426.

[9] Gupta P., McKeown N. Classifying packets with hierarchical intelligent cuttings. *IEEE Micro*, 2000, vol. 20, no. 1, pp. 34–41.

[10] Singh S., Baboescu F., Varghese G., Wang J. Packet classification using multidimensional cutting. *Computer Communication Review*, 2003, vol. 33.

[11] Qi X., Xu L., Yang B. Packet classification algorithms: from theory to practice. *Proceedings - IEEE INFOCOM*, 2009, pp. 648–656.

[12] Li W., Li X., Li H., Xie G. CutSplit: a decision-tree combining cutting and splitting for scalable packet classification. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2645–2653.