



- (51) **International Patent Classification:**
H04L 12/721 (2013.01) H04L 12/935 (2013.01)
H04L 12/741 (2013.01)
- (21) **International Application Number:**
PCT/US2014/036274
- (22) **International Filing Date:**
30 April 2014 (30.04.2014)
- (25) **Filing Language:** English
- (26) **Publication Language:** English
- (30) **Priority Data:**
61/878,032 15 September 2013 (15.09.2013) US
14/069,284 31 October 2013 (31.10.2013) US
- (71) **Applicant:** NICIRA, INC. [US/US]; 3401 Hillview Avenue, Palo Alto, CA 94304 (US).
- (72) **Inventors:** PETTIT, Justin; 3401 Hillview Avenue, Palo Alto, CA 94304 (US). JACKSON, Ethan, J.; 3401 Hillview Avenue, Palo Alto, CA 94304 (US). GROSS, Jesse, E.; 3401 Hillview Avenue, Palo Alto, CA 94304 (US). ZHOU, Andy; 3401 Hillview Avenue, Palo Alto, CA 94304 (US).

- (74) **Agent:** ADELI, Mani; Adeli LLP, 11859 Wilshire Blvd, Suite 500, Los Angeles, CA 90025 (US).
- (81) **Designated States** (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.
- (84) **Designated States** (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) **Title:** DYNAMICALLY GENERATING ENTRIES IN FLOW TABLES FROM ENTRIES HAVING WILDCARD FIELDS

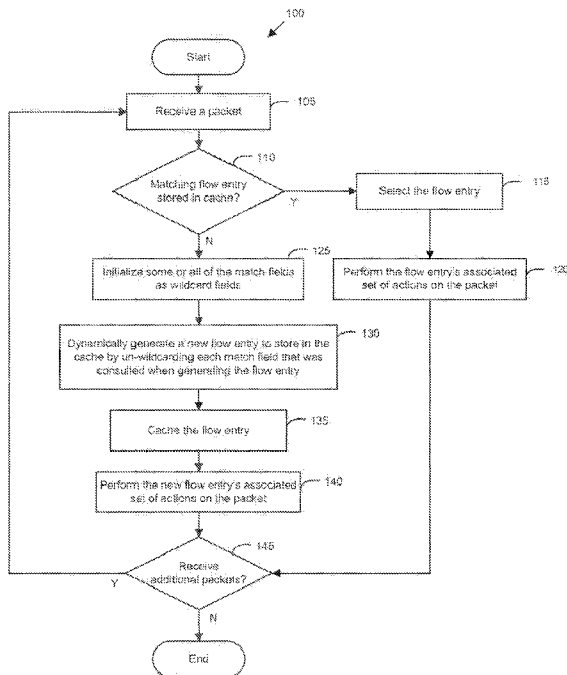


FIG. 1

(57) **Abstract:** Some embodiments of the invention provide a switching element that receives a packet and processes the packet by dynamically generating a flow entry with a set of wildcard fields. The switching element then caches the flow entry and processes any subsequent packets that have header values that match the flow entry's non-wildcard match fields. In generating the flow, the switching element initially wildcards some of all of match fields and generates a new flow entry by un-wildcarding each match field that was consulted or examined to generate the flow entry.



Published:

— *with international search report (Art. 21(3))*

DYNAMICALLY GENERATING ENTRIES IN FLOW TABLES FROM ENTRIES HAVING WILDCARD FIELDS

BACKGROUND

[0001] Many current enterprises have large and sophisticated networks comprising switches, hubs, routers, servers, workstations and other network devices, which support a variety of connections, applications and systems. The increased sophistication of computer networking, including virtual machine migration, dynamic workloads, multi-tenancy, and customer specific quality of service and security configurations require a better paradigm for network control. Networks have traditionally been managed through low-level configuration of individual components.

[0002] In response, there is a growing movement, driven by both industry and academia, towards a new network control paradigm called Software-Defined Networking (SDN). In the SDN paradigm, a network controller, running on one or more servers in a network, controls, maintains, and implements control logic that governs the forwarding behavior of shared network switching elements on a per user basis.

[0003] Typically, the shared network switching elements are software switching elements. A software switching element brings many features that are standard in a hardware switch to virtualized environments. The software switching element can also be more flexible than the hardware switch. For instance, the software switching element can be programmed to emulate a traditional switch pipeline or can be programmed to extend for new models.

[0004] One of the main challenges in developing such a software switching element is performance. A hardware switch has application-specific integrated circuits (ASICs) that are specifically designed to support in-hardware forwarding. The problem with a software switch is that it operates on a host (e.g., x86 box), such as a hypervisor. Thus, methods for increasing the performance of the software switching element are needed.

BRIEF SUMMARY

[0005] Some embodiments of the invention provide a switching element that receives a packet and processes the packet by dynamically generating a flow entry with a set of wildcard fields. The switching element then caches the flow entry and processes any subsequent packets that have header values that match the flow entry's non-wildcard match fields. In other words, each subsequent packet does not have to have header values that match all of the flow entry's match fields but only its non-wildcard fields.

[0006] In generating the flow, the switching element initially wildcards some of all of match fields and generates a new flow entry by un-wildcarding each match field that was consulted or examined to generate the flow entry. The switching element of some embodiments generates a flow entry with wildcard fields based on one or more flows in a set of one or more flow tables. As an example, when a packet is received, the switching element may select a flow from a flow table and un-wildcards each match field that is compared against a packet header value. In some embodiments, the match field is un-wildcarded regardless of whether there was a match between the match field and the header value.

[0007] The switching element may iteratively select the next flow in the flow table until a matching flow is found. If a matching flow is found, it then generates a flow entry with zero or more wildcard fields, depending on the number of remaining wildcard match fields. In some cases, the switching element may recirculate to find one or more other matching flows to generate one consolidated flow entry for multiple flows from one or more flow tables. In some embodiments, the switching element uses one of several different classification algorithms to find a matching flow. Examples of such algorithms include a tuple space search algorithm, a staged lookup algorithm, and a decision tree algorithm.

[0008] Alternatively, or conjunctively with such matching, the switching element of some embodiments un-wildcards match fields based on an action associated with a matching flow. As an example, the switching element of some embodiments can be configured to do normal L2 processing. In such cases, the switching element may use a media access control address (MAC) learning algorithm to derive an output port for a given packet based on a set of one or more packet header values. The switching element can use the algorithm to identify a MAC address of a machine (e.g., a virtual machine) that is connected to its port or attached to its virtual interface. The switching element can then compare the MAC address to the header's

destination Ethernet address and specify an output port if the addresses matches one another. Since the destination MAC address has been looked at, the switching element can then un-wildcards the corresponding match field when generating the wildcard flow.

[0009] Generating a new flow entry can be an expensive or processor intensive operation. This is because the switching element might have to derive match field values and/or iterate through flows in a flow table to find a matching flow. Even if a matching flow is found, one of the flow's associated actions may specify a resubmit operation to find another flow from the flow table. As such, the switch element may potentially have to perform a number of different tasks just to generate one flow entry. However, by generating a flow entry with such wildcard fields, the switching element does not have to generate a new flow entry when it receives a similar packet with one or more different header values. The switching element can continue to process such a similar packet as long as its header values match the non-wildcard fields.

[0010] The preceding Summary is intended to serve as a brief introduction to some embodiments as described herein. It is not meant to be an introduction or overview of all subject matter disclosed in this document. The Detailed Description that follows and the Drawings that are referred to in the Detailed Description will further describe the embodiments described in the Summary as well as other embodiments. Accordingly, to understand all the embodiments described by this document, a full review of the Summary, Detailed Description and the Drawings is needed. Moreover, the claimed subject matters are not to be limited by the illustrative details in the Summary, Detailed Description and the Drawings, but rather are to be defined by the appended claims, because the claimed subject matters can be embodied in other specific forms without departing from the spirit of the subject matters.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] The novel features of the invention are set forth in the appended claims. However, for purposes of explanation, several embodiments of the invention are set forth in the following figures.

[0012] **Figure 1** conceptually illustrates a process that some embodiments use to generate a flow with zero or more wildcard fields.

[0013] **Figure 2** illustrates an example of a switching element that dynamically generates a flow with a wildcard field.

[0014] **Figure 3** shows an example how the switching element uses the flow with the wildcard field to process subsequent packets.

[0015] **Figure 4** conceptually illustrates an architectural diagram of a software-switching element of some embodiments.

[0016] **Figure 5** conceptually illustrates several processes, which show the interactions between several switch components when generate and cache flow with zero or more wildcard fields.

[0017] **Figure 6** illustrates an example of a switching element that finds a matching flow that is stored in a cache.

[0018] **Figure 7** illustrates an example of transferring control to a switch daemon when there is no matching flow stored in the cache.

[0019] **Figure 8** illustrates an example of the switch daemon generating a flow to store in the cache.

[0020] **Figure 9** shows an example of un-wildcarding a match field because it was compared against a header value.

[0021] **Figure 10** illustrates an example of iterating through multiple flows and un-wildcarding multiple match fields.

[0022] **Figure 11** provides an illustrative example of a resubmit operation.

[0023] **Figure 12** illustrates an example of the switching element un-wildcarding a match field that is not explicitly specified in a flow from a flow table.

[0024] **Figure 13** illustrates an example of the switching element un-wildcards the corresponding MAC address fields based on an action associated with a flow.

[0025] **Figure 14** provides an illustrative example of how the switching element utilizes a number of different components to generate the flow entry.

[0026] **Figure 15** illustrates an example of how the datapath flow generator generates a flow by interfacing with different components.

[0027] **Figure 16** conceptually illustrates a process that some embodiments perform to dynamically generate a new flow with zero or more wildcard fields.

[0028] **Figure 17** illustrates several examples of flows that are stored in a datapath cache.

[0029] **Figure 18** conceptually illustrates an electronic system with which some embodiments of the invention are implemented.

DETAILED DESCRIPTION

[0030] In the following detailed description of the invention, numerous details, examples, and embodiments of the invention are set forth and described. However, it will be clear and apparent to one skilled in the art that the invention is not limited to the embodiments set forth and that the invention may be practiced without some of the specific details and examples discussed.

[0031] Some embodiments of the invention provide a switching element that receives a packet and processes the packet by dynamically generating a flow entry with a set of wildcard fields. The switching element then caches the flow entry and processes any subsequent packets that have header values that match the flow entry's non-wildcard match fields. In other words, each subsequent packet does not have to have header values that match all of the flow entry's match fields but only its non-wildcard fields. By generating a flow entry with such wildcard fields, the switching element does not have to generate a new flow entry when it receives a similar packet with one or more different header values. The switching element can continue to process such a similar packet as long as its header values match the non-wildcard fields.

[0032] **Figure 1** conceptually illustrates a process 100 that some embodiments use to process packets. Specifically, the process 100 illustrates an example of dynamically generating a flow with zero or more wildcard fields. In some embodiments, the process 100 is performed by a switching element. The process 100 begins when it receives (at 105) a packet. The process 100 then determines (at 110) whether there is a flow entry stored in the cache that can process the packet. In particular, it determines whether there is a flow entry in which all of its non-wildcard fields match the corresponding header values. If a matching flow entry is found, the process 100 selects (at 115) the flow entry. The process 100 then performs (at 120) a set of actions on the packet according to the selected flow entry.

[0033] When there is a miss in the cache, the process 100 dynamically generates a new flow entry. In generating the flow entry, the process 100 initializes (at 125) some or all match as wildcard fields. The process 100 then generates (at 130) a new flow entry by un-wildcarding each match field that was consulted or examined when generating the flow entry. That is, it un-wildcards each match field that it cared about (e.g., looked at) during the generation of the flow entry.

[0034] The process 100 of some embodiments generates a flow entry with wildcard fields based on one or more flows in a set of one or more flow tables. As an example, when a packet is received, the process 100 may select a flow from a flow table and un-wildcards each match field that is compared against a packet header value. In some embodiments, the match field is un-wildcarded regardless of whether there was a match between the match field and the header value. The process 100 may iteratively select the next flow in the flow table until a matching flow is found. If a matching flow is found, it then generates a flow entry with zero or more wildcard fields, depending on the number of remaining wildcard match fields. In some cases, the process 100 may recirculate to find one or more other matching flows to generate one consolidated flow entry for multiple flows from one or more flow tables.

[0035] In some embodiments, the process 100 uses one of several different classification algorithms to find a matching flow. Examples of such algorithms include a tuple space search algorithm, a staged lookup algorithm, and a decision tree algorithm. These algorithms will be described below in sub-section III.C below.

[0036] Alternatively, or conjunctively with such matching, the process 100 of some embodiments un-wildcards match fields based on an action associated with a matching flow. As an example, the process 100 of some embodiments can be configured to do normal L2 processing. In such cases, the process 100 may use a media access control address (MAC) learning algorithm to derive an output port for a given packet based on a set of one or more packet header values. The process 100 can use the algorithm to identify a MAC address of a machine (e.g., a virtual machine) that is connected to its port or attached to its virtual interface. The process 100 can then compare the MAC address to the header's destination Ethernet address and specify an output port if the addresses matches one another. Since the destination MAC address has been looked at, the process 100 can then un-wildcards the corresponding match field when generating the wildcard flow.

[0037] As shown in **Figure 1**, upon generating the flow entry, the process 100 then stores (at 135) the flow entry in the cache. The process 100 caches the flow entry so that it can process 100 any subsequent packets with header values that match all of the flow's non-wildcard fields. The process 100 performs (at 140) the new flow entry's associated set of actions of the received packet. The process 100 then waits (at 145) for additional packets. If

there is another packet, the process 100 returns to 105, which is described above. Otherwise, the process 100 then ends.

[0038] Some embodiments perform variations on the process 100. The specific operations of the process 100 may not be performed in the exact order shown and described. The specific operations may not be performed in one continuous series of operations, and different specific operations may be performed in different embodiments.

[0039] Having described one example process, several additional examples of generating and using wildcard flows will now be described by reference to **Figures 2** and **3**. **Figure 2** illustrates an example of dynamically generating a wildcard flow. Specifically, this figure shows a switching element 205 that examines a flow entry in a flow table and generates a cache flow entry with at least one wildcard field. This example is shown in terms of five stages of operations 245-265 of the switching element. The switching element 205 includes a datapath flow generator 210 and a datapath manager 220.

[0040] The switching element 205 forwards data (e.g., data frames, packets, etc.) from one network node (e.g., machine) to one or more other network nodes. The switching element 205 can receive packets and can process those packets according to a set of flow entries in one or more flow tables (e.g., the flow table 215). The switching element 205 of some embodiments is a software or virtual switch. In some embodiments, a software is formed by storing its flow table(s) and logic in the memory of a standalone device (e.g., a standalone computer), while in other embodiments, it is formed by storing its flow table(s) and logic in the memory of a device (e.g., a computer) that also executes a hypervisor and one or more virtual machines on top of that hypervisor.

[0041] The datapath flow generator 210 is a component of the switching element 205 that makes switching decisions. The datapath manager 220 receives the switching decisions, caches them, and uses them to process packets. For instance, when a packet comes in, the datapath manager 220 first checks the datapath cache 225 to find a matching flow entry. If no matching entry is found, the control is shifted to the datapath flow generator 210. The datapath flow generator 210 then examines a flow table (e.g., the flow table 215) to generate a flow to push down to the datapath cache 225. In this manner, when any subsequent packet is received, the datapath manager 220 can quickly process the packet using the cached flow entry. The

datapath manager 220 provides a fast path to process each packet. However, the switching decisions are ultimately made through the datapath flow generator 210, in some embodiments.

[0042] The process of determining what to do when there is a missed in the cache can be an expensive operation. The process must be performed each time there is a miss in the cache. The switching element 205 must perform a number of different tasks to generate a flow entry to store in the datapath cache 225. As an example, the datapath flow generator 210 must iterate through one or more flow tables to find a matching flow entry. This can entail dynamically generating a flow based on a default rule if no matching table entry is found. For instance, the switching element can be configured to do normal L2 and/or L3 processing for any packet without a matching flow entry. The generation of a flow entry to store in the cache can also entail performing a number of resubmits, and deriving or learning output ports, etc.

[0043] To speed up processing, the switching element 205 of some embodiments supports flows with wildcards instead of only exact match flows. In particular, it generates a flow with one or more wildcards and caches that flow. This flow is also referred to as a megaflow because it allows packets that have different wildcard values to be quickly processes. In other words, this caching now collapses similar flows into one userspace "megaflow" that makes many flow misses a cache lookup instead of a full translation (e.g., through the datapath flow generator 210).

[0044] Having described several component of the switching element 205, an example of dynamically generating a wildcard flow will now be described by reference to the five operational stages 245-265 that are illustrated in **Figure 2**. The first stage 245 illustrates the switching element 205 receiving a packet 230. The switching element 205 is associated with the flow table 215. To simplify the description, the flow table 215 includes only one flow 235. However, one of ordinary skill in the understand that the table can include many more flows.

[0045] The flow 235 in some embodiments is a table entry that is used to match and process packets. It includes a set of match fields to match packets and one or more actions to perform on matching packets. In the example of **Figure 2**, the match fields are ingress port and Ethernet destination address (e.g., destination MAC addresses). The action is shown as a number two. This number represents an output port. Accordingly, the flow 235 provides a rule that states that all packets received through ingress port one and that has the matching Ethernet destination address should be output to port two.

[0046] Aside from the match fields and the action, the flow entry 235 can include other variables, such as priority and timeout values. The priority value identifies the matching precedence of a flow. For example, the flow table 215 can include multiple entries that can handle a same packet. In some embodiments, the switching element iterates through flow entries based on the priority value. The switching element might examine a flow entry with a higher priority value before a lower one. In this way, a flow entry with a higher value will match before a lower one. The first match can then be used to generate a cache flow entry. So, typically, the matching flow with the highest priority is used to process a packet, but this may not be true in all cases. As mentioned above, the switching element might re-search the same flow table to find one or more other flows. In such cases, the switching element might consolidate the flows into one cache flow entry. The switching element might perform multiple sets of action or just one set of action (e.g., the matching flow with the highest priority value). Different from the priority value, the timeout value represents how long the flow stays in the flow table 215 before it expires. This can be an idle timeout (e.g., if it is inactive) or even a hard timeout (e.g., regardless of its activity).

[0047] The first stage 245 also illustrates the packet 230 that is sent to the switching element. The term "packet" is used here as well as throughout this application to refer to a collection of bits in a particular format sent across a network. One of ordinary skill in the art will recognize that the term "packet" may be used herein to refer to various formatted collections of bits that may be sent across a network, such as Ethernet frames, TCP segments, UDP datagrams, IP packets, etc.

[0048] In some embodiments, the packet is a unit of data that comprises header and user data. The header of a packet includes control information that is necessary for the delivery network to deliver the packets from the node that transmits the data packets to the node to which the packets are to be delivered. The user data of the packet is data that is to be delivered from one node to another. The user data is also called a payload. A data packet may include headers of one or more communication protocols. When a data packet includes more than one header, the outer-most header is followed by other headers. The innermost header is usually the last header of the headers before the payload of the packet. For the network that implements the protocol of the outer-most header of a packet, the network considers only the outer-most header of the packet as the header and the other headers and the user data of the packet are together

considered as the payload of the packet. To simplify the discussion, the packet 230 is shown with only two header values, namely Ethernet and IP destination addresses. However, a typical packet contains more header values.

[0049] In the first stage 245, the switching element 205 receives the packet 230. The second stage 250 shows that datapath manager 220 looks for a matching flow that is stored in the datapath cache 225. The datapath cache 225 provides a fast path to process incoming packet because it does not involve any translation by the datapath flow generator 210. In the second stage 250, there are no flow entries stored in the datapath cache 225. Accordingly, the packet processing is transferred to the datapath flow generator 210. In some embodiments, the transferring entails sending the packet 230 to the datapath flow generator 210.

[0050] The third stage 255 illustrates the datapath flow generator 210 dynamically generating a flow entry 240 with at least one wildcard field. In wildcarding, the datapath flow generator of some embodiments initially wildcards some or all match fields. When a packet is received, the datapath flow generator 210 selects a flow from the flow table and un-wildcards each match field that it consults or examines. This can include comparing match field and header values, and deriving or learning output ports.

[0051] In the example of the third stage 255, the datapath flow generator 210 has specified some or all of the match fields to be wildcard fields. That is, some or all of the match fields has been initialized to be wildcard fields, including destination Ethernet address field and IP destination address field. The datapath flow generator 210 has also selected the flow entry 235 from the flow table 215 to determine if it is a match for the packet 230. In making this determination, the datapath flow generator 210 has compared the ingress port of the flow entry 235 with the ingress port at which the packet 230 was received. The datapath flow generator 210 has also compared the destination Ethernet addresses match field against the corresponding header field. As the destination Ethernet address match field was consulted, the datapath flow generator has also specified the destination Ethernet address as a non-wildcard field. However, the destination IP match field remains a wildcard field. This is conceptually shown by the asterisk symbol in the cache flow entry 240.

[0052] In the third stage 255, the datapath flow generator 210 has generated the cache entry 240 based on the matching flow 235. The fourth stage 260 illustrates that the flow entry 240 has been pushed down and stored in the datapath cache 225. The cached entry 240 includes

an ingress port value, Ethernet destination address value, IP destination address value, and an action value. Although the IP address is shown, that field has been specified as a wildcard field. Here, the slash mark followed by the asterisk symbol indicates that the entire field has completely been wildcarded. This field has been wildcarded because it was not consulted when generating the flow entry 240.

[0053] As shown in the fourth stage 260, the cached flow entry 240 is also different from the flow 235 from the flow table 215. Aside from the wildcard, in some embodiments, the cache flow entry does not have a priority value. This is because the datapath cache 240 does not store multiple flows that can match a same packet. In other words, the datapath cache stores only one flow that can match a given packet, instead of multiple entries. Therefore, there are no resubmit operations with the datapath cache, in some such embodiments. Alternatively, the flow entry 240 of some embodiments is associated with a priority value. In addition, the datapath manager 220 may perform one or more resubmit operations operation to find any other matching flows from the datapath cache.

[0054] The fifth stage 260 illustrates the switching element 205 processing the packet 230. The cache entry 240 specifies that any packet that is received at ingress port one and has an Ethernet destination address "A" should be output to port two. The IP destination address match field has been wildcarded so that field can have any different value. Following this rule, the datapath manager 220 outputs the packet 230 through port two.

[0055] In the preceding example, the switching element 205 dynamically generates a flow 240 with at least one wild card field. As mentioned above, this flow is also referred to as a "megaflow" because it allows packets that have different wildcard values to be quickly processed. This caching technique collapses similar into flows into one userspace megaflow that makes many flow misses a cache lookup instead of a full translation.

[0056] **Figure 3** shows an example how the switching element 205 uses the flow entry 240 to process subsequent packets 305 and 310. Specifically, this figure illustrates the switching element 205 at time one when it receives one subsequent packet 305 and at time two when it receives another subsequent packet 310. At time one, the switching element 205 receives the packet 305. The datapath manager 220 receives the packet and parses it to extract or strip its header values. The datapath manager 220 also identifies the ingress port through which the packet 305 was received. The datapath manger 220 selects the flow entry 240 from the datapath

cache 225 and compares the identified ingress port value with the value from the flow entry. As the port values match, the datapath manager compares the non-wildcard match field value (i.e., destination Ethernet address field) to the corresponding header field value. As those two values match, the datapath manager performs the associated action, which is to output the packet to port two. The datapath manager 220 ignores the IP destination match field from the flow entry 240 because it has been wildcarded. In this case, even if the IP destination address match field has not been wildcarded, the packet would have been output to port two. This is because the IP destination address values from the flow entry and the packet header match one another.

[0057] At time two, the switching element 205 receives another subsequent packet 310. Similar to time one, the datapath manager 220 receives the packet and parses it to extract or strip its header values. The datapath manager also identifies the ingress port through which the packet was received. The datapath manager selects the cache entry 240 and compares the ingress port and the non-wildcard match field (i.e., destination Ethernet address field) to the header fields. As those two values match, the datapath manager performs the associated action, which is to output the packet to port two.

[0058] At time two, the destination IP address fields of the flow entry 240 and the packet's header do not match. Therefore, there would have been a miss in the datapath cache 225, and the packet processing would have to be shifted to the datapath flow generator 210. As stated above, the process of determining what to do with a flow when it is missed in the cache can be an expensive operation. The datapath flow generator 205 must perform a number of different tasks to generate a cache flow entry, such as iterating through flows in the flow table 215 to find a matching flow and/or deriving match field values. At time two, those tasks do not have to be performed. This is because the destination IP address match field has been wildcarded and the packet 310 is processed regardless of the mismatch in the field values. Accordingly, by caching the flow with the wildcard field, the switching element avoids having to do another translation to cache another flow. Such generation of wildcard flows can provide significant flow setup performance, especially when the switching element is able to wildcard many flows.

[0059] Several more detailed examples of dynamically generating and using wildcard flows. Specifically, Section I describes several an example software-switching element that implements some embodiments of the invention. Section II then describes a more detailed

example of how the switching element processes packets. This is followed by Section III that describes various examples of generating flows with zero or more wildcard fields. Lastly, Section IV describes an electronic system for implementing some embodiments of the invention.

I. Example Switching Element

[0060] As method above, the switching element of some embodiments receives a packet and processes the packet by dynamically generating a flow entry with a set of wildcard fields. The switching element then stores that flow entry in a cache and processes any subsequent packets that have header values that match the flow entry's non-wildcard match fields. In some embodiments, the switching element is a software or virtual switch. An example of such a software switch will not be described by reference to **Figure 4**.

[0061] **Figure 4** conceptually illustrates an architectural diagram of a software-switching element 405 of some embodiments that is implemented in a host 400. In this example, the software-switching element 405 operates on a virtual machine (VM) 495 and includes several components. These components includes an Open vSwitch (OVS) kernel module 420, which runs in the kernel of the VM 455, and an OVS daemon 440 and an OVS database server 445, which run in the userspace 450 of the VM 495.

[0062] As shown in **Figure 4**, the host 400 includes hardware 470, hypervisor 465, and VMs 402 and 404. The hardware 470 may include typical computer hardware, such as processing units, volatile memory (e.g., random access memory (RAM)), nonvolatile memory (e.g., hard disc drives, optical discs, etc.), network adapters, video adapters, or any other type of computer hardware. As shown, the hardware 470 includes NICs 485 and 490, which are typical network interface controllers for connecting a computing device to a network.

[0063] The hypervisor 465 is a software abstraction layer that runs on top of the hardware 470 and runs below any operation system. The hypervisor 465 handles various management tasks, such as memory management, processor scheduling, or any other operations for controlling the execution of the VMs 402 and 404. Moreover, the hypervisor 465 communicates with the VM 495 to achieve various operations (e.g., setting priorities). In some embodiments, the hypervisor 465 is one type of hypervisor (Xen or KVM hypervisor) while, in other embodiments, the hypervisor 465 may be any other type of hypervisor for providing hardware virtualization of the hardware 470 on the host 400.

[0064] As shown, the hypervisor 465 includes device drivers 475 and 480 for the NICs 485 and 490, respectively. The device drivers 475 and 480 allow an operating system to interact with the hardware of the host 400. VMs 402 and 404 are virtual machines running on the hypervisor 465. As such, the VMs 402 and 404 run any number of different operating systems. Examples of such operations systems include Linux, Solaris, FreeBSD, or any other type of UNIX based operating system. Other examples include Windows based operating systems as well.

[0065] In some embodiments, the VM 495 is a unique virtual machine, which includes a modified Linux kernel, running on the hypervisor 465. In such cases, the VM 495 may be referred to as domain 0 or dom0 in some embodiments. The VM 495 of such embodiments is responsible for managing and controlling other VMs running on the hypervisor 465 (e.g., VMs 490 and 495). For instance, the VM 495 may have special rights to access the hardware 470 of the host 400. In such embodiments, other VMs running on the hypervisor 465 interact with the VM 495 in order to access the hardware 470. In addition, the VM 495 may be responsible for starting and stopping VMs on the hypervisor 465. The VM 495 may perform other functions for managing and controlling the VMs running on the hypervisor 465. Some embodiments of the VM 495 may include several daemons (e.g., Linux daemons) for supporting the management and control of other VMs running on the hypervisor 465. Since the VM 495 of some embodiments is manages and controls other VMs running on the hypervisor 465, the VM 495 may be required to run on the hypervisor 465 before any other VM is run on the hypervisor 465.

[0066] As shown in **Figure 4**, the VM 495 includes a kernel 455 and a userspace 450. In some embodiments, the kernel is the most basic component of an operating system that runs on a separate memory space and is responsible for managing system resources (e.g., communication between hardware and software resources). In contrast, the userspace is a memory space where all user mode applications may run.

[0067] As shown, the userspace 450 of the VM 495 includes the OVS daemon 440 and the OVS database server 445. Other applications (not shown) may be included in the userspace of the VM 495 as well. The OVS daemon 440 is an application that runs in the background of the userspace of the VM 495. The OVS daemon 440 of some embodiments receives switch configuration from the network controller 406 (in a network controller cluster) and the OVS database server 445. The management information includes bridge information, and the switch

configuration includes various flows. These flows are stored in the flow table 415. Accordingly, the software-switching element 405 may be referred to as a managed forwarding element.

[0068] In some embodiments, the OVS daemon 440 communicates with the network controller using OpenFlow Protocol. In some embodiments, the OVS database server 445 communicates with the network controller 406 and the OVS daemon 440 through a database communication protocol (e.g., OVS database protocol). The database protocol of some embodiments is a JavaScript Object Notation (JSON) remote procedure call (RPC) based protocol.

[0069] The OVS database server 445 is also an application that runs in the background of the userspace of the VM 495. The OVS database server 445 of some embodiments communicates with the network controller 406 in order to configure the OVS switching element (e.g., the OVS daemon 440 and/or the OVS kernel module 420). For instance, the OVS database server 445 receives management information from the network controller 406 for configuring bridges, ingress ports, egress ports, QoS configurations for ports, etc., and stores the information in a set of databases.

[0070] As illustrated in **Figure 4**, the kernel 455 includes the OVS kernel module 420. This module processes and routes network data (e.g., packets) between VMs running on the host 400 and network hosts external to the host (i.e., network data received through the NICs 485 and 490). For example, the OVS kernel module 420 of some embodiments routes packets between VMs running on the host 400 and network hosts external to the host 400 couple the OVS kernel module 420 through a bridge 408.

[0071] In some embodiments, the bridge 408 manages a set of rules (e.g., flow entries) that specify operations for processing and forwarding packets. The bridge 408 communicates with the OVS daemon 440 in order to process and forward packets that the bridge 408 receives. For instance, the bridge 408 receives commands, from the network controller 406 via the OVS daemon 445, related to processing and forwarding of packets.

[0072] In the example of **Figure 4**, the bridge 408 includes a packet processor 430, a classifier 460, and an action processor 435. The packet processor 430 receives a packet and parses the packet to strip header values. The packet processor 430 can perform a number of different operations. For instance, in some embodiments, the packet processor 430 is a network stack that is associated with various network layers to differently process different types of data

that it receives. Irrespective of all the different operations that it can perform, the packet processor 430 passes the header values to the classifier 460.

[0073] The classifier 460 accesses the datapath cache 425 to find matching flows for different packets. The datapath cache 425 contains any recently used flows. The flows may be fully specified, or may contain one or more match fields that are wildcarded. When the classifier 460 receives the header values, it tries to find a flow or rule installed in the datapath cache 425. If it does not find one, then the control is shifted to the OVS Daemon 440. One main distinction between the fast path cache 425 and the set of flow tables 415 is that there is at most only one matching flow entry in the fast path cache 425.

[0074] If the classifier 460 finds a matching flow, the action processor 435 receives the packet and performs a set of action that is associated with the matching flow. The action processor 435 of some embodiment also receives, from the OVS daemon 440, a packet and a set of instructions to perform on the packet. For instance, when there is no matching flow in the datapath cache 425, the packet is sent to the OVS daemon 440. The OVS daemon 440 may generate a flow and install that flow in the datapath cache 425. The OVS daemon 440 might also send the packet to the action processor 435 with the set of actions to perform on that packet.

[0075] The OVS daemon 440 of some embodiments includes a datapath flow generator. The datapath flow generator 440 is a component of the software switching element 405 that makes switching decisions. Each time there is a miss in the datapath cache 425, the datapath flow generator 440 generates a new flow to install in the cache. In some embodiments, the datapath flow generator works in conjunction with its own separate classifier (not shown) to find one or more matching flows from a set of one or more flow table 415. However, different from the classifier 460, the OVS daemon's classifier can perform one or more resubmits. That is, a packet can go through the daemon's classifier multiple times to find several matching flows from one or more flow table 415. When multiple matching flows are found, the datapath flow generator 410 of some embodiments generates one consolidated flow entry to store in the datapath cache 425. In some embodiments, the switching element allows flows with wildcards to be specified in the flow table 415. However, different from the datapath flows, these flows are not dynamically generated on the fly (e.g., in response to a miss in the datapath).

[0076] One of ordinary skill in the art would understand that the architecture is an example architecture and that different embodiments can include different sets of components. The naming of the various components is arbitrary and can change from one implementation to another. Also, the architecture shows two different layers (e.g., the kernel layer and the userspace layer) performing various operations. In some embodiments, these operations occur at just one layer (e.g., at the userspace layer) or are further split into other layers.

II. Packet Processing Operations

[0077] The previous section described an example architecture of a switching element that generates flows with zero or more wildcard fields. The architecture showed various components, including a kernel module and an OVS daemon. **Figure 5** conceptually illustrates several processes 500A-D, which show the interactions between such components to generate and cache a flow with zero or more wildcard fields. The processes 500A-D will be described by reference to **Figures 6-9**.

[0078] The process 500A shows example operations performed by a kernel module when it receives a packet. The process 500A begins when it receives (at 502) a packet. The process 500A then performs a lookup operation on the cache to identify a matching flow entry for the packet. Specifically, the process 500A iteratively selects (at 504) a flow entry that is cached to find one flow entry that matches each of the entry's non-wildcard fields. If a matching entry is found, the process 500A selects (at 508) the flow entry. The process 500A then performs (at 510) a set of actions that is specified by that flow entry. If no matching entry is found, the process 500A proceeds to 506, which is described below. In some embodiments, the packet may be sent the OVS daemon (e.g., the userspace) even if there is a match in the kernel. This is because some packets are too complicated for the kernel to handle. Thus, in some embodiments, a "userspace" action is installed in the datapath cache (e.g., the kernel flow table), which specifies pushing all packets to the OVS daemon.

[0079] **Figure 6** illustrates an example of a switching element 405 that finds a match in a datapath cache 425. Two operational stages 605 and 610 of the switching element's kernel module 420 are shown in this figure. The bridge 408 has been configured, and two flow entries 620 and 625 are stored in the datapath cache 425. In this example, each of the flow entries 620 and 625 has at least one wildcard match field. These entries might have been pushed down earlier by the OVS daemon (not shown) based on two previously received packets.

[0080] The first stage 605 illustrates the kernel module 420 receiving a packet 615. In particular, the packet 615 is received by the packet processor 430 through the bridge 408. The packet includes a number of header fields. To simplify the discussion, the header fields are specified as field 1 through field N. The header field 1 has a value of "A", field 2 has a value of "B", and field N has a value of "C".

[0081] The second stage 610 illustrates an example of processing the packet after finding a matching flow in the datapath cache 425. In processing the packet, the packet processor 430 first strips the headers off the packet 615. The classifier 460 then selects the first flow entry 620 and compares its non-wildcard match field values against the corresponding header values. Here, the first flow entry 620 is a match for the packet 615. Accordingly, the action processor 435 performs the flow entry's associated set of actions on the packet 615, which is to output the packet to output port three.

[0082] Referring to **Figure 5**, when there is no matching flow, the process 500A sends (at 506) the packet (e.g., the header values) to the OVS daemon for processing. **Figure 7** illustrates an example of transferring control to the OVS daemon 440 when there is a miss in the datapath cache 425. Two operational stages 705 and 710 are illustrated in this figure. The first stage 705 illustrates the kernel module 420 receiving a packet 715. In particular, the packet 715 is received by the packet processor 430 through the bridge 408. The packet includes a number of header fields. The header field 1 has a value of "E", field 2 has a value of "F", and field N has a value of "G".

[0083] The second stage 710 illustrates an example of how control is shifted to the OVS daemon when there is a miss in the datapath cache 425. In particular, the packet processor 430 first parses the packet to strip the headers from the packet 715. The classifier 460 then selects the first flow entry 620 and compares its non-wildcard match field values against the corresponding header values. The non-wildcard fields of the first flow entry 620 do not match the corresponding header values. Accordingly, the classifier 460 selects the second flow entry 625 and compares its non-wildcard match field values against the corresponding header values. The second flow entry 625 is also not a match for the packet 715. As there is no matching entry, the classifier 460 sends the packet to the OVS daemon 440.

[0084] Referring to **Figure 5**, Process 500B shows several example operations that can occur at the OVS daemon when it receives a packet from the kernel module. As shown, the

process 500B begins when it receives (at 512) the packet from the kernel module. The process then dynamically generates (at 514) a new flow entry with zero or more wildcard fields.

[0085] **Figure 8** illustrates an example of the OVS daemon 440 generating a flow and sending it to the kernel module 420. Two operational stages 805 and 810 are illustrated in this figure. These stages 805 and 810 are a continuation of the stages 705 and 710 shown in **Figure 7**. As shown, the OVS daemon 440 includes a datapath flow generator 410 to dynamically generate a new flow to store in the datapath cache 425. Similar to the kernel module 420, the datapath flow generator 410 is associated with a classifier 720. This classifier 720 is used to find a matching flow from one or more flow tables 415. Different from the classifier 460, the OVS daemon's classifier 415 can perform one or more resubmits. That is, a packet can go through the daemon's classifier multiple times to find several matching flows from one or more flow tables (e.g., the flow table 415). For instance, even if a matching flow is found, the flow's associated action may specify a resubmit operation. The resubmit operation re-searches the same flow table (or another specified flow table) and creates a flow entry that specifies additional actions found, if any, in addition to any other actions in the original matching flow. In some embodiments, the datapath flow generator specifies performing only a set of actions associated with the flow from the flow table with the highest priority value.

[0086] The first stage 805 illustrates the OVS daemon 440 after it has received the packet 715 from the kernel module 420. This stage also shows the OVS daemon 440 sending a new flow 820 to the kernel module 420. The datapath flow generator 410 has generated the new flow 820. In generating the flow, the datapath flow generator 410 of some embodiments initially wildcards some or all match fields. Upon the datapath flow generator 410 receiving the packet, it calls the classifier 410 to iterate through flows in the flow table 415. The classifier 410 selects a flow entry from the flow table and un-wildcards each match field that was compared against the packet's header field.

[0087] Alternatively, or conjunctively with such matching, the data flow generator 440 of some embodiments generates a flow entry by deriving or learning output ports. In deriving, the data flow generator 440 may consult one or more match field values, un-wildcard the match fields, and specify those match field values as non-wildcard field values in the flow entry. The data flow generator 440 of some embodiments generates a flow entry by communicating with one or more different forwarding decision modules 725, such as a MAC learning module. This

MAC learning module may learn MAC addresses in a typical manner that layer 2 switches learn MAC addresses. For instance, when a MAC address (i.e., a destination MAC address of a packet is not included in a set of tables of learned MAC addresses), the MAC learning module may flood all of the ports of the bridge 408 and record the MAC address of the packet that responds to the flood.

[0088] Referring to **Figure 5**, after generating the flow entry, the process 500B sends (at 516) instructions to the kernel module to cache the flow entry. The process then sends (at 518) the packet to the kernel module with instructions to perform a set of actions on the packet. The process 500B then ends.

[0089] The process 500C shows operations performed by the kernel module after the OVS daemon has generated a new flow entry and sent instructions to install the new flow entry in the cache. As shown, the process 500C begins when it receives (at 520) the instructions from the OVS daemon to cache the new flow entry. The process 500C then caches (at 522) the flow entry. The process 500C then ends.

[0090] The process 500D shows operations performed by the kernel module after the OVS daemon has generated a new flow entry and sent the packet to the kernel module with instructions to perform a set of actions on the packet. As shown, the process 500D begins when it receives (at 524) the packet with instructions to perform a set of actions on the packet. The process 500D then performs (at 526) the set of action on the packet. The process 500D then ends.

[0091] The second stage 810 of **Figure 8** illustrates the kernel module 420 after receiving the flow entry 820. The classifier 460 has received the flow entry 820 through the bridge 408. The classifier 460 has installed the flow entry 820 in the datapath cache 425. To quickly process similar packets without causing a miss in the datapath cache 425, the third stages 815 illustrates that the flow entry 820 includes a number of wildcard match fields. The packet is then received at the action processor from the OVS daemon. The packet is received with instructions to perform a set of actions. The set of actions may be the same as the one associated with the cached flow entry 820. In the example of the second stage 805, the action processor 435 performs the flow entry's associated action on the packet 715, which is to output the packet to output port five.

III. Dynamically Generating Flows with Wildcards

[0092] As mentioned above, the switching element of some embodiments dynamically generates flows with wildcards. In generating, the switching element initially wildcards some of all of match fields and generates a new flow entry by un-wildcarding each match field that was consulted to generate the flow entry. The switching element of some embodiments generates a flow by un-wildcarding each match field that was compared against a header value. Several such examples will now be described below by reference to **Figures 9-17**.

A. Examples of Generating Flows

[0093] **Figure 9** shows an example of un-wildcarding a match field because it was compared against a header value. Four operational stages 905-920 of the switching element 405 are shown in the figure. The switching element 405 includes the datapath flow generator 410, which was described above by reference to **Figure 4**.

[0094] Stage 905 begins when there is a miss in the datapath cache. The packet processing is shifted from the kernel module to the OVS daemon 440. In particular, the kernel module sends the packet 945 to the OVS daemon 440. The daemon's datapath flow generator 410 receives the packet 410 and generates a new flow entry to store in the datapath cache.

[0095] The first stage 905 illustrates the datapath flow generator 410 receiving the packet 945. The packet 945 has the following header field values: Ethernet source value of "A", Ethernet destination value of "B", and field N value of "C". To find a matching flow, the datapath flow generator 410 selects a first flow or rule 930 from the flow table 415. If there are multiple flows, the flow may be sorted by priority values (e.g., from highest to lowest).

[0096] The second stage 910 illustrates that the datapath flow generator 410 initializing a group of match field values as wildcards. The datapath flow generator 410 of some embodiments generates a flow entry 940 by keeping track of the data value of each match field, and a mask associated with that field. If a field is masked, that match field value was not consulted (e.g., compared against a corresponding header field value) to generate the flow entry. As such, a masked field represents a wildcard field. In some embodiments, the entire match field may be masked or wildcarded. Alternatively or conjunctively, the switching element 405 of some embodiments allows masking or wildcarding at the sub-value level. In other words, the switching element supports masking portion of the match field (e.g., a portion of the IP address field) rather than the entire match field value. In the example of **Figure 9**, a mask value of zero

indicates that the match field has been completely wildcarded, and a mask value of one indicates that the match field was consulted.

[0097] The third stage 915 illustrates the datapath flow generator 410 generating the cache flow entry 940 based on the selected flow 930 and the packet header values. Specifically, the datapath flow generator 410 has selected the flow 930 and compared the flow's Ethernet destination value with the corresponding header value. As the Ethernet destination match field was compared against a header field value, the datapath flow generator 410 has unmasked the match field. This is shown in the third stage 915 with the zero value being replaced by the one value in the cache flow entry 940, namely from the mask of the Ethernet destination address match field.

[0098] The third stage 915 also illustrates that the datapath flow generator 410 has found a matching flow for the packet. This is because the flow's only match field matches the corresponding header value. The fourth stage 920 shows the OVS daemon 440 sending the cache flow entry 940 to the kernel module. Specifically, the datapath flow generator 410 has associated the action from the flow 930 with the cache flow entry 940. The cache flow entry 940 reads that Ethernet source address has a value of "A", Ethernet destination address has a value of "B", and field N has a value of "C". Although the Ethernet source address and Field N are associated with values, the zero value after the slash mark indicates that each of these match fields is completely masked or wildcarded. In addition, the cache flow entry 940 is associated with an action, which is to drop any packet that has an Ethernet destination address value of "B".

[0099] The previous example illustrated the OVS daemon 440 finding a match with a first flow 930 from the flow table 415. **Figure 10** illustrates an example of iterating through multiple flows to find a match. Specifically, this figure illustrates that a wildcard match field is specified to be a non-wildcard field when it is compared against a corresponding header value, regardless of whether there was a match or a mismatch. This figure is similar to the previous figure, except that the flow 930 is the second flow in the flow table 415.

[00100] Four operational stages 1005-1020 of the switching element 405 are shown in **Figure 10**. The first stage 1005 illustrates the datapath flow generator 410 receiving the packet 945. The packet has the following header field values, Ethernet source value of "A", Ethernet destination value of "B", and field N value of "C". To find a matching flow, the datapath flow

generator 410 selects a first flow or rule 1030 from the flow table 415. The flow 1030 has instructions to output every packet with an Ethernet source value of "D" to output port two.

[00101] The second stage 1010 illustrates the datapath flow generator 410 building the cache flow entry 1025 based on the selected flow 1030 and the packet header values. The field values are extracted from the packet header and each value is associated with a mask that identifies whether the value is associated with a wildcard match field.

[00102] In the example of the second stage 1010, the datapath flow generator has selected the flow 1030 with the highest priority value and compared the flow's Ethernet source value with the corresponding header value. As the Ethernet source match field was compared against a header field value, the datapath flow generator 410 has unmasked the match field. This is shown in the second stage 1010 with the zero value being replaced with the one value in the cache flow entry 1025, namely from the mask of the Ethernet source address match field.

[00103] In the second stage 1010, the flow 1030 was not match for the packet 945 because their Ethernet sources address values did not match. Accordingly, in the third stage 1015, the datapath flow generator has selected the flow 930 with the next highest priority value. Specifically, the datapath flow generator 410 has selected the flow 930 and compared the flow's Ethernet destination value with the corresponding header value. As the Ethernet destination match field was compared against a header field value, the datapath flow generator 410 has unmasked the match field. This is shown in the third stage 1015 with the zero value being replaced with the one value in the cache flow entry 1025, namely from the mask of the Ethernet destination address match field.

[00104] The third stage 1015 also illustrates that the datapath flow generator 410 has found a matching flow for the packet. This is because the second flow's only match field matches the corresponding header value. The fourth stage 1020 shows the OVS daemon 440 sending the cache flow entry 1025 to the kernel module. Specifically, the datapath flow generator 410 has associated the action from the flow 930 with the cache flow entry 1025. The cache flow entry 1025 reads that Ethernet source address has a value of "A", Ethernet destination address has a value of "B", field N has a value of "C". Although Field N is associated with a value, the zero value after the slash mark indicates that the match field is completely masked or wildcarded. In addition, the cache flow entry 1025 is associated with an action, which is to drop any packet that has an Ethernet destination address value of "B".

[00105] As mentioned above, even if a matching flow is found in a flow table, one of the flow's associated actions may specify a resubmit operation to find another flow from the flow table. **Figure 11** provides an illustrative example of generating a flow entry based on multiple matching flows from a flow table. Four operational stages 1105-1120 are illustrated in this figure. These stages 1105-1120 are similar to the previous figure, except that the first flow from the flow table is a match for packet. The first flow is associated with a resubmit action.

[00106] The first stage 1105 illustrates the datapath flow generator 410 receiving the packet 945. The second stage 1110 illustrates the datapath flow generator 410 building the cache flow entry 1125 based on the selected flow 1030 and the packet header values. Specifically, the datapath flow generator has selected the flow 1130 (e.g., with the highest priority value) from table zero and compared the flow's Ethernet source value with the corresponding header value. As the Ethernet source match field was compared against a header field value, the datapath flow generator 410 has unmasked the match field. This is shown in the second stage 1110 with the zero value being replaced with the one value in the cache flow entry 1125, namely from the mask of the Ethernet source address match field.

[00107] In the second stage 1110, the flow 1030 was a match for the packet 945 because their Ethernet sources address values matches one another. However, the flow 1130 is associated with a resubmit operation (e.g., to concatenate multiple flows into one datapath flow). The resubmit operation specifies resubmitting into another flow table (i.e., flow table two). Accordingly, in the third stage 1115, the datapath flow generator 410 has selected the flow 1135 (e.g., with the highest priority value) from flow table two. Specifically, the datapath flow generator 410 has selected the flow 1135 and compared the flow's Ethernet destination value with the corresponding header value. As the Ethernet destination match field was compared against a header field value, the datapath flow generator 410 has unmasked the match field. This is shown in the third stage 1115 with the zero value being replaced with the one value in the cache flow entry 1125, namely from the mask of the Ethernet destination address match field.

[00108] The third stage 1115 also illustrates that the datapath flow generator 410 has found a matching flow for the packet. This is because the only match field of the flow 1135 matches the corresponding header value. In this example, all match fields that were specified as wildcards or non-wildcards carries over when there is a resubmit operation. That is, the OVS

daemon does not reinitialize all the match fields as wildcards when there is a resubmit. However, the OVS daemon might reinitialize them as wildcards, in some other embodiments.

[00109] The fourth stage 1120 shows the OVS daemon 440 sending the cache flow entry 1125 to the kernel module. Specifically, the datapath flow generator 410 has associated the action from the flow 1135 with the cache flow entry 1125. The cache flow entry 1125 reads that Ethernet source address has a value of "A", Ethernet destination address has a value of "B", and field N has a value of "C". Although Field N is associated with a value, the zero value after the slash mark indicates that the match field is completely masked or wildcarded. In addition, the cache flow entry 1125 is associated with an action, which is to drop any packet that has an Ethernet destination address value of "B".

[00110] In the example described above, the OVS daemon finds a matching flow that has a resubmit action. The resubmit action specifies performing a resubmit to another flow table. In some embodiments, the resubmit action can specify a resubmit operation on the same flow table. One example way of resubmitting the packet to the same flow table is to modify the packet in some way before the resubmission. For instance, the action of the initial matching flow could specify changing a matching header value (e.g., the Ethernet source value). This is because if the matching header value remains the same, the same initial flow will once again match the packet.

[00111] In some cases, a flow in a flow table may have a value for a match field value that requires the datapath flow generator 410 to examine one or more other match fields. For example, when a match field relates to an IP address or an Ethernet address, the datapath flow generator 410 might first consult the Ethertype match field and determine if the corresponding header value matches the match field value. The datapath flow generator may then un-wildcard the Ethertype match field.

[00112] **Figure 12** illustrates an example of the switching element 405 un-wildcarding a match field that is not explicitly specified in a flow 1240. Five operational stages 1205-1225 of the switching element 405 are shown in this figure. The first stage 1205 illustrates the datapath flow generator 410 receiving a packet 1235. The packet has the following header field values, Ethertype of "0x0800", IP sources address value of "1.1.1.1", and IP destination address value of "1.1.1.2". To find a matching flow, the datapath flow generator 410 selects a first flow or rule

1240 from the flow table 415. The flow 1230 has a rule that states that any packet with an IP source address of "1.1.1.1" should be output to port two.

[00113] The second stage 1210 illustrates the first comparison was made for a match field that is not explicitly specified in the flow 1240. The first comparison was made to a related field that indicates which protocol is encapsulated in the payload of the packet or frame. The header value "0x0800" indicates that the Ethertype is for Internet Protocol version 4 (IPv4). The comparison of the Ethertype match field provides the datapath flow generator 410 with a quick feedback on whether to compare the match field of the flow 1240 to the corresponding header value. In this case, as the Ethertype value relates to the IP source or destination address, the datapath flow generator 410 unmask the Ethertype match field and proceeds to stage three 1215. In cases where the Ethertype values does not relate to IP source or destination address, the datapath flow generator 410 may unmask the Ethertype match field and select another flow to perform the matching.

[00114] The third stage 1215 illustrates comparing a match field associated with the flow 1240 to the corresponding header value. Specifically, the datapath flow generator 410 has selected the flow 1240 and compared the flow's IP source address value with the corresponding header value. As the IP source address match field was compared against a header field value, the datapath flow generator 410 has unmasked the match field. This is shown in the third stage 1215 with the zero being replaced by a one in the cache flow entry 1235, namely from the mask of the IP source address match field.

[00115] The third stage 1215 also illustrates that the datapath flow generator 410 has found a matching flow for the packet 1230. This is because the flow's only match field matches the corresponding header value. The fifth stage 1225 shows the OVS daemon 440 sending the cache flow entry 1235 to the kernel module. Specifically, the datapath flow generator 410 has associated the action from the flow 1240 with the cache flow entry 1235. The cache flow entry 1235 reads that any packet having the EtherType value of "0x0800" and IP source address value of "1.1.1.1" should be output to port two. The cache flow entry 1235 includes a value for the IP destination address match field. However, the zero value after the slash mark indicates that the IP destination address match field is completely masked or wildcarded.

[00116] In several of the examples described above, the switching element un-wildcards each match field that was compared against a header value to find a matching flow.

Alternatively, or conjunctively with such comparison, the switching element of some embodiments un-wildcards match fields after it finds the matching flow. For example, an action associated with the flow may specify consulting one or more header field values. When those field values are consulted, the switching element of some embodiments un-wildcards the corresponding match fields.

[00117] **Figure 13** illustrates an example of the switching element 405 that examines one or more match fields based on an action of a match rule. In this example, the flow table includes a rule 1325 that specifies doing normal L2/L3 processing for all packets. Four operational stages 1305-1320 of the switching element 405 are shown in this figure. This example assumes that the switching element 405 is performing a bonding operation that allows more than one interfaces for a port.

[00118] The first stage 1305 illustrates the datapath flow generator 410 receiving a packet 1330. The packet has the following header field values, Ethernet type of "0x0800", Ethernet source of "Foo", Ethernet destination of "Bar", IP source address value of "1.1.1.1", IP destination address value of "1.1.1.2", and a field N value of "A". To find a matching flow, the datapath flow generator 410 selects a first flow or rule 1325 from the flow table 415.

[00119] The second stage 1315 illustrates EtherType value being consulted for the cache flow entry 1335. Here, the datapath flow generator 410 has examined the EtherType value and unmasked the same field. The third stage 1315 illustrates IP source address and destination address values being consulted for the cache flow entry 1335. Here, the IP address values are derived from the corresponding packet header values. The third stage 1315 also shows that that two IP source and destination match fields are specified to be non-wildcard fields. This is because the switching element has been configured to do normal L2/L3 processing by matching at least these two field values against incoming packets.

[00120] The fourth stage 1320 illustrates Ethernet source address and destination address being consulted to the cache flow entry 1335. Here, the Ethernet address values are derived from the corresponding packet header values. The fourth stage 1320 also shows that that the two match fields are specified to be non-wildcard fields as the corresponding masks are removed from the cache flow entry 1335. This is because the switching element has been configured to do normal L2/L3 processing by matching at least these two additional field values against incoming packets.

[00121] In the fourth stage 1320, the switching element 405 has associated an action to the cache flow entry 1335. In some embodiments, the switching element 405 assigns the action based on results of a learning algorithm. As an example, the datapath flow generator might have chosen an output port based on the results of the MAC learning algorithm. In some embodiments, the OVS daemon includes a MAC learning module. The MAC learning module of some embodiments identifies, for a given packet, one or more ports to output the packet based on the packet's header field values.

[00122] **Figure 14** provides an illustrative example of how the switching element 405 utilizes a number of different components to generate the flow entry 1335. Three operational stages 1405-1415 of the switching element 405 are shown in this figure. As shown, the datapath flow generator 410 operates in conjunction with a number of modules to dynamically generate a flow entry to cache in a datapath cache. These modules include the classifier 720, a bonding module 1420, and a MAC learning module 1425. The cache flow 410 generator may operate in conjunction with a set of other modules 1430 to match and/or derive field values.

[00123] The first stage 1305 illustrates the OVS daemon 440 receiving the packet 1330. The packet is passed to the classifier 720 through the datapath flow generator 410. As shown in the second stage 1310, the classifier 720 derives various field values relating to the packet. This includes (1) the ingress port through which the packet was received, (2) the EtherType value, (3) the source IP address, and (4) the destination IP address. The second stage 1410 also shows that the packet processing operation is then shifted to the bonding module 1420.

[00124] The third stage 1415 illustrates the bonding module 1420 calling the MAC learning module to associate the cache flow entry with MAC address values. In some embodiments, the MAC learning module 1430 of some embodiments identifies, for a given packet, one or more ports to output the packet based on the packet's header field values. As shown in the third stage 1415, the bonding module 1420 adds the Ethernet source and destination addresses returned by the MAC learning module 1430 to the cache flow entry.

[00125] **Figure 15** illustrates an example of how the datapath flow generator 410 generates a flow by interfacing with different components. Here, the components include the classifier 720, the bonding module 1420, the MAC learning module 1425, Bidirectional Forwarding Detection (BFD) module 1505, and learning action module 1510. As mentioned above, the MAC learning module 1425 identifies, for a given packet, one or more ports to

output the packet based on the packet's header field values (e.g., the destination Ethernet field value).

[00126] The classifier 720 of some embodiments is called to match certain match fields. The bonding module 1420 is called to perform bonding operation that allows more than one interfaces for a port. The BFD module 1505 is used to detect whether a packet is a BFD packet. For example, if a packet comes in, the BFD module 1505 may be called to determine whether the packet is a BFD packet or not. This may cause a change in the masking or wildearding of match fields relating to BFD. If the packet is a BFD, the BFD module 1505 may consume the packet and generation of the cache flow will terminate at that point. Different from the BFD module 1505, the learn action module 1510 installs a rule in the classifier, which can affect traffic. The learn action module 1510 can be used to more abstractly learn MAC addresses. For example, if a packet with a particular Ethernet source address comes in through port 1, the learn action module 1510 can be used to install a rule that specifies that any packet whose destination MAC address field has the same address should be output to port 1.

[00127] The datapath flow generator 410 may call any one or more of these modules. One or more of these modules may call another module. Each one of the different modules may be called an arbitrary number of times when generating the flow to store in the cache. As an example, the data flow generator may call the classifier and receive a result; however, depending on the flow, it can call other modules such as the MAC learning module. One of ordinary skilled in the art would understand that the modules shown in **Figure 15** are example modules. For example, different embodiments can include even more modules, fewer modules, or different combination of modules.

B. Example Flow

[00128] Having described various examples of generating cache flow entries, an example process will now be described. **Figure 16** conceptually illustrates a process 1600 that some embodiments perform to dynamically generate a new flow with zero or more wildcard fields. In some embodiments, the process is performed by the switching element's OVS daemon.

[00129] The process 1600 begins when it receives (at 1605) a packet (e.g., from the kernel module). The process 1600 then initializes or sets (at 1610) all match fields as wildcard match fields. Alternatively, the process 1600 might initialize some but not all match fields as wildcard fields. In addition, the process 1600 might initialize an ingress port field as a wildcard

field. The ingress port is the port through which the switching element has received the packet. The ingress port may be a logical port. In some embodiments, the process never wildcards the ingress port field.

[00130] At 1615, the process 1600 determines if there any available flows (e.g., in a flow table). At 1610, the process selects a flow from a flow table. If there are no flows, the process might select a rule (e.g., a default rule). The switching element can have such one or more rules that specifies performing normal L2 processing, dropping packet, and/or sending the packet to the network controller. In some embodiments, the process 1600 selects a flow according to its associated priority value. For example, the process 1600 might iteratively select flows, starting from the one with the highest priority value to the lowest priority value. This is because there might be two flows in a flow table that match a packet but only one matching flow is cached in a datapath cache.

[00131] After selecting a flow, the process 1600 determines (at 1620) whether any one or more match fields have been compared with header values. If so, the process 1600 marks (at 1625) each of the one or more match fields as non-wildcard fields. Otherwise, the process determines (at 1630) whether the flow is a match for the packet. If the flow is not a match, the process returns to 1615, which is described above. Otherwise, the process 1600 determines (at 1635) whether the action of the matching flow specifies consulting one or more match fields. For example, based on the action, a switching element might consult a destination Ethernet address value of a packet to compare against a MAC address of a virtual machine that is connected to its port. As another example, the process 1600 might check if the packet is a BFD packet. If the process performs such consultation, the process 1600 marks (at 1640) the one or more match fields as non-wildcard fields.

[00132] If the action does not specify examining other match fields, the process 1600 then determines (at 1645) whether the matching flow is associated with a resubmit operation. In some embodiments, the resubmit operation is used to concatenate multiple flows into one datapath flow. For example, the process might be set up as multiple pipelines the packet goes through (e.g., to do a look-up in the L2 pipeline, then do a resubmit, and do a look-up in the L3 pipeline, etc.). Also, each wildcard and non-wildcard fields carries over from one hop to the next.

[00133] If the action specifies a resubmit operation, the process 1600 returns to 1615, which is described above. If the flow is not associated with such resubmit action, the process 1600 generates (at 1650) a new flow entry. The process 1600 of some embodiments generates the new flow entry by taking into account each remaining wildcard match fields. That is, the generated flow may include zero or more wildcard fields based on whether any of the wildcarded fields were marked as non-wildcard files (e.g., at operation 1625 and 1640). Upon generating the flow entry, the process 1600 sends (at 1655) the flow entry to the kernel. The process 1600 then ends.

[00134] Some embodiments perform variations on the process 1600. The specific operations of the process 1600 may not be performed in the exact order shown and described. For example, some embodiments optimize the process through a series of hash table look-ups when matching flows. Accordingly, the specific operations may not be performed in one continuous series of operations, and different specific operations may be performed in different embodiments.

C. Example Classification Algorithms

[00135] In several of the examples described above, the datapath flow generator utilizes a “linear search” algorithm to find a matching flow. The “linear search” algorithm does not require much memory, but it may be not be very fast. The switching element of some embodiments can utilize one of a number of different algorithms when un-wildcarding match fields. Several examples such classification algorithms will now be described in this subsection.

1. Example Tuple Space Search Algorithm

[00136] In some embodiments, the switching element uses a tuple search algorithm to find a matching flow. The tuple space search algorithm is a hashed-based search algorithm. It is similar to the “linear search” algorithm. However, instead of linearly traversing through every rule, the tuple space algorithm linearly traverses through different groups of rules that are organized by match fields. In some cases, the tuple space search can be much faster than a “linear search” because it can perform a lookup within a group of rules that have the same wildcard pattern using a hash table.

[00137] An example of a tuple space search will now be described. Suppose that a switching element maintains a flow table with the following three rules:

priority 5, in_port=1, eth_src=2 → Action 1;
priority 4, in_port=2, eth_src=3 → Action 2; and
priority 3, in_port=5 → Action 3.

[00138] In some embodiments, at flow table creation time, the switching element organizes these rules into different groups based on what fields (or partial fields) the rules match on. In this case, there are two groups:

group 1 (in_port, eth_src); and
group 2 (in_port).

[00139] Here, each rule is placed in a hash table that belongs to a particular group (e.g., group 1 or group 2). The hash table can have very fast (e.g., nearly instant) lookup. Thus, instead of a linearly traversing through all of the rules, the switching element can traverse through each hash table, doing a hash table lookup on each one, and un-wildcarding the fields that the switching element looks at or consults. In some embodiments, each hash table carries with it the priority of the highest priority rule it contains. Hence, if there is a match in the first hash table, the switching element is programmed to know that the rule has a higher priority value than every other rule in a subsequent table. The switching element can therefore skip the lookup and un-wildcarding in the subsequent hash table.

2. Example Staged Lookup Algorithm

[00140] In some embodiments, the switching element uses a staged lookup algorithm to search for one or more matching flows. The staged lookup algorithm is similar to the tuple space algorithm; however, it takes into account the fact that some header fields may change more frequently than others. Based on this fact, it performs a multi-staged search starting with infrequently changing fields, and progressing to frequently changing ones.

[00141] In utilizing this algorithm, the switching element of some embodiments does not look at all the fields (e.g., in a hash table) at once but first looks at those fields that do not change frequently. If none of those fields matches, the switching element terminates the lookup operation without having to lookup fields that change frequently. For instance, suppose that there is a particular hash table, which looks at the fields, in_port, eth_src, ip_src, and tcp_src.

[00142] With standard tuple space search, the software switching element looks at all those fields irrespective of whether the fields changes frequently or infrequently. With the staged lookup algorithm, the lookup is broken into different stages. For instance, in the first

stage, the algorithm can look up the `in_port` in a hash table and get a simple “yes” or “no” as to whether there is a match on the `in_port`. If the answer is “no”, the algorithm can terminate knowing that no further stages match. If the answer is “yes”, the algorithm can proceed to the next stage, which looks up the `in_port` and `eth_src` in a separate hash table. If successful, the algorithm may move onto `in_port`, `eth_src`, and `ip_src`. Thereafter, if successful again, the algorithm may look up the full `in_port`, `eth_src`, `ip_src`, and `tp_src`. One of the main ideas here is that, at each stage, if there is a miss, the algorithm can terminate without looking at the higher layer headers. This is important because the higher layer headers are the ones that are most likely to change from packet to packet and therefore the most problematic when trying to improve performance (e.g., megaflow performance).

3. Other Example Algorithms

[00143] In some embodiments, the switching element uses a decision tree algorithm to perform the classification operations. In some embodiments, the decision tree algorithm entails performing several phases of operations. For example, in the first phase, some fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. Each of the parallel lookups yields an output value. The contents of each memory are chosen so that the result of the lookup is narrower than the index. In subsequent phases, the index into each memory is formed by combining the results of the lookups from earlier phases. In the final phase, there is one result left from the lookup. This is because of the way the memory contents have been pre-computed. The decision tree algorithm is also described in the document entitled “Packet Classification on Multiple Fields” by Pankaj Gupta and Nick McKeown. The document was published in October of 1999 and is incorporated herein by reference. In each phase of the decision tree, the switching element of some embodiments does not wildcard a match field if it has to look at a field in the packet header. This same principle applies to the various different versions of the decision tree algorithm, in some embodiments.

D. Example Datapath Flows

[00144] **Figure 17** illustrates several examples of flows 1700 that are stored in a datapath cache. In some embodiments, the flows are stored in the datapath cache based on the number of flows that are currently stored in the cache. For example, if the switching element is not heavily loaded, a flow will stay in the cache if it was not used within a particular time period (e.g., five seconds). If the flow was not used within the particular time period, the flow may be dropped

from the cache. Once the datapath reaches a certain threshold number, the switching element may change how long a flow stays in the cache. For example, if the cache is heavily loaded, a flow might only last a hundred milliseconds if it was not used. In some embodiments, the userspace makes the decisions about how long a flow stays in the cache based on how recently it was used and/or the amount of flows in the datapath cache.

[00145] In the example of **Figure 17**, the datapath 1700 includes two flows. Each of these flows has the Ethernet sources and destination match fields un-wildcarded. The remaining fields are all wildcarded. Specifically, the IP source, IP destination, protocol, time to live, time of service, fragment, (Internet Control Message Protocol) ICMP type, and ICMP code match fields have all been wildcarded. Each flow is also associated with several other values such as byte size, last used, packets, and action.

IV. Electronic System

[00146] Many of the above-described features and applications are implemented as software processes that are specified as a set of instructions recorded on a computer readable storage medium (also referred to as computer readable medium). When these instructions are executed by one or more computational or processing unit(s) (e.g., one or more processors, cores of processors, or other processing units), they cause the processing unit(s) to perform the actions indicated in the instructions. Examples of computer readable media include, but are not limited to, CD-ROMs, flash drives, random access memory (RAM) chips, hard drives, erasable programmable read-only memories (EPROMs), electrically erasable programmable read-only memories (EEPROMs), etc. The computer readable media does not include carrier waves and electronic signals passing wirelessly or over wired connections.

[00147] In this specification, the term "software" is meant to include firmware residing in read-only memory or applications stored in magnetic storage, which can be read into memory for processing by a processor. Also, in some embodiments, multiple software inventions can be implemented as sub-parts of a larger program while remaining distinct software inventions. In some embodiments, multiple software inventions can also be implemented as separate programs. Finally, any combination of separate programs that together implement a software invention described here is within the scope of the invention. In some embodiments, the software programs, when installed to operate on one or more electronic systems, define one or

more specific machine implementations that execute and perform the operations of the software programs.

[00148] **Figure 18** conceptually illustrates an electronic system 1800 with which some embodiments of the invention are implemented. The electronic system 1800 may be a computer (e.g., a desktop computer, personal computer, tablet computer, etc.), server, dedicated switch, phone, PDA, or any other sort of electronic or computing device. Such an electronic system includes various types of computer readable media and interfaces for various other types of computer readable media. Electronic system 1800 includes a bus 1805, processing unit(s) 1810, a system memory 1825, a read-only memory 1830, a permanent storage device 1835, input devices 1840, and output devices 1845.

[00149] The bus 1805 collectively represents all system, peripheral, and chipset buses that communicatively connect the numerous internal devices of the electronic system 1800. For instance, the bus 1805 communicatively connects the processing unit(s) 1810 with the read-only memory 1830, the system memory 1825, and the permanent storage device 1835.

[00150] From these various memory units, the processing unit(s) 1810 retrieves instructions to execute and data to process in order to execute the processes of the invention. The processing unit(s) may be a single processor or a multi-core processor in different embodiments.

[00151] The read-only-memory (ROM) 1830 stores static data and instructions that are needed by the processing unit(s) 1810 and other modules of the electronic system. The permanent storage device 1835, on the other hand, is a read-and-write memory device. This device is a non-volatile memory unit that stores instructions and data even when the electronic system 1800 is off. Some embodiments of the invention use a mass-storage device (such as a magnetic or optical disk and its corresponding disk drive) as the permanent storage device 1835.

[00152] Other embodiments use a removable storage device (such as a floppy disk, flash memory device, etc., and its corresponding drive) as the permanent storage device. Like the permanent storage device 1835, the system memory 1825 is a read-and-write memory device. However, unlike storage device 1835, the system memory 1825 is a volatile read-and-write memory, such a random access memory. The system memory 1825 stores some of the instructions and data that the processor needs at runtime. In some embodiments, the invention's processes are stored in the system memory 1825, the permanent storage device 1835, and/or the

read-only memory 1830. From these various memory units, the processing unit(s) 1810 retrieves instructions to execute and data to process in order to execute the processes of some embodiments.

[00153] The bus 1805 also connects to the input and output devices 1840 and 1845. The input devices 1840 enable the user to communicate information and select commands to the electronic system. The input devices 1840 include alphanumeric keyboards and pointing devices (also called "cursor control devices"), cameras (e.g., webcams), microphones or similar devices for receiving voice commands, etc. The output devices 1845 display images generated by the electronic system or otherwise output data. The output devices 1845 include printers and display devices, such as cathode ray tubes (CRT) or liquid crystal displays (LCD), as well as speakers or similar audio output devices. Some embodiments include devices such as a touchscreen that function as both input and output devices.

[00154] Finally, as shown in **Figure 18**, bus 1805 also couples electronic system 1800 to a network 1865 through a network adapter (not shown). In this manner, the computer can be a part of a network of computers (such as a local area network ("LAN"), a wide area network ("WAN"), or an Intranet, or a network of networks, such as the Internet. Any or all components of electronic system 1800 may be used in conjunction with the invention.

[00155] Some embodiments include electronic components, such as microprocessors, storage and memory that store computer program instructions in a machine-readable or computer-readable medium (alternatively referred to as computer-readable storage media, machine-readable media, or machine-readable storage media). Some examples of such computer-readable media include RAM, ROM, read-only compact discs (CD-ROM), recordable compact discs (CD-R), rewritable compact discs (CD-RW), read-only digital versatile discs (e.g., DVD-ROM, dual-layer DVD-ROM), a variety of recordable/rewritable DVDs (e.g., DVD-RAM, DVD-RW, DVD+RW, etc.), flash memory (e.g., SD cards, mini-SD cards, micro-SD cards, etc.), magnetic and/or solid state hard drives, read-only and recordable Blu-Ray® discs, ultra density optical discs, any other optical or magnetic media, and floppy disks. The computer-readable media may store a computer program that is executable by at least one processing unit and includes sets of instructions for performing various operations. Examples of computer programs or computer code include machine code, such as is produced by a compiler,

and files including higher-level code that are executed by a computer, an electronic component, or a microprocessor using an interpreter.

[00156] While the above discussion primarily refers to microprocessor or multi-core processors that execute software, some embodiments are performed by one or more integrated circuits, such as application specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs). In some embodiments, such integrated circuits execute instructions that are stored on the circuit itself. In addition, some embodiments execute software stored in programmable logic devices (PLDs), ROM, or RAM devices.

[00157] As used in this specification and any claims of this application, the terms “computer”, “server”, “processor”, and “memory” all refer to electronic or other technological devices. These terms exclude people or groups of people. For the purposes of the specification, the terms display or displaying means displaying on an electronic device. As used in this specification and any claims of this application, the terms “computer readable medium,” “computer readable media,” and “machine readable medium” are entirely restricted to tangible, physical objects that store information in a form that is readable by a computer. These terms exclude any wireless signals, wired download signals, and any other ephemeral signals.

[00158] While the invention has been described with reference to numerous specific details, one of ordinary skill in the art will recognize that the invention can be embodied in other specific forms without departing from the spirit of the invention. In addition, a number of the figures (including **Figures 1, 5, and 16**) conceptually illustrate processes. The specific operations of these processes may not be performed in the exact order shown and described. The specific operations may not be performed in one continuous series of operations, and different specific operations may be performed in different embodiments. Furthermore, the process could be implemented using several sub-processes, or as part of a larger macro process. Thus, one of ordinary skill in the art would understand that the invention is not to be limited by the foregoing illustrative details, but rather is to be defined by the appended claims.

CLAIMS

What is claimed is:

1. A method for a switching element that forwards packets, the method comprising:
receiving a packet;
initializing a plurality of match fields as wildcard fields;
dynamically generating a flow entry by un-wildcarding each match field that is consulted in generating the flow entry, wherein the flow entry is associated with an action;
performing the action on the packet; and
storing the flow entry in a cache to process any subsequent packet that matches each non-wildcard field of the flow entry.
2. The method of claim 1, wherein dynamically generating comprises comparing a value of a match field against the packet's header value and un-wildcarding the match field.
3. The method of claim 1, wherein dynamically generating comprises selecting one or more flows from a flow table until a matching flow is found.
4. The method of claim 3, wherein dynamically generating comprises un-wildcarding one or more fields based on an action associated with the matching flow.
5. The method of claim 3, wherein dynamically generating comprises researching the same flow table or another flow table for another matching flow if the matching flow specifies a resubmit action.
6. The method of claim 1, wherein dynamically generating comprises using a hash-based algorithm to search for one or more matching flows while un-wildcarding each match field that is consulted during the search.
7. The method of claim 6, wherein the hash-based algorithm performs a multi-staged search, starting with infrequently changing match fields, and progressing to frequently changing fields.
8. The method of claim 1, wherein dynamically generating comprises using a decision tree algorithm to search for one or more matching flows while un-wildcarding each match field that is consulted during the search.
9. The method of claim 1, wherein the packet is a first packet, the method further comprising:
receiving a second packet;

determining that the second packet's header has values that matches each non-wildcard field of the flow entry; and

in response to the determination, performing the action of the second packet.

10. The method of claim 1, wherein the flow includes a plurality of wildcard match fields.

11. A non-transitory machine readable medium storing a program that when executed by at least one processing unit forwards packets, the program comprising sets of instructions for:

receiving a packet;

initializing a plurality of match fields as wildcard fields;

dynamically generating a flow entry by un-wildcarding each match field that is consulted in generating the flow entry, wherein the flow entry is associated with an action;

performing the action on the packet; and

storing the flow entry in a cache to process any subsequent packet that matches each non-wildcard field of the flow entry.

12. The non-transitory machine readable medium of claim 11, wherein the set of instructions for dynamically generating comprises a set of instructions for comparing a value of a match field against the packet's header value and un-wildcarding the match field.

13. The non-transitory machine readable medium of claim 11, wherein the set of instructions for dynamically generating comprises a set of instructions for selecting one or more flows from a flow table until a matching flow is found.

14. The non-transitory machine readable medium of claim 13, wherein the set of instructions for dynamically generating comprises a set of instructions for un-wildcarding one or more fields based on an action associated with the matching flow.

15. The non-transitory machine readable medium of claim 13, wherein the set of instructions for dynamically generating comprises a set of instructions for researching the same flow table or another flow table for another matching flow if the matching flow specifies a resubmit action.

16. The non-transitory machine readable medium of claim 11, wherein the set of instructions for dynamically generating comprises a set of instructions for using a hash-based algorithm to search for one or more matching flows while un-wildcarding each match field that

is consulted during the search.

17. The non-transitory machine readable medium of claim 16, wherein the hash-based algorithm performs a multi-staged search, starting with infrequently changing match fields, and progressing to frequently changing fields.

18. The non-transitory machine readable medium of claim 11, wherein the set of instructions for dynamically generating comprises a set of instructions for using a decision tree algorithm to search for one or more matching flows while un-wildcarding each match field that is consulted during the search.

19. The non-transitory machine readable medium of claim 11, wherein the packet is a first packet, the program further comprising sets of instructions for:

receiving a second packet;

determining that the second packet's header has values that matches each non-wildcard field of the flow entry; and

in response to the determination, performing the action of the second packet.

20. The non-transitory machine readable medium of claim 11, wherein the flow includes a plurality of wildcard match fields.

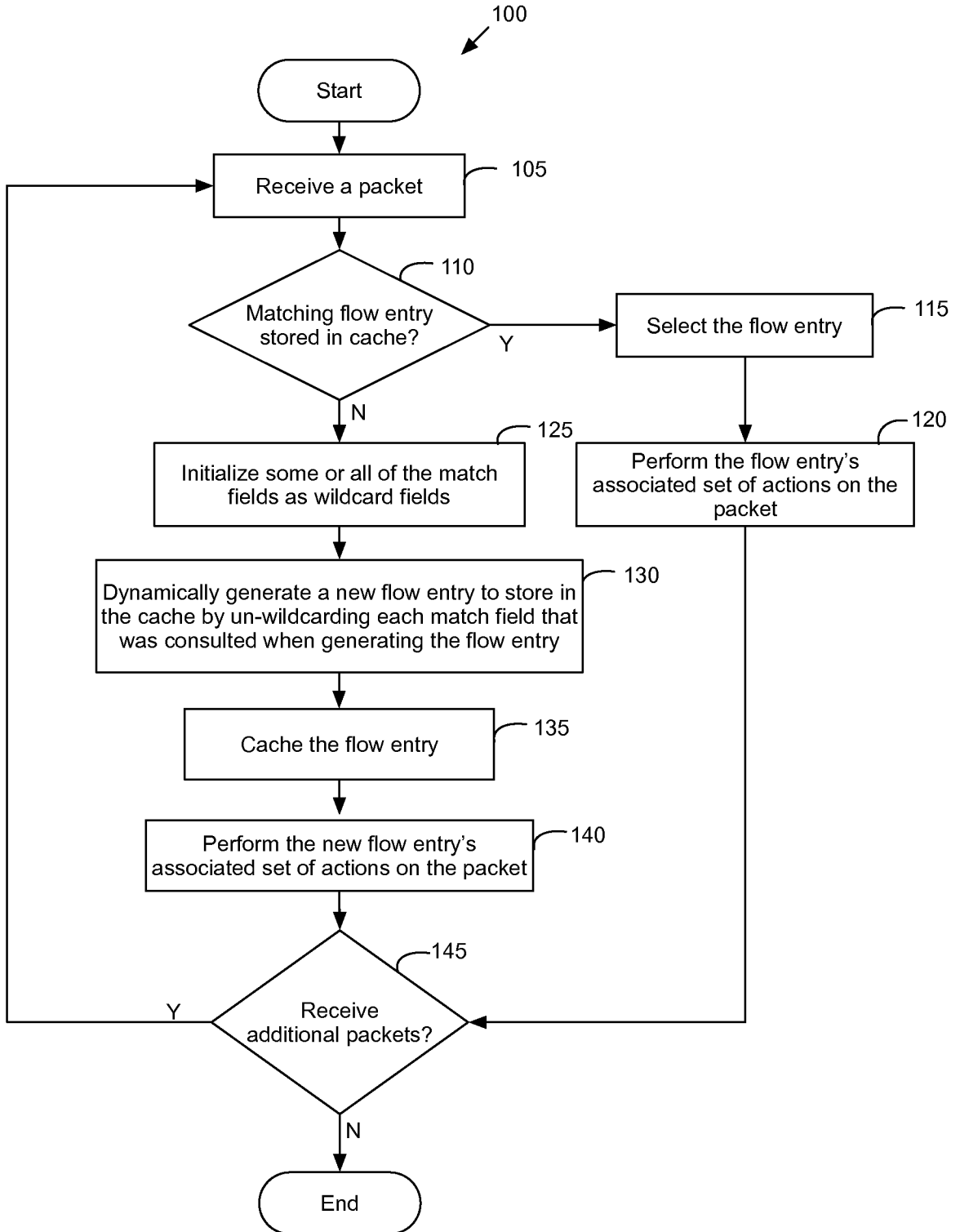


FIG. 1

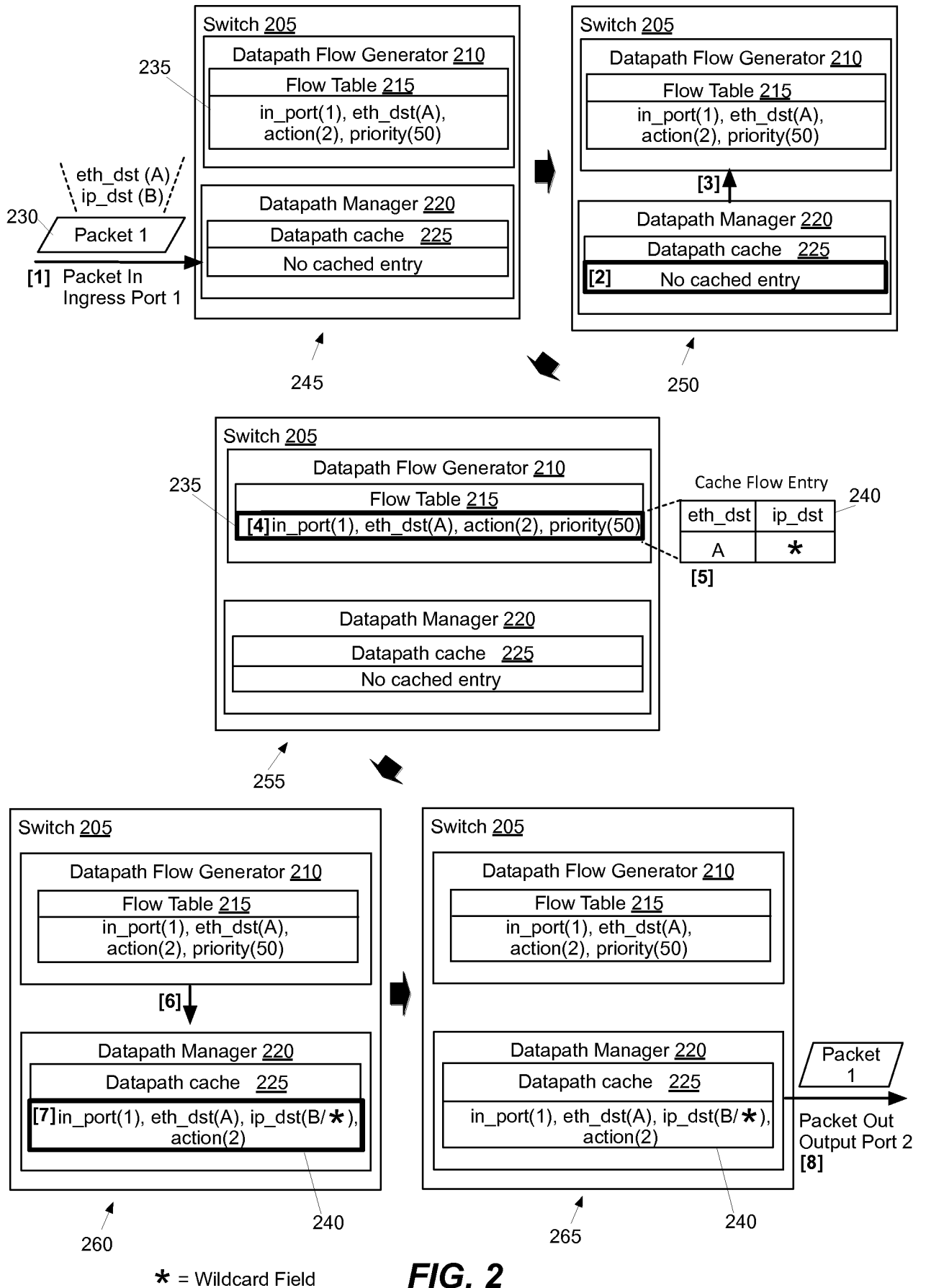


FIG. 2

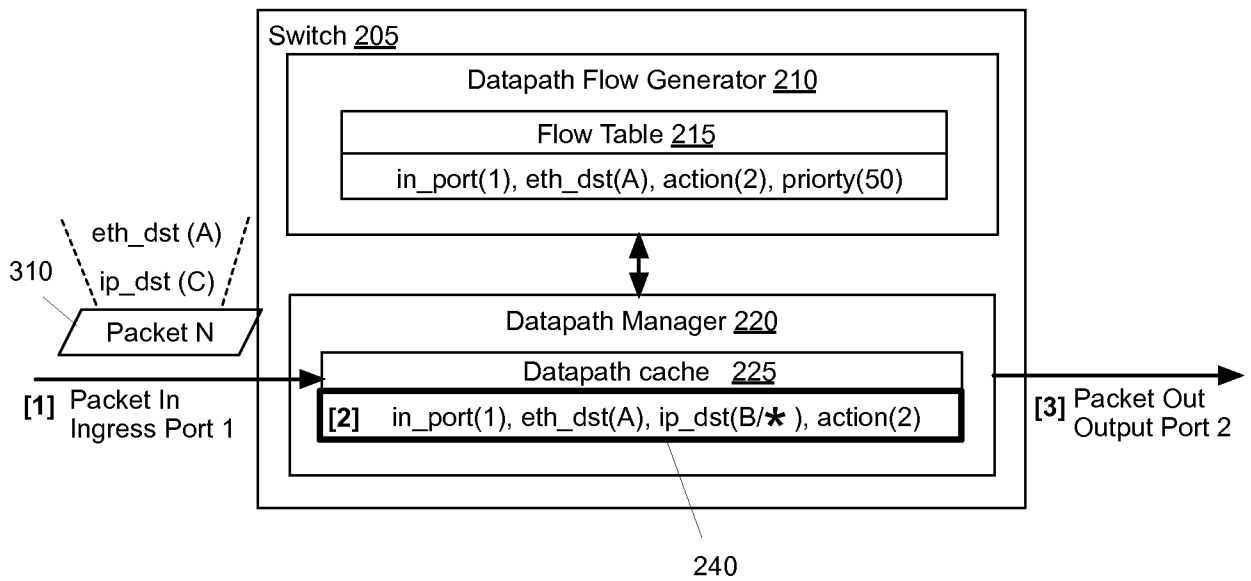
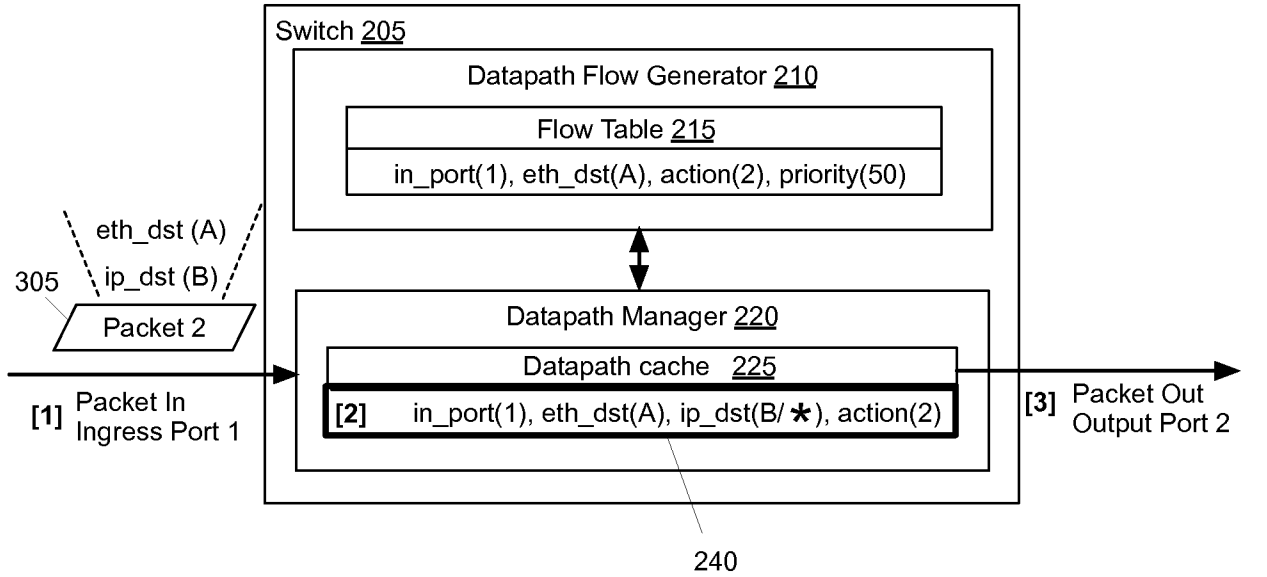


FIG. 3

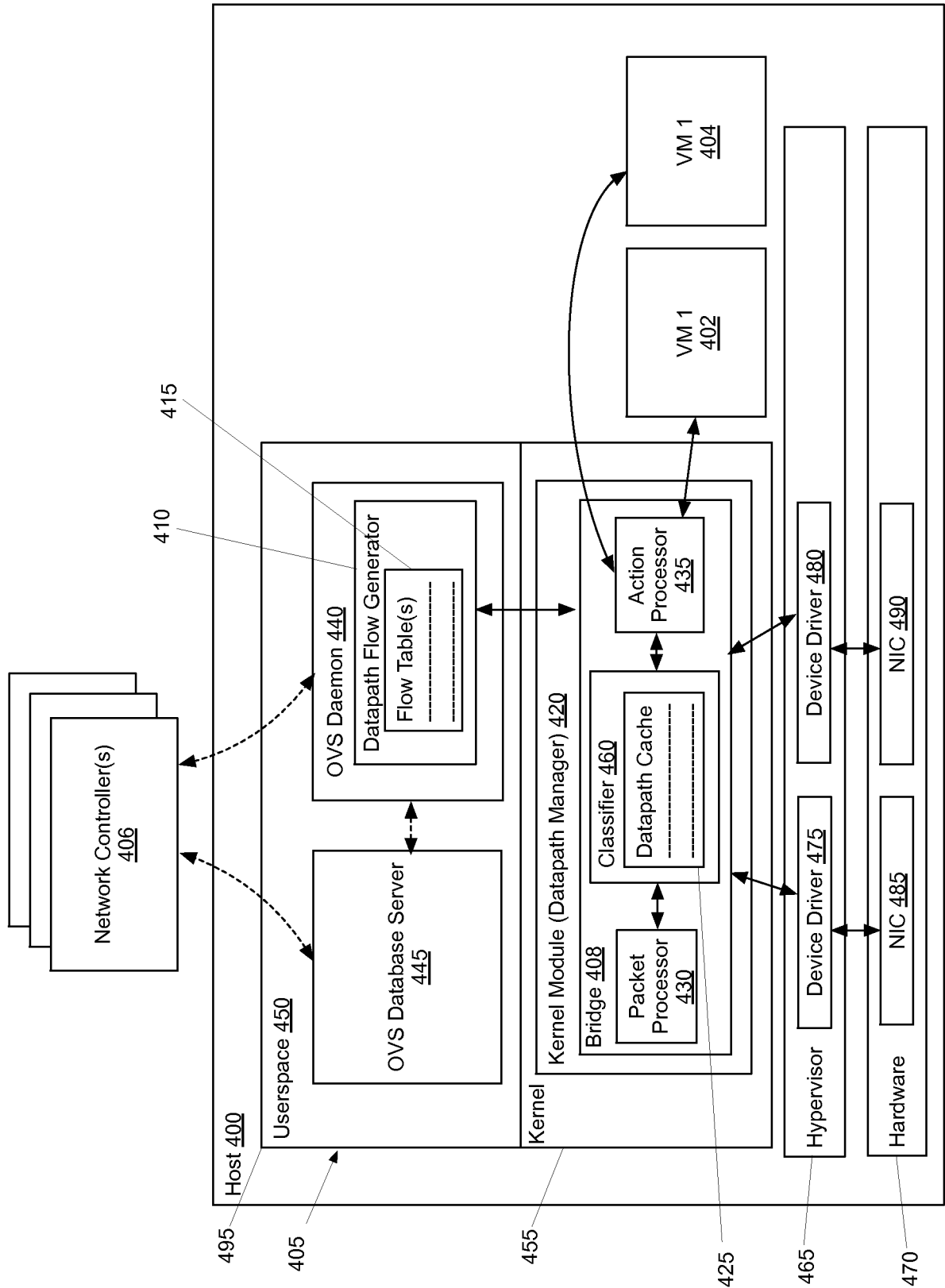


FIG. 4

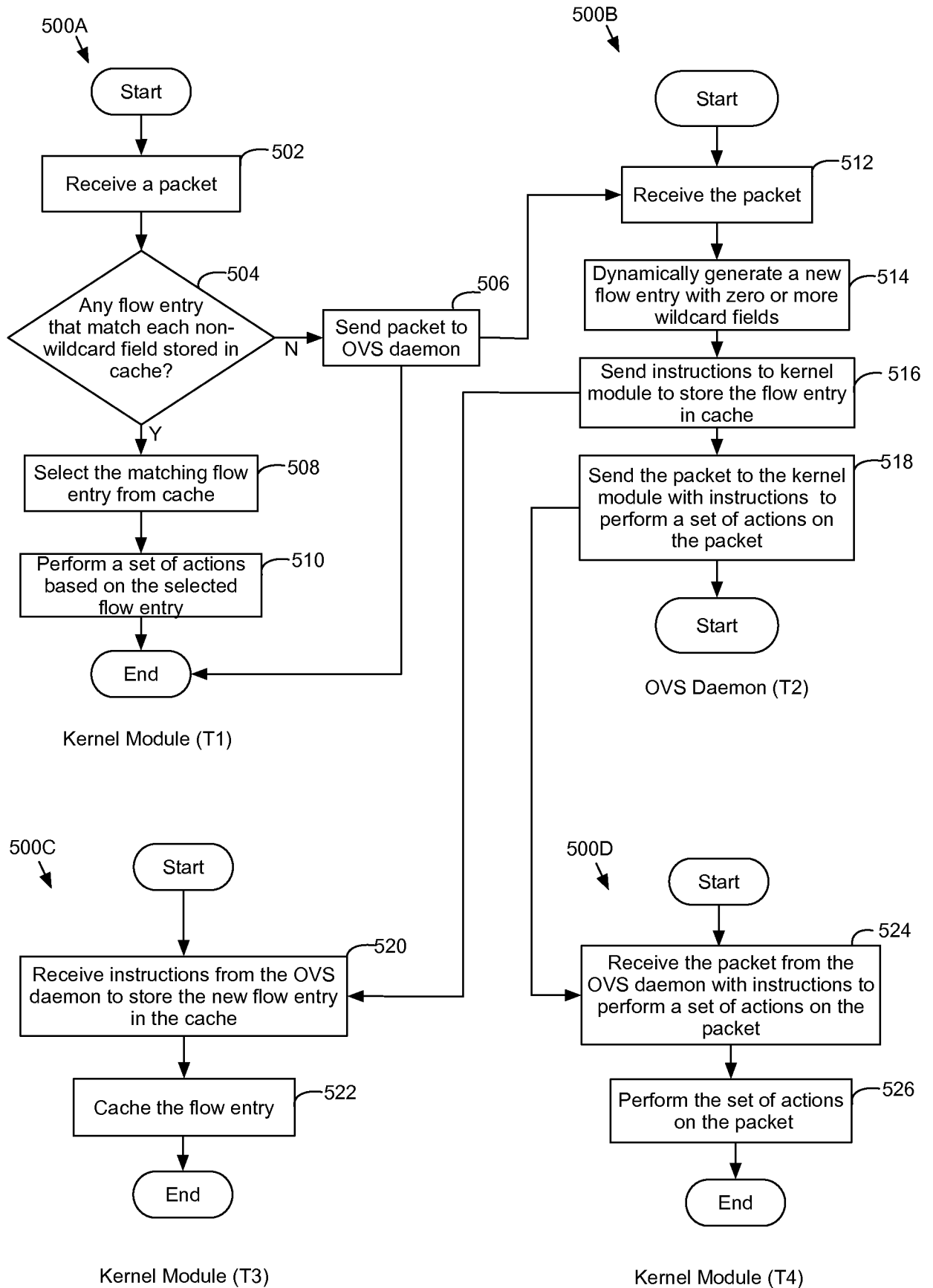


FIG. 5

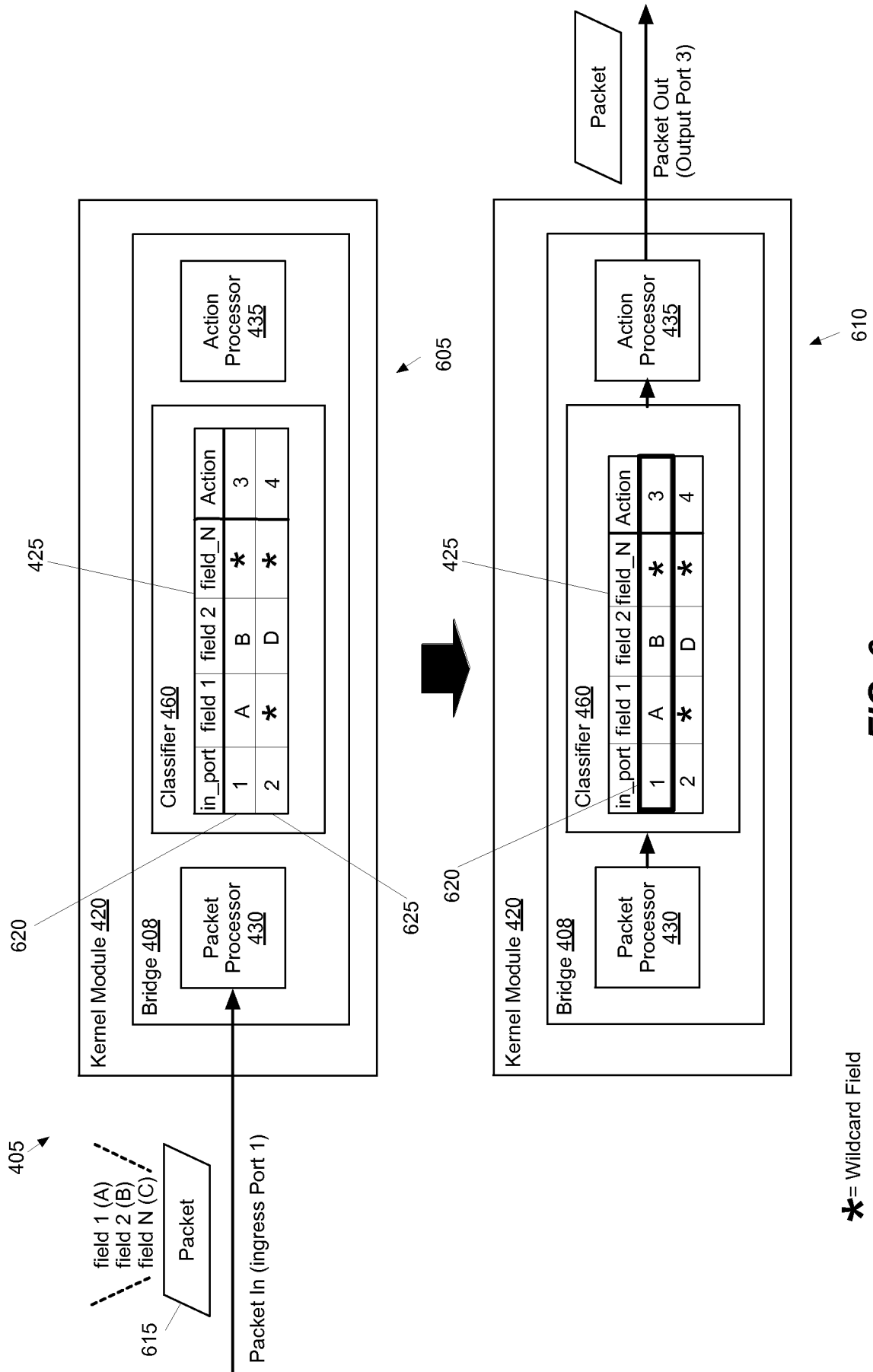


FIG. 6

(7/18)

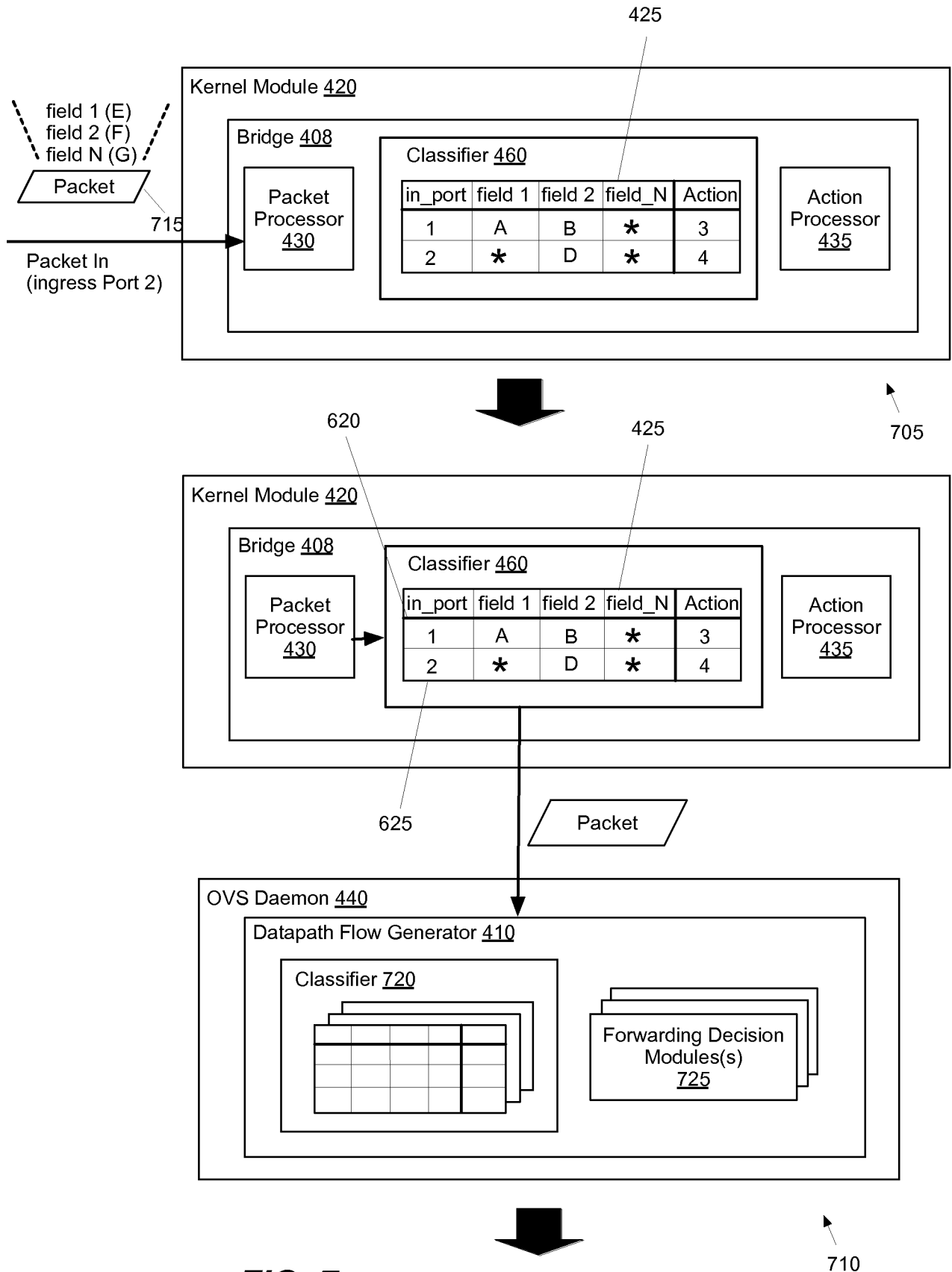


FIG. 7

To FIG. 8

From FIG. 7

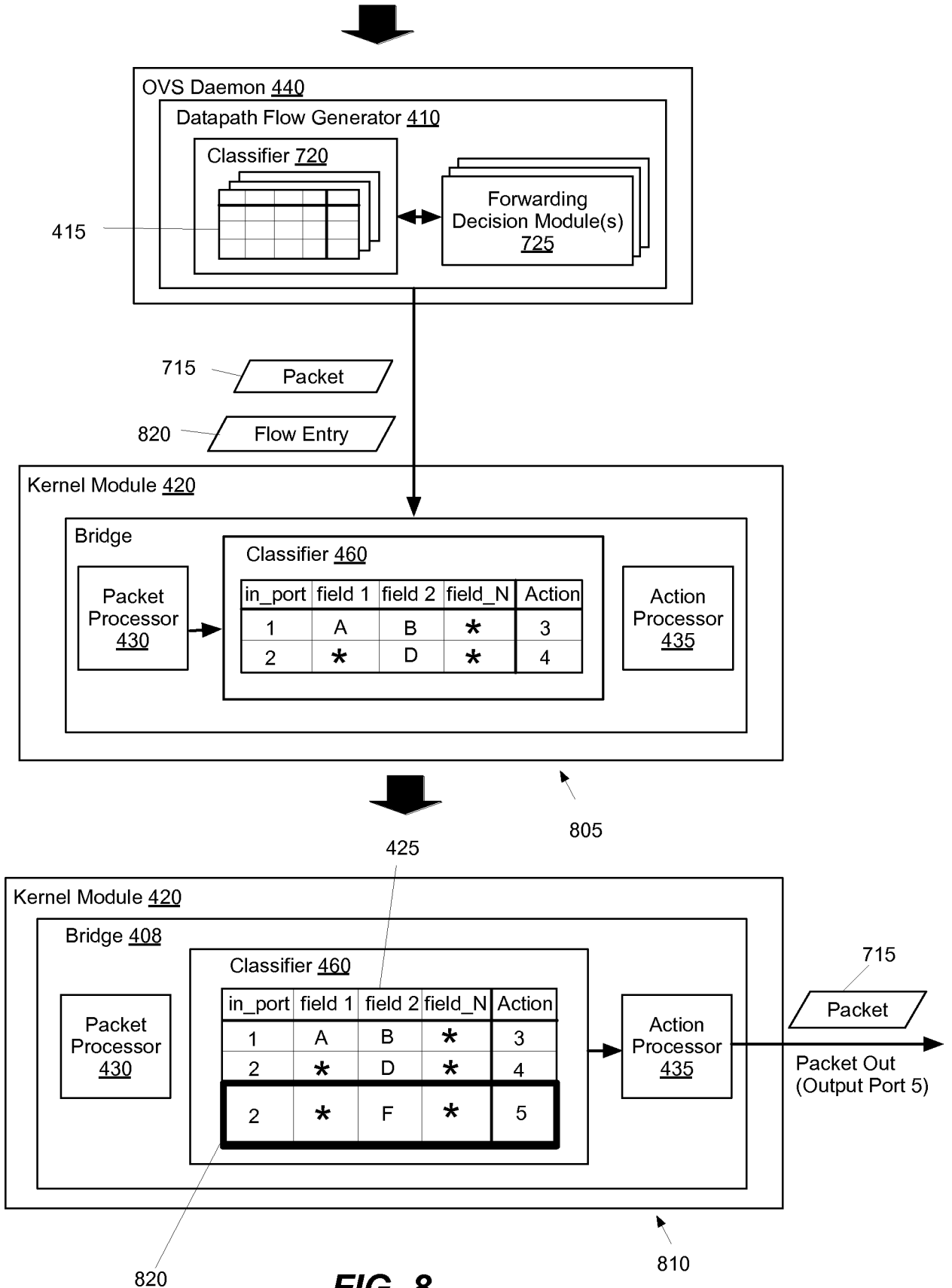


FIG. 8

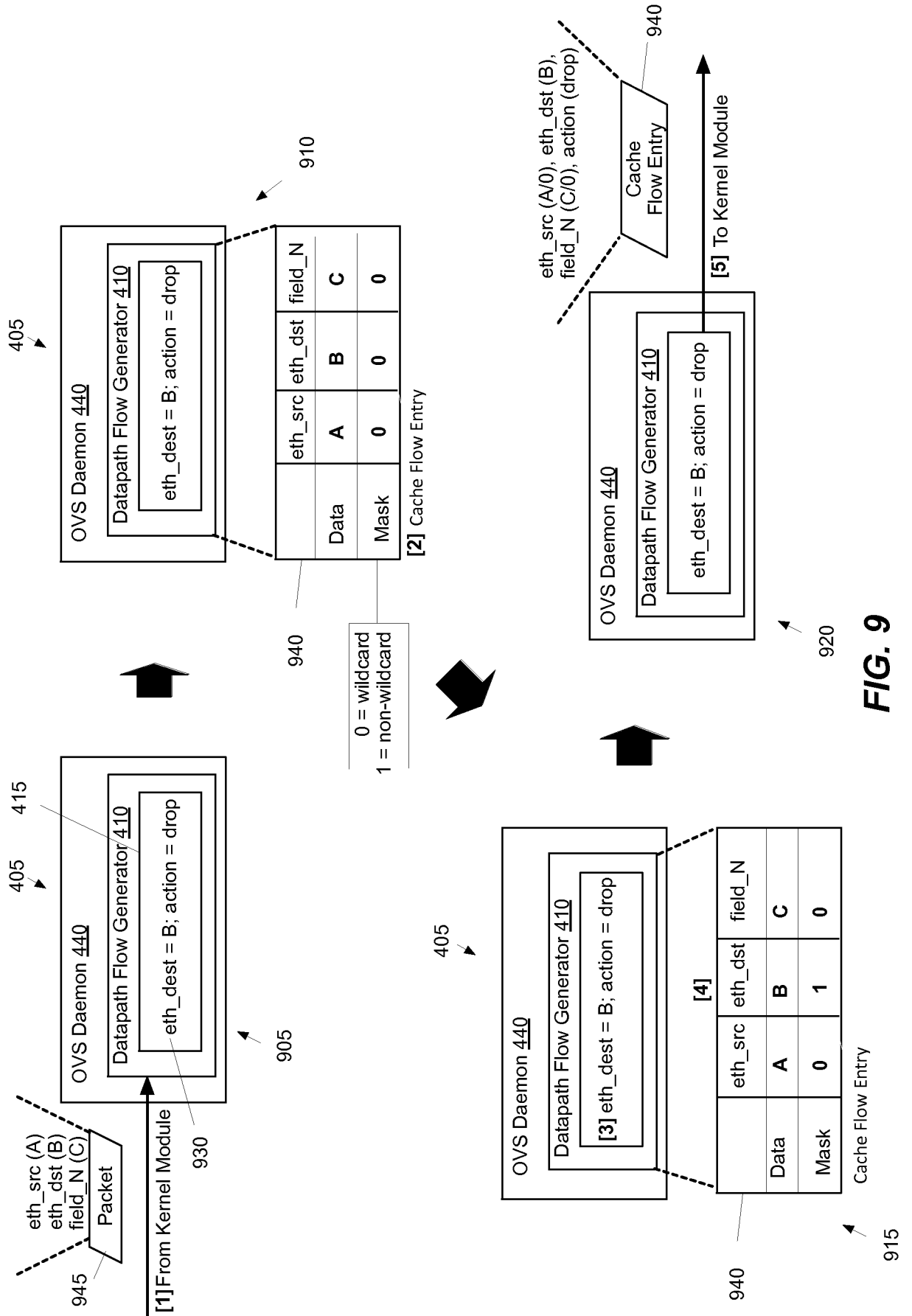


FIG. 9

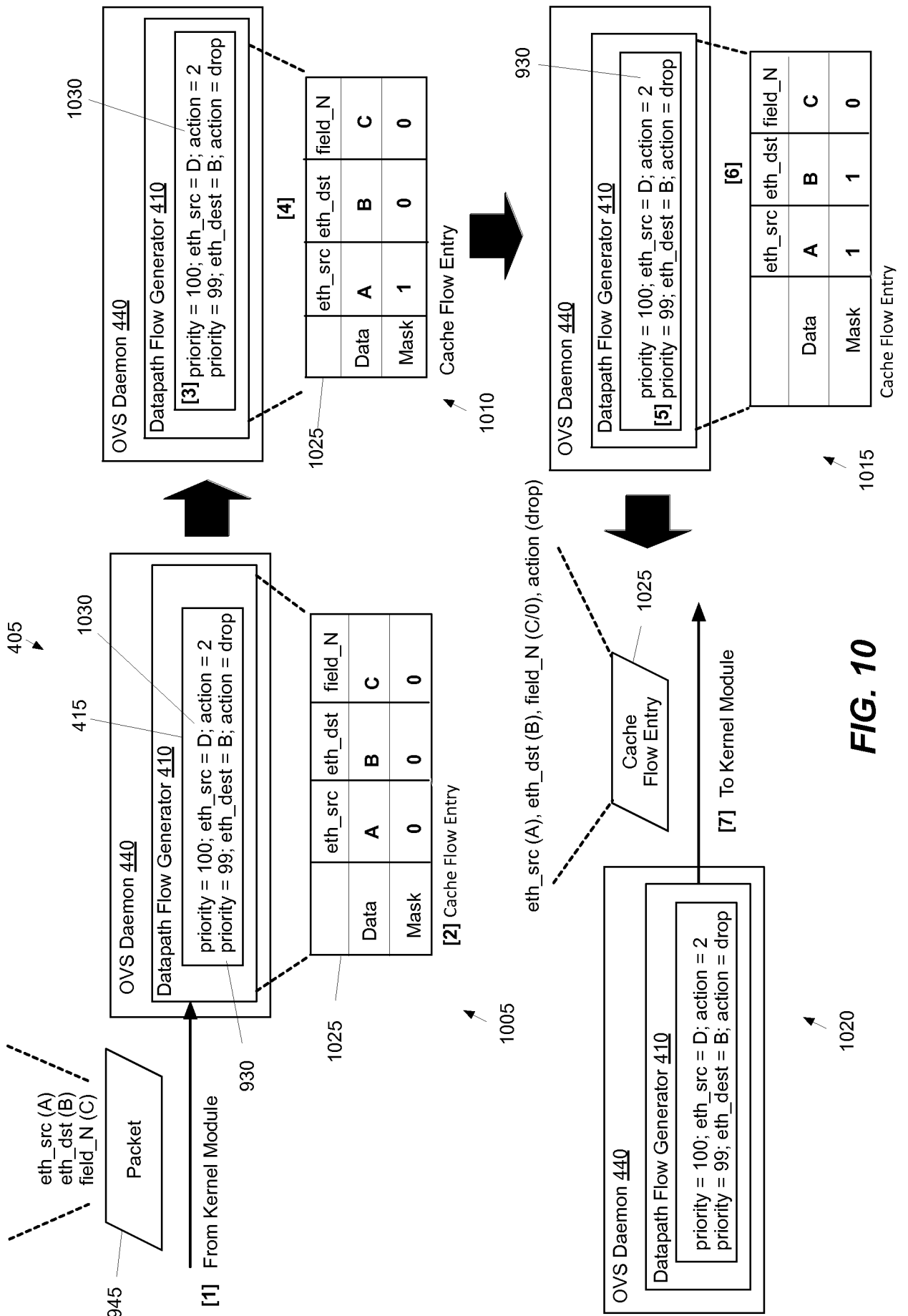


FIG. 10

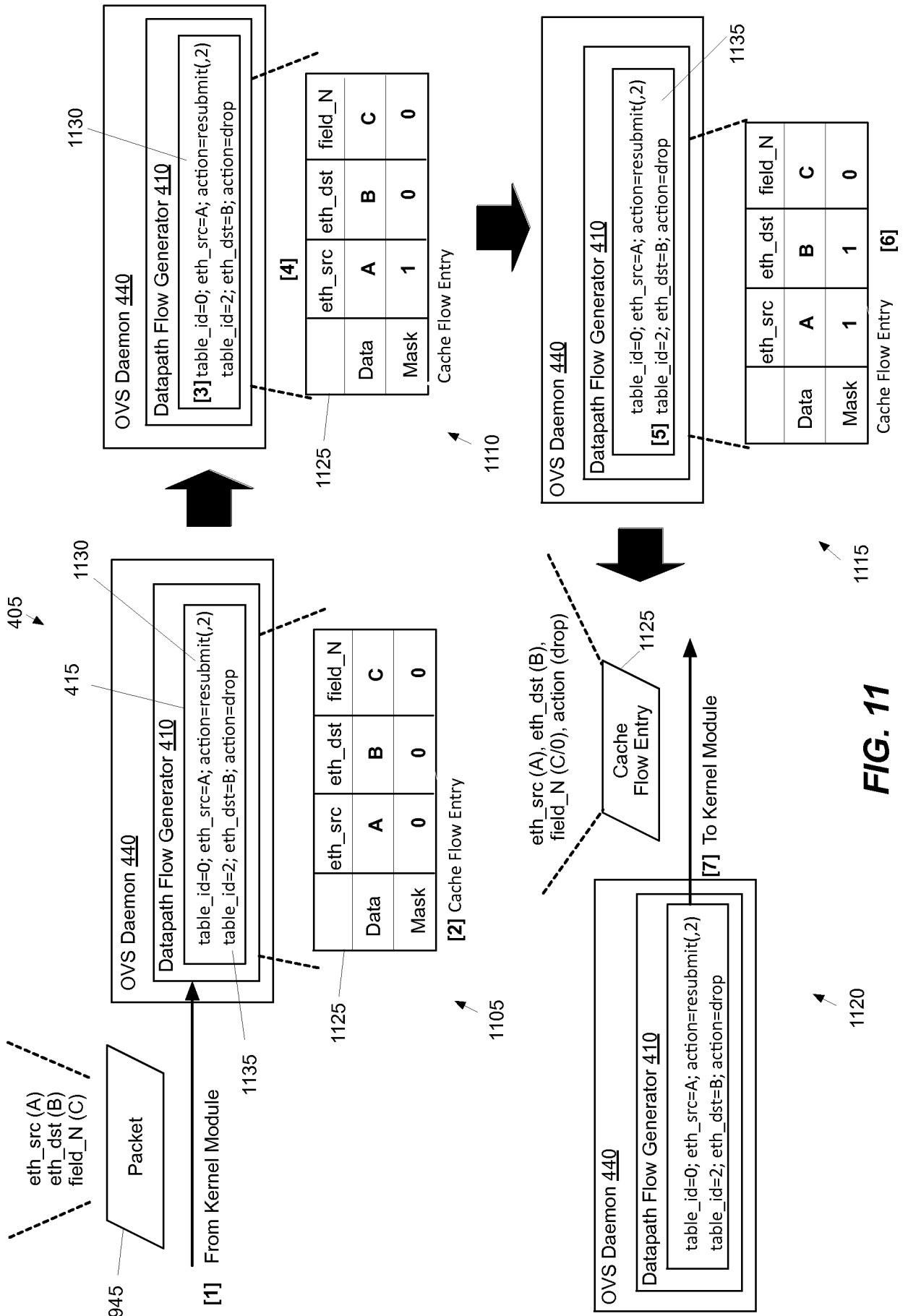


FIG. 11

(12/18)

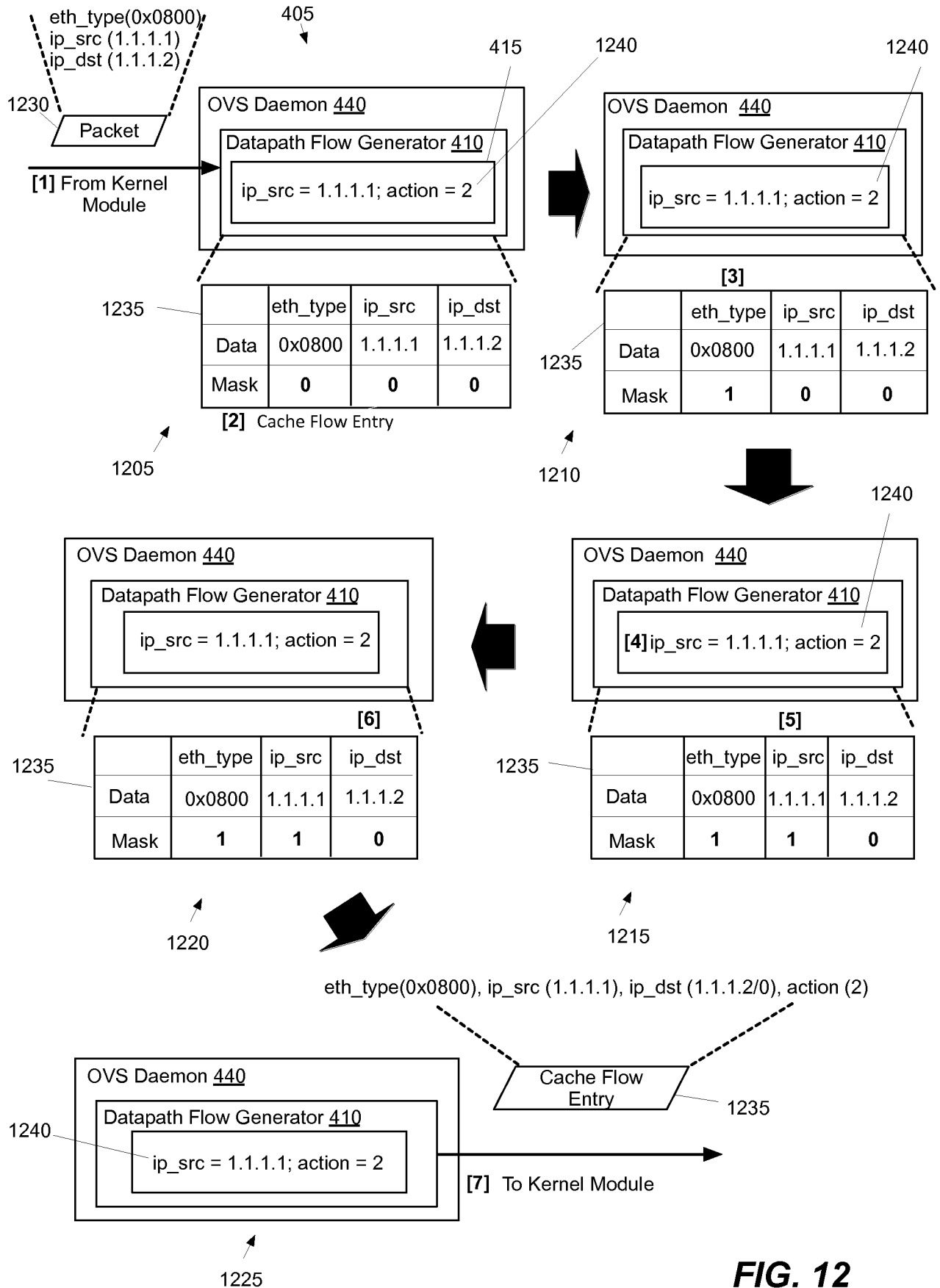


FIG. 12

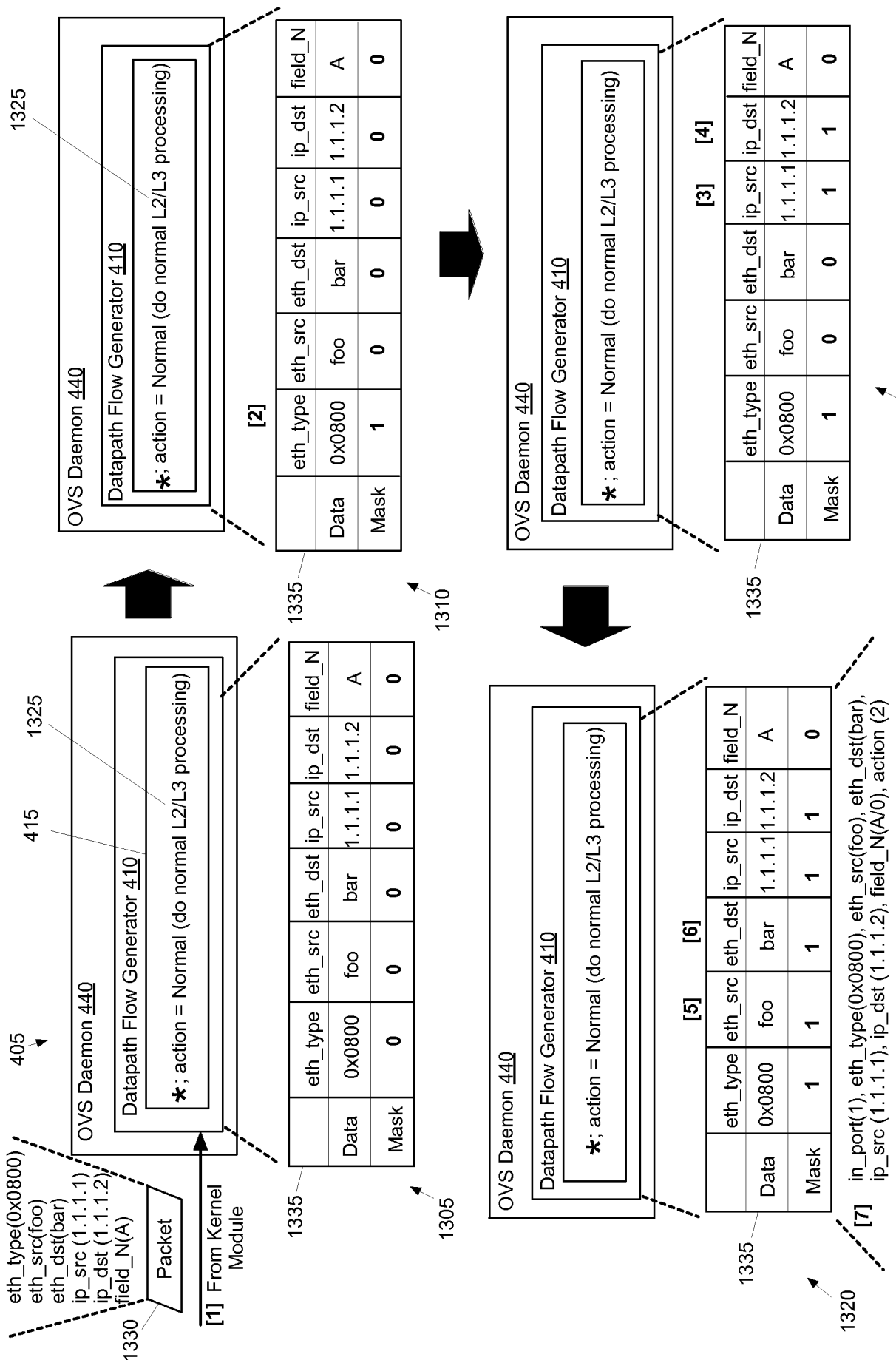


FIG. 13

(14/18)

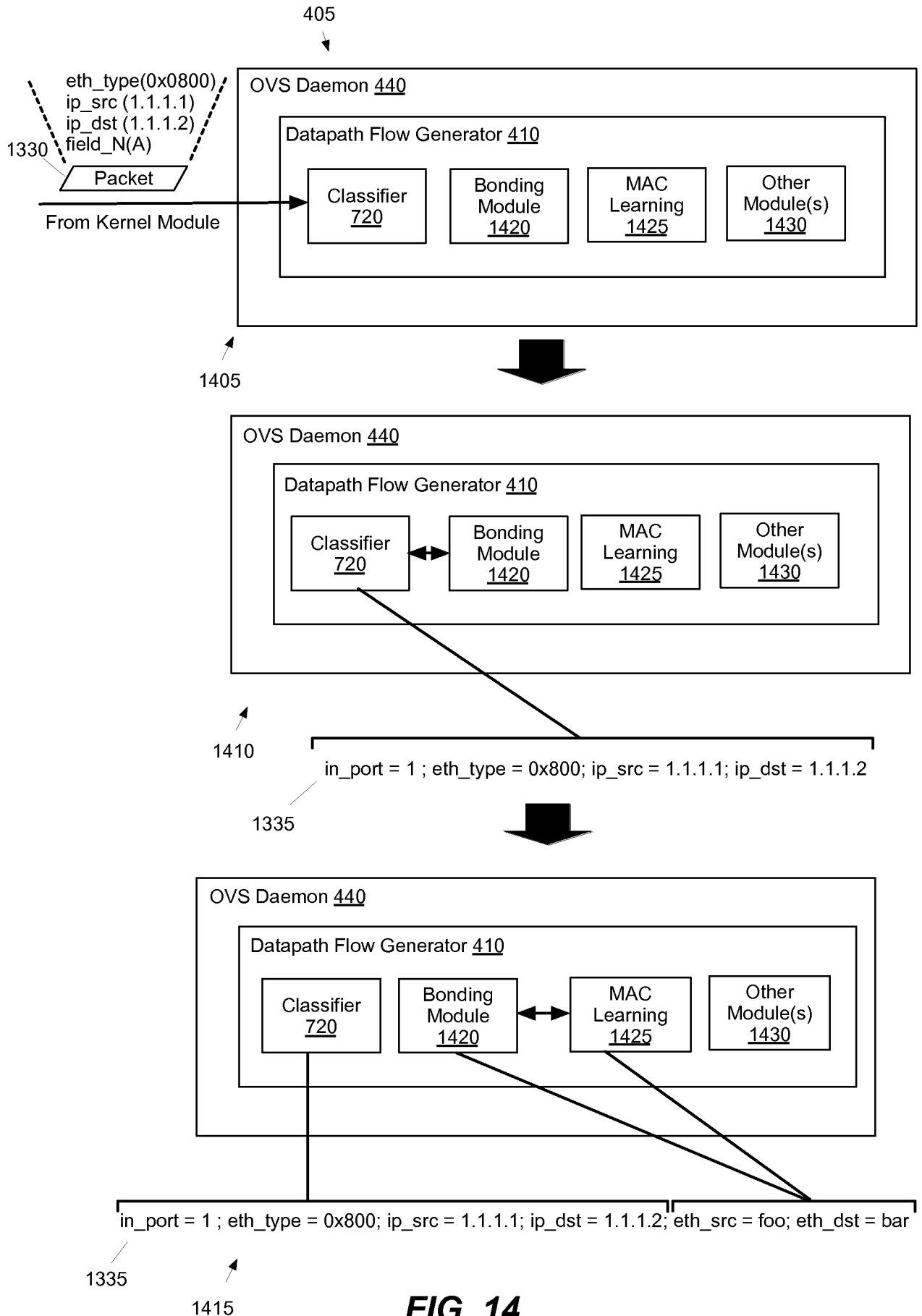


FIG. 14

(15/18)

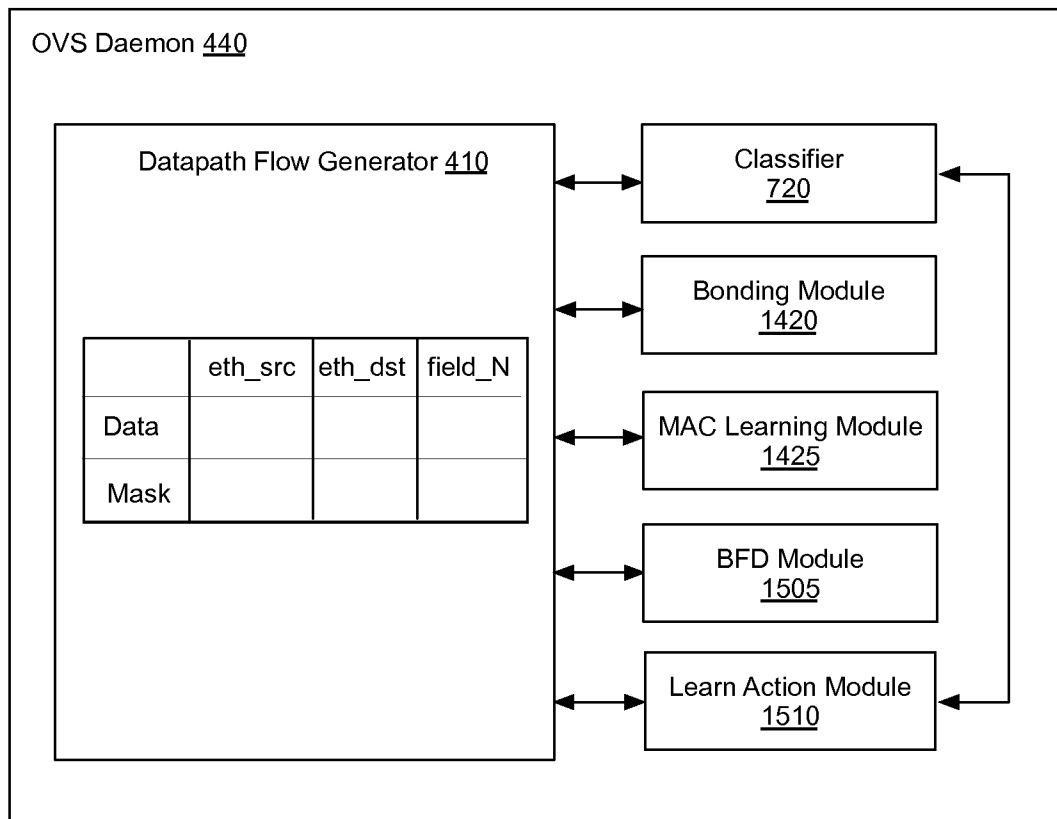


FIG. 15

(16/18)

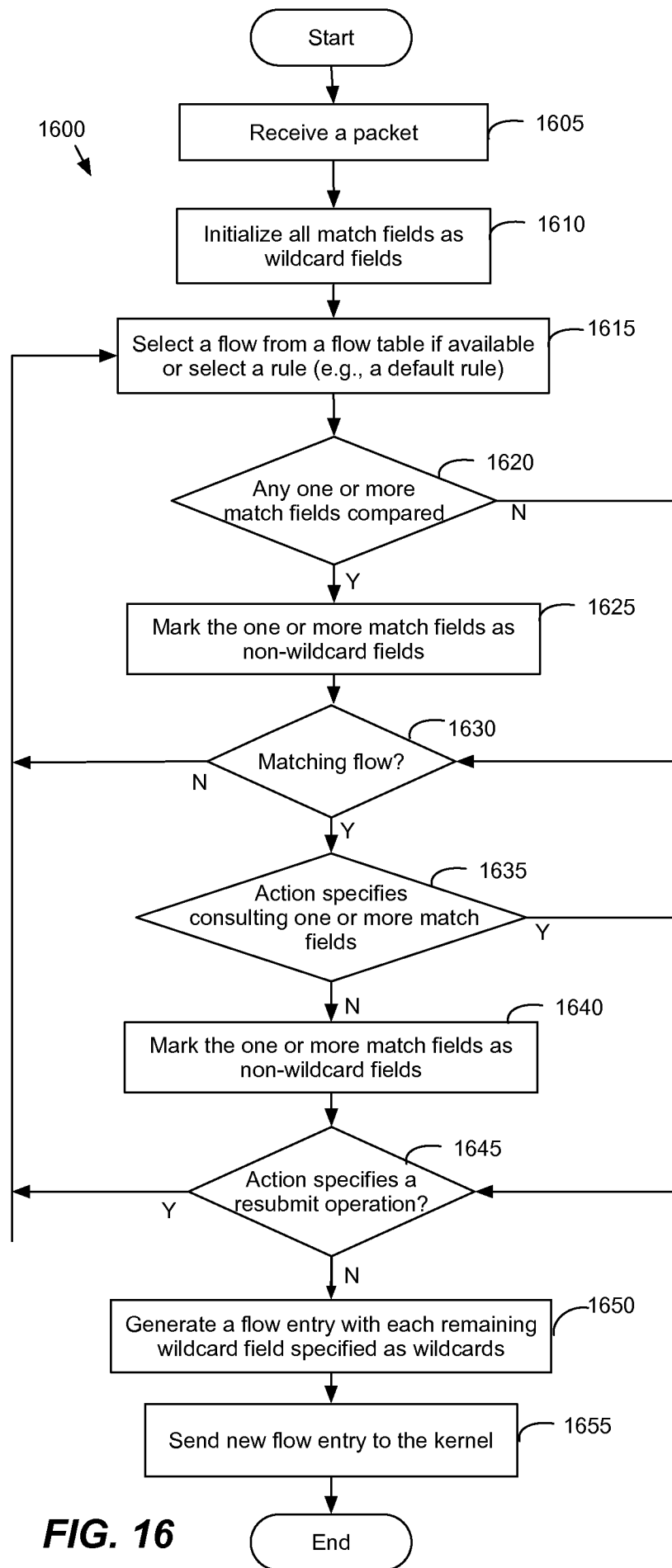


FIG. 16

(17/18)

1700 

| in port | eth src | eth dst | eth type | ip src | ip dst | proto | tos | ttl | frag | icm typ | imcp code | packets | bytes | used | action |
|---------|-------------------|-------------------|----------|----------------------|----------------------|-------|-----|------|--------|---------|-----------|---------|-------|--------|--------|
| 3 | 50:54:00:00:00:03 | 50:54:00:00:00:01 | 0x0800 | 192.168.0.02/0.0.0.0 | 192.168.0.01/0.0.0.0 | 1/0 | 0/0 | 64/0 | no/0x2 | 0/0 | 0/0 | 95 | 9310 | 0.425s | 2 |
| 2 | 50:54:00:00:00:01 | 50:54:00:00:00:03 | 0x0800 | 192.168.0.01/0.0.0.0 | 192.168.0.02/0.0.0.0 | 1/0 | 0/0 | 64/0 | no/0x2 | 0/0 | 0/0 | 95 | 9310 | 0425s | 3 |

FIG. 17

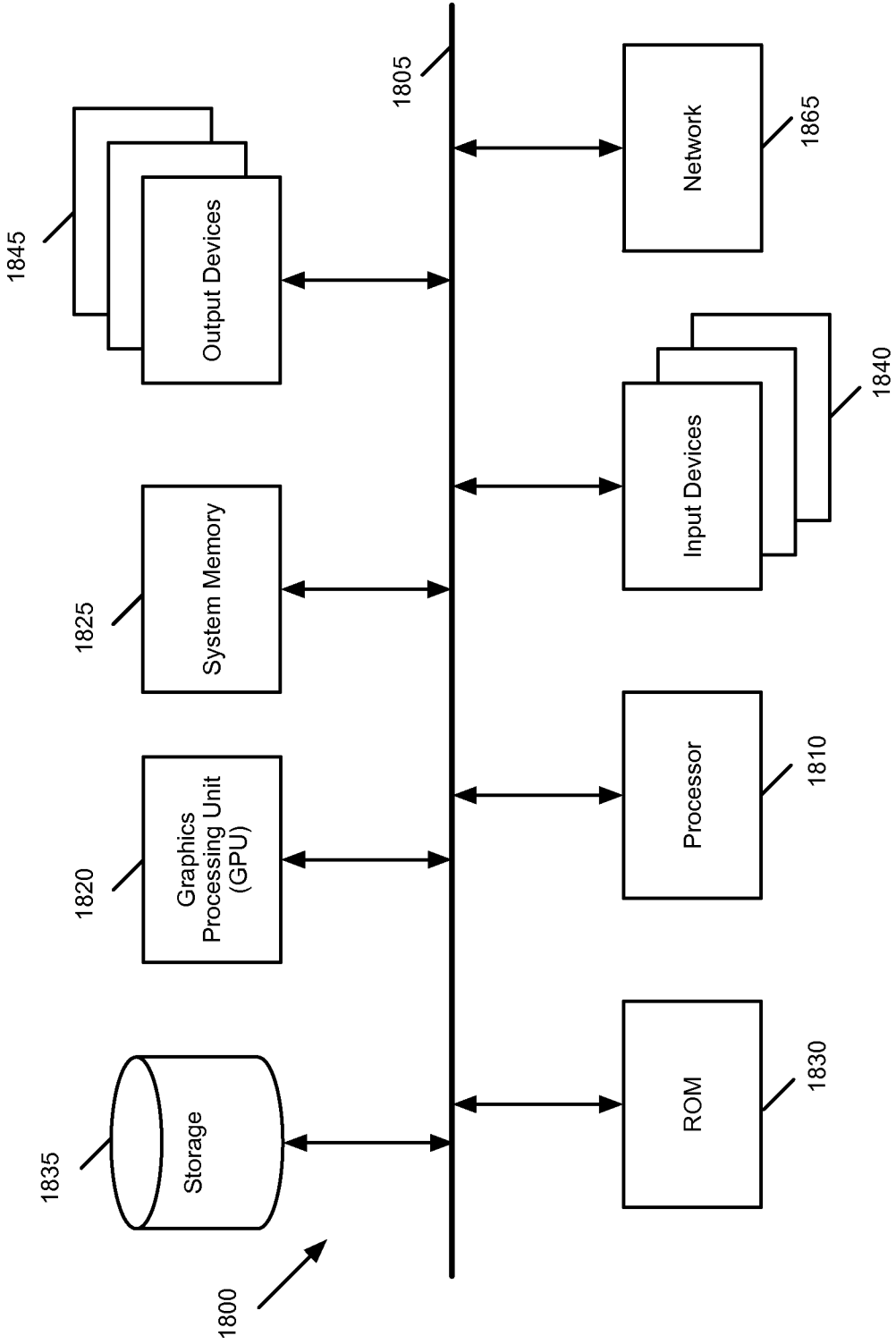


FIG. 18

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2014/036274

A. CLASSIFICATION OF SUBJECT MATTER
INV. H04L12/721 H04L12/741 H04L12/935
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-Internal

C. DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|-----------|--|-----------------------|
| X | NOBUTAKA MATSUMOTO ET AL: "LightFlow: Speeding up GPU-based flow switching and facilitating maintenance of flow table", HIGH PERFORMANCE SWITCHING AND ROUTING (HPSR), 2012 IEEE 13TH INTERNATIONAL CONFERENCE ON, IEEE, 24 June 2012 (2012-06-24), pages 76-81, XP032217636, DOI: 10.1109/HPSR.2012.6260831 ISBN: 978-1-4577-0831-2 | 1,3-5, 11,13-15 |
| Y | the whole document ----- -/-- | 2,6-10, 12,16-20 |

Further documents are listed in the continuation of Box C.

See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

| | |
|---|--|
| Date of the actual completion of the international search | Date of mailing of the international search report |
| 5 September 2014 | 15/09/2014 |

| | |
|--|--|
| Name and mailing address of the ISA/ European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016 | Authorized officer Von Der Straten, G |
|--|--|

INTERNATIONAL SEARCH REPORT

International application No

PCT/US2014/036274

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

| Category* | Citation of document, with indication, where appropriate, of the relevant passages | Relevant to claim No. |
|-----------|---|-----------------------|
| Y | ANDREW R CURTIS ET AL: "DevoFlow", SIGCOMM, ACM, 2 PENN PLAZA, SUITE 701 NEW YORK NY 10121-0701 USA, 15 August 2011 (2011-08-15), pages 254-265, XP058006656, DOI: 10.1145/2018436.2018466 ISBN: 978-1-4503-0797-0 abstract page 259, column 2 ----- | 2,6-10, 12,16-20 |
| A | US 2013/163427 A1 (BELIVEAU LUDOVIC [US] ET AL) 27 June 2013 (2013-06-27) abstract paragraph [0044] - paragraph [0102]; figures 1,3-6 ----- | 1-20 |

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No

PCT/US2014/036274

| Patent document cited in search report | Publication date | Patent family member(s) | Publication date |
|--|------------------|-------------------------|------------------|
| US 2013163427 A1 | 27-06-2013 | CN 103999431 A | 20-08-2014 |
| | | US 2013163427 A1 | 27-06-2013 |
| | | WO 2013093860 A1 | 27-06-2013 |
| ----- | | | |