Dynamic Analysis of Application Delivery Network for Leveraging Software Defined Infrastructures

Lav Gupta*, *Sr. Member, IEEE*, Raj Jain, *Fellow, IEEE,* Mohammed Samaka

Abstract— **Application Service Providers (ASPs) may obtain resources from a number of cloud service providers (CSPs) in an attempt to improve latency and minimize operational expenses (OpEx). The CSPs may use management and control platforms, such as OpenStack and EC2 and the network service providers (NSPs) may use network management platforms, such as, OpenDaylight. However, today the ASPs do not have a common management and control platform that would present to them a converged view of all the cloud and network resources. OpenADN being developed at Washington University in Saint Louis aims to allow the ASPs dynamic and real time control of virtual resources across multiple clouds and networks to provide efficient application delivery. The OpenADN platform itself is a complex distributed and multi-threaded system. Performance evaluation and assessment of need for optimization of such a complex platform requires precise and fine-grained behavioral data. In this paper we establish the need for profiling OpenADN like platforms so that the ASPs can optimize its behavior and control their cost, performance (latency) and energy consumption.[1]**

***Index Terms*—Software defined infrastructure, profiling, multi-cloud, cloud services, network services, application service providers, OpenADN, distributed systems, optimization**

I. Introduction

Software-defined infrastructure (SDI) is a generic term that is used to refer to virtual infrastructures with software-based control and management systems. The physical devices on which these virtual infrastructures are created could themselves be spread over a number of datacenters or cloud platforms, each controlling a number of datacenters. Some examples of software that enable SDI implementations include OpenStack, EC2 and OpenDaylight. SDI allows application specific virtual clouds to be carved out of physical resources from multiple clouds and to dynamically control and manage them. Through SDI, businesses and enterprises, which we call, the Application Service Providers (ASPs), get a converged view of resources provided by the Cloud Service Providers (CSPs) and the Network Service Providers (NSPs). This allows them to use resources available from many providers, through their APIs, in a manner that enables optimization of flexibility, reliability, latency and operational expenses (OpEx).

Software presenting virtualized environment of distributed physical resources under disparate management tend to be complex systems. They generally use multithreading technique of software modularization and concurrent execution. If the modules, of such a system, do not work in harmony, performance suffers resulting in inefficient resource utilization and greater energy consumption [1]. Responding reactively to performance degradation for optimization results in higher expenses being incurred during operation of the existing system and its modification during its lifetime. At the same time, understanding the performance of the software platform through profiling should invariably precede optimization. Using well-known techniques software engineers are able isolate hot spots that consume disproportionate share of resources. Multithreaded systems become difficult to profile because characterizing the effects of interactions between threads is difficult [16]. Efficient abstractions need to be developed to capture this behavior without resulting in exponential analysis times.

OpenADN, being developed at Washington University in St Louis, is an application delivery platform that creates application clouds employing, controlling and managing resources across multiple CSP clouds. OpenADN functions as an interface between the applications and the SDI layer. It allows ASPs to dynamically acquire and manage resources from multiple CSPs and optimize operational expenses. In principle one would expect these benefits from any such platform that might be developed in future. However, if the platform software has not been optimized from the ASPs point of view, then the resources would be inefficiently utilized resulting in sub-optimal system behavior and increase in operational expenditure. Such systems also lead to higher energy consumption and are contradictory to the notion of reducing the carbon footprint. Taking advantage of the first such system being available to us we have attempted to characterize the behavior of such a platform under operation and used several profiling techniques to see what could cause the system to behave sub-optimally. This should spur the

The manuscript was submitted on 1st December, 2014.

Lav Gupta and Raj Jain are at Washington University in St Louis, MO 63130 USA (email: {lavgupta, jain}@wustl.edu).

Mohammed Samaka is with Qatar University, Doha, Qatar (email: samaka.m@qu.edu.qa)

[1] This work has been supported under the grant ID NPRP 6 - 901 - 2 - 370 for the project entitled "Middleware Architecture for Cloud Based Services Using Software Defined Networking (SDN)", which is funded by the Qatar National Research Fund (QNRF). The statements made herein are solely the responsibility of the authors.

*Corresponding Author

978-1-4799-8218-9/15 $31.00 © 2015 IEEE
DOI 10.1109/IC2E.2015.51

IEEE computer society

developers of such systems to fine-tune their platforms saving money for the users and the planet from harmful effects of higher energy consumption. Section II describes the OpenADN platform highlighting its distributed and multi-threaded nature. Section III deals with profiling approaches that can be used for platforms like OpenADN dealing with resources spread across multiple clouds. Section IV takes up the discussion on profiling OpenADN and directions for optimization. We conclude our results in Section V.

II. Multi-cloud SDI - OpenADN

A. OpenADN Architecture

OpenADN is a multi-cloud management system. As shown in Fig 1, on the north side, it offers interfaces for application developers, application architects, and application deployment administrators to define the application resource requirements and deployment policies. On the south side it has many modules, one for each of the cloud/network management systems.

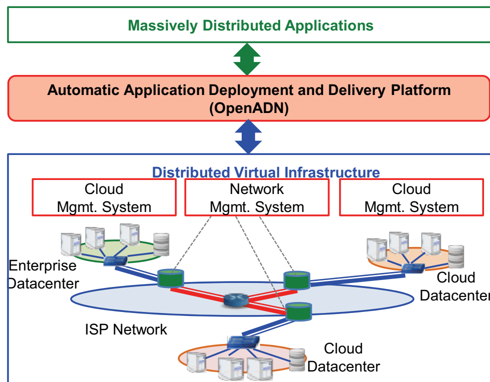


Fig. 1. OpenADN Multi-Cloud Management System

Notice that the OpenADN architecture has a modular structure similar to the OpenDaylight SDN controller [2] with many southbound interfaces. The northbound interfaces of OpenStack/ OpenDaylight become one of the southbound interfaces of OpenADN. While OpenStack allows implementing client policies in one cloud, OpenADN allows implementing client policies uniformly among all the clouds.

OpenADN does not directly manipulate the resources inside the clouds; it simply requests the respective cloud manager to create those resources. The Application Deployment Manager specifies the policies regarding when and where to create the resources.

Most contemporary and future application deployments like Internet-of-Things (IOT), Cyber-Physical Systems, mobile apps, massively parallel gaming and virtual reality tend to be distributed and need to use multiple clouds primarily due to cost and latency considerations and can use OpenADN to manage the entire application as if it was in one cloud.

OpenADN is an integrated infrastructure comprising both, message-level devices and packet-level devices, hosting application-layer services as well as network-layer services. It consists of a central global manager and a set of local managers - one per cloud or network. The administrator starts only the global manager manually and the rest of the process is bootstrapped by it. For massively distributed applications, OpenADN allows multiple zones with each zone consisting of multiple clouds [14].

B. OpenADN As A Massively Distributed System

OpenADN has a distributed data plane to optimize application service deployment. Considering the geographical spread of resources, a part of control activities are centralized in the global controller making it easy to introduce new service, propagate new polities and troubleshoot problems. However, keeping view the latencies of a highly distributed data plane, tasks relating to a data center are controlled by a local controller. The management plane is completely centralized and ensures that the policies are being enforced properly and record non-compliance. The system as a whole can perform many different tasks at the same time leading to better utilization of the hardware resources and ensuring that the system as a whole makes progress all the time. OpenADN is essentially a multi-threaded system where performance is determined by the execution environment.

III. Profiling Multi-cloud Delivery Platform

In this section we discuss the techniques that we have selected from in profiling OpenADN.

A. *Profiling and optimization*

Profiling gives insight into program performance. Program analysis tools are extremely important for understanding program behavior. Most software has code that consumes disproportionate amount of resources and produces higher CPU loads. Reading of code does not provide reliable information about program behavior under execution. Using intuition on multi-threaded programs does not usually give right results.

B. *Profiling techniques for Application Delivery Platforms*

New profiling, characterization, and modeling methodologies are required to understand the nature of architectural behavior under full system virtualization. In order to pinpoint the sections of code that should be optimized, a programmer needs detailed data on how that program behaves [3]. We shall see here the techniques that can be applied to distributed, multi-threaded systems [16]. We'll divide these techniques into static, dynamic and concurrency profiling

1) *Static Profiling*

Static analysis is the formal construction of program execution models [4]. Model checking is a static analysis technique that is often applied to multi-threaded programs to explore all feasible inter-leavings exhaustively to ensure correctness properties [5]. Model checking becomes computationally expensive due to a vast number of feasible inter-leavings in a large multi-threaded system. Static-analysis techniques give assessment of relative time and temporal ordering and do not give absolute time [6]. For assessment of

absolute times dynamic we applied dynamic profiling [15].

2) *Dynamic Profiling*

Dynamic profiling is the only practical approach to behavioral analysis that can incorporate aspects of absolute time by inspecting behavior of a running system. It is an active form of profiling in which the system being measured explicitly generates information about its execution parameters. Conversely, passive profiling relies on explicit inspection of control flow and execution state through an external entity, such as a probe or modified runtime environment. Three main families of dynamic profiling techniques code instrumentation, statistical sampling and concurrent profiling.

a) Code Instrumentation: An instrument is a set of additional instructions injected into the target program to generate the required information. Instrumenting a program can cause changes in the performance of the program, potentially causing inaccurate results and has to be carried out carefully in a controlled manner. These instructions count how many times various parts of a program get executed. Some instrumentation systems [5] count function activations while others [7] count more fine-grained control flow transitions.

b) Statistical Profiling: randomly samples the effective instruction pointer/program counter, and deduces where time is being spent. They are not as intrusive to the target program. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing and also the user time out of the total execution time [17,18]. In OpenADN environment this could, for example, provide valuable information on whether optimization should at all be attempted.

c) Deterministic profiling refers to all function calls, function returns, and exception events being monitored and precise timings made for the intervals between these events. In Python, since there is an interpreter active during execution, the presence of instrumented code is not required to do deterministic profiling. Python automatically provides a *hook* (optional callback) for each event. Call count statistics can be used to identify bugs in code and to identify possible inline-expansion points. Internal time statistics can be used to identify ``hot loops'' that should be carefully optimized.

3) *Concurrency Profiling*

Concurrency profiling can be additionally used for multithreaded applications. Resource contention profiling collects detailed call stack information every time that competing threads are forced to wait for access to a shared resource. Concurrency visualization also collects more general information about how multithreaded application interacts with itself, the hardware, the operating system, and other processes on the hosts. It can help locate performance bottlenecks, CPU underutilization and synchronization delays [8,9]

IV. OpenADN Profiling

The complexity of the multi-threaded OpenADN platform required precise and fine-grained behavioral data while in execution, coupled with off-line analysis to help characterize the performance of the platform and possible need for optimization. Profiling of OpenADN was, therefore, carried out at multiple levels. To validate the functionality, we ran OpenADN in a virtual environment created by Mininet [10]. Mininet allows emulating a whole virtual network running real kernel, switch and application code, on shared physical resources of a machine. The following virtual resources were created for profiling OpenADN: One zone consisting of a global controller, two data center sites with a local controller each, a name-server, 7 hosts per site and client host with 10000 users. The selection of stimuli (set-up and input data) and multiple runs of the platform ensured that behavioral data for most control paths are collected.

To recapitulate the virtual network bootstrap process, the resource manager adds resources from different sites to the global resource pool. It then assigns a role to each virtual node that is started. The client host simulates around 10,000 users, each starting a separate user session with the application. The global controller (GC) initializes a workflow manager (WFM). The WFM spawns an initial workflow thread (WFT) for an application, say ABC, in the only zone, say US-E. The WFT registers this application instance with the name server. The name server advertises this mapping when the WFT explicitly activates the mapping entry. WFT also gets a proxy node allocated to it that will be the interface between the workflow services and the external users. The WFM is responsible for allocating a proxy node to each WFT. However, the WFM can do so only when it has the resources. WFM does not keep the state so WFT has to retry. The WFT runs an exponential back-off mechanism to repeat its request instead of flooding the system with useless request messages. The WFM on the other hand independently attempts to get the resources required to allocate a proxy node. While this happens, each datacenter controller boots up independently and tries to register itself with the GC.

The WFM makes a resource request to site 1 that offers resources to the WFM only after its data plane nodes have registered their resources with it and so it does not have any available resources. The WFM would try other sites and repeat these requests till it gets the required resources. In the meantime, the WFT thread keeps on polling the WFM for a proxy node. Eventually WFM is flooded with resource updates from the different sites. Each data plane node in our experiment reports 1000 units of resources and hence the total resource available per-site is 5000. WFM sends the request to allocate a data plane node to run the proxy service to the first site that reports enough available resource. The proxy node is initialized and the WFM starts gathering the resources to deploy the other services within the workflow.

Once the required data plane nodes with enough resources to run the workflow have been identified, the next step is to actually start the application services on these nodes and setup the message and packet routing services. WFT gets to the job of starting the services for the workflow after the WFM has allocated it the required resources. After each service is

initialized it connects to the OpenADN socket that opens a communication channel between the service and the platform. The OpenADN socket also starts a heartbeat reply service to reply to aliveness queries from the platform. The WFT attaches itself to one of the ports of the shared proxy service. This concludes the bootstrap process.

A. Execution Time Analysis

To get the broad idea of the efficiency of the platform code executing in a virtualized environment, the Unix time utility was used. The platform software “driver_mininet” created virtual hosts over which the platform modules – global controller, local controller, name server, node controller and clients ran. The program was run to bootstrap the process and run it till all the modules were added and services started running. The Unix built-in time command was invoked with: /usr/bin/time -f "\n%E elapsed,\n%U user,\n%S system,\n%M memory\n%x status" driver_mininet.py. A number of runs were performed for the same virtual environment and data from five of them are given in Table I.

The elapsed time is the total platform run time for booting and starting new services, user-space time is for non-system calls or CPU time spend outside the kernel and system-calls is time spent in kernel specific functions.

Table I
Run time used for user and system activities

(time unit: seconds)							
Runs	I	II	III	IV	V	Averages	% Run time
User Space	0.53	0.55	0.62	0.6	0.61	0.58	1.65
System Calls	0.76	0.75	0.65	0.67	0.68	0.7	1.99
Run time	35.82	35.6	34.65	34.8	35.06	35.19	100
Res Memory Block(kB)	19216	19216	19232	19216	19232	19222	-

Of the average total elapsed time of 35.19 seconds for which the platform software was executed, the time spent in user functions and kernel space was 1.65% and 1.99%, respectively. This gives a sense that a large part of the time is spent in I/O waits and sleep times for dealing with dependent asynchronous concurrent processes. However, it cannot be yet said whether this time relates to unavoidable delays and the situation can be improved through optimization. This called for the next level of profiling, i.e., at module/function level to see which of the modules are more CPU intensive.

The same modules were also run on separate physical machines for comparison and the results obtained are given in Table II.

Table II
Time used for user and system activities on physical machines

Function	**User Space**	**System Calls**	**Run Time**	**User(%)**
Name Server	14.161	5.072	229.438	6.17
Global Controller	83.637	15.797	200.835	41.64
Local Controller	18.549	7.16	175.57	10.57
Node Controller	19.95	8.86	156.99	12.71
Client	0.428	0.036	18.855	2.27
	136.725	36.925	781.688	17.49

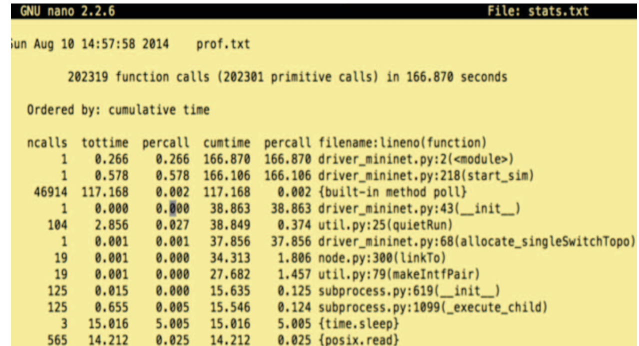
On physical machines, the platform does not have to spend time creating virtual machines for its own modules as well as for running services. Even in this case the overall user-space time is 17.49% and even lesser for kernel calls. Among these the global controller used the time more effectively with user functions taking up to 41.64% of run time on an average. However, in the actual operational environment, these modules will be hosted on VMs that will take finite amount of time to create, start and augment.

This simple profiling indicates the possibility of higher load on the CPU because of potentially wasteful activities like waiting on I/O calls and sleep functions. While in many cases where asynchronous linking of threads are used some waiting would be unavoidable. However, one needs to see whether these could be optimized for 1) making the platform more efficient 2) correctly dimensioning the resources leased, and 3) distributing the workload properly.

B. Deterministic Profiling of OpenADN

Deterministic profiling of OpenADN programs was carried out to see execution pattern and the resultant CPU loads of various functions. This was done through *cProfile* provided by the Python library. The profile of these programs gives a set of statistics that describes how many times different functions are called and how much time the CPU is spending in various modules. The module ‘*pstats*’ [11] was used to format these statistics to make them amenable to analysis. Fig 2 gives a sample output.

It can be seen that the total time that the *driver_mininet.py* was executed in the run above was 166.870 seconds. Out of this the simulator module took 166.106 seconds.



```
GNU nano 2.2.6                                              File: stats.txt

Sun Aug 10 14:57:58 2014    prof.txt

         202319 function calls (202301 primitive calls) in 166.870 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.266    0.266  166.870  166.870 driver_mininet.py:2(<module>)
        1    0.578    0.578  166.106  166.106 driver_mininet.py:218(start_sim)
    46914  117.168    0.002  117.168    0.002 {built-in method poll}
        1    0.000    0.000   38.863   38.863 driver_mininet.py:43(__init__)
      104    2.856    0.027   38.849    0.374 util.py:25(quietRun)
        1    0.001    0.001   37.856   37.856 driver_mininet.py:68(allocate_singleSwitchTopo)
       19    0.001    0.000   34.313    1.806 node.py:300(linkTo)
       19    0.001    0.000   27.682    1.457 util.py:79(makeIntfPair)
      125    0.015    0.000   15.635    0.125 subprocess.py:619(__init__)
      125    0.655    0.005   15.546    0.124 subprocess.py:1099(_execute_child)
        3   15.016    5.005   15.016    5.005 {time.sleep}
      565   14.212    0.025   14.212    0.025 {posix.read}
```

Fig. 2. Sample Deterministic Profile Run

Legend: **ncalls:** the total number of calls, **tottime:** total time spent in the given function (excluding sub functions) seconds, **percall**: tottime divided by ncalls, **cumtime**: total time in this and all sub-functions seconds, **Percall**: cumtime divided by primitive calls, **filename**: data for each function

It was initially suspected that the Zeromq messaging library [12] poller takes up a lot of CPU time. A cursory analysis of the output shows that the poller is called 46914 times and the total time spent is 117.168 seconds, which is 70.21% of the total time. However, these calls are distributed and on each call the time spent is just 0.002 seconds. Creation of the virtual network topology takes 37.856 seconds and starting the virtual network takes another 6.151 seconds. It was also thought that the sleep module might also be wasting a lot of CPU time. It took 15.016 seconds of the total time. Different important modules take up the times given in Table III.

Table III
Time for different functional modules

Module	CPU time taken (seconds)
Name server	5.009
Global controller	5.008
Local controller	0.014
Hosts	6.013
Client host	0.065

We would see later that the Name Server and the Global Controller sleep through most of the time. Their job is largely reactive in nature, getting activated when other modules need their services.

C. *Detailed Line-by-line Analysis*

Profiling at the platform and function levels gives a good idea of the time spent by the CPU in kernel space calls, user space calls and waiting for I/O and in various modules of OpenADN. It was observed that a large proportion (96.36%) time was spent in waiting for I/O. Polling operations took about 70% of the execution time. The program spent 15 seconds of the total 166.87 seconds in sleep mode. OpenADN functional modules took up to about 6 seconds each.

Profiling at this level gave a good indication of overall execution times and of the functions that were consuming unduly large amount of CPU time. It was not enough to tell us which modules to look into to locate the potential hot spots and optimize the software. As is often the case, the reason for a particular module or functionality taking a large amount of time could be pin-pointed to some small part which may seem to be innocuous on simple reading of the code. Some statements could trigger a library function or call to a special method that may not be so obvious. The function level profiling only times the explicit function calls and not the special methods called. Such profiling would not identify a slow operation in the library function like Zeromq. If a statement triggers computation when using libraries, when there is no explicit call, function profiler will not usually break it down.

A more detailed line-by-line analysis of the program was undertaken to find out which parts of the program take more CPU time. A more intrusive line profiler that could go into each function and time execution of each statement was used for this purpose. The Kernprof python script and the @profile decorator used in a judicious manner allow this kind of analysis [13]. This profiler keeps track of multiple statement executions, sums up the total time each statement takes in multiple passes and avoids profiling overheads. The profiling result is a binary file that could be deciphered with 'pstats' or a similar function. The output consists of the following:

a) Hits: Number of times that line was executed.
b) Time: Total execution time
c) Per Hit: Average amount of execution time
d) % Time: Percentage of time spent on that line relative to the total amount of recorded time spent in the function.
e) Line Contents: Actual source code.

We give snapshots of some sample outputs in Fig. 3 (a) through (d) and then discuss some important aspects revealed by these.

Fig. 3 is part of a typical profiling run. However, the averages of multiple runs were quite close to individual runs and so this figure provides sound grounds for discussion. The overall execution times for different functions are given in Table IV.

The illustrations in Fig. 3 show some of the portions of profiling data that indicate possible need for optimization. Fig. 3a and 3b show linking to the switch takes up a major percentage of the execution time. The name-server takes 6.8% while the global controller takes 4.2%(not shown). The hosts take the longest accounting for 68.5% of the time.

```
1  Timer unit: 1e-06 s
2
3  Total time: 10.1736 s
4  File: driver_mininet.py
5  Function: allocate_singleSwitchTopo at line 68
6
7  Line #      Hits         Time  Per Hit   % Time  Line Contents
8  ==============================================================
9      68                                           @profile
10     69                                           def allocate_singleSwitchTopo(self):
11     70                                             #pr=cProfile.Profile() #remove
12     71                                             #pr.enable() #remove
13     72         1        59938  59938.0      0.6    self.switch = self.net.addSwitch('s1')
14     73         1        72290  72290.0      0.7    self.net.addController( 'c0' )
15     74
16     75                                             # Fake name server
17     76         1        75581  75581.0      0.7    self.gc_fakeNameServer = self.net.addHost("FakeNS", mac=self.ns_mac, ip=self.ns_ip )
18     77         1       693808 693808.0      6.8    self.gc_fakeNameServer.linkTo(self.switch)
19     78         1         1819   1819.0      0.0    print ("adding global nameserver<%s, %s, %s>"%(self.gc_fakeNameServer.name, self.ns_mac,
   self.ns_ip))
20     79
```

(a) Profile run for creation of topology

```
120      14        37      2.6     0.0        hostName = "h" + str(j-1)+ "s"+ str(i)
121      14     53512   3822.3     0.5        print ("adding host to site %s: <%s, %s, %s>" %(i, hostName,  mac, ip) )
122      14    352694  25192.4     3.5        host = self.net.addHost(hostName,mac=mac, ip=ip)
123      14   6964839 497488.5    68.5        host.linkTo( self.switch )
124                                           # store info for each host
125      14        40      2.9     0.0        hostInfo = {}
126      14        14      1.0     0.0        hostInfo["host"] = host
127      14        44      3.1     0.0        hostInfo["lc_controller_addr"]= lc_lighthouseController.defaultIP
128      14        16      1.1     0.0        hostInfo["host_addr"] = host.defaultIP
```

(b) Profile run for linking hosts to a switch

```
148                                       def start_gc_lighthouseController(self):
149       1         0      0.0     0.0      print ("2. Starting Global Lighthouse Controller"),
150       1       483    483.0     0.0      self.gc_lighthouseController.cmd('export HOST_NAME=%s'%(self.gc_lighthouseController.name)
)
151       1       162    162.0     0.0      self.gc_lighthouseController.cmd('export NAME_SERVER_ADDR=%s'%(self.gc_fakeNameServer.
defaultIP))
152       1       137    137.0     0.0      self.gc_lighthouseController.cmd('export NAME_SERVER_UPDATE_PORT=%s'%(self.
ns_update_mapping_port))
153       1      1865   1865.0     0.0      self.gc_lighthouseController.cmd("python3 %s &"%(GLOBAL_CONTROLLER))
154       1         0      0.0     0.0      print (".... started\n")
155
156
157       1   5004523 5004523.0   99.9       sleep(5)
```

(c) Start of global controller

```
273        1      24689   24689.0     0.0   simNetwork.start_client_host()
274        1       1499    1499.0     0.0   print ("------------\n")
275        1       1301    1301.0     0.0   print ("checkpoint 5...after client host")
276                                         #start the monitoring
277        1         17      17.0     0.0   endTime = time() + _runTime
278   102526     481979       4.7     0.2   while time()< endTime:
279   102526  208269149    2031.4    87.2     readable = poller.poll(1)
280   102810     495496       4.8     0.2     for fd, _mask in readable:
281      285       1032       3.6     0.0       node = Node.outToNode[ fd ]
282      285      11422      40.1     0.0       outString = node.monitor().strip()
```

(d) Simulation run of the platform

Fig. 3. Results of Line by Line Analysis

Fig. 3c and 3d indicate large amount of times taken up by the sleep function and the polling function. The global

controller sleeps most of its execution time and similar is the case with the name-server. This could mean that these functions are demanding more virtual resources than necessary and are leading to higher OpEx. Also the function to check the ports for inter-process messages takes up 87.2% of the entire simulation time.

Table IV
Overall function execution times

Function	Execution time in seconds
Creation of virtual topology	10.174
Starting the virtual network	2.032
Starting global controller	5.007
Starting local controllers	0.014
Starting name server	0.012
Starting client host	0.025
Starting hosts	5.334
Simulation	238.897
Total	261.495

V. DISCUSSION, CONCLUSIONS AND FUTURE WORK

Multi-cloud management systems, like OpenADN, have parts of the code that account for unduly bigger share of time elapsed during execution. This could lead to suboptimal application delivery, increased resource usage and operational expenses. ASPs who would use such systems would like to optimize their platforms to control their expenses and for other desirable features like reduced latency and reduced energy consumption.

Visual examination of the code does not provide reliable information about what could be wrong with it. Using intuition on multi-threaded programs is still worse. It becomes necessary to generate program profile with data collected at various levels-platform, functions and statements.

For OpenADN, the top level analysis reveals that the overall execution time has a large component of non-user, non-kernel time that could be explained by I/O waits. A concern that arises is that some part of this time could be spent unproductively using up resources and contributing to energy consumption. A function level analysis makes apparent the functions that have potential hot spots. Line profiling on all of the modules simultaneously allows interplay of threads and reveals the parts of the functions that could be helped with optimization efforts.

In our experiments the detailed profiling points to the parts where processes wait for other processes to finish their jobs and provide inputs and polling of ports to see if any inter-process messages have arrived as taking a large amount of time.

Optimization could simply mean fine-tuning the sleep/wait times of processes built into the platform. On the other hand there could be more serious issues and optimization would involve changing of code to do something differently.

Based on the profiling data, optimization could involve the following:

1. Critically examine the time spent in I/O waits and take remedial measures wherever possible
2. Examine the use of sleep statements and fine-tune their durations
3. Examine the use of heartbeat and ways to make it efficient
4. Optimize the time take to dynamically create and destroy virtual resources

Profile-led optimization makes use of the results generated by deterministic, functional and line-by-line profiling to get optimized code. If the execution environment fairly represents the usage scenario then profile guided feedback benefits optimization. Future work will involve demonstrating usefulness of the approach in carrying out optimization of OpenADN.

The scope of the problem at hand, however, was to see whether a combination of carefully selected profiling tools, working at different levels of the OpenADN program hierarchy (and by extension other similar platforms), would be able to pin-point the bottlenecks that could cause higher consumption of virtual resources. From the results discussed above it is clear that it would be in the interest of reduced cost and increased agility of doing the ASP business to carry out profiling at different levels as a precursor to optimization.

REFERENCES

[1] M. A. Khan, C. Hankendi, A.K. Coskun, M.C. Herbordt, “Software Optimization for Performance, Energy, and Thermal Distribution: Initial Case Studies,” International Green Computing Conference and Workshops (IGCC), 2011, pp. 1-6.

[2] OpenDaylight, http://www.opendaylight.org/project/technical-overview

[3] D. Eklov, N. Nikoleris and E. Hagersten, “A Profiling Method for Analyzing Scalability Bottlenecks on Multicores,” ACM, 2012.

[4] D. Jackson, & M. Rinard,“Software Analysis: A Roadmap,” *Proceedings of the IEEE International Conference on Software Engineering*, 2000, pp. 133-145.

[5] E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking.* The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2000.

[6] M. Rinard, “Analysis of Multithreaded Programs,” Proceedings of the 8th International Symposium on Static Analysis, 2001, pp. 1-19.

[7] D. Chen, N. Vachharajani, and R. Hundt, ‘‘Taming Hardware Event Samples for FDO Compilation, Proc. 8th Ann. IEEE/ ACM Int’l Symp. Code Generation and Optimization (CGO 10), ACM Press, 2010, pp. 42-52.

[8] Concurrency Profiling, http://msdn.microsoft.com/en-us/library/dd264994.aspx, 2013.

[9] Multithreaded Programming Guide, “Timers, Alarms, and Profiling,” Oracle, 2012, https://docs.oracle.com/cd/E26502_01/html/E35303/gen-90808.html.

[10] B. Lantz, N. Handigol, B. Heller, and V. Jeyakumar, “Introduction to Mininet,” https://github.com/mininet/mininet/wiki/Introduction-to-Mininet.

[11] The Python Profilers, https://docs.python.org/2/library/profile.html

[12] The Zeromq messaging library, www.zeromq.org

[13] Kernprof Line_Profiler, https://github.com/rkern/line_profiler.

[14] S. Paul, "Software Defined Application Delivery Networking" (2014). *All Theses and Dissertations (ETDs).* Paper 1331 http://openscholarship.wustl.edu/ etd/1331

[15] J. Mars and R. Hundt, ‘‘Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations,’’ Proc. 2009 Int’l Symp. Code Generation and Optimization (CGO 09), IEEE CS Press, 2009, pp. 169-179.

[16] G. D. Waddington, N. Roy and D.C. Schmidt, “Dynamic Analysis and Profiling of Multi-threaded Systems,” IGI Global 2009.

[17] Wikibooks,“Introduction to Software Engineering/Testing/Profiling,” 20,http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing/Profiling.

[18] Intel Whitepaper, “Optimizing Software for Multi-core Processors,” http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/multicore-optimizing-software.html