Send Orders for Reprints to reprints@benthamscience.ae

Recent Advances in Communications and Networking Technology, **2016,** *5,* **000-000**

Performance Evaluation of Multi-Cloud Management and Control Systems

Lav Gupta[1], Raj Jain[1,*], Mohammed Samaka[2], Aiman Erbad[2] and Deval Bhamare[2]

[1]*Department of Computer Science and Engineering, Washington University in St Louis, St. Louis, MO, USA and* [2]*Department of Computer Science and Engineering, Qatar University, Doha, Qatar*

Raj Jain

Abstract: Most global enterprises and application service providers need to use resources from multiple clouds managed by different cloud service providers, located throughout the world. The ability to manage these geographically distributed resources requires use of specialized management and control platforms. Such platforms allow enterprises to deploy and manage their applications across remote clouds that meet their objectives. Generally, these platforms are multi-threaded, distributed and highly complex. They need to be optimized to perform well and be cost effective for all players. For optimization to succeed, it has to be preceded by profiling and performance evaluation. In this paper we present techniques to profile such platforms using OpenADN as a running example. The effectiveness of using profiling data with the two factor full factorial design to analyze the effect of workloads and other important factors on the performance, has been demonstrated. It is seen that the workload, of varying number of users and hosts, does not have a significant impact on the performance. On the other hand, functions like host creation and polling have significant impact on the execution time of the platform software, indicating potential gains from optimization.

Keywords: Cloud Management and Control, Cloud Service Provider, Multi-Cloud, Network Service Provider, OpenADN, Optimization, Performance Evaluation, Profiling.

Received: November 11, 2015 **Revised: February 10, 2016** **Accepted: February 11, 2016**

1. INTRODUCTION

The ability to deploy and manage resources across multiple clouds is becoming increasingly important for enterprises. This paper focuses on enterprises that use virtual resources for their own applications or who provide services to others. We refer to both the types as application service providers (ASPs). To manage applications across multi-cloud infrastructure these enterprises need an appropriate control and management platform.

Each individual Cloud Service Provider (CSP) offers virtualized resources through diverse control and management platforms, like OpenStack and Amazon's EC2. Similarly, Network Service Providers (NSPs) offer virtual network services, to link up multiple clouds, through management and control platforms like OpenDaylight. In such a situation enterprises obtaining resources from many CSPs and NSPs would have to deal with many different management and control platforms. This would make their tasks difficult unless there is a multi-cloud management platform to assist them. The OpenADN platform (Open Application Delivery Network platform being developed at the Washington University in St Louis, MO, USA) does exactly the same [1]. It is interposed between user enterprises and multiple public clouds and provides an integrated view of all the resources to the ASPs so that they can deploy and manage multi-cloud applications with ease and in best possible manner.

Multi-cloud management and control platforms need to be optimized to efficiently utilize resources and minimize operational costs. Optimization can be expensive and time consuming and needs apriori understanding of a platform's behavior. Multithreading technique of software modularization and concurrent execution used in these platforms makes understanding their behavior difficult. One method of analyzing such a platform is through profiling based performance evaluation.

*Address correspondence to this author at the Department of Computer Science and Engineering, Washington University in St Louis, 509 Bryan Hall, Campus Box 1045, 1 Brookings Drive, St Louis, MO, USA; Tel: 314-825-0063; E-mail: jain@wustl.edu

2215-0811/16 $58.00+.00

The main contributions of this paper are twofold 1) To elaborate how behavior of such a platform can be analyzed to obtain data for optimization, while it is in operation *i.e.* obtaining and using virtual resources from multiple clouds and 2) To evolve a methodology to examine the usefulness of the identified factors for optimization of the platform and avoid unnecessary optimization efforts.

Section 2 gives some of the useful related research. Section 3 gives an understanding of architecture of the OpenADN platform that was profiled and evaluated. In Section 4 we take up the performance modeling of OpenADN, selection of profiling techniques and the experimental technique used for performance evaluation. Section 5 describes the profiling methodology used, the virtual set up for conducting the experiment and the actual profiling data obtained. Section 6 discusses the experimental design and the results obtained. Section 7 summarizes the paper and indicates future directions.

2. RELATED RESEARCH

Profiling and optimization have been well documented as important constituents of software systems design and implementation. Use of these techniques, to improve the performance of cloud management systems and applications running on cloud resources, is in the nascent stage. G. Ren *et al* have in [2] presented an elaborate exposition of using continuous profiling technique for improving performance of datacenter applications. The authors argue that performance and utilization characteristics are critically important, because even minor performance improvements translate into huge cost savings. Their contention is that the traditional performance analysis is complicated for datacenter applications and it is easier to monitor them on live traffic. To be useful, the tools must be non-intrusive and introduce minimal overhead. The authors explain Google Wide Profiling (GWP) as a continuous profiling methodology. It samples usage across machines in multiple data centers and collects data about events such as stacks, lock contentions, heap profile and kernel events. GWP profiles provide performance insights for cloud applications.

In [3] the authors propose MIMIR, a dynamic profiling framework that can be used in conjunction with cache service such as memcached. The profiler enables cache operators to dynamically project the cost and performance impact from adding or removing memory resources within a distributed in-memory cache. The authors claim 98% accuracy and 2−5% overhead on request latency and throughput. They conclude that online cache profiling can be a practical tool for improving provisioning of large caches. For system wide optimization Hung *et al*. in [4] assert that energy and computational resources are most critical limitations. For accurate energy and performance prediction they suggest modeling energy-states of each hardware component and time spent in each state. For predicting resources accurately software execution needs to be tracked in actual environment. According to them conventional tools do not fare well with simulators so they have developed a framework called the virtual performance analyzer (VPA). This analyzer vests virtual machines with profiling/tracing capabilities and effective tools to analyze important hardware-software interactions in the system.

Palanisamy *et al*. have proposed Cura for provisioning cost-effective MapReduce services in a cloud [5]. It leverages MapReduce profiling to automatically create the best cluster configuration for the jobs. They have reported upto 80% reduction in cloud compute infrastructure cost with upto 65% reduction in job response times for Facebook-like workloads.

Profile-guided optimization has a huge potential to save costs for datacenters. In [6] authors argue that hardware features are inflexible limiting the types of data that can be gathered. On the other hand, instrumentation-based profiling can provide more flexible and targeted information gathering. In order for these techniques to be useful for datacenters, overhead needs to be contained to less than a few percent in terms of both throughput and latency. The authors propose instant profiling, an instrumentation sampling technique using dynamic binary translation. In this technique normal execution is interleaved with instrumented execution. They have achieved less than 6% slowdown and 3% computational overhead on average.

While there are some good works that deal with profiling and optimizing cloud based applications, to the best of our knowledge no other work has focused on behavioral analysis and optimization of multi-cloud management and control platform.

3. ANATOMY OF A MULTI-CLOUD MANAGEMENT PLATFORM

A brief description of the architecture of OpenADN[1] is provided in this section to help the readers fully appreciate the discussion on the main theme of profiling and optimization of such platforms.

3.1. Components of OpenADN

OpenADN is interposed between various single cloud management systems, owned by different CSPs, and the enterprise intending to use resources on multiple public clouds. It has two types of external interfaces. The first set of interfaces, also called northbound interfaces, is for the application developers, application architects, and application deployment administrators to define the application resource requirements and deployment policies. The second set of interfaces, or southbound interfaces, allow OpenADN to interact with the management and control systems of the cloud and network service providers to manage their virtual resources.

The key components of the OpenADN multi-cloud management platform are shown in Fig. (**1**). The global manager boots up the platform at application run time and co-ordinates with other clouds for acquisition of resources. The global controller is part of the hybrid control plane of the OpenADN that consists of a global controller and one or more local controllers. After bootstrap, the global controller takes over and launches one workflow manager for each workflow. The workflow manager checks for resources and launches workflow instances. One local controller is instantiated for each data center from which resources are leased. The local controller launches a new thread for managing each new virtual machine. The data plane is distributed in which each node has a control agent through which OpenADN control plane manages and controls the data plane node.

For multi-cloud deployments, the application administrator has to configure appropriate policies in the global controller. These policies include specifying how to distribute the application delivery network deployment initially and during runtime. It is important to decide when and where to instantiate new instances and shutdown or move existing instances to support change in the application context. This massively distributed data plane structure makes the performance evaluation of OpenADN difficult and calls for specialized techniques that we shall discuss in the following sections.

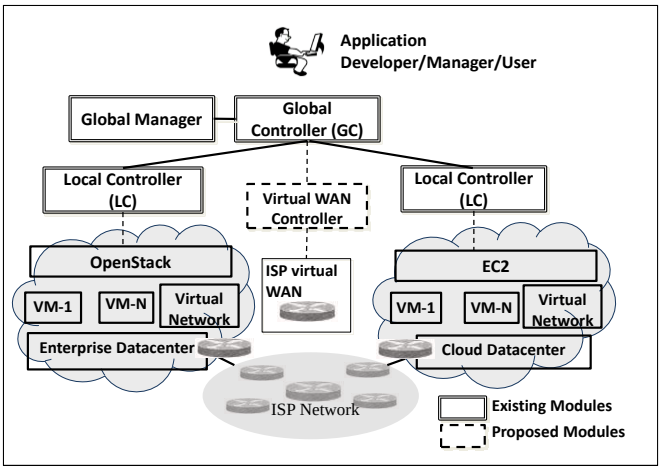


Fig. (1). Key Components of OpenADN Platform.

3.2. Design and Coding Considerations

Most of the control plane code has been implemented using Python while the data plane has been implemented with a mix of C and Python. The total size of the code base is currently about 10,000 lines of code. OpenADN has been designed as modular software to provide code readability and maintainability. Modularity also restricts inter-module interference in case of failures. Multiple operations are performed simultaneously to support multiple clouds, multiple users and multiple services. Partitioning of OpenADN into concurrently executable modules leads to better utilization of the hardware resources and ensures that the system as a whole makes progress all the time [3, 7]. The platform modules and application services are executed in separate processes. Application services are designed as external modules that connect to the platform through an external communication interface. A failed application service(s) can be handled by the platform without affecting other services. The services may run on the same or separate hosts. On the same host they use inter-process communication (IPC) while on different hosts they communicate using the network transport layer.

In each host, platform modules may run within the same address space (same process) but in sepa-

[1] Open Application Delivery Network is a multi-cloud management plat under development at Washington University in St Louis.

rate threads to achieve concurrency. The ports handling packet level services and message level services run in separate processes because kernel network stack has been used for packet level communication. These threads share the process heap, which provides them a way to communicate with each other. However, in order to avoid fate-sharing completely, threads communicate inside the platform process through messaging. Depending on the profiling techniques used, these design considerations could affect the outcome of profiling.

4. PERFORMANCE MODELING OF A MULTI-CLOUD MANAGEMENT PLATFORM

This section defines the experimental modeling that assists in gathering behavioral data, while the platform is in operation, and carries out performance evaluation decision for optimization. While we have considered the OpenADN as a representative platform, the technique described here would apply to any other platform.

4.1. Goal of the Study and System Definition

The main goal of the study is to first methodically and scientifically locate areas of code in the platform that might cause it to consume unduly large amount of computing resources during bootstrap and normal operation. Then we apply an experimental design technique to find whether any of the located hotspots have significant impact on the metric described in sub-section 4.3. The actual setup of the experiments described in detail in subsection 4.4 would be used as the basis for carrying out the collection of profiling data using techniques mentioned in Section 5.

4.2. Services And Their Outcomes

OpenADN offers all the basic services expected of such a platform, *e.g.*, allocation of resources from multiple clouds, distribution of applications, scaling/de-scaling, and performance assurance of workflows. Its uniqueness, however, lies in the additional application and network layer services it offers for highly distributed and multi-threaded applications to run on multiple clouds [1]. These services include:

1) *Application layer services including message and packet level services* (sometimes called middleboxes)
   a) *Message level services:* webservers, database servers, and web firewalls.
   b) *Packet level services:* Intrusion detection and intrusion prevention systems.
2) *Network level services* like packet forwarding and routing

The expected outcome is effective use of resources, assurance of meeting quality of service and dynamically ensuring efficient operation of the system. However, if the system operates sub-optimally, say under a computationally demanding application, it results in higher cost, exactly opposite of what it was supposed to achieve. Performance parameters like latency may be met for some applications and may not be met for others at all times. Communication among message level or packet level devices may take unduly long time. These issues were kept in mind for deciding metrics and parameters as discussed in the next sub-section.

4.3. Metrics, Factors and Parameters

The main metric is the CPU time taken to execute the platform software during the complete process of bootstrap and as the services start. Execution times for individual functions that consume a large amount of time would be of interest. The *system parameters* include: the type of virtual machines setup, storage capacities, intra- and inter-cloud network bandwidth. The *workload parameters* that affect the metric are the users' requests for services, types of services – message or packet service and amount of resources available.

4.4 Evaluation Techniques And Experiment Design

Platforms managing resources across clouds tend to be multi-threaded and distributed. Deterministic multi-layered profiling technique can be applied in such a situation for gathering data for performance evaluation.

The existing prototype of OpenADN was used to set up experiment and take measurements. The platform software was loaded and executed in the virtual environment as described in the next section. Experiments were conducted to observe effect of different workloads (involving varying number of clients and hosts) and also various functions of platform on CPU time required.

The experiment was designed as a two factor full factorial design without replications [8]. The reason for choosing this design actually became obvious while conducting profiling studies and collecting data. We had a situation where two sets of parameters, *i.e.*, functions (host creation, polling and sleep) and workloads (users and the number of hosts) were affecting the CPU time. A careful control of these two sets of parameters was required. We assumed that the factors are categorical. A full factorial design with two factors functions (Aj) and workloads (Bi) having i, j=3 levels each. The results are deterministic in nature and, therefore, single replication of each experiment was considered sufficient.

The methods used and data collected are given in Section 5. Experimental results and analysis are discussed in Section 6.

5. GATHERING BEHAVIORAL DATA

5.1. Selection of profiling techniques and experimental setup

Complex software of a multi-cloud management platform would always have regions of code that consume disproportionate amount of computing, storage or network resources. This would lead to leasing more resources than are necessary, pushing up the deployment cost and increasing latency. In [9], the authors have reported detailed work in the area of understanding behavior of software under execution and conclude that dynamic analysis is the only practical way to get absolute timing of events. Three profiling methods were found to be useful, to varying extent, for multi-cloud management platforms: static, dynamic and concurrent analysis [10]. Static analysis involves model checking to explore loops and their interactions exhaustively to ensure correctness properties. Static-analysis techniques give assessment of relative time and temporal ordering and do not give absolute time [11]. Dynamic analysis could be statistical where state of the program is sampled to make a relative assessment of timing of events or deterministic where events can be precisely timed by using instrumented code. Instrumentation systems can monitor coarse or fine-grained behavior [5]. While the event timing with deterministic profiling takes into account interaction of threads, using concurrent analysis, a more precise thread level examination can be made.

The layered profiling model shown in Fig. (**2**) helped us to progressively get more detailed information and zero in on the problem areas.

Platform level profiling provided overall execution data for the complete platform. However, as we shall see, while it gives useful information to start with, it does not pinpoint the problems in the code. Function level profiling enables us to measure the CPU times for execution of various functions so that we could isolate the blocks those took disproportionate time to execute. It does not, however, tell us the exact location of these time consuming operations. Some functions are called repeatedly in different modules. Thus, statement level profiling was carried out to get the location of the calls that were resulting in inappropriate behavior. Deterministic and concurrent techniques were used to be able to measure absolute timing of events for carrying out the experimental study.

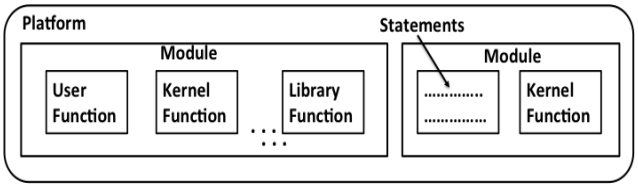


Fig. (2). The Layered Profiling Model.

The experimental setup consisted of one global controller, resources from datacenters of two clouds with one local controller each, a distributed data plane with 7 hosts per local controller and a 'fakenameserver' (labeled as "Nameserver"), all implemented on virtual machines. A client node was also created to simulate different number of users, as we shall see later in the experimental results. The setup is shown in Fig. (**3**).

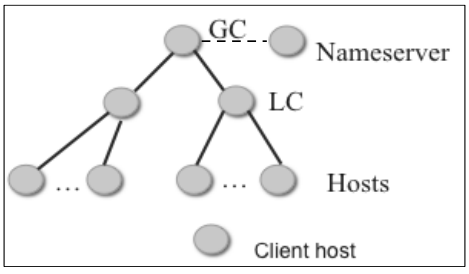


Fig. (3). Virtual resources used for the experiments.

5.2. Collection and consolidation of data

1) Platform level analysis: To get a broad idea of the efficiency of the platform code executing in a virtualized environment, the built in timing utility

of the operating system was used. Table 1 shows the CPU times (in seconds) for user space functions, kernel (system) functions, total of user and kernel and the overall run time of the platform across seven runs.

Graphical representation of the data is given in Fig. (**4**). Of the average total elapsed time of 49.269 seconds for which the platform software was executed, the time spent in user functions and kernel space was 1.3% and 1.77%, respectively.

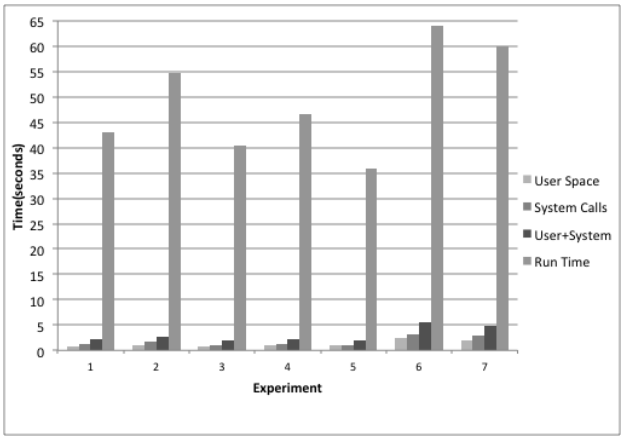


Fig. (4). User, System and overall CPU time for OpenADN.

This gives a sense that a large part of the time is spent in activities such as 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.

Table 1. Platform level execution data.

Run	User Space	System Calls	User+System	Run Time
1	0.872	1.248	2.120	43.068
2	0.948	1.796	2.744	54.675
3	0.864	1.064	1.928	40.464
4	0.940	1.256	2.196	46.650
5	1.016	1.040	2.056	35.936
6	2.500	3.096	5.596	64.050
7	2.004	2.910	4.914	60.041
Averages	1.306	1.773	3.079	49.269
% of Run Time	2.651	3.598	6.250	

2) Function Level Profiling and Analysis: Python library provides routines to collect behavioral data at the function level. These routines provide a set of statistics that describes how many times different functions are called and how much time the CPU is spending to execute various modules. The statistical data collected needs to be processed through some other conversion routintes like 'pstats' to make them amenable to analysis. A large volume of data was produced of which a part of output is shown in Fig. (**5**).

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.001    0.001   59.801   59.801 driver_mininet.py:2(<module>)
     1    0.675    0.675   59.792   59.792 driver_mininet.py:209(start_sim)
101047   42.045    0.000   42.045    0.000 {built-in method poll}
     3   15.012    5.004   15.012    5.004 {time.sleep}
     1    0.001    0.001    5.151    5.151 driver_mininet.py:186(start_hosts)
     1    0.000    0.000    5.012    5.012 driver_mininet.py:163(start_fakeNameServer)
     1    0.000    0.000    5.010    5.010 driver_mininet.py:134(start_gc_lighthouseController)
   104    0.161    0.002    0.981    0.009 util.py:25(quietRun)
     1    0.000    0.000    0.896    0.896 driver_mininet.py:42(__init__)
  3502    0.015    0.000    0.876    0.000 driver_mininet.py:264(write)
     1    0.001    0.001    0.856    0.856 driver_mininet.py:67(allocate_singleSwitchTopo)
    19    0.000    0.000    0.813    0.043 node.py:300(linkTo)
    19    0.001    0.000    0.627    0.033 util.py:79(makeIntfPair)
  3502    0.593    0.000    0.593    0.000 {method 'write' of 'file' objects}
  3502    0.268    0.000    0.268    0.000 {method 'flush' of 'file' objects}
   195    0.002    0.000    0.259    0.001 node.py:235(cmd)
   125    0.003    0.000    0.248    0.002 subprocess.py:619(__init__)
```

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 subfunctions seconds, **Percall**: cumtime divided by primitive calls, **filename**: data for each function

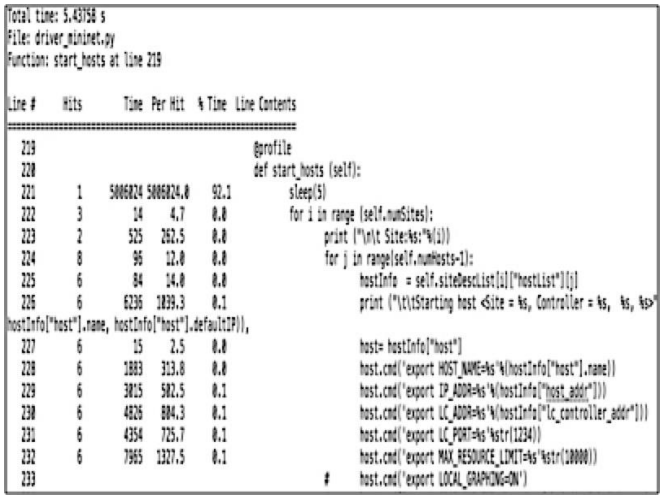
Fig. (5). Extract of function level profile.

From the 2nd and 4th lines of Fig. (**5**), it can be seen that the platform was executed for a total of 59.801 seconds. Out of this the polling function took 42.045 seconds. OpenADN uses the ZeroMQ™ polling function that provides communication between modules on different virtual hosts.

The communicating services have to poll the sockets to check for the new incoming message. If a large amount of time is taken then this may be an indication that the entire process of platform execution may be slowing down. To know the exact location of this time consuming operation and other such operations statement level profiling was done.

3) Statement Level Profiling: 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 a spe-

cial method that may not be so obvious. A more detailed line-by-line analysis of the program was undertaken to find out which parts of the program take more CPU time. Workload was varied to get CPU times for various statements and identify the functions that should be taken up for further analysis. Fig. (**6**) shows a section of the profiling output with a large proportion of sleep time (92%) and also the time taken for creation of hosts.



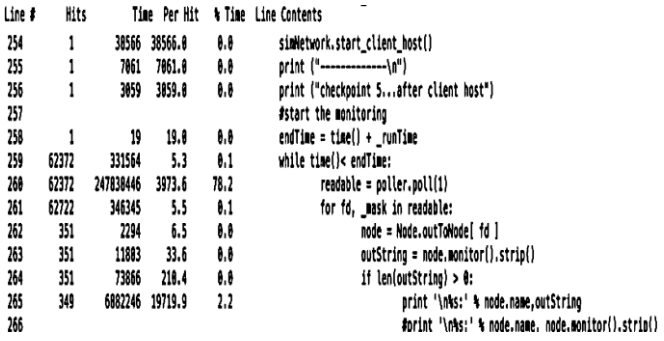
```
Total time: 5.43758 s
File: driver_mininet.py
Function: start_hosts at line 219

Line #    Hits       Time  Per Hit   % Time  Line Contents
==============================================================
   219                                           @profile
   220                                           def start_hosts (self):
   221         1    5006824 5006824.0     92.1          sleep(5)
   222         3         14      4.7      0.0          for i in range (self.numSites):
   223         2        526    262.5      0.0              print ("\n\t Site=%s:"%(i))
   224         8         96     12.0      0.0              for j in range(self.numHosts-1):
   225         6         84     14.0      0.0                  hostInfo  = self.siteDescList[i]["hostList"][j]
   226         6       6236   1039.3      0.1                  print ("\t\tStarting host <Site = %s, Controller = %s,  %s, %s>"
hostInfo["host"].name, hostInfo["host"].defaultIP)),
   227         6         15      2.5      0.0                  host= hostInfo["host"]
   228         6       1883    313.8      0.0                  host.cmd('export HOST_NAME=%s'%(hostInfo["host"].name))
   229         6       3015    502.5      0.1                  host.cmd('export IP_ADDR=%s'%(hostInfo["host_addr"]))
   230         6       4826    804.3      0.1                  host.cmd('export LC_ADDR=%s'%(hostInfo["lc_controller_addr"]))
   231         6       4354    725.7      0.1                  host.cmd('export LC_PORT=%s'%str(1234))
   232         6       7965   1327.5      0.1                  host.cmd('export MAX_RESOURCE_LIMIT=%s'%str(10000))
   233                                                   #      host.cmd('export LOCAL_GRAPHING=ON')
```

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

Fig. (6). Extract of statement level profile showing large sleep time 2000 users and 4 hosts

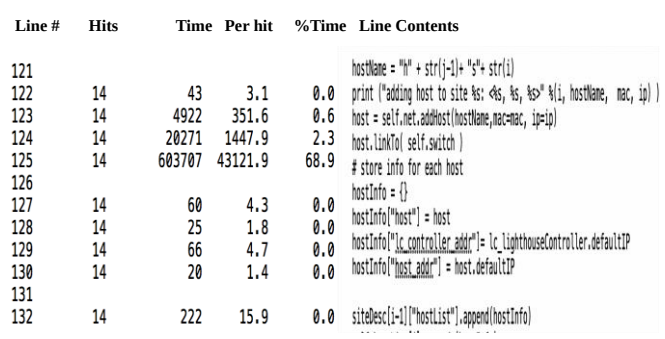
Fig. (**7**) shows that, in this section of the profile, the polling function takes 78.2% of the time. Legends of Fig. (**6**) are applicable.



```
Line #    Hits       Time  Per Hit   % Time  Line Contents
   254       1      38566  38566.0      0.0      simNetwork.start_client_host()
   255       1       7061   7061.0      0.0      print ("------------\n")
   256       1       3059   3059.0      0.0      print ("checkpoint 5...after client host")
   257                                            #start the monitoring
   258       1         19     19.0      0.0      endTime = time() + _runTime
   259   62372     331564      5.3      0.1      while time()< endTime:
   260   62372  247838446   3973.6     78.2              readable = poller.poll(1)
   261   62722     346345      5.5      0.1              for fd, _mask in readable:
   262     351       2294      6.5      0.0                  node = Node.outToNode[ fd ]
   263     351      11883     33.6      0.0                  outString = node.monitor().strip()
   264     351      73866    210.4      0.0                  if len(outString) > 0:
   265     349    6882246  19719.9      2.2                      print '\n%s:' % node.name,outString
   266                                                        #print '\n%s:' % node.name, node.monitor().strip()
```

Fig. (7). Extract of Statement Level Profile showing large time taken by Poller 2000 users and 4 hosts.

Fig. (**8**) shows 68.9% of the CPU time taken by host creation and linking. Legends of Fig. (**8**) are applicable.



```
Line #    Hits       Time  Per hit   %Time  Line Contents
   121                                          hostName = "h" + str(j-1)+ "s"+ str(i)
   122        14         43      3.1     0.0   print ("adding host to site %s: <%s, %s, %s>" %(i, hostName,  mac, ip) )
   123        14       4922    351.6     0.6   host = self.net.addHost(hostName,mac=mac, ip=ip)
   124        14      20271   1447.9     2.3   host.linkTo( self.switch )
   125        14     603707  43121.9    68.9   # store info for each host
   126                                          hostInfo = {}
   127        14         60      4.3     0.0   hostInfo["host"] = host
   128        14         25      1.8     0.0   hostInfo["lc_controller_addr"]= lc_lighthouseController.defaultIP
   129        14         66      4.7     0.0   hostInfo["host_addr"] = host.defaultIP
   130        14         20      1.4     0.0
   131                                          siteDesc[i-1]["hostList"].append(hostInfo)
   132        14        222     15.9     0.0
```

Fig. (8). Extract of profile for 2000 users and 8 hosts showing 68.9% host creation and linking time.

5.3. Concurrency Profiling Data

While the recursive function level profiling, that includes timing of execution of sub-functions and statement level profiling, reflects the effect of execution of various threads, individual thread behavior may not be evident. To get a better understanding of the multi-threaded platform, thread level profiling was carried out while the program was in execution. A sample of concurrency profile is given in Fig. (**9**). This aspect of profiling is a part of future work.

The internal polling operation at line 8 of the output in Fig. (**9**) shows that this function was called 65804 times after the global controller was started (even before the local controllers were activated) and a total of about 0.25 sec were spent in this operation. This amounts to about 26% of the time the thread spends in this function and the subfunctions it calls.

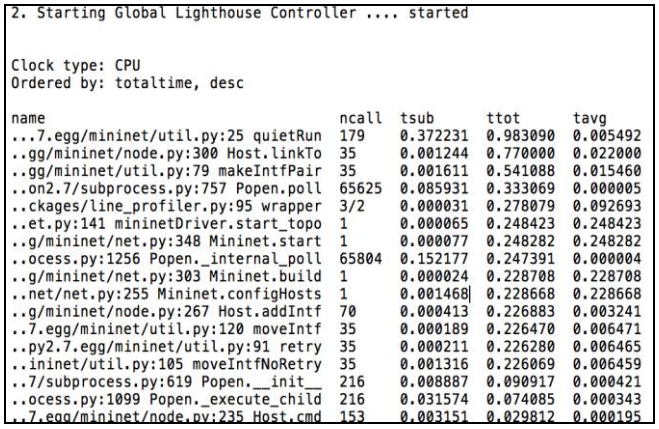
6. EXPERIMENTAL RESULTS AND ANALYSIS

From all the profiling runs with different workloads it is observed that three types of activities are consuming a large amount of time during the execution of the platform software:

1. Creation and linking of the host to the network
2. Polling of sockets for inter-service communication
3. Sleep function

The percentage time consumed by these functions varied with the workload. The workload was varied by changing the number of users from 500 to 2000, each accessing from a list of web pages, and also by creating different number of hosts per

cloud varying from 4 to 16 for hosting the platform modules as well as the application. It was seen from the measurements taken that the execution times across functions varied more with the number of hosts created than with the number of users. It was, therefore, decided to carry out a detailed performance analysis for a fixed number of 2000 users and the number of hosts varying from 4 to 16 in steps of 4.



```
2. Starting Global Lighthouse Controller .... started


Clock type: CPU
Ordered by: totaltime, desc

name                                    ncall  tsub      ttot      tavg
...7.egg/mininet/util.py:25 quietRun    179    0.372231  0.983090  0.005492
..gg/mininet/node.py:300 Host.linkTo    35     0.001244  0.770000  0.022000
..gg/mininet/util.py:79 makeIntfPair    35     0.001611  0.541088  0.015460
..on2.7/subprocess.py:757 Popen.poll    65625  0.085931  0.333069  0.000005
..ckages/line_profiler.py:95 wrapper    3/2    0.000031  0.278079  0.092693
..et.py:141 mininetDriver.start_topo    1      0.000065  0.248423  0.248423
..g/mininet/net.py:348 Mininet.start    1      0.000077  0.248282  0.248282
..ocess.py:1256 Popen._internal_poll    65804  0.152177  0.247391  0.000004
..g/mininet/net.py:303 Mininet.build    1      0.000024  0.228708  0.228708
..net/net.py:255 Mininet.configHosts    1      0.001468  0.228668  0.228668
..g/mininet/node.py:267 Host.addIntf    70     0.000413  0.226883  0.003241
..7.egg/mininet/util.py:120 moveIntf    35     0.000189  0.226470  0.006471
..py2.7.egg/mininet/util.py:91 retry    35     0.000211  0.226280  0.006465
..ininet/util.py:105 moveIntfNoRetry    35     0.001316  0.226069  0.006459
..7/subprocess.py:619 Popen.__init__    216    0.008887  0.090917  0.000421
..ocess.py:1099 Popen._execute_child    216    0.031574  0.074085  0.000343
..7.egg/mininet/node.py:235 Host.cmd    153    0.003151  0.029812  0.000195
```

name: function name, ncall: callcount of the function, tsub: time spent in the function, ttot:time spend in the function and sub-functions, tavg: ttot/ccnt

Fig. (9). Concurrency Profiling with 2000 users and 16 hosts.

As noted in subsection 4.4, a two factor full factorial design was used with the CPU time (the observed function time rationalized with total module execution time) being observed and the factors as functions and workloads. Table **2** gives the CPU times for these factors. The CPU time required for execution of the three functions has been rationalized with the total module times to make them comparable across runs.

Table 2. CPU Time for functions and workloads.

	Functions				
Work-loads	**Host creation**	**Polling**	**Sleep**	**Row Mean**	**Row effects**
2000/4 hosts	0.3307	0.7865	0.9641	0.6938	-0.0063
2000/8 hosts	0.4316	0.7571	0.9325	0.7071	0.0070
2000/16 hosts	0.5186	0.7174	0.862	0.6993	-0.0007
Column Mean	0.4270	0.7537	0.9195	0.7001	
Column Effects	-0.2731	0.0536	0.2195		

6.1. Effect of the selected factors

Table **2** data can be used for calculation of row (workload) effects (β_i) and column (functions) effects (α_j). The grand mean (μ) is 0.7001. From the table it is seen that the effect of host creation is 39% less than the average CPU time while that of Polling and Sleep are 7.66% and 31.35% more. The workload effects are small and within 1% of average workload either way.

6.2. Explanation of Variation

The total variation of CPU time (y) can be attributed to the two factors: Functions and Workloads and to the experimental errors. Next we calculate the sum of squares explained by these factors. These values are shown in the second column of Table **3**. From these values we calculate the variations explained by various factors. The percentage variation explained by functions and workloads is 68.29% and 0.05%, respectively. The unexplained variation (due to errors) comes out to be 31.68%. From the above we conclude the functions selected are important for optimization regardless of the workloads.

6.3. Analysis of Variance and Visual Results

Now we have sufficient information to complete the ANOVA table to test the significance of the two factors as far as the CPU time is concerned. The number of functions are a=3 and the number of workloads are b=3.

We obtain the respective mean squares by dividing the corresponding sum of squares by its degree of freedom and compute F-ratio by finding the ratio with mean square of errors.

Table 3. ANOVA table for functions and workloads.

Component	Sum of Squares	% Variation	Degrees of Freedom	Mean square	F computed	F Table
y	4.8131		9			
μ	4.4107		1			
y-μ	0.4024	100	8			
α_j	0.2748	68.29	2	0.1374	8.6212	$F_{0.90,2,4}$= 4.32
β_i	0.0002	0.05	2	0.0001	0.0031	$F_{0.90,2,4}$= 4.32
e_i	0.1275	31.68	4	0.0159		

From Table **3** we observe that the workloads had comparable runtimes. This ensured that effect of the functions selected is not overshadowed by

the differences in the workload run times. The F-ratio calculated for Functions is greater than that obtained from the table (at 90% confidence level) so they are significant for our study. F Level of workload is less so they are not significant.

Visual examination of residuals and responses can be seen from the graphs below (Fig. **10**). In order to check the homogeneity of the error variance we obtain errors (Table **4**) and plot them against predicted response.

Table 4. The estimated y and the residuals.

$\hat{y}_{ij}=\mu+\alpha_j+\beta_i$			$e_i=y_{ij}-\hat{y}_{ij}$		
0.4207	0.7474	0.9132	-0.0900	0.0391	0.0509
0.4340	0.7607	0.9265	-0.0024	-0.0036	0.0060
0.4263	0.7530	0.9188	0.0923	-0.0356	-0.0568

The residuals and responses are given in Table **4** and the corresponding graph in Fig. (**10a**). The residuals are scattered uniformly with zero mean and do not show a trend. A normal quantile-quantile plot was plotted from the data in Table **7b** and is shown in Fig. (**10b**). The plot is approximately linear suggesting normal distribution of residuals.

6.4 Confidence Intervals for the effects

To check the sanity of our results we took the analysis further by calculating standard deviations (SDs) and 90% confidence intervals (CIs) for the effects related to functions and workloads (Table **5**). The 't' value used for the calculation of CIs is for 90% confidence interval and 4 degrees of freedom (the degrees of freedom for the errors).

Table 5. Calculation of CI of effects.

MSE =0.0159	s_e= 0.1261
SD of grand mean	s_μ=0.0420
SD of α_j	$s_{\alpha j}$ 0.0594
SD of β_is	$s_{\beta i}$ 0.0594
90% confidence interval for α	90% confidence interval for β
(0.3003, 0.5537)	(0.5670, 0.8205)
(0.6269, 0.8804)	(0.5803, 0.8338)
(0.7928, 1.0463)	(0.5726, 0.8261)

It can be seen that CIs of the functions are all significant. Also the means of α do not lie in the CI of each other so they are significantly different from each other. This implies that optimization with respect to these three functions should result in improvement in performance of the platform. The workloads on the other hand have their means in the CI of the others so they are not significantly different from each other.

7. CONCLUSIONS

In this paper, we presented the performance evaluation of OpenADN, a complex multi-threaded and distributed multi-cloud management platform in operation. Behavior of the platform is

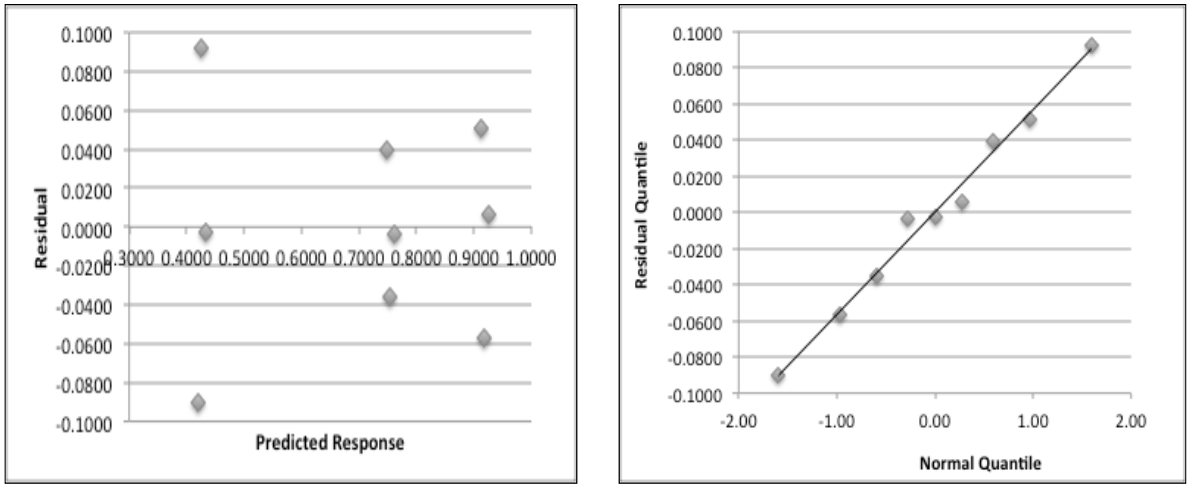


Fig. (10). a) Residual vs. Predicted Response, b) Quantile-Quantile Plot.

studied as it starts operating, obtains virtual resources from multiple clouds, allocates the resources to processes and then continues normal operation. The profiling data obtained through deterministic methods coupled with the two-factor full-factorial design helped us to determine whether the factors selected are significant and should be used for further optimization of the platform. The factors *viz*. Functions (host creation time, polling and sleep times) and Workloads (2000 users with 4, 8 and 16 hosts) were selected based on the various levels of profiling data collected. Workloads, involving varying number of users and hosts, did not have a significant impact on the performance and their effect was low and indistinguishable. On the other hand, after a detailed analysis we have reached the conclusion that all the functions, identified through profiling, have a significant effect on the CPU time of the platform. The CPU time in turn is an indicator of the performance of the platform. These functions can, therefore, be considered for optimization to achieve improved performance of the platform. It is, therefore, concluded that behavioral analysis through layered profiling and full factorial analysis of the data can reveal vital information about for optimization of multi-cloud management platforms.

CONFLICT OF INTEREST

The authors confirm that this article content has no conflict of interest.

ACKNOWLEDGEMENTS

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.

REFERENCES

[1] S. Paul, R. Jain, J. Pan, J. Iyer, D. Oran, "OpenADN: A Case for Open Application Delivery Networking," 22nd International Conference on Computer Communications and Networks (ICCCN), pp. 1-7, 2013

[2] Gang Ren Eric Tune Tipp Moseley Yixin Shi Silvius Rus Robert Hundt, "Google-Wide Profiling: A Continuous Profiling Infrastructure For Data Centers," IEEE Micro, Volume 30, Issue 4, pp 65-90, 2010

[3] T Saemundsson, H Bjornsson, Gregory Chockler, "Dynamic Performance Profiling of Cloud Caches," ACM Symposium on Cloud Computing, pp. 1-14, 2014

[4] S-H. Hung *et al*."System-Wide Profiling and Optimization with Virtual Machines," 17th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 395-400, 2012

[5] B. Palanisamy, A. Singh, and L. Liu, "Cost-effective resource provisioning for mapreduce in a cloud," Parallel and Distributed Systems, IEEE Transactions on, vol. PP, no. 99, pp. 1–1, 2014.

[6] H. K. Cho, T. Moseley, R. Hank, D. Bruening, S. Mahlke, "Instant Profiling: Instrumentation Sampling for Profiling Datacenter Applications," IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1-10, 2013

[7] Subharthi Paul, Raj Jain, Mohammed Samaka, Jianli Pan, "Application Delivery in Multi-Cloud Environments using Software Defined Networking, " Computer Networks Special Issue on cloud networking and communications, pp. 166-186, Feb 2014

[8] R. Jain, "The Art of Computer Systems Performance Analysis," Wiley, 1991

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

[10] L. Gupta, R. Jain and M. Samaka, "Dynamic Analysis of Application Delivery Network for Leveraging Software Defined Infrastructures," Proceedings of IEEE International Workshop on Software Defined Systems, pp. 305-310, 2015

[11] 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, pp. 169-179, 2009

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