Efficient Conflict Detection in Flow-Based Virtualized Networks

Sriram Natarajan*, Xin Huang†, Tilman Wolf*
*University of Massachusetts, Amherst, MA 01002
Email: {snataraj,wolf}@ecs.umass.edu
† Deutsche Telekom R&D Lab, USA
Email: xin.huang@telekom.com

***Abstract*—In the current Internet, traffic is routed at the level of destination prefixes. The next-generation Internet requires control of traffic at the level of flows or flow aggregates. To accommodate fine-grained control, modern switching substrates (e.g., OpenFlow) maintain forwarding information for each active flow in a flow table. A separate control plane manages flows within a subnetwork by updating this flow information within switches. When using network virtualization, a technique that allows sharing of networking resources among different logical networks, the physical switch and its flow table need to be shared. Current virtualization solutions in OpenFlow do not support hardware isolation in the flow table and thus lead to hidden conflicts and misconfiguration of flows. To maintain network-level flow integrity, we propose two new conflict detection techniques. The first algorithm uses an hybrid hash-trie structure to represent the flow table and determine the conflicting flows using a divide and conquer strategy. The second mechanism uses an ontology based logic inference system to represent and infer the conflicting flow entries. We performed extensive experimental evaluation of both the techniques. For flow table with thousands of flow entries, the proposed techniques effectively resolve the conflicts in approximately 100 milliseconds.**

I. Introduction

In the current Internet, network traffic is routed based on the destination address prefixes. While this approach allows an efficient implementation of shortest-path (and more complex) routing protocols, it does not provide fine-grained control over network traffic. However, many proposals for the future Internet architecture require that the network data plane implement routing and forwarding at the level of individual connections or connection aggregates e.g., for network virtualization [1] or for network services [2].

An increasingly widely used platform for flow-based networking is OpenFlow [3]. OpenFlow is based on commercial switching hardware and clearly separates the control plane from the data plane. An OpenFlow network consists of hardware switches and controllers that communicate with a dedicated protocol to set up and maintain "flow tables" used in packet lookup and forwarding. Each flow entry consists of set of header fields and multiple actions to be performed on the matching packets. Table I shows the 12-tuple header fields defined in the OpenFlow 1.0 [4]. When the switch receives an incoming packet, a matching flow entry is determined in the flow table and the packet is forwarded based on the specified actions. If no flow matches, the packet is forwarded to the "OpenFlow Controller," which adds a new flow entry to the flow table and subsequently all matching packets will be handled by the added flow entry.

Network virtualization, which is the ability to support multiple logical networks on a common infrastructure, is an important aspect of the future Internet [1]. OpenFlow implements virtualization through a "FlowVisor," which can interact with multiple different controllers, each handling a separate network slice. The architecture allows a packet to be a part of multiple slices as mentioned in [5]. While this approach achieves effective virtualization of an OpenFlow network, it also presents an important technical challenge. The parallel operation of multiple OpenFlow controllers implies that multiple entities can install flow entries in the flow table of an OpenFlow switch. It is essential that these entries do not conflict, i.e., do not specify different processing and forwarding actions for the same packets.

To maintain configuration integrity, it is essential to implement an effective isolation mechanism for the flow table between virtual network slices. Current OpenFlow switches do not provide flow table isolation in hardware. Instead, OpenFlow handles the flow conflict problem by assigning a priority to each flow table entry. When using multiple controllers, effective isolation cannot be achieved with priorities since individual controllers may make modifications to the flow table that lead to hidden flow conflicts (i.e., a flow entry with higher priority shadows another flow entry). As a result, virtual OpenFlow networks may exhibit network-level routing that is inconsistent with the view of each controller.

In this paper, we present two new techniques to detect and resolve the flow conflicts, which is essential for effectively managing flow-based (e.g., OpenFlow) networks that support virtualization. The specific contributions of our work are the following: 1) Identification of the flow conflict problem in the context of flow-based virtualized networks, 2) Design of an efficient flow conflict detection system to solve the above problem, and 3) Implementation and experimental evaluation of the performance of the proposed techniques.

The remainder of the paper is organized as follows. Section II states the flow conflict problem in the context of flow-based virtualized networks. Section III discusses the proposed flow conflict detection system. Section IV focuses on the experimental evaluation of the proposed system. Section V

Ingress Port	Ether Src	Ether Dst	Ether Type	VLAN ID	VLAN Priority	IP Src	IP Dst	IP Proto	IP ToS	Src Port	Dst Port

TABLE I
OPENFLOW HEADER FIELDS

Flows	Ingress Port	Ether Src	Ether Dst	Ether Type	VLAN ID	VLAN Priority	IP Src	IP Dst	IP Proto	IP ToS	Src Port	Dst Port	Actions
e1	5	00:24:D7: 63:2C:14	58:B0:35: F6:12:F1	0x800	*	*	*	01*	TCP	*	*	2211	action 1
e2	1	*	*	0x800	*	*	010*	100*	UDP	*	*	2210	action 2
e3	*	*	*	0x800	*	*	101*	011*	*	*	*	2211	action 3
e4	*	*	*	*	*	*	*	*	*	*	*	*	action 4

TABLE II
EXAMPLE FLOW TABLE

discusses the related work. Section VI summarizes and concludes this paper.

II. PROBLEM STATEMENT

A flow-based (e.g., OpenFlow) network virtualization architecture allows multiple logical networks to share the same physical infrastructure. Network virtualization layer (e.g., FlowVisor) allows set of controllers to manage multiple switches per slice. Controllers are responsible for installing flow entries in the assigned domain of switches. In such a design, one physical switch could belong to multiple virtual networks and thus could be controlled by set of controllers, leading to flow conflicts.

The flow conflict detection identifies the conflicting flows in the same flow table. In our work, we model the flow table as $E = \{e_1, e_2, ...e_n\}$, where n is the number of flow entries. Flow entry e_i contains d fields (e.g., in OpenFlow V1.0, $d = 12$). Let r_i^j denote the range of values that are permissible for field j of flow e_i. Hence e_i can be represented as, $e_i = [r_i^1, r_i^2, ...r_i^d]$. Let an incoming packet p be represented as the set of header fields, $p = [f^1, f^2, ...f^d]$, where f^j is the value of field j. Packet p matches flow e_i (i.e., $p \in e_i$), if $\forall j, f^j \in e_i^j$. Flows e_i and e_k conflict each other if a packet p matches both entries (i.e., $\exists\ p, p \in e_i$ and $p \in e_k$). Table II shows an example OpenFlow flow table, where flow e_3 conflicts with e_4 and e_1, flow e_4 conflicts with e_1, e_2 and e_3.

OpenFlow associates a priority field to each flow entry to solve the conflict problem. In an OpenFlow switch, flow entries are stored in the flow table TCAM from the highest priority to the lowest priority order. When a packet is received, the matching flow with the highest priority (i.e., the first matching entry in the TCAM) is chosen and the associated actions are performed. This approach works well when a physical switch is controlled by one controller. However, in the context of virtualized networks, the highest priority match leads to hidden flows and eventually causes a network-level misconfiguration. Hence, when a new flow is added, an efficient flow conflict detection mechanism helps to maintain configuration consistency among multiple virtual networks and thus should be an integral function of network management.

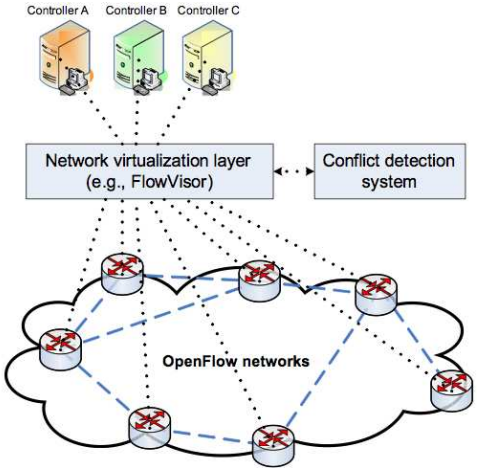

Fig. 1. Flow Conflict Detection.

III. FLOW CONFLICT DETECTION SYSTEM

In this section, we propose a flow conflict detection system that evaluates new flow entries to be added in the flow tables of the physical switches. The working of our conflict detection system is depicted in Figure 1, which shows an example of an OpenFlow-based virtualized network where multiple controllers install set of flow entries on the same set of switches. The network virtualization layer (e.g., FlowVisor) forwards the new flow entries to our conflict detection system, which maintains a database of existing flow entries for all switches in the network domain. The conflict detection system can either include an hybrid hash-trie based algorithm or an ontology based logical inference mechanism to evaluate and report the set of all conflicting entries. The network virtualization layer then resolves the conflicting flows, ensuring consistency in the flow table. In the next section, we describe both methods and evaluate their performance.

A. Hash-Trie based Conflict Detection

The hash-trie based conflict detection algorithm (HTCD) exhibits a divide and conquer design paradigm to detect flow conflicts. In our approach we handle the exact match and wildcard entries with different representations as discussed

Flows	f^1	f^2	f^3	f^4	f^5	f^6	f^7	f^8	f^9	f^{10}	f^{11}	f^{12}
e1	1	1	1	1	1	1	1	1	1	1	1	1
e2	0	1	1	1	1	1	0	0	0	1	1	1
e3	1	1	1	1	1	1	1	1	1	1	1	1
e4	1	1	1	1	1	1	1	1	1	1	1	1

TABLE III
INTERSECTION MATRIX

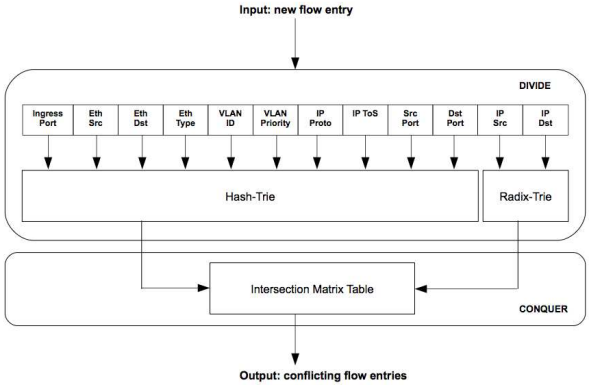

Fig. 2. Hash-Trie based Conflict Detection Workflow.

below. Figure 2 shows the workflow of our conflict detection algorithm in the context of OpenFlow-based virtualized networks.

- *Divide*: The algorithm first divides each flow entry into set of header fields and detects conflicts on each field separately. We categorize the conflict detection process based on the type of the header field as follows: 1) The prefix-based fields (e.g., source IP and destination IP) are represented using radix trie structure to determine the conflicts, and 2) The exact value fields (e.g., the remaining 10 fields in OpenFlow) are represented using hybrid hash and trie structure. The divide step returns the set of FlowIDs (in our design, each flow is associated with a FlowID), which represents the conflicts in each field.
- *Conquer*: The algorithm then combines the returned set of FlowIDs from all fields and determines the conflicting flow entries using an Intersection Matrix (IM) table.

1) Radix Trie: In our algorithm, the prefix-based fields are represented using radix trie structure for the following reasons: 1) A trie structure is a natural selection to represent prefixes, as discussed in [6], and 2) A radix trie maintains a good balance between lookup time, update time, and space requirement.

The conflict detection process determines all the nodes that overlap with the new entry in the corresponding IP field. The detection process is as follows:

- We determine the Matching Node (MN) using the Longest Prefix Match (LPM) mechanism.
- For prefix-based fields, all parent nodes of the MN and all child nodes of the MN (subtrie of MN) represent flow entries conflicting with the input flow. Thus, we record all nodes along the path while finding the MN. As a result, the process returns all FlowIDs associated with the MN, the parents of MN, and the subtrie of MN.

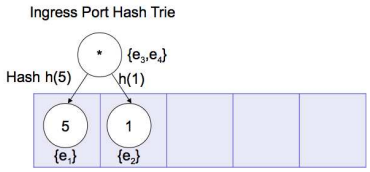

Fig. 3. Hash-Trie Representation

2) Hash Trie: The exact value header fields are stored in a unique hash and trie combination structure. The root of the hash-trie represents the wildcard entries. The subtrie of the root in the trie structure stores all the exact match entries. For performance improvements, the subtrie entries are stored using a hash table as shown in Figure 3. In the hash structure, multiple flow entries can be mapped to the same value, hence to maintain uniqueness, each header field value is added as hash key and the array of FlowIDs are stored as hash values. Hence, the hash table can be represented as a (Field Value, FlowIDs) pair. For each new flow entry, the corresponding field values are compared. If the value is a wildcard, all FlowIDs are returned. Else, if the value matches with an exact value in the hash, the FlowIDs associated with the exact match field and the FlowIDs associated with the wildcard entry in the root are returned. When no match is found, the new (key,value) pair will be added to the hash table.

3) Intersection Matrix: As described above, each field determines the conflicting flow entries and returns the associated FlowIDs. The IM then combines the results from all fields and determines the intersection of FlowIDs. The rows in the IM table represent the flow entries and the columns represent the header fields. All entries in IM are initialized to zero, which denotes no conflicts. For every FlowID returned, a corresponding flow entry in the IM for the specified header field is set to 1. An entire row of 1's in the IM corresponds to the flow entry conflicting with the input. For a new flow entry [5,*,*,*,*,*,10*,01*,TCP,*,*,*], Table III shows the conflicting flows in the IM, for the flow table in Table II. As a result, IM reports flows e_1, e_3, and e_4 to be conflicting with the new flow.

B. Ontology based Conflict Detection

In this section we introduce our second technique, an ontology based conflict detection (OCD) mechanism to determine the conflicting flow entries. An ontology based logic system provides a standardized mechanism for knowledge representation and automated reasoning (inference) with well defined syntaxes and semantics. A representation language transforms the knowledge from the real world into a logic representation using the syntaxes and a reasoning methodology deduces and infers the required solutions from the built up knowledge.

In our work, we consider the Description Logic (DL) mechanism [7], which provides a logical formalism to represent the flow table ontology and determine the conflicting flow

Constructor	DL Syntax	Manchester OWL Syntax
Intersection	$e_i \sqcap ... \sqcap e_k$	e_i *and...and* e_k
Union	$e_i \sqcup ... \sqcup e_k$	e_i *or...or* e_k
Complement	$\neg e_i$	*not* e_i
Max Cardinality	$\leq_n P$	$maxP_n$
Min Cardinality	$\geq_n P$	$minP_n$
Exact Cardinality	$=_n P$	$exactlyP_n$
Universal Quantifier	$\forall_n P$	$only\ P_n$
Existential Quantifier	$\exists_n P$	$some\ P_n$

TABLE IV
DESCRIPTION LOGIC SYNTAXES

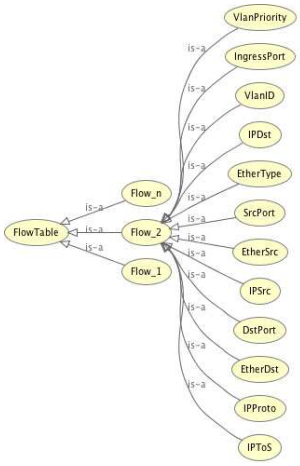

Fig. 4. Flow Table Class Taxonomy.

entries. The DL based logic representation consists of classes (concepts), individuals and properties (relations or roles). The classes in the logic represent a set of individuals and properties represent a relationship between individuals. Complex classes can be constructed from a set of atomic classes using the description logic constructors. The set of available constructors and the corresponding DL and Manchester OWL syntaxes are shown in Table IV (P denotes the set of properties).

1) Requirements: The following logic representation and reasoning requirements are necessary to determine the conflicting flow entries: 1) A logical component to represent the flow entries and their header fields in the flow table (E), 2) A set of defined object and data properties for each fields that ensures consistency and functioning of the system, and 3) A logic function to determine the flow conflicts by querying an ontology based reasoner.

2) Representation: The flow table entities are represented using a set of OWL components as shown here. A class (representing the flow table) consists of set of individuals (representing flow entries). An individual should satisfy certain conditions (properties) to be a member of a class. The permissible value for each header fields are represented using the object properties and the data values for each field are represented using the data properties in the OWL ontology. An ontology based flow table class taxonomy is shown in Figure 4. The $FlowTable$ represents a class that consists of set of individuals ($Flow_1, Flow_2, ...Flow_n$). Each individual consists of d object properties (e.g., hasDomainRange) that checks if a given individual satisfies the range of values that are permissible for every field and d data properties, representing the header field values (e.g., hasIngress, hasIP-Proto,..hasDstPort). The following properties provide the logic representation to determine the conflicting entries.

- **Domain Property:** The domain property ensures that each field value is within the range of values that are permissible for the field j, by creating a constraint on the individuals with a set of object property restrictions. The restrictions are based on the cardinality constructors as shown in Table IV ($\leq, =, and \geq$). A domain property is constructed for each field as per the OpenFlow 1.0 specification and is evaluated when a new individual (field) is added to the class (flow) as follows.

$$FlowValue \equiv (\sqsubseteq FlowClass) \sqcap_1^d (\exists hasFieldDomainRange_j) \quad (1)$$

  The above representation specifies that the $FlowValue$ is a subsumption of the $FlowClass$ and the data instance for each field is within the permissible value for each header field in the flow entry.
- **Data Property:** A set of data properties link individuals to the associated data values. Each data property (denoting a field) is associated with a specific data type (e.g., integer, string). When an individual satisfies the domain property, the corresponding data value for every field is added to the data property of the individual (flow entry). An individual is associated with d (For the OpenFlow 1.0 specification, d=12) data properties and is represented as follows.

$$FlowValue \equiv \sqcap_1^d (\exists (hasDataProperty_j(value)) \quad (2)$$

3) Reasoning: The description logic based query language provides an effective mechanism in searching the defined flow table ontology and determine the conflicting flow entries. The following steps indicate the querying process using an OWL reasoner:

- The entities (classes, properties and individuals) representing the flow entries are first loaded in the flow table ontology.
- Once loaded, a class hierarchy is built and the data structures representing the class instances are initialized.
- The input flow request is transformed into a description logic syntax and a query is sent to the reasoner to evaluate and determine the conflicting flow entries.

To handle the exact matching fields and the wildcard header fields in an input flow, the class expression that represents the logical query is generated as follows:

a) Exact Match: The input header field value conflicts with the flow entries that exactly matches with the input and with all the flow entires that are wildcard for the specified header field. The class expression (conflict) corresponding to the exact matching input for $field_j$ is generated as:

$$Field\ Conflict \equiv (hasDataProperty_j = Input \sqcup *) \quad (3)$$

The intersection ($\sqcap$) of all field values ($1 \leq j \leq d$) gives the conflicting flow entries and is represented as:

$$Flow\ Conflict \equiv \sqcap_1^d (hasDataProperty_j = (Input \sqcup *)) \quad (4)$$

b) Wildcard Match: A wildcard input value conflicts with all the flow entries for the corresponding header field. When all field values of the input flow entry is wildcard, then all the flow entries in the flow table conflicts as shown in Table II (e.g., Flow $e4$). The class expression (conflict) corresponding to the wildcard input for $field_j$ is generated as:

$$Field\ Conflict \equiv \exists (hasDataProperty_j(*)) \quad (5)$$

When all the field values are wildcard, the intersection of all field values ($1 \leq j \leq d$) gives the conflicting flow entries and is represented as:

$$Flow\ Conflicts \equiv \sqcap_1^d (\exists hasDataProperty_j(*)) \quad (6)$$

IV. Evaluation

In this section we evaluate and discuss the performance of our proposed flow conflict detection techniques with analysis and implementation results.

A. Analysis

1) Search Time:

a) HTCD Algorithm: The total search time to determine the conflicting flows using the HTCD algorithm involves the following three stages: 1) Cumulative search time for the exact value fields, 2) Aggregate search time for the prefix-based fields and 3) Intersection Matrix (IM) evaluation time.

The hash-trie search time is dominated by the hash lookup time. The load factor, α (u/m) is given as the ratio of the number of unique field values (u) over the number of slots (m). For an efficient hash implementation, α is usually kept below a small constant, preferably less than 1 as shown in [8]. Thus, the total hash search time using the HTCD algorithm is $\mathcal{O}(1)$. For set of d fields represented by the hash-trie structure, the cumulative hash-trie search time is $\mathcal{O}(d)$. The radix trie search time is determined as a combination of LPM search time and the subtrie search time. For each radix trie, the LPM search time is proportional to $\mathcal{O}(D^j)$, where D^j denotes the depth of the radix trie structure of field j. When the depth of a prefix trie is close to 32, the worst case LPM search time is $\mathcal{O}(1)$. For d fields, the search time is $\mathcal{O}(d)$.

For each radix trie, the subtrie search time is proportional to the size of the subtrie rooted from the MN. In the worst case scenario, this is proportional to the size of the whole radix trie (i.e., when MN is the root of the trie structure), that is, $\mathcal{O}(n)$, where n denotes the number of entries in the flow table. Assume that d fields are represented by the radix trie structure, then the worst case scenario for the subtrie search time is $\mathcal{O}(nd)$. Thus, the aggregate worst case radix trie search time is $\mathcal{O}(nd)$, where n is the number of flow entries and d is the number of prefix-based header fields. The IM combination time is proportional to the number of flow entries and the number of header fields, that is, $\mathcal{O}(nd)$. Combining the above three stages, the total worst case search time is proportional to the number of flow entries and the number of header fields and is given by $\mathcal{O}(nd)$.

b) OCD Inference System: The total search time to determine the conflicting flow entries using the OCD inference system involves the following two stages: 1) Building and initializing the class instance data structure, and 2) Querying the ontology with an effective OWL based reasoner. The class initialization time to build the data structure and reasoner querying time are dependent on the number of entities defined in the ontology. In our case, we have n individuals, with each individual containing d properties. Hence the worst case total search time using the OCD inference system is (similar to the HTCD algorithm) proportional to the number of flow entries and the number of header fields and is given by $\mathcal{O}(nd)$. In both the techniques, for the OpenFlow scenario the number of header fields, d, is constant. Hence the total worst case search time is dependent only on the number of flow entires and is given by, $\mathcal{O}(n)$.

2) Memory Requirements:

a) HTCD Algorithm: The worst case memory requirement for the hash-trie structure (exact match fields), radix trie (prefix-based fields) and the Intersection Matrix is proportional to the number of flow entries in the flow table and the number of header fields, and is given by $\mathcal{O}(nd)$. In the OpenFlow scenario the number of header fields, d, is constant (10 for hash-trie structure and 2 for radix trie). Hence the total worst case memory requirement is dependent only on the number of flow entires and is given by, $\mathcal{O}(n)$.

b) OCD Inference System: The worst case memory requirement for the ontology based mechanism is proportional to the number of individuals in the flow table class and the associated properties (object and data) for each individual, and is given by $\mathcal{O}(nd)$. As shown in the HTCD algorithm, the worst case memory requirement is dependent only on the number of flow entires and hence is given by, $\mathcal{O}(n)$.

B. Implementation

In this section we discuss the experimental setup, performance metrics and evaluation results of both the HTCD and OCD mechanisms in the OpenFlow scenario.

1) Setup: The experiment was conducted in our lab using a conflict detection server based on an Intel Core 2 Duo processor (2.53 GHz, 4 GB memory, and 3 MB L2 cache). Since performance evaluations using real flow entries are restricted by the size and structure, we generate synthetic flows that reflect the characteristics of the flows used in real OpenFlow scenarios. The synthetic flow entries are generated as follows:

The 5-tuple fields (i.e., IP source, IP destination, source port, destination port, IP protocol) are generated using Class-

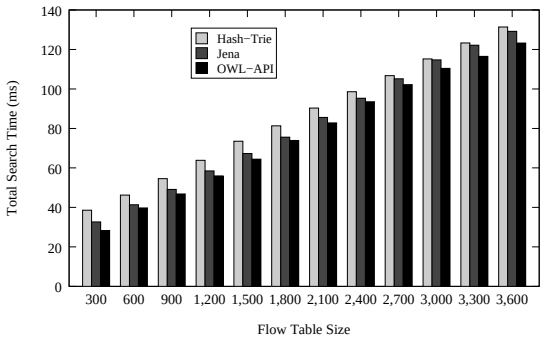

Fig. 5. Flow Table Size vs Total Search Time (ms).

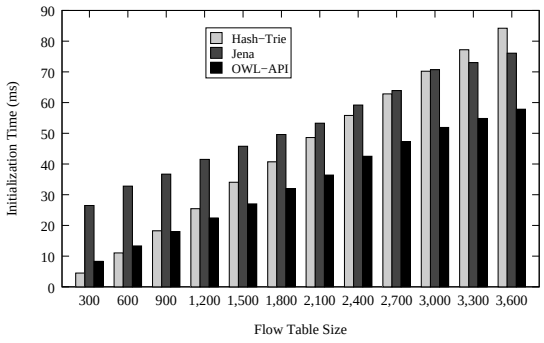

Fig. 6. Flow Table Size vs Initialization Time (ms).

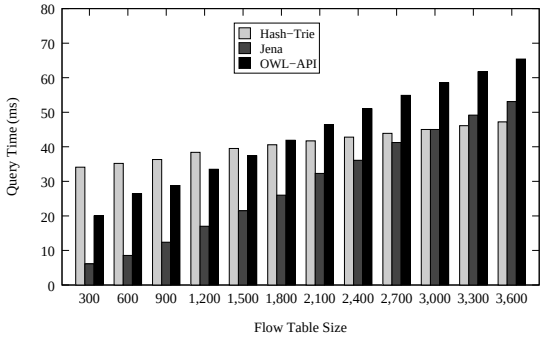

Fig. 7. Flow Table Size vs Query Time (ms).

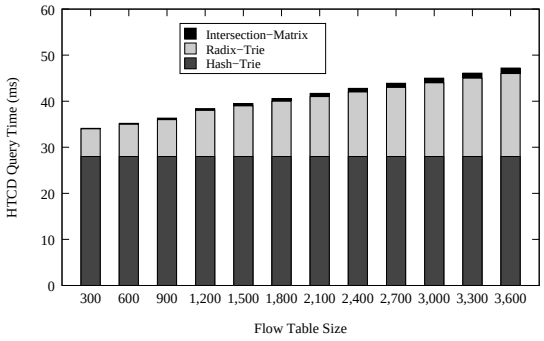

Fig. 8. Flow Table Size vs HTCD Query Time (ms).

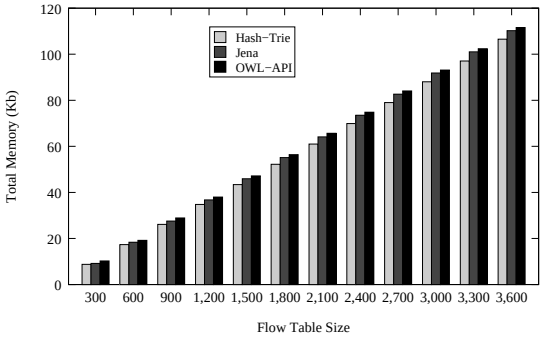

Fig. 9. Flow Table Size vs Total Memory (Kb).

Bench [9], a packet classification benchmark. The source and destination ethernet addresses are generated by taking a random value in the range of valid MAC addresses, ensuring significant wildcard entries added to the flow table. A typical OpenFlow hardware switch (e.g., NEC IP8800) has 48 1G ports and 4 10G ports. Hence the valid ingress port numbers in our experiment were from 1 to 52. According to [4], the Ethernet type values that are most widely used in OpenFlow networks are $0x0806$ (i.e., ARP) and $0x0800$ (i.e., IP). The IP ToS field consists of 6 bits in the IP header, and hence ranges from 0 to $2^6 - 1$. VLAN id and VLAN priority fields are used for packets with Ethernet type 0x8100 and hence are set to wildcard.

The experiment was conducted for different flow table sizes (i.e., number of flow entries) ranging from 300 to 3600 with step size of 300. During the flow entry set generation process, we also ensured that the entries have sufficient number of wildcard fields to test the efficiency of our conflict detection system. The HTCD algorithm was implemented using the MultiMap hash functionality provided in the C++ STL library and the RadixTrie Java library. For an extensive set of evaluations, the ontology based inference system was implemented using the OWL-API library and using the Jena library. For the OWL-API, the Hermit OWL reasoner was used to infer the flow conflicts by generating queries using the Manchester OWL syntax. For the Jena implementation, we used the inbuilt Jena inference engine to query the ontology using OWLReasoner.

C. Results

1) Search Time: The total search time performance for the three techniques (HTCD, OWL-API, Jena) are shown in Figure 5. The OWL-API based ontology implementation performs better than compared to the HTCD and Jena based technique. To further analyze the search time contributions of each entities, we breakdown the total search time into the initialization time and query time. Figure 6 shows the initialization time for the three techniques. For smaller flow table sizes, the HTCD scheme initializes faster. However, with increase in flow table size the OWL-API technique initializes the individuals and properties in the ontology faster than the other two techniques. Figure 7 shows the query time for different flow table sizes. The query time for both ontology

schemes (OWL-API and Jena) linearly increases with increase in the flow table size. For the HTCD algorithm: 1) The hash search time is constant, 2) The radix trie search and the Intersection Matrix evaluation time increases linearly with the flow table size and hence the total search time also increases linearly as shown in Figure 8. For larger flow table sizes, the query time for HTCD algorithm performs better than the ontology based techniques. On the average case, the OWL-API based technique is 8.5% faster than the HTCD technique and 4% faster than the Jena based model in determining the flow conflicts.

2) Memory: The memory requirement for the three techniques were evaluated and is shown in Figure 9. The total memory required for all the three techniques (HTCD, OWL-API and Jena) linearly increases with increase in the flow table size. The results prove our analysis in Section IV-A2. On the average case, the HTCD has 6.5% less memory requirement than the OWL-API scheme and 4.5% less memory requirement than the Jena scheme. From the above analysis and our experimentation results we see that for efficient search time requirements, the OWL-API based ontology model performs the best and for limited memory constraints, the HTCD based algorithm performs better.

V. Related Work

Future Internet architectures are being actively explored in the networking research community [10]. Network virtualization is at the core of many proposed solutions [1] and prototypes that are being developed [11]. OpenFlow is a potential substrate for such a future Internet [3]. The problem of managing flows within OpenFlow has received attention in related work [12]. Reference [13] addresses the requirement of flow-based policy languages and policy representations. [14] shows the flexibility of decoupling policy and configuration using OpenFlow. Such solutions define flow-based policies and emphasize the requirement of expressive policy specification languages to maintain consistency. FlowChecker [15] proposes a verification tool to determine OpenFlow flow table misconfigurations using a Binary Decision Diagram. Improved flow matching schemes have been proposed using FPGA-based techniques [16]. A conflict detection technique for multi-dimensional rule sets have been explored for packet classification in general [17]. However, the fundamental problem of avoiding hidden, conflicting flow entries due to wildcard fields in a single flow table has not yet been addressed in the context of flow-based virtualized networks. Our work provides a practical solution to this problem.

VI. Conclusion

To support future Internet architectures, control over routing of individual flows or flow aggregates is essential. OpenFlow provides a useful substrate for managing network traffic at the flow-level. With a move toward network virtualization, it is important that parallel OpenFlow controllers do not introduce conflicts into the flow tables of OpenFlow switches. In our work, we have identified this problem of hidden conflicts. We present multiple methodologies for identifying and resolving the conflicts. We observe a performance trade-off between the proposed techniques. The ontology based technique (OWL-API) performs better in the initialization time, while the hash-trie algorithm performs better in the query time. The performance evaluation of our mechanisms show that we can effectively determine flow conflicts with low time and space complexities. Thus, our work can solve a key management problem in any flow-based virtualized network and ensure correct network-level operation of a virtualized network infrastructure.

References

[1] T. Anderson, L. Peterson, S. Shenker, and J. Turner, "Overcoming the Internet impasse through virtualization," *Computer*, vol. 38, no. 4, pp. 34–41, Apr. 2005.

[2] T. Wolf, "In-network services for customization in next-generation networks," *IEEE Network*, vol. 24, no. 4, pp. 6–12, Jul. 2010.

[3] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[4] *OpenFlow Switch Specification Version 1.0*, OpenFlow Switch Consortium, Dec. 2010.

[5] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," Department of Computer Science, Stanford University, Technical Report, 2009.

[6] P. Gupta and N. McKeown, "Algorithms for packet classification," *IEEE Network*, vol. 15, no. 2, pp. 24–32, Mar. 2001.

[7] F. Baader, "Description logic terminology," in *The Description Logic Handbook: Theory, Implementation, and Applications*, F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds. Cambridge University Press, 2003, pp. 485–495.

[8] S. Kumar and P. Crowley, "Segmented hash: an efficient hash table implementation for high performance networking subsystems," in *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, Princeton, NJ, Oct. 2005.

[9] D. E. Taylor and J. S. Turner, "Classbench: A packet classification benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, Sep. 2007.

[10] A. Feldmann, "Internet clean-slate design: what and why?" *SIGCOMM Computer Communication Review*, vol. 37, no. 3, pp. 59–64, Jul. 2007.

[11] *Global Environment for Network Innovation*, National Science Foundation, http://www.geni.net/.

[12] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee, "Devoflow: Cost-effective flow management for high performance enterprise networks," in *Proc. of The Ninth ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, Montreal, CA, Oct. 2010.

[13] T. L. Hinrichs, N. Gude, M. Casado, J. C. Mitchell, and S. Shenker, "Expressing and enforcing flow-based network security policies," Department of Computer Science, University of Chicago, Technical Report, 2008.

[14] N. Feamster, A. Nayak, H. Kim, R. Clark, Y. Mundada, A. Ramachandran, and M. bin Tariq, "Decoupling policy from configuration in campus and enterprise networks," in *Proc. of the 17th IEEE Workshop on Local and Metropoliton Area Networks (LANMAN)*, Long Branch, NJ, May 2010, pp. 1–6.

[15] E. Al-Shaer and S. Al-Haj, "Flowchecker: configuration analysis and verification of federated openflow infrastructures," in *Proc. of the 3rd ACM workshop on Assurable and usable security configuration (SafeConfig)*, Chicago, IL, Oct. 2010.

[16] W. Jiang, V. Prasanna, and N. Yamagaki, "Decision forest: A scalable architecture for flexible flow matching on fpga," in *Proc. of the 20th IEEE International Conference on Field Programmable Logic and Applications (FPL)*, Milano, Italy, Aug. 2010.

[17] F. Baboescu and G. Varghese, "Fast and scalable conflict detection for packet classifiers," in *Network Protocols, 2002. Proceedings. 10th IEEE International Conference on*, nov. 2002, pp. 270 – 279.