SPIRIT: A Framework for Profiling SDN

Heedo Kang, Seungsoo Lee, Chanhee Lee, Changhoon Yoon and Seungwon Shin
Graduate School of Information Security, School of Computing, KAIST
Email : {kangheedo, lss365, mitzvah, chyoon87, claude}@kaist.ac.kr

***Abstract*—Software-Defined Networking (SDN), which separates the control and data plane of network, is strongly considered as a promising future networking architecture. Compared with legacy networking architecture, it allows to enable a variety of innovative network functions at much less cost and effort. Accordingly, each component of SDN is also being rapidly realized, and one of the most noticeable SDN component implementations would be SDN controllers, such as ONOS or Floodlight. One advantage of these SDN controllers is capability of hosting various network applications to enable innovative network functions; however, it is crucial to analyze these applications before the actual deployment as they may directly affect the performance of the managed network. To be more specific, SDN applications may contain performance bugs that unnecessarily consume significant system resource or produce critical bottlenecks in the controller. In this paper, we introduce an automatic SDN application profiling framework, SPIRIT, which reduces the human effort in revealing any performance bugs that might exist in SDN applications. In order to show the effectiveness of our framework, we reveal new performance bugs exist in ONOS and Floodlight applications.**

I. Introduction

One of the distinctive advantages of SDN over legacy networking is that SDN monitors and manages all the underlying network devices in a centralized manner. In particular, such centralized architecture offers a bird's eye view of the network infrastructure, and accordingly, it is possible to easily compute optimal network paths. Meanwhile, it introduces disadvantages as well. One disadvantage is that the control plane itself becomes a potential bottleneck or a single point of failure, and any performance overhead caused within the control plane will directly affect the quality of network services provided in the managed network.

Some researchers have already noticed this performance issue, and they have investigated the performance of SDN by measuring the performance of several SDN control planes (or network operating systems, NOS) [1]. Their results present that one NOS instance can handle more than millions of new flows in a second [1], and it implies that the performance issue of SDN is not so significant. However, surveying their results, we notice that there is a missing point that should be considered. Most previous studies investigating the performance of SDN have measured the performance of SDN, when they run quite simple network applications, such as layer-2 switching. Indeed, SDN can manage a network with such simple network applications, but in reality, network administrators would use more advanced network applications (e.g., load balancing and cost-based routing) as well as those simple applications. What if a network administrator deploys a much complex and heavy network application? Would it be able to handle multi-millions of new network flows as the layer-2 switching application was capable of?

This paper appeared in the CoolSDN Workshop of ICNP 2015.

These questions could be regarded as trivial as a simple technical issue; however, they lead to a more fundamental research question - *how can we understand the performance of NOS and its applications in detail?* The answers to this question will allow us to design more efficient and effective SDN applications and also help us to understand how to properly operate our SDN networks. This has motivated us to propose an automatic SDN application profiling framework - with the name of SPIRIT - that can analyze the operations of each network application and NOS itself automatically. Basically, SPIRIT employs the concept of dynamic program analysis (or profiling) to investigate the operations of NOS and applications minutely, and it provides a convenient and flexible environment to make the analysis process simple and easy.

With SPIRIT, people can understand the overall performance of the current adopted applications on NOS and monitor other critical features (e.g., hotspot and critical path of an application) clearly, and our tool enables people to do this job without much complexity. This kind of analysis usually requires setting up a test environment by emulating a real world scenario. For example, we need to construct a network topology and generate network traffic to induce Packet-In messages. SPIRIT relieves people from the burden of these cumbersome tasks, and it helps them to solely focus on the analysis itself.

To verify the effectiveness and efficiency of SPIRIT, we have analyzed real world NOSs and their applications (e.g., Floodlight controller and its topology managing application) with our tool. Through this analysis, we discover the bottlenecks in the target network applications, and we draw ideas of reducing the effect of bottleneck based on the analysis. Our work is still in progress, and we are planning to improve SPIRIT to provide more useful information and even suggest an idea of how we can enhance the performance of the analyzed application.

In summary, the contributions of this paper are as follows:

- We suggest the methodology for profiling SDN applications, and we design a framework that automatically profiles SDN applications without the human intervention.
- We investigate the performance and operations of several well-known SDN controllers with advanced and complicated network applications.
- We reveal the bottlenecks (or hotspots) exist in SDN applications and discuss about the results.

II. Problem Statement

A. Motivating Example

Understanding the performance of a network application running on an SDN environment is very important, because the overall network throughput highly depends on the performance of a network application. In this context, there are previous studies discussing about this issue [1], yet they mostly check the performance of a simple network application (i.e., learning switch). Then, what about the performance of other applications? and are they similar or different to each other?

To understand any possible performance gaps among different network applications, we measure the performance of two different network applications - learning switch and forwarding with routing path calculation - that are provided by Floodlight. In this experiment, we use *cbench* [2] to emulate a network with 64 switches and generate control traffic. The result of the experiment is presented in Table I, and the learning switch application outperforms the forwarding application by three times. As shown, the performance of SDN applications differs from each other, and it is not surprising because each application has different computational logics and bottlenecks. Thus, to enhance the overall network performance, each application must be carefully analyzed and any bottlenecks exist in the application must be revealed and improved.

Application	**Throughput**
Learning Switch	130,804.35 responses/sec
Forwarding (routing path)	48,328.79 responses/sec

TABLE I. THROUGHPUT OF FLOODLIGHT APPLICATIONS

B. Research Challenge

In this paper, we aim to profile basic applications of some well-known NOSs to reveal their bottlenecks. Not reinventing the wheel, we have attempted to profile applications leveraging existing profiling tools; however, we have noticed that such tools are not suitable for profiling SDN applications due to the following reasons:

1) The Network Composition Challenge: Since most of SDN applications operate reactively to the network events generated from the data plane, the performance of SDN applications basically depends on the network status. Therefore, in order to properly profile the applications, they must be profiled under various network topologies using network emulation tools and manually generated network traffic, which is a time-consuming and daunting task.

2) The Language-Specific Challenge: Most existing software profiling tools are specialized in analyzing the programs written in a certain programming language. If the user wants to use the existing tool for profiling the SDN application, the user have to check if the tool supports the language of target NOS as each NOS is written in different languages, such as C, Java, or Python.

3) The Information Sharing Challenge: Most of the popular NOSs implement fundamental network management services, and SDN applications often leverage those services to carry out various tasks. For example, ONOS has a dedicated device management service by default and most ONOS applications use that service to access and control the network devices. In other words, such core services are commonly used by various applications. If a performance bug exists in one of those core modules, the bug will repeatedly appear in the diagnosis results for different SDN applications. Thus, such redundant detection results make the profiling process inefficient.

In order to solve the aforementioned challenges, a new type of automated profiling framework for SDN application must be presented to enhance the user satisfaction. Thus, we aim to suggest a new automated profiling framework, which is specialized on SDN in this research.

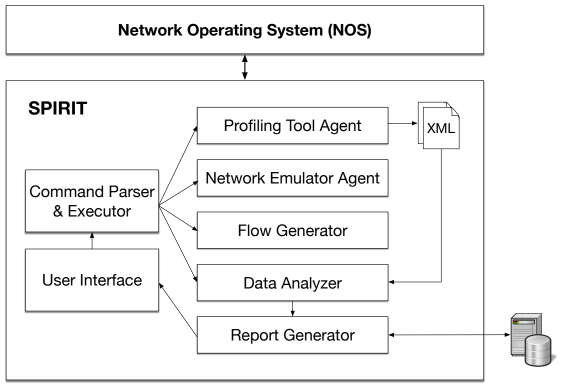

Fig. 1. Architecture of the SPIRIT

III. SYSTEM DESIGN

A. Overall Architecture

The basic idea of the SPIRIT is to automatically analyze the bottleneck of SDN applications running on NOS and provide the information about found critical path and hotspot of SDN applications to the user. As shown in Figure 1, our framework consists of seven modules. We detail the operation of each module as follows.

User Interface principally provides a graphical interface for the user to help him use intuitively. The user can set the configuration information about target NOS, network topology and so on. When the user completes their own environment setting, the user can initiate the profiling of the SDN application from SPIRIT.

Command Parser & Executor is kind of a command center. It parses the user inputs received by user interface module, and then executes modules of the SPIRIT for profiling with sending the input variables to each module.

Profiling Tool Agent manages a well-known profiling tool and adopts some APIs provided by the tool for profiling target NOS. When command parser & executor module executes this agent, it is attached to the instance of target NOS automatically to collect profiling data of the SDN application. Then, it measures the latency time and CPU resource usage of each application. These measured data are stored as XML file automatically.

Flow Generator consists of a flow initiator and an ARP responder. The flow initiator generates the distinct flow per second to the network based on user demands continuously. The ARP responder sends proper response messages to the captured ARP request packet because the flow initiator first requests a mac address of destination before the sending packets.

Network Emulator Agent manages a well-known network emulator and adopts some APIs provided by the emulator. It provides the environment for automatic configuration the virtual network by constructing the virtual network what user wanted easily. After the virtual network is configured by the user, the agent tries to create the virtual network and connect to the NOS.

Data Analyzer parses the XML file generated by the profiling tool agent, which contains profiling information about target NOS. To find the hotspot of the each SDN application, we have applied Algorithm 1 and 2 on this module. We

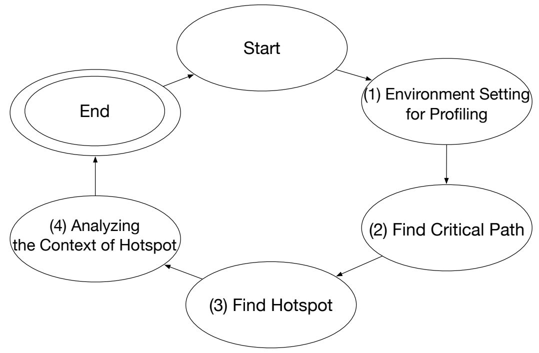
Start
(1) Environment Setting for Profiling
(2) Find Critical Path
(3) Find Hotspot
(4) Analyzing the Context of Hotspot
End

Fig. 2. State diagram of the profiling step

will detail these algorithms in the next subsection. Also, this module analyzes the basic context of found critical path and hotspot automatically.

Report Generator module sends the query to an external database to check whether the same hotspot information exists or not. If the information is found, this module will get the information and displays it to the user.

Operational Scenario. After the user completes setting configuration through *User Interface* module, the *Command Parser & Executor* conducts profiling target NOS. First, it executes the profiling tool, *Profiling Tool Agent* module. Then, SDN application needs the network flow for profiling, *Flow Generator* module creates and sends the network packet to the configured network. Because most of the NOS pull down the rule to the network devices based on the knowledge learned from ARP service, *Flow Generator* module also automatically responses to the arp packet that is coming from the network for imitating the real network environment. And then, attached profiling tool starts profiling when flow generating is started, and profiling data is stored as XML file if profiling is terminated. The saved XML file is parsed by *Data Analyzer* module. This module finds the critical path and hotspot of the desired application automatically, which need to be profiled by the user. Also, it analyzes the context of the found hotspot and critical path based on each class specification. When the critical path and the hotspot of the specific application are found completely, *Report Generator* module explores the external database that stores the various findings about hotspots by many researchers for finding whether information about found hotspot is stored or not. If information exists about found hotspot on the external database, *Report Generator* module carries it to the SPIRIT and informs to the user.

B. Profiling Methodology

SDN application profiling methodology is not differed significantly from the general method of application profiling. However, because SDN application is kind of network program, some steps have to be added to profiling methodology. The overall state diagram that we have performed for profiling SDN application is shown in Figure 2, and description of each step is following.

(1) The first step for the SDN application profiling is to set the environment. For doing that, we first run the NOS such as Floodlight [3] and ONOS [4]. Then, we attach a profiling tool to the running instance of the NOS and organize the virtual network topology for imitating the realistic network. Also, we construct the not only network topology but also network conditions (e.g., a link up/down), because network condition may affect SDN application performance. After the network configuration is completed, we generate distinct flow to the network and start the recording of CPU.

Algorithm 1 Find the Ciritical Path of each NOS Application

```
procedure FINDCIRITICALPATH($Node$)
    $Highest = null$
    $Child = null$
    if $Node.NumberOfChild == 0$ then return
    for each integer $i$ in $Node.NumberOfChild$ do
        $Child = Node.ChildNode[i]$
        if $i == 0$ then
            $Highest = Child$
        else if $i > 0$ then
            if $Highest.totaltime < Child.totaltime$ then
                $Highest = Child$
    $CriticalPath.Element+ = Highest$
    $FindCriticalPath(Highest)$
```

(2) The second step is to figure out the critical path of the specific applications by analyzing the profiling data to find out performance critical applications. The Algorithm 1 is a pseudo code for discovering the application's critical path on the entire call graph. A brief word of the Algorithm 1 is just searching a node on the entire call graph that has the highest total execution time among the same level nodes and adding the found nodes to critical path from the root node to the leaf node repeatedly.

(3) The third step is identifying any hotspots on the critical path. This step is divided into four detailed work phase. The first phase is the call graph pruning that removes unnecessary nodes on the entire call graph. The second phase is getting subgraphs to get the call path of major SDN application. The next phase is to find the critical paths of the applications based on the color information of the function node, and the last phase is about to find the hotspots based on the total time of each node.

(4) The last step is analyzing the found hotspot of each application for understanding the context.

All profiling step has to be repeated a few times with varying the number of network nodes to identify the critical hotspots of each application that get hotter respect to the size of a network. The critical path of some applications can be presented as the same result regardless of network conditions, in this case, the Algorithm 2 shows the example of how to figure out the most hotspot on two distinct critical paths which one is found at small network and another is extracted from the large network environment. Conceptually, the Algorithm 1 performs finding the node, which has the highest value by subtracting the inherent time of each node between two distinct critical paths. And conversely, it is possible that some applications have different critical path according to change the network size. In this case, the user has to find the critical path on the large network profiling data firstly, and then, find the same critical path on the small network profiling data. The process of the finding the hotspot, in this case, is same as the Algorithm 2.

In order to avoid any redundancy of manual profiling

Algorithm 2 Reveal the Hotspot on Two Distinct Critical Path

```
procedure FINDOUTHOTSPOT($Path_1, Path_2$)
    $HotSpot = 0$
    $Temp = 0$
    if $Path_1.Nodes == Path_2.Nodes$ then
        $Temp = Path_1[0].Inherent - Path_2[0].Inherent$
        for each integer $i$ in $Path_1.NumberOfNode$ do
            $Sub = Path_1[i].Inherent - Path_2[i].Inherent$
            if $Temp < Sub$ then
                $Temp = Sub$
                $HotSpot = i$
    return $Path_1[HotSpot].Node$
```

steps, we develop a new automated SDN application profiling framework, which profiles SDN applications without human intervention and offers flexible profiling environment to the user. Furthermore, this framework could be extended to suggest possible improvements that could be made to reduce the impact of the bottlenecks. The summarized contributions of the our framework are as following.

- Our framework offers easy and comfortable SDN application profiling environment by providing automated SDN application profiling process. With the existing profiling tool, user has to conduct annoying tasks manually for SDN application profiling, such as attaching the profiling tool to NOS, organizing the network, creating the flow and analyzing the profiled data. However, our framework performs all of the things automatically.
- Our framework provides flexible profiling environment. While user has to find profiling tool that can support written programming language of NOS previously, such work does not necessary anymore using the our framework.
- Our framework offers a possible solution to enhance the performance of detected hotspot using information sharing system. Information sharing system in the our framework stores the possible solution that is suggested by others and if the someone detects same hotspot when they do profile SDN application, user can know solutions about found hotspot easily through the stored possible solution at the our framework.

Simple description of how we have implemented automatic profiling framework is introduced in the following subsection IV-A.

IV. Evaluation

In this section, we elaborate on the implementation of SPIRIT and demonstrate the effectiveness of SPIRIT by real use cases. Then, we measure the performance overhead that SPIRITaffects to NOS.

A. Implementation

To verify the feasibility of SPIRIT, we have implemented a prototype system. It is written in Java language to leverage its rich functionalities, user-friendliness and portability. SPIRIT is fundamentally based on JProfiler v9.0.1 [5], which is a well-known Java application profiling tool. In order to implement

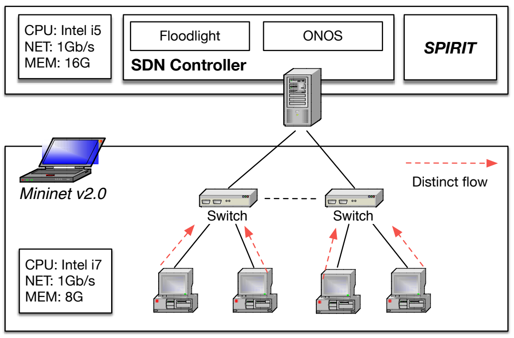

Fig. 3. Test Environment

our prototype system, we extend JProfiler to profile SDN network applications. In this paper, we only focus on profiling Java applications as most of the SDN controller implementations and applications available today are written in Java. Note that our tool can be easily extended to support other language, and we are looking forward to support different programming languages as a future work.

In order to construct an automated profiling platform, Mininet v2.0 [6], which is a well-known network emulator, is used for automatically creating a functional network environment. Our system also includes a program that uses libpcap [7] library to continuously capture and generate distinct network flows (test traffic generator).

The profile data analysis, which is a key feature of our system, involves hotspot and critical path examination. Here, our Profiling Tool Agent module parses the data collected from the profiler, and it uses Java DOM Parser since the data is stored as an XML file. To determine the specific hotspot of the target SDN application, we implement Algorithm 1 and Algorithm 2 in the Data Analyzer module of SPIRIT. Finally, to analyze the context of the found hotspot and critical path, we take advantage API description of the each NOS.

Currently, our prototype system only supports some of the SDN environments, such as Floodlight and ONOS, but we look forward to extend this prototype to cover various SDN controller implementations.

B. Test Environment

For the accurate evaluation of SPIRIT, two different physical machines are used; a desktop and a laptop. We run the network emulator and the flow generator on the laptop machine, while running the rest of the component of our system on the desktop machine. Such separated deployment scheme significantly improves the accuracy of the profile analysis result as it eliminates any potential interruptions caused by test modules. Thus, the test components, such as network emulator and flow generator, are deployed to the separate laptop machine. Figure 3 illustrates our experimental environment.

C. Use Case: Topology Application

As aforementioned, we analyze the default applications that comes with open source NOS distributions. Specifically

These applications are mostly basic network applications (e.g., network routing) that are usually enabled by default on boot.

in this use case scenario, we inspect the default topology applications, which maintain network topology information base, of Floodlight version 1.0 and ONOS version 1.1.

Here, we attempt to find the hotspots that are the most critical in terms of performance under consistently changing network environment. In order to create such dynamic network environment, we have composed four test cases, each emulating 4, 8, 16, 32 switches to effectively identify the hotspots that are sensitive to the size of network. Furthermore, we also periodically connect and disconnect one of the network links to artificially unstabilize the emulated network. To generate a natural flow, we have configured the flow generator module to generate one distinct flow per second. The duration of each test case was 60 seconds.

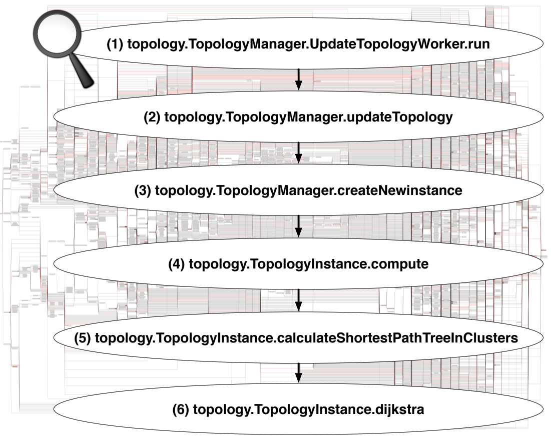
(1) topology.TopologyManager.UpdateTopologyWorker.run
(2) topology.TopologyManager.updateTopology
(3) topology.TopologyManager.createNewinstance
(4) topology.TopologyInstance.compute
(5) topology.TopologyInstance.calculateShortestPathTreeInClusters
(6) topology.TopologyInstance.dijkstra

Fig. 4. Critical Path of the Floodlight(Topology Application)

1) Floodlight: When we have profiled Floodlight and its basic applications using the SPIRIT, we have found one critical path within the entire call graph and a hotspot.

Critical Path. Figure 4 depicts the critical path of the Floodlight, which is extracted by the SPIRIT. The critical path of the Floodlight is taken to draw the network topology newly to their internal storage and calculate the shortest paths between each node on the newly drawn topology for updating the network topology. The detailed descriptions of each node on the path are summarized : When the Floodlight receives a control message that informs network link state is updated, (1) thread for recomputing topology detects network link state is changed. Then this thread calls (2) the entry point method that starts updating the stored network topology information at their internal storage. And then, this method calls (3) a method that creates a new instance of changed network topology. The updated network topology information is reflected in their internal storage, and it is used to create the new instance of network topology as parameters of topology instance constructor. Once the new instance is created, this method calls (4) a method that construct clusters of updated network topology. If making clusters is done, this method calls (5) the method that compute shortest path trees in each cluster for unicast routing. For doing that, this method calls (6) the function that implements Dijkstra algorithm.

HotSpot. To correctly identify the critical hotspots that get hotter respect to the size of a network, we have measured the taken time by each method on the critical paths by varying the number of switches.

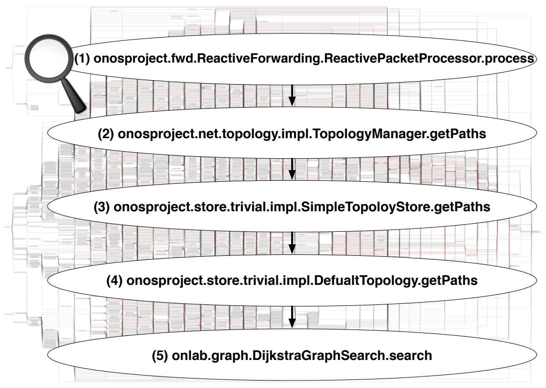
(1) onosproject.fwd.ReactiveForwarding.ReactivePacketProcessor.process
(2) onosproject.net.topology.impl.TopologyManager.getPaths
(3) onosproject.store.trivial.impl.SimpleTopoloyStore.getPaths
(4) onosproject.store.trivial.impl.DefualtTopology.getPaths
(5) onlab.graph.DijkstraGraphSearch.search

Fig. 5. Ciritical Path of the ONOS(Topology Application)

One of the critical hotspots is observed at the very last method on the critical path. Of those methods along the critical path, the most dramatic increase in taken time is found at the very last method. As shown in Figure 6, latency time of found hotspot is increased linearly and take up to 1274 milliseconds when we have experimented with 32 switches. Furthermore, the resource consumption percentage of the found hotspot on the Floodlight is measured up to 32 percentage.

This result can be implied that calculating the shortest path of every nodes in their network using the Dijkstra algorithm whenever network topology is changed will likely add even more latency to the overall operation time of the Floodlight for a choppy network.

2) ONOS: We have found one critical path within the entire call graph and a hotspot through the SPIRIT.

Critical Path. Figure 5 shows extracted critical path via SPIRIT on the ONOS. The presented critical path is taken to calculate shortest paths for forwarding the packet from the source to the destination whenever the ONOS receives a Packet-In control message. The detailed descriptions of each node on the path are summarized : Whenever the Packet-In control message coming up to the ONOS from their network node, (1) ReactiveForwarding application, which is installed on the ONOS for simple reactive forwarding, receives context of incoming Packet-In control message through this method. This method plays a role as finding a reachable and reasonable path for forwarding packets along their paths and installing rules to network by calling other methods. So, first of all, this method has a necessary to know all shortest paths between the source and destinations of received packet on current network topology. For that reason, this method call (2,3,4,5) methods that calculate the all possible shortest paths between the source and destinations using Dijkstra algorithm.

HotSpot. We have found a hotspot of the ONOS in the same way as the Floodlight use case. The hotspot of the ONOS has been found at the very last method on the critical path, which performs the same function as the hotspot of the Floodlight.

Figure 6 represents the how much time and resources are taken on the hotspot of the ONOS when the number of switches is increased. Found hotspot takes the latency time up to 8977 milliseconds and consumes the resources of the ONOS up to approximately 54 percentage with the number of 32 switches. The latency time graph of the ONOS can be seen as it is increased exponentially in accordance with the number of switches. Also, the hotspot of the ONOS has the higher latency time than the case of the Floodlight hotspot although it performs the same function.

These results tell us that computation the shortest path for forwarding the packet whenever Packet-In control message comes up to the ONOS will surely impose very large overhead to the ONOS for a larger network. Even more, we can know that kind of adopted solution on ONOS to calculate the shortest path for forwarding incur much higher overhead than the solution of the Floodlight case.

Without SPIRIT helping us identify the hotspot and analyze the context of the found hotspot automatically, it is hard to know why the found hotspot affects the overall performance degradation of the controller.

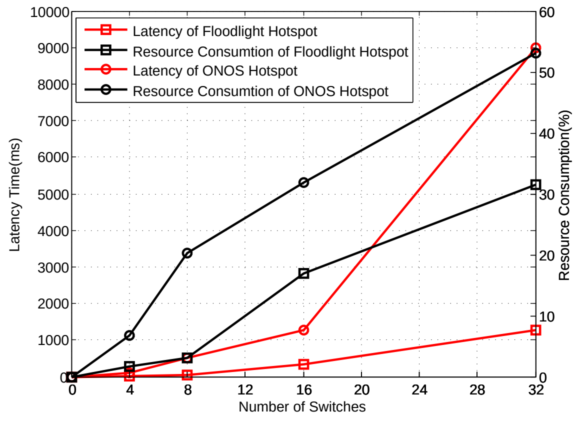

Fig. 6. Latency and Resource Consumption of each NOS Hotspot

D. Use Case: Debugcounter Application

We introduce another use case of SPIRIT that determines critical paths and hotspots of a NOS when a large number of requests were made by the data plane. For the testing, we generate diverse network flows to invoke Packet-In messages, which will ultimately arrive at the our target NOS. *Due to page constraint, we only introduce the Floodlight case in this paper.*

Critical Path. One critical path is discovered in this case, and the detailed descriptions of the path are as follows: (1) The Forwarding application receives a Packet-In message. (2) If Packet-In is verified to represent a valid network flow, it passes the analyzed information to the Routing module. (3) The Routing module attempts to issue the flow rule via OFMessageDamper of the Util package. (4) The OFMessageDamper module writes a rule to switch through OFConnection, which is one of the Floodlight core modules. (5) Before enforcing the rule to switch, the OFConnection module calls the method of Debugcounter application for updating OpenFlow counters. (6) Debugcounter application updates the value of the counters. Each counter indicate that the information how many each type of rule is enforced to the switch and when is the last seen time of each type of rule on that connection for collecting the statistics.

HotSpot. The most critical hotspot turns out to be the last method on the critical path shown in Figure 7. Of the methods executed within the path, DebugCounterImpl.increment method recorded the highest execution time for DebugCounter application.

Debugcounter application implements a central store that maintains all of the counters for the system debugging purpose. To be more specific, `increment method`, which is a hotspot, is called by Debugcounter application is a simple method that increments the counters that counts the number of each type of OpenFlow messages observed on each connection between the NOS and switch. At the same time, it also keeps a record of last seen time of each OpenFlow message by invoking *System.currentTimeMillis()* function. Although such operation is not very heavy, performing the operation for each OpenFlow message may significantly impact the overall performance of the system.

To understand how much overhead that this hotspot adds to the overall performance of Floodlight, we have measured the total time spent on getting system time within Debugcounter application. As a result, the application solely spent 2,139 milliseconds (7 percents of total time) on system time determination in 30 seconds. Therefore, we recommend to avoid using Debugcounter application if the network is performance-sensitive.

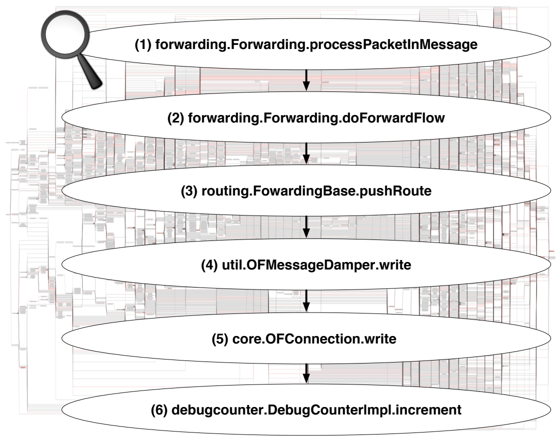
(1) forwarding.Forwarding.processPacketInMessage
(2) forwarding.Forwarding.doForwardFlow
(3) routing.FowardingBase.pushRoute
(4) util.OFMessageDamper.write
(5) core.OFConnection.write
(6) debugcounter.DebugCounterImpl.increment

Fig. 7. Critical Path of the Floodlight(Debugcounter Application)

E. Overhead

In order to determine the performance impact of SPIRIT that may affect the performance of NOS, we measure and compare the throughputs of Floodlight and ONOS under two different environment; with and without SPIRIT. In this experiment, we emulate 4 dummy switches to generate the control traffic using *cbench* with throughput mode enabled, and each test case is executed for 10 seconds. The same machine shown in Figure 3 is used, and the experiment is repeated 10 times to improve the data quality.

As shown in Figure8, ONOS achieved 70,217 responses/sec and Floodlight achieved 117,689 responses/sec on average without SPIRIT. Meanwhile, with SPIRIT, ONOS and Floodlight achieved 5,907 responses/sec and 10,623 responses/sec respectively.

The performance evaluation indicates that SPIRIT may significantly degrade the overall performance of NOS; however, the result is not surprising at all. JProfiler, which SPIRIT employs to profile Java-based NOS, attempts to collect detailed timing information about every method calls invoked within the JVM that hosts NOS, and this is an obvious extremely-heavy task. In addition, to improve the accuracy, our prototype system uses the instrumentation profiling method that dynamically injects code to time each method call, and it makes the profiling process even more costly.

Meanwhile, such performance degradation does not affect SPIRIT's accuracy in determining critical paths or hotspots. Although the numbers measured by our system do not represent the actual performance of NOS, the relative comparison of those numbers is still meaningful and valid to discover critical paths and hotspots. Furthermore, the performance impact to NOS caused by our system is not a concern, since it is a testing tool, which is not supposed to be attached to the NOS that is deployed to real networks at real time.

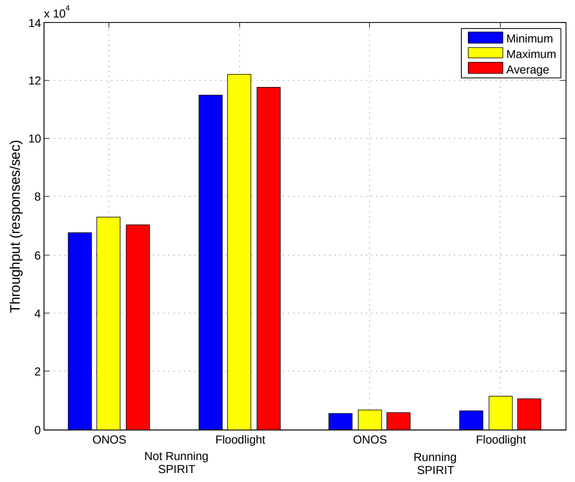

Fig. 8. Throughput Results

V. Discussion

Profiling NOS and its applications is useful in understanding their operations and performance bottleneck points, and it will provide many insights that can guide us to devise more efficient and robust NOS and network applications. In addition, this kind of approach can be used in other areas, such as security and capacity planning. Hence, although not much attention has not been put on this area so far, soon researchers and practitioners dive into this area.

However, as we have presented here, profiling NOS and its applications is not an easy job, and requires many annoying things. Moreover, it is very hard to apply existing profiling tools to this area without changes due to the differences between the operational characteristics of NOS (and its applications) and those of legacy OS and applications.

To the best of our knowledge, this work is the first trial to profile NOS and its applications in an (nearly) automatic fashion (minimizing human intervention). Of course, we are not in the final stage yet, and we need to improve our proposal in many places. We do not think that this job is done by a small community, and thus we will open this tool soon in public to get more feedbacks from diverse communities.

VI. Related Work

Enhancing the SDN performance: In the SDN environment, performance is the most critical problem because the controller must handle the amount of large flow. Understanding these problems, the most of recent research trends in the leveraging performance of the controller is based on their architectural design. Centralized controller architecture such as the Beacon [8] and Floodlight [3] are designed as a highly concurrent system that is based on multi-thread in order to accomplish the desired throughput. Some research showed that such a single NOS is enough for supporting the amount of large flow [1]. However, these results have a limit on that they tested only with pretty simple network applications. Many researchers point out the limitation of that and thought a single NOS is not enough to manage the large network. To cover the performance issue of a single NOS, latest released controller, such as OpenDaylight [9],ONOS [4] have been adopted distributed-controller architecture, which binds multiple physical controllers as a single logical controller. Contrary to the centralized controller, distributed controller can increase the overall network performance through load balancing with other clustered NOS.

SDN diagnostic framework: In the SDN community, various diagnostic frameworks have been researched. Frameworks such as VeriFlow [10], Header Space Analysis(HSA) [11] and VeriCon [12] are the very recent SDN diagnostic framework, however, those cannot analyze the SDN performance because they are focused on the network correctness. In the research area of the SDN network diagnosing, some researches are focused on the debugging the SDN network, for example, ndb [13], NetSight [14] and OFRewind [15]. While they cannot analyze the deep inspection of the SDN application, SPIRIT can analyze the SDN application with source code level automatically.

Application profiling: Coppa et al. [16] have proposed a profiling methodology to discover hidden inefficiencies in the code and Zhuang et al. [17] presented an approach to build the calling context tree for constructing accurate profiles. However, these previous works did not consider the data from the network. ProfileDroid [18] and ARO [19] suggested application profiling tools to locate the performance bottlenecks of the application by considering the multi-layer that includes the network. Although these works considered the data from the network for profiling the application, these are not suitable for SDN application profiling because it has to be considered the various network events. On the other hand, our framework SPIRIT is specialized on SDN application profiling by taking into account the various network events.

Profiling technology: Profiling technology has been researching continuously. In case of the JAVA, there is some JVM profiling technology. For example, JVMPI [20] is the basic technology for profiling the JVM, which is available from the SDK(Software Development Kit) version 1.1.0. Various JAVA profiling tools were developed by using this technology, such as JavaTreeProfiler [21] and DJProf [22]. However, JVMPI is an old technology now. Currently, more recent technology than the JVMPI, which is called JVMTI [23], replaces the JVMPI. JVMTI adopts dynamic bytecode instrumentation methodology for profiling, and it provides all of the functional capabilities of the JVMPI. JVMTI is available from the SDK version 1.5.0, and many JAVA profiling tools have been developing based on JVMTI. JProfiler that is adopted on our prototype of SPIRIT is also developed based on JVMTI.

VII. Conclusion and future work

While the networking community considers SDN as coming under the spotlight and future networking technology, SDN falls short of performance to deploy on the real network for now. To improve the SDN performance, most of the previous approaches focused on NOS platform design, the performance of devices and architectures. However, there was no research that analyze and optimize the SDN application to improve the performance of the SDN until now. This is regrettable for ours because SDN performs many control-plane functions as software application.

In this work, we do not analyze comprehensive applications of NOS, but we investigate the performance of default applications in some well-known NOSs - Floodlight and ONOS. Also, we suggest new framework - SPIRIT for providing comfortable SDN application profiling environment. We believe that methods and findings presented in this paper can encourage the SDN researchers or developers to devise more and better SDN applications.

In the future, we desire to investigate the performance of more comprehensive SDN applications on various controllers. Speaking more specifically, extending the SPIRIT to support the OpenDaylight is our ongoing project. This project is almost finished now, so we expect that we can profile the comprehensive application of the OpenDaylight in the near future. Moreover, we have a plan to adding the functions to the SPIRIT for supporting the commercial controller, such as HP VAN SDN [24], by providing our custom API or application.

Acknowledgment

This work was supported by the ICT R&D program of MSIP/IITP, Korea (2014-044-072-003, Development of Cyber Quarantine System using SDN Techniques) project.

References

[1] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, “On controller performance in software-defined networks,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, ser. Hot-ICE’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228283.2228297

[2] R. Sherwood and K.-K. Yap, “Cbench controller benchmarker,” 2011, http://www.openflow.org/wk/index.php/Oflops.

[3] B. S. Networks, “Project floodlight,” 2012, http://www.projectfloodlight.org.

[4] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, and G. Parulkar, “Onos: Towards an open, distributed sdn os,” in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’14. New York, NY, USA: ACM, 2014, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620744

[5] EJ-technologies, “Project jprofiler v9.0.1,” 2015, https://www.ej-technologies.com/products/jprofiler/overview.html.

[6] B. Lantz, B. Heller, and N. McKeown, “A network in a laptop: Rapid prototyping for software-defined networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466

[7] Tcpdump/Libpcap, “Project libpcap v1.7.4,” 2015, http://www.tcpdump.org.

[8] D. Erickson, “The beacon openflow controller,” in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN ’13. New York, NY, USA: ACM, 2013, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/2491185.2491189

[9] J. Medved, R. Varga, A. Tkacik, and K. Gray, “Opendaylight: Towards a model-driven sdn controller architecture,” in *2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 1–6.

[10] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, “Veriflow: Verifying network-wide invariants in real time,” in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 15–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482630

[11] P. Kazemian, G. Varghese, and N. McKeown, “Header space analysis: Static checking for networks,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2228298.2228311

[12] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, “Vericon: Towards verifying controller programs in software-defined networks,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 282–293. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594317

[13] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, “Where is the debugger for my software-defined network?” in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN ’12. New York, NY, USA: ACM, 2012, pp. 55–60. [Online]. Available: http://doi.acm.org/10.1145/2342441.2342453

[14] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, “I know what your packet did last hop: Using packet histories to troubleshoot networks,” in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol

[15] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, “Ofrewind: Enabling record and replay troubleshooting for networks,” in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 29–29. [Online]. Available: http://dl.acm.org/citation.cfm?id=2002181.2002210

[16] E. Coppa, C. Demetrescu, and I. Finocchi, “Input-sensitive profiling,” in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 89–98.

[17] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi, “Accurate, efficient, and adaptive calling context profiling,” in *ACM SIGPLAN Notices*, vol. 41, no. 6. ACM, 2006, pp. 263–271.

[18] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Profiledroid: multi-layer profiling of android applications,” in *Proceedings of the 18th annual international conference on Mobile computing and networking*. ACM, 2012, pp. 137–148.

[19] F. Qian, Z. Wang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck, “Profiling resource usage for mobile applications: a cross-layer approach,” in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 321–334.

[20] Oracle, “Jvmpi,” 1997, http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/.

[21] W. Jung, “Javatreeprofiler,” 2002, http://jcoverage.sourceforge.net/.

[22] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly, “Profiling with aspectj,” *Softw., Pract. Exper.*, vol. 37, no. 7, pp. 747–777, 2007. [Online]. Available: http://dx.doi.org/10.1002/spe.788

[23] Oracle, “Jvmti,” 2004, http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/.

[24] HP, “Project hp virtual application networks(van),” 2013, hp.com/networking/sdn.